

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

David Trachtenherz

February 24, 2011

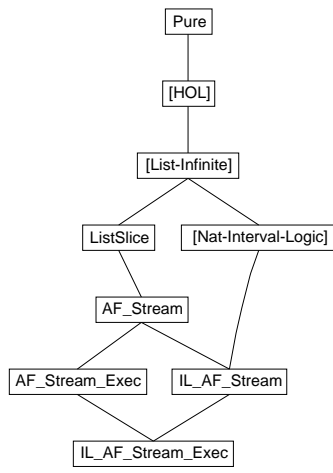
Abstract

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	ListSlice: Additional definitions and results for lists	4
1.1	Slicing lists into lists of lists	4
2	AF-Stream: AutoFocus message streams	11
2.1	Basic definitions	11
2.1.1	Time-synchronous streams	11
2.1.2	Time abstraction	13
2.2	Expanding and compressing lists and streams	15
2.2.1	Expanding message streams	15
2.2.2	Aggregating lists	23
2.2.3	Compressing message streams	27
2.2.4	Holding last messages in everly cycle of a stream . . .	36
2.2.5	Compressing lists	42
3	AF-Stream-Exec: Processing of message streams	46
3.1	Executing components with state transition functions	46
3.1.1	Basic definitions	46

3.1.2	Basic results	47
3.1.3	Connected streams	67
3.1.4	Additional auxiliary results	71
3.2	Components with accelerated execution	72
3.2.1	Equivalence relation for executions	72
3.2.2	Idle states	79
3.2.3	Basic definitions for accelerated execution	83
3.2.4	Basic results for accelerated execution	85
3.2.5	Basic results for accelerated execution with initial state in the resulting stream	95
3.2.6	Rules for proving execution equivalence	97
3.2.7	Idle states and accelerated execution	106
4	IL-AF-Stream: AutoFocus message streams and temporal logic on intervals	112
4.1	Stream views – joining streams and intervals	113
4.1.1	Basic definitions	113
4.1.2	Basic results	114
4.1.3	Results for intervals from <i>IL-Interval</i>	128
4.2	Streams and temporal operators	134
5	IL-AF-Stream-Exec: AutoFocus message stream processing and temporal logic on intervals	143
5.1	Correlation between Pre/Post-Conditions for <i>f-Exec-Comp-Stream</i> and <i>f-Exec-Comp-Stream-Init</i>	144
5.2	<i>i-Exec-Comp-Stream-Acc-Output</i> and temporal operators with bounded intervals.	145
5.3	<i>i-Exec-Comp-Stream-Acc-Output</i> and temporal operators with unbounded intervals and start/finish events.	147
5.4	<i>i-Exec-Comp-Stream-Acc-Output</i> and temporal operators with idle states.	149



1 ListSlice: Additional definitions and results for lists

```
theory ListSlice
imports ListInf
begin
```

1.1 Slicing lists into lists of lists

definition

```
list-slice :: 'a ilist  $\Rightarrow$  nat  $\Rightarrow$  'a list ilist
```

where

```
list-slice f k  $\equiv$   $\lambda x$ . map f [x * k..Suc x * k]
```

primrec

```
list-slice-aux :: 'a list  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  'a list list
```

where

```
list-slice-aux xs k 0 = []
```

```
| list-slice-aux xs k (Suc n) = take k xs # list-slice-aux (xs  $\uparrow$  k) k n
```

definition

```
list-slice :: 'a list  $\Rightarrow$  nat  $\Rightarrow$  'a list list
```

where

```
list-slice xs k  $\equiv$  list-slice-aux xs k (length xs div k)
```

definition

```
list-slice2 :: 'a list  $\Rightarrow$  nat  $\Rightarrow$  'a list list
```

where

```
list-slice2 xs k  $\equiv$  list-slice xs k @ (
```

```
if length xs mod k = 0 then [] else [xs  $\uparrow$  (length xs div k * k)])
```

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

definition

```
ilist-unslice :: 'a list ilist  $\Rightarrow$  'a ilist
```

where

```
ilist-unslice f  $\equiv$   $\lambda n$ . f (n div length (f 0)) ! (n mod length (f 0))
```

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

by (*induct* n, *simp*+))

lemma *list-slice-aux-nth*:

```
 $\bigwedge m xs$ . m < n  $\implies$  (list-slice-aux xs k n) ! m = (xs  $\uparrow$  (m * k)  $\downarrow$  k)
```

apply (*induct* n)

apply *simp*

apply (*simp* add: *nth-Cons'* *diff-mult-distrib*)

done

lemma *list-slice-length*: length (*list-slice* xs k) = length xs div k

by (*simp add: list-slice-def list-slice-aux-length*)
lemma *list-slice-0*: $list\text{-}slice\ xs\ 0 = []$
by (*simp add: list-slice-def*)
lemma *list-slice-1*: $list\text{-}slice\ xs\ (Suc\ 0) = map\ (\lambda x. [x])\ xs$
by (*fastsimp simp: list-eq-iff list-slice-def list-slice-aux-nth list-slice-aux-length*)

lemma *list-slice-less*: $length\ xs < k \implies list\text{-}slice\ xs\ k = []$
by (*simp add: list-slice-def*)
lemma *list-slice-Nil*: $list\text{-}slice\ []\ k = []$
by (*simp add: list-slice-def*)

lemma *list-slice-nth*:
 $m < length\ xs\ div\ k \implies list\text{-}slice\ xs\ k\ !\ m = xs\ \uparrow\ (m * k)\ \downarrow\ k$
by (*simp add: list-slice-def list-slice-aux-nth*)
lemma *list-slice-nth-length*:
 $m < length\ xs\ div\ k \implies length\ ((list\text{-}slice\ xs\ k)\ !\ m) = k$
apply (*case-tac length xs < k*)
apply *simp*
apply (*simp add: list-slice-nth*)
thm *less-div-imp-mult-add-divisor-le*
apply (*drule less-div-imp-mult-add-divisor-le*)
apply *simp*
done

thm *take-drop-eq-sublist-list*
lemma *list-slice-nth-eq-sublist-list*:
 $m < length\ xs\ div\ k \implies list\text{-}slice\ xs\ k\ !\ m = sublist\text{-}list\ xs\ [m * k..<m * k + k]$
apply (*simp add: list-slice-nth*)
apply (*rule take-drop-eq-sublist-list*)
apply (*rule less-div-imp-mult-add-divisor-le, assumption+*)
done

lemma *list-slice-nth-nth*:
 $[m < length\ xs\ div\ k; n < k] \implies$
 $(list\text{-}slice\ xs\ k)\ !\ m\ !\ n = xs\ !\ (m * k + n)$
apply (*frule list-slice-nth-length[of m xs k]*)
apply (*simp add: list-slice-nth*)
done

lemma *list-slice-nth-nth-rev*:
 $n < length\ xs\ div\ k * k \implies$
 $(list\text{-}slice\ xs\ k)\ !\ (n\ div\ k)\ !\ (n\ mod\ k) = xs\ !\ n$
apply (*case-tac k = 0, simp*)
thm *list-slice-nth-nth*
apply (*simp add: list-slice-nth-nth div-less-conv*)
done

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

```

apply (case-tac k = 0)
  apply (simp add: list-slice-0)
apply (simp add: list-eq-iff list-slice-length)
apply (simp add: div-mult-le min-eqR list-slice-nth)
apply (clarify, rename-tac i)
apply (subgoal-tac k ≤ length xs div k * k - i * k)
  prefer 2
  apply (drule-tac m=i in Suc-leI)
  apply (drule mult-le-mono1[of - - k])
  apply simp
apply (subgoal-tac length xs div k * k - i * k ≤ length xs - i * k)
  prefer 2
  apply (simp add: div-mult-cancel)
apply (simp add: min-eqR)
apply (simp add: drop-take)
done

```

```

lemma list-slice-append-mult:
   $\bigwedge xs. \text{length } xs = m * k \implies$ 
   $\text{list-slice } (xs @ ys) k = \text{list-slice } xs k @ \text{list-slice } ys k$ 
apply (case-tac k = 0)
  apply (simp add: list-slice-0)
apply (induct m)
  apply (simp add: list-slice-Nil)
apply (simp add: list-slice-def)
apply (simp add: list-slice-def add-commute[of - length ys] add-assoc[symmetric])
done

```

```

lemma list-slice-append-mod:
   $\text{length } xs \bmod k = 0 \implies$ 
   $\text{list-slice } (xs @ ys) k = \text{list-slice } xs k @ \text{list-slice } ys k$ 
apply clarify
thm list-slice-append-mult
apply (simp only: mult-commute[of k] list-slice-append-mult)
done

```

```

lemma list-slice-div-eq-1[rule-format]:
   $\text{length } xs \text{ div } k = \text{Suc } 0 \implies \text{list-slice } xs k = [\text{take } k \text{ } xs]$ 
by (simp add: list-slice-def)
lemma list-slice-div-eq-Suc[rule-format]:
   $\text{length } xs \text{ div } k = \text{Suc } n \implies$ 
   $\text{list-slice } xs k = \text{list-slice } (xs \downarrow (n * k)) k @ [xs \uparrow (n * k) \downarrow k]$ 
apply (case-tac k = 0, simp)
apply (subgoal-tac n * k < length xs)
  prefer 2
  apply (case-tac length xs = 0, simp)
  apply (drule-tac arg-cong[where f=λx. x - Suc 0], drule sym)
  apply (simp add: diff-mult-distrib div-mult-cancel)
thm list-slice-append-mult[of take (n * k) xs n k drop (n * k) xs]

```

apply (*insert list-slice-append-mult*[of *take (n * k) xs n k drop (n * k) xs*])
apply (*simp add: min-eqR*)
apply (*rule list-slice-div-eq-1*)
apply (*simp add: div-diff-mult-self1*)
done

term *concat*

term *list-unslice*

lemma *list-slice2-mod-0*:

$length\ xs\ mod\ k = 0 \implies list-slice2\ xs\ k = list-slice\ xs\ k$

by (*simp add: list-slice2-def*)

lemma *list-slice2-mod-gr0*:

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

by (*simp add: list-slice2-def*)

lemma *list-slice2-length*:

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

by (*simp add: list-slice2-def list-slice-length*)

lemma *list-slice2-0*:

$list-slice2\ xs\ 0 = (if\ (length\ xs = 0)\ then\ []\ else\ [xs])$

by (*simp add: list-slice2-def list-slice-0*)

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

by (*simp add: list-slice2-def list-slice-1*)

lemma *list-slice2-le*:

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

apply (*case-tac k = 0*)

apply (*simp add: list-slice2-0*)

apply (*drule order-le-less[THEN iffD1], erule disjE*)

apply (*simp add: list-slice2-def list-slice-def*)

apply (*simp add: list-slice2-def list-slice-div-eq-1*)

done

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

by (*simp add: list-slice2-def list-slice-Nil*)

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

$m < length\ xs\ div\ k \implies list-slice2\ xs\ k ! m = list-slice\ xs\ k ! m$

by (*simp add: list-slice2-def list-slice-length nth-append*)

lemma *list-slice2-last*:

$[length\ xs\ mod\ k > 0; m = length\ xs\ div\ k] \implies$

$list-slice2\ xs\ k ! m = xs\ \uparrow\ (length\ xs\ div\ k * k)$

by (*simp add: list-slice2-def nth-append list-slice-length*)

lemma *list-slice2-nth*:

$[m < length\ xs\ div\ k] \implies$

$list-slice2\ xs\ k ! m = xs\ \uparrow\ (m * k) \downarrow k$

by (*simp add: list-slice2-def list-slice-length nth-append list-slice-nth*)

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

$m < length\ xs\ div\ k \implies length\ (list-slice2\ xs\ k ! m) = k$

by (*simp add: list-slice2-def nth-append list-slice-length list-slice-nth-length*)

lemma *list-slice2-nth-length-eq2*:
 $\llbracket \text{length } xs \text{ mod } k > 0; m = \text{length } xs \text{ div } k \rrbracket \implies$
 $\text{length } (\text{list-slice2 } xs \ k \ ! \ m) = \text{length } xs \ \text{mod } k$
by (*simp add: list-slice2-def list-slice-length nth-append mod-div-equality'*)

lemma *list-slice2-nth-nth-eq1*:
 $\llbracket m < \text{length } xs \ \text{div } k; n < k \rrbracket \implies$
 $(\text{list-slice2 } xs \ k) \ ! \ m \ ! \ n = xs \ ! \ (m * k + n)$
by (*simp add: list-slice2-list-slice-nth list-slice-nth-nth*)

lemma *list-slice2-nth-nth-eq2*:
 $\llbracket m = \text{length } xs \ \text{div } k; n < \text{length } xs \ \text{mod } k \rrbracket \implies$
 $(\text{list-slice2 } xs \ k) \ ! \ m \ ! \ n = xs \ ! \ (m * k + n)$
by (*simp add: mult-commute[of - k] mult-div-cancel list-slice2-last*)

lemma *list-slice2-nth-nth-rev*:
 $n < \text{length } xs \implies (\text{list-slice2 } xs \ k) \ ! \ (n \ \text{div } k) \ ! \ (n \ \text{mod } k) = xs \ ! \ n$
apply (*case-tac k = 0*)
apply (*clarsimp simp: list-slice2-0*)
apply (*case-tac n div k < length xs div k*)
apply (*simp add: list-slice2-nth-nth-eq1*)
apply (*frule div-le-mono[OF less-imp-le, of - - k]*)
apply *simp*
apply (*drule sym*)
apply (*subgoal-tac n mod k < length xs mod k*)
prefer 2
apply (*rule ccontr*)
apply (*simp add: linorder-not-less*)
thm *less-mod-ge-imp-div-less*[*of n length xs k*]
apply (*drule less-mod-ge-imp-div-less*[*of n length xs k*], *simp+*)
apply (*simp add: list-slice2-nth-nth-eq2*)
done

lemma *list-slice2-append-mult*:
 $\text{length } xs = m * k \implies$
 $\text{list-slice2 } (xs \ @ \ ys) \ k = \text{list-slice2 } xs \ k \ @ \ \text{list-slice2 } ys \ k$
apply (*case-tac k = 0*)
apply (*simp add: list-slice2-0*)
apply (*clarsimp simp: list-slice2-def list-slice-append-mult*)
apply (*simp add: add-commute*[*of m * k*] *add-mult-distrib*)
done

lemma *list-slice2-append-mod*:
 $\text{length } xs \ \text{mod } k = 0 \implies$
 $\text{list-slice2 } (xs \ @ \ ys) \ k = \text{list-slice2 } xs \ k \ @ \ \text{list-slice2 } ys \ k$
apply *clarify*
thm *list-slice2-append-mult*
apply (*simp only: mult-commute*[*of k*] *list-slice2-append-mult*)
done

lemma *ilist-slice-nth*:

$(\text{ilist-slice } f \ k) \ m = \text{map } f \ [m * k .. < \text{Suc } m * k]$
by (*simp add: ilist-slice-def*)

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

lemma *ilist-slice-nth-nth*:

$n < k \implies (\text{ilist-slice } f \ k) \ m \ ! \ n = f \ (m * k + n)$
by (*simp add: ilist-slice-def*)

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

$0 < k \implies (\text{ilist-slice } f \ k) \ (n \ \text{div } k) \ ! \ (n \ \text{mod } k) = f \ n$
by (*simp add: ilist-slice-nth-nth*)

lemma *list-slice-concat*:

$\text{concat } (\text{list-slice } xs \ k) = xs \ \downarrow \ (\text{length } xs \ \text{div } k * k)$
(is ?P xs k)
apply (*case-tac k = 0*)
apply (*simp add: list-slice-0*)
apply *simp*
apply (*subgoal-tac $\bigwedge m. \forall xs. \text{length } xs \ \text{div } k = m \implies ?P \ xs \ k, \text{ simp}$*)
apply (*induct-tac m*)
apply (*intro allI impI*)
apply (*simp add: in-set-conv-nth div-eq-0-conv' list-slice-less*)
apply *clarify*
apply (*simp add: add-commute[of k]*)
apply (*subgoal-tac $n * k + k \leq \text{length } xs$*)
prefer 2
thm *le-less-div-conv*
apply (*simp add: le-less-div-conv[symmetric]*)
thm *list-slice-div-eq-Suc*
apply (*simp add: list-slice-div-eq-Suc*)
apply (*drule-tac $x = xs \ \downarrow \ (n * k)$ in spec*)
apply (*simp add: min-eqR*)
apply (*simp add: take-add*)
done

lemma *list-slice-unslice-mult*:

$\text{length } xs = m * k \implies \text{concat } (\text{list-slice } xs \ k) = xs$
apply (*case-tac k = 0*)
apply (*simp add: list-slice-Nil*)
apply (*simp add: list-slice-concat*)
done

lemma *ilist-slice-unslice*: $0 < k \implies \text{ilist-unslice } (\text{ilist-slice } f \ k) = f$
by (*simp add: ilist-unslice-def ilist-slice-nth-length ilist-slice-nth-nth*)

lemma *i-take-ilst-slice-eq-list-slice*:

$0 < k \implies \text{ilst-slice } f \ k \ \Downarrow \ n = \text{list-slice } (f \ \Downarrow \ (n * k)) \ k$
apply (*simp add: list-eq-iff list-slice-length ilst-slice-nth list-slice-nth*)
apply (*clarify, rename-tac i*)
apply (*subgoal-tac $k \leq n * k - i * k$*)
prefer 2
apply (*drule-tac $m=i$ in Suc-leI*)
apply (*drule mult-le-mono1[of - - k]*)
apply *simp*
apply *simp*
done

lemma *list-slice-i-take-eq-i-take-ilst-slice*:

$\text{list-slice } (f \ \Downarrow \ n) \ k = \text{ilst-slice } f \ k \ \Downarrow \ (n \ \text{div } k)$
apply (*case-tac $k = 0$*)
apply (*simp add: list-slice-0*)
apply (*simp add: i-take-ilst-slice-eq-list-slice*)
thm *list-slice-eq-list-slice-take[of f \Downarrow n]*
apply (*subst list-slice-eq-list-slice-take[of f \Downarrow n, symmetric]*)
apply (*simp add: div-mult-le min-eqR*)
done

thm *list-slice-append-mod*

lemma *ilst-slice-i-append-mod*:

$\text{length } xs \ \text{mod } k = 0 \implies$
 $\text{ilst-slice } (xs \ \frown \ f) \ k = \text{list-slice } xs \ k \ \frown \ \text{ilst-slice } f \ k$
apply (*simp add: ilst-eq-iff ilst-slice-nth i-append-nth list-slice-length*)
apply (*clarsimp simp: mult-commute[of k], rename-tac n i*)
apply (*intro conjI impI*)
apply (*simp add: list-slice-nth*)
apply (*subgoal-tac $k \leq n * k - i * k$*)
prefer 2
apply (*drule-tac $m=i$ in Suc-leI*)
apply (*drule mult-le-mono1[of - - k]*)
apply *simp*
apply (*fastsimp simp: list-eq-iff i-append-nth min-eqR*)
apply (*simp add: ilst-eq-iff list-eq-iff i-append-nth linorder-not-less*)
apply (*clarify, rename-tac j*)
apply (*subgoal-tac $n * k \leq i * k + j$*)
prefer 2
apply (*simp add: trans-le-add1*)
apply (*simp add: diff-mult-distrib*)
done

corollary *ilst-slice-append-mult*:

$\text{length } xs = m * k \implies$
 $\text{ilst-slice } (xs \ \frown \ f) \ k = \text{list-slice } xs \ k \ \frown \ \text{ilst-slice } f \ k$
by (*simp add: ilst-slice-i-append-mod*)

end

2 AF-Stream: 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
syntax (latex)
  NoMsg :: 'a ( $\varepsilon$ )
  Msg   :: 'a (Msg)
syntax (HTML output)
  NoMsg :: 'a ( $\varepsilon$ )
  Msg   :: 'a (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
typ 'a fstream-af
typ 'a istream-af
```

```
thm not-None-eq
```

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

```
by (case-tac m, simp-all)
```

```
thm not-Some-eq
```

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

```
by (case-tac m, simp-all)
```

```
primrec
```

```
  the-af :: 'a message-af  $\Rightarrow$  'a
```

```
where
```

```
  the-af (Msg x) = x
```

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

```
consts
```

```
  is-NoMsg :: 'a  $\Rightarrow$  bool
```

overloading

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

$$is-NoMsg \varepsilon = True$$

$$| is-NoMsg (Msg x) = False$$
end**overloading**

$$is-NoMsg \equiv is-NoMsg :: ('a \times 'b) \Rightarrow bool$$
begin
definition $is-NoMsg\text{-tuple-def} :$

$$is-NoMsg (p :: 'a \times 'b) \equiv (is-NoMsg (fst p) \wedge is-NoMsg (snd p))$$
end**overloading**

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

$$is-NoMsg (A :: 'a \text{ set}) \equiv (\forall x \in A. is-NoMsg x)$$
end
record $SomeRecordExample =$

$$Field1 :: nat \text{ message-af}$$

$$Field2 :: int \text{ message-af}$$

$$Field3 :: int \text{ message-af}$$
overloading

$$is-NoMsg \equiv is-NoMsg :: SomeRecordExample \Rightarrow bool$$
begin
definition $is-NoMsg\text{-SomeRecordExample-def} :$

$$is-NoMsg (r :: SomeRecordExample) \equiv Field1 r = \varepsilon \wedge Field2 r = \varepsilon \wedge Field3 r = \varepsilon$$
end
definition $is-Msg :: 'a \Rightarrow bool$ **where**

$$is-Msg\text{-def} : is-Msg x \equiv (\neg is-NoMsg x)$$
lemma $is-NoMsg\text{-message-af-conv} : is-NoMsg m = (case m of \varepsilon \Rightarrow True \mid Msg x \Rightarrow False)$
by ($case\text{-tac } m, simp+$)

lemma $is-NoMsg\text{-message-af-conv2} : is-NoMsg m = (m = \varepsilon)$
by ($case\text{-tac } m, simp+$)

lemma $is-Msg\text{-message-af-conv} : is-Msg m = (case m of \varepsilon \Rightarrow False \mid Msg x \Rightarrow$

True)
by (*unfold is-Msg-def, case-tac m, simp+*)
lemma *is-Msg-message-af-conv2*: *is-Msg m = (m ≠ ε)*
by (*unfold is-Msg-def, case-tac m, simp+*)

Collection for definitions for *is_NoMsg*.

ML ⟨⟨
structure is-NoMsg-defs =
Named-Thms(val name = is-NoMsg-defs val description = is-NoMsg-defs rules)
 ⟩⟩
setup *is-NoMsg-defs.setup*

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*]

thm *is-NoMsg-defs*

lemma *not-is-NoMsg*: (\neg *is-NoMsg m*) = *is-Msg m*

by (*simp add: is-NoMsg-defs*)

lemma *not-is-Msg*: (\neg *is-Msg m*) = *is-NoMsg m*

by (*simp add: is-NoMsg-defs*)

lemma *is-NoMsg* ($\varepsilon::(\text{nat message-af})$)

by *simp*

lemma *is-NoMsg* ($\varepsilon::(\text{nat message-af}), \varepsilon::(\text{nat message-af})$)

by (*simp add: is-NoMsg-defs*)

lemma *is-NoMsg* ($\varepsilon::(\text{nat message-af}), \varepsilon::(\text{nat message-af}), \varepsilon::(\text{nat message-af})$)

by (*simp add: is-NoMsg-defs*)

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

by (*simp add: is-NoMsg-defs*)

lemma *is-NoMsg* { $\varepsilon::(\text{nat message-af}), \varepsilon$ }

by (*simp add: is-NoMsg-defs*)

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

by (*simp add: is-NoMsg-defs*)

lemma *is-NoMsg* ($\lfloor \text{Field1} = \varepsilon, \text{Field2} = \varepsilon, \text{Field3} = \varepsilon \rfloor$)

by (*simp add: is-NoMsg-defs*)

lemma *is-Msg* ($\lfloor \text{Field1} = \varepsilon, \text{Field2} = \text{Msg } 1, \text{Field3} = \varepsilon \rfloor$)

by (*simp add: is-NoMsg-defs*)

2.1.2 Time abstraction

primrec

untime :: *'a fstream-af* \Rightarrow *'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 (λx. M_{sg} x) (untime s) = filter (λx. x ≠ ε) s
apply (induct s, simp)
apply (case-tac a, simp-all)
done

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 (λx. the-af x) (filter (λx. x ≠ ε) s)
by (induct s, simp-all)
thm
untime-eq-filter
untime-eq-filter2

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
by (induct s, simp-all add: *untime-length-def*)

definition
untime-length-filter :: 'a fstream-af ⇒ nat
where
untime-length-filter s ≡ length (filter (λx. x ≠ ε) s)

lemma *untime-length-filter-eq-untime-length*:
untime-length-filter s = *untime-length* s
apply (unfold *untime-length-def* *untime-length-filter-def*)
apply (simp add: *untime-eq-filter2*)
done

lemma *untime-empty-conv*: $(\text{untime } s = []) = (\forall n < \text{length } s. s ! n = \varepsilon)$
apply (*induct s*)
apply *simp*
apply (*force simp add: nth.simps split: nat.split*)
done
lemma *untime-not-empty-conv*: $(\text{untime } s \neq []) = (\exists n < \text{length } s. s ! n \neq \varepsilon)$
by (*simp add: untime-empty-conv*)
corollary *untime-empty-imp-NoMsg*[*rule-format*]:
 $\llbracket \text{untime } s = [] ; n < \text{length } s \rrbracket \implies s ! n = \varepsilon$
thm *untime-empty-conv*[*THEN iffD1, rule-format*]
by (*rule untime-empty-conv*[*THEN iffD1, rule-format*])

lemma *untime-nth-eq-filter*:
 $n < \text{untime-length } s \implies$
 $\text{Msg } (\text{untime } s ! n) = (\text{filter } (\lambda x. x \neq \varepsilon) s) ! n$
thm *untime-eq-filter*
by (*simp add: untime-eq-filter[symmetric] untime-length-def*)
corollary *untime-nth-eq-filter2*:
 $n < \text{untime-length } s \implies$
 $\text{untime } s ! n = \text{the-af } ((\text{filter } (\lambda x. x \neq \varepsilon) s) ! n)$
by (*simp add: untime-length-def untime-nth-eq-filter[symmetric]*)

lemma *untime-hd-eq-filter-hd*:
 $\text{untime } s \neq [] \implies$
 $\text{Msg } (\text{hd } (\text{untime } s)) = \text{hd } (\text{filter } (\lambda x. x \neq \varepsilon) s)$
by (*simp add: untime-eq-filter[symmetric] hd-eq-first[symmetric]*)
corollary *untime-hd-eq-filter-hd2*:
 $\text{untime } s \neq [] \implies$
 $\text{hd } (\text{untime } s) = \text{the-af } (\text{hd } (\text{filter } (\lambda x. x \neq \varepsilon) s))$
by (*simp add: untime-hd-eq-filter-hd[symmetric]*)

lemma *untime-last-eq-filter-last*:
 $\text{untime } s \neq [] \implies$
 $\text{Msg } (\text{last } (\text{untime } s)) = \text{last } (\text{filter } (\lambda x. x \neq \varepsilon) s)$
by (*simp add: untime-eq-filter[symmetric] last-nth*)
corollary *untime-last-eq-filter-last2*:
 $\text{untime } s \neq [] \implies$
 $\text{last } (\text{untime } s) = \text{the-af } (\text{last } (\text{filter } (\lambda x. x \neq \varepsilon) s))$
by (*simp add: untime-last-eq-filter-last[symmetric]*)

2.2 Expanding and compressing lists and streams

2.2.1 Expanding message streams

primrec

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

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

definition

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

where

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

primrec

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

where

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

definition

$i\text{-expand-Suc} :: 'a \text{ istream-af} \Rightarrow \text{nat} \Rightarrow 'a \text{ istream-af} \text{ (infixl } \odot_{i\text{Suc}} \text{ 100)}$

where

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

syntax (*xsymbols*)

$-f\text{-expand} :: 'a \text{ fstream-af} \Rightarrow \text{nat} \Rightarrow 'a \text{ fstream-af} \text{ (infixl } \odot \text{ 100)}$

$-i\text{-expand} :: 'a \text{ istream-af} \Rightarrow \text{nat} \Rightarrow 'a \text{ istream-af} \text{ (infixl } \odot \text{ 100)}$

syntax (*HTML output*)

$-f\text{-expand} :: 'a \text{ fstream-af} \Rightarrow \text{nat} \Rightarrow 'a \text{ fstream-af} \text{ (infixl } \odot \text{ 100)}$

$-i\text{-expand} :: 'a \text{ istream-af} \Rightarrow \text{nat} \Rightarrow 'a \text{ istream-af} \text{ (infixl } \odot \text{ 100)}$

translations

$-f\text{-expand } xs k \equiv \text{CONST } f\text{-expand } xs k$

$-i\text{-expand } xs k \equiv \text{CONST } i\text{-expand } xs k$

term $s \odot_f k$

term $s \odot_i k$

lemma $\text{length-}f\text{-expand-Suc}[\text{simp}]$: $\text{length } (f\text{-expand-Suc } xs k) = \text{length } xs * \text{Suc } k$
by (*induct xs, simp+*)

thm $f\text{-expand.simps}$

thm $i\text{-expand-def}$

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

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

by (*simp add: i-expand-def ilist-eq-iff*)

lemma $f\text{-expand-one}$: $0 < k \implies [a] \odot_f k = a \# \varepsilon^k - \text{Suc } 0$
by *simp*

lemma *f-expand-0*[simp]: $xs \odot_f 0 = []$

by (*induct xs, simp+*)

corollary *f-expand-0-is-zero-element*: $xs \odot_f 0 = ys \odot_f 0$

by *simp*

lemma *i-expand-0*[simp]: $f \odot_i 0 = (\lambda n. \varepsilon)$

by (*simp add: i-expand-def*)

corollary *i-expand-0-is-zero-element*: $f \odot_i 0 = g \odot_i 0$

by *simp*

lemma *f-expand-gr0-f-expand-Suc*: $0 < k \implies xs \odot_f k = f\text{-expand-Suc } xs (k - \text{Suc } 0)$

by (*induct xs, simp+*)

lemma *i-expand-gr0-i-expand-Suc*: $0 < k \implies f \odot_i k = i\text{-expand-Suc } f (k - \text{Suc } 0)$

by (*simp add: i-expand-def i-expand-Suc-def ilist-eq-iff*)

lemma *i-expand-gr0*:

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

by (*simp add: i-expand-if*)

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

by (*induct xs, simp+*)

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

by (*simp add: i-expand-gr0*)

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

apply (*case-tac k, simp*)

apply (*simp add: f-expand-gr0-f-expand-Suc*)

done

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

by (*simp add: length-0-conv[symmetric] del: length-0-conv*)

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

by (*simp add: f-expand-empty-conv*)

lemma *f-expand-Cons*:

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

by *simp*

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

apply (*case-tac k = 0, simp*)

apply (*induct xs, simp+*)

done

lemma *f-expand-snoc*:

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

by *simp*

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

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

apply (*induct xs*)

```

apply simp
apply (case-tac n, simp)
apply (simp add: nth-append append-Cons[symmetric] del: append-Cons)
done

thm f-expand-nth-mult
lemma i-expand-nth-mult:  $0 < k \implies (f \odot_i k) (n * k) = f n$ 
by (simp add: i-expand-gr0)
lemma f-expand-nth-if:  $\bigwedge n.$ 
   $n < \text{length } xs * k \implies$ 
   $(xs \odot_f k) ! n = (\text{if } n \bmod k = 0 \text{ then } xs ! (n \text{ div } k) \text{ else } \varepsilon)$ 
apply (case-tac  $k = 0$ , simp)
apply (simp, intro conjI impI)
  thm f-expand-nth-mult
  apply (clarsimp simp: f-expand-nth-mult mult-commute[of k])
  apply (induct xs, simp)
  apply (simp add: nth-append append-Cons[symmetric] del: append-Cons)
  apply (intro conjI impI)
  apply (simp add: nth-Cons')
  apply (case-tac  $\text{length } xs = 0$ , simp)
  apply (simp add: add-commute[of k] diff-less-conv[symmetric] mod-diff-self2)
done
corollary f-expand-nth-mod-eq-0:
   $\llbracket n < \text{length } xs * k; n \bmod k = 0 \rrbracket \implies (xs \odot_f k) ! n = xs ! (n \text{ div } k)$ 
by (simp add: f-expand-nth-if)
corollary f-expand-nth-mod-neq-0:
   $\llbracket n < \text{length } xs * k; 0 < n \bmod k \rrbracket \implies (xs \odot_f k) ! n = \varepsilon$ 
by (simp add: f-expand-nth-if)

thm f-expand-nth-if
lemma f-expand-nth-0-upto-k-minus-1-if:
   $\llbracket t < \text{length } xs; n = t * k + i; i < k \rrbracket \implies$ 
   $(xs \odot_f k) ! n = (\text{if } i = 0 \text{ then } xs ! t \text{ else } \varepsilon)$ 
apply (subst f-expand-nth-if)
  apply (drule Suc-leI[of t])
  apply (drule mult-le-mono1[of - - k])
  apply simp+
done

lemma f-expand-take-mult:  $xs \odot_f k \downarrow (n * k) = (xs \downarrow n) \odot_f k$ 
apply (clarsimp simp add: list-eq-iff min-def)
apply (rename-tac i)
apply (case-tac  $\neg i < n * k$ , simp)
apply (subgoal-tac  $i < \text{length } xs * k$ )
prefer 2
apply (rule-tac  $y = n * k$  in order-le-less-trans, simp+)

```

apply (*clarsimp simp: f-expand-nth-if*)
done

thm *f-expand-take-mult[no-vars]*

lemma *f-expand-take-mod*:

$$n \bmod k = 0 \implies xs \odot_f k \downarrow n = xs \downarrow (n \operatorname{div} k) \odot_f k$$

by (*clarsimp simp: mult-commute[of k] f-expand-take-mult*)

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

apply (*insert arg-cong[OF append-take-drop-id, of $\lambda x. x \odot_f k n xs$]*)

apply (*drule ssubst[OF append-take-drop-id, of - $xs \odot_f k n * k$]*)

apply (*simp only: f-expand-append*)

apply (*simp only: f-expand-take-mult*)

apply *simp*

done

lemma *f-expand-drop-mod*:

$$n \bmod k = 0 \implies xs \odot_f k \uparrow n = xs \uparrow (n \operatorname{div} k) \odot_f k$$

by (*clarsimp simp: mult-commute[of k] f-expand-drop-mult*)

lemma *f-expand-take-mult-Suc*:

$$\llbracket n < \operatorname{length} xs; i < k \rrbracket \implies$$

$$xs \odot_f k \downarrow (n * k + \operatorname{Suc} i) = (xs \downarrow n) \odot_f k @ (xs ! n \# \varepsilon^i)$$

apply (*subgoal-tac $n * k + \operatorname{Suc} i \leq \operatorname{length} xs * k$*)

prefer 2

apply (*drule Suc-leI[of n]*)

apply (*drule mult-le-mono1[of $\operatorname{Suc} n - k$]*)

apply *simp*

apply (*clarsimp simp: list-eq-iff min-eqR nth-append f-expand-nth-if min-def nth-Cons'*)

apply (*simp add: mult-commute[of k] linorder-not-less*)

apply (*drule-tac $n=q$ in le-neq-implies-less, simp+*)

apply (*drule-tac $n=q$ in Suc-leI*)

apply (*drule-tac $j=q$ in mult-le-mono1[of - - k]*)

apply *simp*

done

lemma *f-expand-take-Suc*:

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

$$xs \odot_f k \downarrow \operatorname{Suc} n = (xs \downarrow (n \operatorname{div} k)) \odot_f k @ (xs ! (n \operatorname{div} k) \# \varepsilon^{n \bmod k})$$

apply (*case-tac $k = 0$, simp*)

thm *f-expand-take-mult-Suc[of $n \operatorname{div} k xs n \bmod k k$]*

apply (*insert f-expand-take-mult-Suc[of $n \operatorname{div} k xs n \bmod k k$]*)

apply (*simp add: div-less-conv*)

done

lemma *i-expand-nth-if*:

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

by (*simp add: i-expand-gr0*)

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

$\llbracket 0 < k; n \bmod k = 0 \rrbracket \implies (f \odot_i k) n = f (n \operatorname{div} k)$

by (simp add: i-expand-gr0)

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

$0 < n \bmod k \implies (f \odot_i k) n = \varepsilon$

apply (case-tac k = 0, simp)

apply (simp add: i-expand-gr0)

done

thm *i-expand-nth-if*

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

$\llbracket n = t * k + i; i < k \rrbracket \implies$

$(f \odot_i k) n = (\text{if } i = 0 \text{ then } f t \text{ else } \varepsilon)$

by (simp add: i-expand-nth-if)

thm *f-expand-take-mult*

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

apply (case-tac k = 0, simp)

apply (clarsimp simp: list-eq-iff i-expand-nth-if f-expand-nth-if)

done

thm *f-expand-take-mod*

lemma *i-expand-i-take-mod*:

$n \bmod k = 0 \implies f \odot_i k \Downarrow n = f \Downarrow (n \operatorname{div} k) \odot_f k$

by (clarsimp simp: mult-commute[of k] i-expand-i-take-mult)

thm *f-expand-drop-mult*

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

apply (case-tac k = 0, simp)

apply (clarsimp simp: ilist-eq-iff i-expand-nth-if)

done

lemma *i-expand-i-drop-mod*:

$n \bmod k = 0 \implies f \odot_i k \Uparrow n = f \Uparrow (n \operatorname{div} k) \odot_i k$

by (clarsimp simp: mult-commute[of k] i-expand-i-drop-mult)

lemma *i-expand-i-take-mult-Suc*:

$i < k \implies f \odot_i k \Downarrow (n * k + \operatorname{Suc} i) = (f \Downarrow n) \odot_f k @ (f n \# \varepsilon^i)$

apply (clarsimp simp: list-eq-iff, rename-tac i')

apply (clarsimp simp: i-expand-nth-if f-expand-nth-if nth-append nth-Cons')

apply (simp add: linorder-not-less mult-commute[of k])

apply (drule-tac n=q in le-neq-implies-less, simp+)

apply (drule-tac n=q in Suc-leI)

apply (drule-tac j=q in mult-le-mono1[of - - k])

apply simp

done

lemma *i-expand-i-take-Suc*:

$0 < k \implies f \odot_i k \Downarrow \operatorname{Suc} n = (f \Downarrow (n \operatorname{div} k)) \odot_f k @ (f (n \operatorname{div} k) \# \varepsilon^{n \operatorname{mod} k})$

thm *i-expand-i-take-mult-Suc*

thm *i-expand-i-take-mult-Suc*[of n mod k k n div k f]

apply (*insert i-expand-i-take-mult-Suc*[of $n \bmod k \ k \ n \ \text{div} \ k \ f$])
apply *simp*
done

lemma *f-expand-nth-interval-eq-nth-append-replicate-NoMsg*[*rule-format*]:
 $\llbracket 0 < k; t < \text{length } xs; t * k \leq t1; t1 \leq t * k + k - \text{Suc } 0 \rrbracket \implies$
 $xs \odot_f k \downarrow \text{Suc } t1 \uparrow (t * k) = xs ! t \# \varepsilon^{t1 - t * k}$
apply (*rule-tac* $t = \text{Suc } t1$ **and** $s = t * k + \text{Suc } (t1 - t * k)$ **in** *subst, simp*)
thm *f-expand-take-mult-Suc*
apply (*subst f-expand-take-mult-Suc*)
apply *simp+*
done

lemma *f-expand-nth-interval-eq-replicate-NoMsg*:
 $\llbracket 0 < k; t * k < t1; t1 \leq t2; t2 \leq t * k + k; t2 \leq \text{length } xs * k \rrbracket \implies$
 $xs \odot_f k \downarrow t2 \uparrow t1 = \varepsilon^{t2 - t1}$
apply (*clarsimp simp: list-eq-iff min-eqR f-expand-nth-if, rename-tac i q*)
apply (*drule-tac* $i = i$ **and** $k = t1$ **in** *add-less-mono2, simp*)
apply (*drule-tac* $i = t * k$ **and** $j = t1$ **and** $m = i$ **in** *trans-less-add1*)
thm *less-mod-eq-imp-add-divisor-le*
apply (*drule-tac* $x = t * k$ **and** $y = t1 + i$ **and** $m = k$ **in** *less-mod-eq-imp-add-divisor-le, simp*)
apply *simp*
done

lemma *i-expand-nth-interval-eq-nth-append-replicate-NoMsg*[*rule-format*]:
 $\llbracket 0 < k; t * k \leq t1; t1 \leq t * k + k - \text{Suc } 0 \rrbracket \implies$
 $f \odot_i k \downarrow \text{Suc } t1 \uparrow (t * k) = f t \# \varepsilon^{t1 - t * k}$
by (*simp add: list-eq-iff Suc-diff-le i-expand-nth-if nth-Cons'*)

lemma *i-expand-nth-interval-eq-replicate-NoMsg*:
 $\llbracket 0 < k; t * k < t1; t1 \leq t2; t2 \leq t * k + k \rrbracket \implies$
 $f \odot_i k \downarrow t2 \uparrow t1 = \varepsilon^{t2 - t1}$
apply (*clarsimp simp: list-eq-iff i-expand-nth-if add-commute*[of k])
apply (*drule-tac* $i = i$ **and** $k = t1$ **in** *add-less-mono2, simp*)
apply (*drule-tac* $i = t * k$ **and** $j = t1$ **and** $m = i$ **in** *trans-less-add1*)
thm *less-mod-eq-imp-add-divisor-le*
apply (*drule-tac* $x = t * k$ **and** $y = t1 + i$ **and** $m = k$ **in** *less-mod-eq-imp-add-divisor-le, simp*)
apply *simp*
done

lemma *f-expand-replicate-NoMsg*[*simp*]: $(\varepsilon^n) \odot_f k = \varepsilon^{n * k}$
by (*clarsimp simp: list-eq-iff f-expand-nth-if*)

lemma *i-expand-const-NoMsg*[simp]: $(\lambda n. \varepsilon) \odot_i k = (\lambda n. \varepsilon)$
by (*simp add: i-expand-def ilist-eq-iff*)

lemma *f-expand-assoc*: $xs \odot_f a \odot_f b = xs \odot_f (a * b)$
apply (*induct xs*)
apply *simp*
apply (*simp add: replicate-add[symmetric] diff-mult-distrib*)
done

lemma *i-expand-assoc*: $f \odot_i a \odot_i b = f \odot_i (a * b)$
by (*fastsimp simp: i-expand-def ilist-eq-iff*)

lemma *f-expand-commute*: $xs \odot_f a \odot_f b = xs \odot_f b \odot_f a$
by (*simp add: f-expand-assoc mult-commute[of b]*)
lemma *i-expand-commute*: $f \odot_i a \odot_i b = f \odot_i b \odot_i a$
by (*simp add: i-expand-assoc mult-commute[of b]*)

thm *f-expand-append*

lemma *i-expand-i-append*: $(xs \frown f) \odot_i k = xs \odot_f k \frown (f \odot_i k)$
apply (*case-tac k = 0, simp*)
apply (*clarsimp simp add: ilist-eq-iff i-expand-gr0 i-append-nth*)
apply (*case-tac x < length xs * k*)
apply (*frule-tac n=x and k=length xs in div-less-conv[THEN iffD2, of k, rule-format], simp*)
apply (*simp add: f-expand-nth-if*)
apply (*simp add: linorder-not-less*)
apply (*frule div-le-mono[of - - k]*)
apply (*simp add: mod-diff-mult-self1 div-diff-mult-self1*)
done

lemma *f-expand-eq-conv*:

$0 < k \implies (xs \odot_f k = ys \odot_f k) = (xs = ys)$
apply (*rule iffI*)
apply (*clarsimp simp: list-eq-iff, rename-tac i*)
apply (*drule-tac x=i * k in spec*)
apply (*simp add: f-expand-nth-mult*)
apply *simp*
done

lemma *i-expand-eq-conv*:

$0 < k \implies (f \odot_i k = g \odot_i k) = (f = g)$
apply (*rule iffI*)
apply (*clarsimp simp: ilist-eq-iff, rename-tac i*)
apply (*drule-tac x=i * k in spec*)
apply (*simp add: i-expand-nth-mult*)
apply *simp*
done

lemma *f-expand-eq-conv'*:

$$(xs' \odot_f k = xs) = \\ (\text{length } xs' * k = \text{length } xs \wedge \\ (\forall i < \text{length } xs. xs ! i = (\text{if } i \bmod k = 0 \text{ then } xs' ! (i \text{ div } k) \text{ else } \varepsilon)))$$

by (*fastsimp simp: list-eq-iff f-expand-nth-if*)

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

$$0 < k \implies (f' \odot_i k = f) = \\ (\forall i. f i = (\text{if } i \bmod k = 0 \text{ then } f' (i \text{ div } k) \text{ else } \varepsilon))$$

by (*fastsimp simp: ilist-eq-iff i-expand-nth-if*)

2.2.2 Aggregating lists

term *list-slice*

definition

$$f\text{-aggregate} :: 'a \text{ list} \Rightarrow \text{nat} \Rightarrow ('a \text{ list} \Rightarrow 'a) \Rightarrow 'a \text{ list}$$

where

$$f\text{-aggregate } s \ k \ ag \equiv \text{map } ag \ (\text{list-slice } s \ k)$$

definition

$$i\text{-aggregate} :: 'a \text{ ilist} \Rightarrow \text{nat} \Rightarrow ('a \text{ list} \Rightarrow 'a) \Rightarrow 'a \text{ ilist}$$

where

$$i\text{-aggregate } s \ k \ ag \equiv \lambda n. ag \ (s \uparrow (n * k) \downarrow k)$$

lemma *f-aggregate-0[simp]*: $f\text{-aggregate } xs \ 0 \ ag = []$

by (*simp add: f-aggregate-def list-slice-0*)

lemma *f-aggregate-1*:

$$(\bigwedge x. ag [x] = x) \implies \\ f\text{-aggregate } xs \ (\text{Suc } 0) \ ag = xs$$

by (*simp add: list-eq-iff f-aggregate-def list-slice-1*)

lemma *f-aggregate-Nil[simp]*: $f\text{-aggregate } [] \ k \ ag = []$

by (*simp add: f-aggregate-def list-slice-Nil*)

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

by (*simp add: f-aggregate-def list-slice-length*)

lemma *f-aggregate-empty-conv*:

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

by (*simp add: length-0-conv[symmetric] div-eq-0-conv' del: length-0-conv*)

lemma *f-aggregate-one*:

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

by (*simp add: f-aggregate-def list-slice-def*)

lemma *f-aggregate-Cons*:

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

by (*simp add: f-aggregate-def list-slice-def*)

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

f-aggregate (*xs* ↓ (*length xs* div *k* * *k*)) *k ag* = *f-aggregate xs k ag*
by (*simp add: f-aggregate-def list-slice-eq-list-slice-take*)

lemma *f-aggregate-nth*:

n < *length xs* div *k* \implies
(*f-aggregate xs k ag*) ! *n* = *ag* (*xs* ↑ (*n* * *k*) ↓ *k*)
by (*simp add: f-aggregate-def list-slice-length list-slice-nth*)

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

n < *length xs* div *k* \implies
(*f-aggregate xs k ag*) ! *n* = *ag* (*sublist-list xs* [*n* * *k*..*n* * *k* + *k*])
apply (*frule less-div-imp-mult-add-divisor-le*)
apply (*simp add: f-aggregate-nth take-drop-eq-sublist-list*)
done

lemma *f-aggregate-take-nth*:

$\bigwedge xs m. \llbracket n < \text{length } xs \text{ div } k; n < m \text{ div } k \rrbracket \implies$
f-aggregate (*xs* ↓ *m*) *k ag* ! *n* = *f-aggregate xs k ag* ! *n*
apply (*simp add: f-aggregate-nth drop-take*)
thm *less-div-imp-mult-add-divisor-le*
apply (*drule-tac n=m in less-div-imp-mult-add-divisor-le*)
apply (*simp add: min-eqL*)
done

lemma *f-aggregate-hd*:

$\llbracket 0 < k; k \leq \text{length } xs \rrbracket \implies$
hd (*f-aggregate xs k ag*) = *ag* (*xs* ↓ *k*)
apply (*drule div-le-mono[of - - k]*)
apply (*simp add: Suc-le-eq*)
apply (*subst hd-eq-first[symmetric]*)
apply (*simp add: length-greater-0-conv[symmetric]*)
apply (*simp add: f-aggregate-nth*)
done

thm *list-slice-append-mod*

lemma *f-aggregate-append-mod*:

length xs mod *k* = 0 \implies
f-aggregate (*xs* @ *ys*) *k ag* =
f-aggregate xs k ag @ *f-aggregate ys k ag*
by (*simp add: f-aggregate-def list-slice-append-mod*)

lemma *f-aggregate-append-mult*:

length xs = *m* * *k* \implies
f-aggregate (*xs* @ *ys*) *k ag* =
f-aggregate xs k ag @ *f-aggregate ys k ag*
by (*simp add: f-aggregate-append-mod*)

lemma *f-aggregate-snoc*:

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

$f\text{-aggregate } (xs \text{ @ } ys) \ k \ ag = f\text{-aggregate } xs \ k \ ag \text{ @ } [ag \ ys]$
by (*simp add: f-aggregate-append-mod f-aggregate-one*)

lemma *f-aggregate-take*:

$f\text{-aggregate } (xs \downarrow n) \ k \ ag = f\text{-aggregate } xs \ k \ ag \downarrow (n \text{ div } k)$
apply (*case-tac k = 0, simp*)
apply (*simp add: list-eq-iff*)
apply (*case-tac length xs ≤ n*)
apply (*simp add: min-eqL div-le-mono f-aggregate-nth*)
apply (*clarsimp simp: linorder-not-le min-eqR div-le-mono f-aggregate-nth drop-take*)
apply (*frule less-div-imp-mult-add-divisor-le*)
apply (*simp add: min-eqL*)
apply (*subgoal-tac i < length xs div k*)
prefer 2
apply (*drule-tac y=n in order-le-less-trans, assumption*)
thm *div-le-mono[OF less-imp-le]*
apply (*drule-tac m=i * k + k and k=k in div-le-mono[OF less-imp-le]*)
apply *simp*
apply (*simp add: f-aggregate-nth*)
done

lemma *f-aggregate-take-mult*:

$f\text{-aggregate } (xs \downarrow (n * k)) \ k \ ag = f\text{-aggregate } xs \ k \ ag \downarrow n$
by (*simp add: f-aggregate-take*)

lemma *f-aggregate-drop-mult*:

$f\text{-aggregate } (xs \uparrow (n * k)) \ k \ ag = f\text{-aggregate } xs \ k \ ag \uparrow n$
by (*simp add: list-eq-iff div-diff-mult-self1 f-aggregate-nth add-mult-distrib add-commute[of n * k]*)

lemma *f-aggregate-drop-mod*:

$n \text{ mod } k = 0 \implies f\text{-aggregate } (xs \uparrow n) \ k \ ag = f\text{-aggregate } xs \ k \ ag \uparrow (n \text{ div } k)$
by (*clarsimp simp: mult-commute[of k] f-aggregate-drop-mult*)

lemma *f-aggregate-assoc*:

$(\bigwedge xs. \text{length } xs \text{ mod } a = 0 \implies ag (f\text{-aggregate } xs \ a \ ag) = ag \ xs) \implies$
 $f\text{-aggregate } (f\text{-aggregate } xs \ a \ ag) \ b \ ag = f\text{-aggregate } xs \ (a * b) \ ag$
apply (*clarsimp simp add: list-eq-iff div-mult2-eq f-aggregate-nth, rename-tac i*)
apply (*simp add: take-drop f-aggregate-take-mult[symmetric]*)
apply (*simp add: add-mult-distrib2 mult-commute[of - a] f-aggregate-drop-mult[symmetric] mult-assoc[symmetric]*)
apply (*drule-tac x=(xs ↓ (a * b + a * i * b) ↑ (a * i * b)) in meta-spec*)
apply (*subgoal-tac a * b + a * i * b ≤ length xs*)
prefer 2
apply (*simp add: div-mult2-eq[symmetric]*)
apply (*drule-tac x=i in less-div-imp-mult-add-divisor-le*)
apply (*simp add: mult-assoc[symmetric] mult-commute[of - a] add-commute[of - a * b]*)
apply (*simp add: min-eqR*)
done

lemma *f-aggregate-commute*:

$\llbracket \bigwedge xs. \text{length } xs \bmod a = 0 \implies ag (f\text{-aggregate } xs \ a \ ag) = ag \ xs;$
 $\bigwedge xs. \text{length } xs \bmod b = 0 \implies ag (f\text{-aggregate } xs \ b \ ag) = ag \ xs \rrbracket \implies$
 $f\text{-aggregate } (f\text{-aggregate } xs \ a \ ag) \ b \ ag = f\text{-aggregate } (f\text{-aggregate } xs \ b \ ag) \ a \ ag$
by (*simp add: f-aggregate-assoc mult-commute[of - b]*)

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

by (*simp add: i-aggregate-def*)

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

by (*simp add: i-aggregate-def i-take-first*)

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

by (*simp add: i-aggregate-def*)

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

by (*simp add: i-aggregate-nth*)

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

by (*simp add: i-aggregate-nth i-take-drop-eq-map*)

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

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

$i\text{-aggregate } (xs \ \frown \ f) \ k \ ag = f\text{-aggregate } xs \ k \ ag \ \frown \ i\text{-aggregate } f \ k \ ag$

apply (*clarsimp simp: ilist-eq-iff i-aggregate-nth i-append-nth f-aggregate-nth mult-commute[of k] diff-mult-distrib, rename-tac i n*)

apply (*drule-tac n=i in Suc-leI*)

apply (*drule mult-le-mono1[of - - k]*)

apply *simp*

done

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

$\text{length } xs = m * k \implies$

$i\text{-aggregate } (xs \ \frown \ f) \ k \ ag = f\text{-aggregate } xs \ k \ ag \ \frown \ i\text{-aggregate } f \ k \ ag$

by (*rule i-aggregate-i-append-mod, simp*)

thm *f-aggregate-Cons*

lemma *i-aggregate-Cons*:

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

$i\text{-aggregate } (xs \ \frown \ f) \ k \ ag = [ag \ xs] \ \frown \ (i\text{-aggregate } f \ k \ ag)$

apply (*insert i-aggregate-i-append-mod[of xs k f ag], simp*)

apply (*simp add: f-aggregate-def list-slice-div-eq-1*)

done

lemma *i-aggregate-take-nth*:

$n < m \ \text{div } k \implies f\text{-aggregate } (f \ \downarrow \ m) \ k \ ag \ ! \ n = i\text{-aggregate } f \ k \ ag \ n$

apply (*simp add: f-aggregate-nth i-aggregate-nth*)

apply (*drule less-div-imp-mult-add-divisor-le*)

apply (*simp add: i-take-drop-map i-take-drop-eq-map take-map*)

done

lemma *i-aggregate-i-take*:

f-aggregate ($f \Downarrow n$) k $ag = i-aggregate$ f k $ag \Downarrow (n \text{ div } k)$

by (*simp add: list-eq-iff i-aggregate-take-nth*)

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

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

by (*simp add: i-aggregate-i-take*)

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

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

by (*simp add: ilist-eq-iff i-aggregate-nth add-mult-distrib*)

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

$n \text{ mod } k = 0 \implies$

i-aggregate ($f \Uparrow n$) k $ag = i-aggregate$ f k $ag \Uparrow (n \text{ div } k)$

by (*clarsimp simp: mult-commute[of k] i-aggregate-i-drop-mult ilist-eq-iff*)

lemma *i-aggregate-assoc*:

$\llbracket 0 < a; 0 < b;$

$\bigwedge xs. \text{length } xs \text{ mod } a = 0 \implies ag (f-aggregate \text{ } xs \ a \ ag) = ag \ xs \rrbracket \implies$

i-aggregate (*i-aggregate* f a ag) b $ag = i-aggregate$ f ($a * b$) ag

apply (*simp add: ilist-eq-iff i-aggregate-nth*)

apply (*simp add: i-aggregate-i-drop-mult[symmetric] i-aggregate-i-take-mult[symmetric] mult-commute[of a] mult-assoc*)

done

lemma *i-aggregate-commute*:

$\llbracket 0 < a; 0 < b;$

$\bigwedge xs. \text{length } xs \text{ mod } a = 0 \implies ag (f-aggregate \text{ } xs \ a \ ag) = ag \ xs;$

$\bigwedge xs. \text{length } xs \text{ mod } b = 0 \implies ag (f-aggregate \text{ } xs \ b \ ag) = ag \ xs \rrbracket \implies$

i-aggregate (*i-aggregate* xs a ag) b $ag = i-aggregate$ (*i-aggregate* xs b ag) a ag

by (*simp add: i-aggregate-assoc mult-commute[of - b]*)

2.2.3 Compressing message streams

Determines the last non-empty message.

primrec

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

where

last-message [] = ε

| *last-message* ($x \# xs$) = (if *last-message* $xs = \varepsilon$ then x else *last-message* xs)

definition

f-shrink :: 'a *fstream-af* \Rightarrow nat \Rightarrow 'a *fstream-af* (**infixl** \div_f 100)

where

f-shrink xs $k \equiv f-aggregate$ xs k *last-message*

definition

i-shrink :: 'a *istream-af* \Rightarrow nat \Rightarrow 'a *istream-af* (**infixl** \div_i 100)

where

$i\text{-shrink } f k \equiv i\text{-aggregate } f k \text{ last-message}$

syntax (*xsymbols*)

$-f\text{-shrink} :: 'a \text{ fstream-af} \Rightarrow \text{nat} \Rightarrow 'a \text{ fstream-af}$ (**infixl** \div 100)

$-i\text{-shrink} :: 'a \text{ istream-af} \Rightarrow \text{nat} \Rightarrow 'a \text{ istream-af}$ (**infixl** \div 100)

translations

$-f\text{-shrink } xs n \equiv \text{CONST } f\text{-shrink } xs n$

$-i\text{-shrink } f n \equiv \text{CONST } i\text{-shrink } f n$

syntax (*HTML output*)

$-f\text{-shrink} :: 'a \text{ fstream-af} \Rightarrow \text{nat} \Rightarrow 'a \text{ fstream-af}$ (**infixl** \div 100)

$-i\text{-shrink} :: 'a \text{ istream-af} \Rightarrow \text{nat} \Rightarrow 'a \text{ istream-af}$ (**infixl** \div 100)

term $s \div_f n$

term $s \div_i n$

lemmas $f\text{-shrink-defs} = f\text{-shrink-def } f\text{-aggregate-def}$

lemmas $i\text{-shrink-defs} = i\text{-shrink-def } i\text{-aggregate-def}$

lemma last-message-Nil : $\text{last-message } [] = \varepsilon$

by *simp*

lemma last-message-one : $\text{last-message } [m] = m$

by *simp*

lemma $\text{last-message-replicate}$: $0 < n \Longrightarrow \text{last-message } (m^n) = m$

apply (*induct n, simp*)

apply (*case-tac n, simp+*)

done

lemma $\text{last-message-replicate-NoMsg}$: $\text{last-message } (\varepsilon^n) = \varepsilon$

apply (*case-tac n = 0, simp*)

apply (*simp add: last-message-replicate*)

done

lemma $\text{last-message-Cons-NoMsg}$: $\text{last-message } (\varepsilon \# xs) = \text{last-message } xs$

by *simp*

lemma $\text{last-message-append-one}$:

$\text{last-message } (xs @ [m]) = (\text{if } m = \varepsilon \text{ then last-message } xs \text{ else } m)$

apply (*induct xs, simp*)

apply (*case-tac m = ε , simp+*)

done

lemma $\text{last-message-append}$: $\bigwedge xs.$

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

$\text{if last-message } ys = \varepsilon \text{ then last-message } xs \text{ else last-message } ys)$

apply (*induct ys, simp*)

apply (*drule-tac x=xs @ [a] in meta-spec*)

apply (*simp add: last-message-append-one*)

done

corollary $\text{last-message-append-replicate-NoMsg}$:

$last\text{-}message (xs @ \varepsilon^n) = last\text{-}message xs$
by (*simp add: last-message-append last-message-replicate-NoMsg*)

lemma *last-message-replicate-NoMsg-append*:
 $last\text{-}message (\varepsilon^n @ xs) = last\text{-}message xs$
by (*simp add: last-message-append last-message-replicate-NoMsg*)

lemma *last-message-NoMsg-conv*:
 $(last\text{-}message xs = \varepsilon) = (\forall i < length\ xs. xs ! i = \varepsilon)$
apply (*induct xs, simp*)
apply (*simp add: nth-Cons'*)
apply (*safe, simp-all*)
apply (*rename-tac i*)
apply (*drule-tac x=Suc i in spec*)
apply *simp*
done

lemma *last-message-not-NoMsg-conv*:
 $(last\text{-}message xs \neq \varepsilon) = (\exists i < length\ xs. xs ! i \neq \varepsilon)$
by (*simp add: last-message-NoMsg-conv*)

lemma *not-NoMsg-imp-last-message*:
 $\llbracket i < length\ xs; xs ! i \neq \varepsilon \rrbracket \implies last\text{-}message xs \neq \varepsilon$
by (*rule last-message-not-NoMsg-conv[THEN iffD2, OF exI, OF conjI]*)

lemma *last-message-exists-nth*:
 $last\text{-}message xs \neq \varepsilon \implies$
 $\exists i < length\ xs. last\text{-}message xs = xs ! i \wedge (\forall j < length\ xs. i < j \longrightarrow xs ! j = \varepsilon)$
apply (*induct xs, simp*)
apply (*rename-tac a xs*)
thm *last-message-NoMsg-conv*
apply (*case-tac last-message xs = \varepsilon*)
apply (*rule-tac x=0 in exI*)
apply *clarsimp*
apply (*rename-tac j, case-tac j, simp*)
apply (*simp add: last-message-NoMsg-conv*)
apply (*rule ccontr*)
apply (*clarsimp, rename-tac i*)
apply (*drule-tac x=Suc i in spec*)
apply (*clarsimp simp: nth-Cons'*)
done

lemma *last-message-exists-nth'*:
 $last\text{-}message xs \neq \varepsilon \implies \exists i < length\ xs. last\text{-}message xs = xs ! i$
by (*blast dest: last-message-exists-nth*)

lemma *last-messageI2-aux*: $\bigwedge i.$
 $\llbracket i < length\ xs; xs ! i \neq \varepsilon;$
 $\forall j. i < j \wedge j < length\ xs \longrightarrow xs ! j = \varepsilon \rrbracket \implies$
 $last\text{-}message xs = xs ! i$
apply (*induct xs, simp*)

```

apply (simp add: nth-Cons')
apply (case-tac i)
  apply simp
  apply (subgoal-tac  $\forall j. j < \text{length } xs \longrightarrow xs ! j = \varepsilon$ )
    prefer 2
    apply (clarify, drule-tac  $x = \text{Suc } j$  in spec)
    apply simp
  apply (simp add: last-message-NoMsg-conv)
apply clarsimp
apply (rename-tac i)
apply (intro conjI impI)
  apply (simp add: not-NoMsg-imp-last-message)
apply (subgoal-tac  $\forall j. i < j \wedge j < \text{length } xs \longrightarrow xs ! j = \varepsilon$ )
  prefer 2
  apply (clarify, drule-tac  $x = \text{Suc } j$  in spec)
  apply simp
apply simp
done
lemma last-messageI2:
   $\llbracket i < \text{length } xs; xs ! i \neq \varepsilon; \bigwedge j. \llbracket i < j; j < \text{length } xs \rrbracket \implies xs ! j = \varepsilon \rrbracket \implies$ 
  last-message xs = xs ! i
by (blast intro: last-messageI2-aux)
lemma last-messageI:
   $\llbracket m \neq \varepsilon; i < \text{length } xs; xs ! i = m; \bigwedge j. \llbracket i < j; j < \text{length } xs \rrbracket \implies xs ! j = \varepsilon \rrbracket \implies$ 
  last-message xs = m
by (blast intro: last-messageI2)

lemma last-message-Msg-eq-last:
   $\llbracket xs \neq []; \text{last } xs \neq \varepsilon \rrbracket \implies \text{last-message } xs = \text{last } xs$ 
by (simp add: last-nth last-messageI2)

lemma last-message-conv:
   $m \neq \varepsilon \implies$ 
  (last-message xs = m) =
  ( $\exists i < \text{length } xs. xs ! i = m \wedge (\forall j < \text{length } xs. i < j \longrightarrow xs ! j = \varepsilon)$ )
apply (rule iffI)
thm last-message-exists-nth
apply (cut-tac  $xs = xs$  in last-message-exists-nth, simp)
apply clarify
apply (rule-tac  $x = i$  in exI)
apply simp
apply (clarsimp simp: last-messageI)
done

lemma last-message-conv-if:
  (last-message xs = m) =
  (if  $m = \varepsilon$  then  $\forall i < \text{length } xs. xs ! i = \varepsilon$ 

```

else $\exists i < \text{length } xs. xs ! i = m \wedge (\forall j < \text{length } xs. i < j \longrightarrow xs ! j = \varepsilon)$
by (*simp add: last-message-NoMsg-conv last-message-conv*)

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

$\llbracket \text{last-message } xs \neq \varepsilon; \text{last-message } ys \neq \varepsilon \rrbracket \Longrightarrow$
 $(\text{last-message } xs = \text{last-message } ys) =$
 $(\exists i j. i < \text{length } xs \wedge j < \text{length } ys \wedge xs ! i \neq \varepsilon \wedge$
 $ys ! j = \varepsilon \wedge$
 $(\forall n < \text{length } xs. i < n \longrightarrow xs ! n = \varepsilon) \wedge$
 $(\forall n < \text{length } ys. j < n \longrightarrow ys ! n = \varepsilon))$
apply (*simp add: last-message-conv[where m=last-message ys]*)
apply (*frule last-message-exists-nth[of xs]*)
apply (*frule last-message-exists-nth[of ys]*)
apply (*rule iffI*)
apply (*clarsimp, rename-tac i j*)
apply (*rule-tac x=j in exI, simp*)
apply (*rule-tac x=i in exI, simp*)
apply (*clarsimp, rename-tac i1 j1 i j*)
apply (*rule-tac x=i in exI, simp*)
apply (*subgoal-tac last-message ys = ys ! j, simp*)
thm *last-messageI2*
apply (*rule last-messageI2*)
apply *simp+*
done

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

by (*simp add: f-shrink-defs list-slice-0*)

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

by (*simp add: f-shrink-def f-aggregate-1*)

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

by (*simp add: f-shrink-def list-slice-Nil*)

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

by (*simp add: f-shrink-def*)

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

by (*simp add: f-shrink-def f-aggregate-empty-conv*)

lemma *f-shrink-Cons*:

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

by (*simp add: f-shrink-def f-aggregate-Cons*)

lemma *f-shrink-one*:

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

by (*simp add: f-shrink-def f-aggregate-one*)

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

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

by (*simp add: f-shrink-defs list-slice-eq-list-slice-take*)

lemma *f-shrink-nth*:

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

by (*simp add: f-shrink-def f-aggregate-nth*)

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

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

by (*simp add: f-shrink-def f-aggregate-nth-eq-sublist-list*)

lemma *f-shrink-take-nth*:

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

by (*simp add: f-shrink-def f-aggregate-take-nth*)

lemma *f-shrink-hd*:

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

by (*simp add: f-shrink-def f-aggregate-hd*)

thm *list-slice-append-mod*

lemma *f-shrink-append-mod*:

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

by (*simp add: f-shrink-defs list-slice-append-mod*)

lemma *f-shrink-append-mult*:

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

by (*simp add: f-shrink-append-mod*)

lemma *f-shrink-snoc*:

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

by (*simp add: f-shrink-append-mod f-shrink-one*)

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

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

apply (*case-tac k = 0, simp*)

thm *append-constant-length-induct*

apply (*rule append-constant-length-induct[of k]*)

apply *simp*

apply (*simp add: f-shrink-Cons last-message-append*)

done

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

apply (*case-tac k = 0, simp*)

apply (*case-tac n < k*)

apply (*simp add: f-shrink-empty-conv*)

apply (*clarsimp simp: list-eq-iff f-shrink-length f-shrink-nth*)

thm *last-message-replicate*

apply (*rule last-message-replicate*)

apply (*clarsimp simp: min-def*)

apply (*drule mult-less-mono1[of - n div k k], simp*)

apply (*simp add: div-mult-cancel*)
done

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

apply (*simp add: list-eq-iff*)

apply (*simp add: f-shrink-length f-shrink-nth f-expand-drop-mult f-expand-take-mod drop-take-1 last-message- replicate-NoMsg*)

done

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

$\llbracket \forall i < \text{length } xs. 0 < i \bmod k \longrightarrow xs ! i = \varepsilon \rrbracket \implies$

$xs \div_f k \odot_f k = xs \downarrow (\text{length } xs \text{ div } k * k)$

apply (*case-tac k = 0, simp*)

thm *append-constant-length-induct*[*of k, rule-format*]

apply (*induct xs rule: append-constant-length-induct*[*of k*])

apply (*simp add: f-shrink-empty-conv*[*symmetric*])

apply (*drule meta-mp*)

apply *clarify*

apply (*drule-tac x=length xs + i in spec, simp*)

apply (*simp add: f-shrink-append-mod*)

apply (*rule-tac t=xs and s=(xs ! 0) # replicate (k - Suc 0) ε in subst*)

apply (*simp (no-asm-simp) add: list-eq-iff nth-Cons'*)

apply (*clarify, rename-tac i*)

apply (*drule-tac x=i in spec*)

apply (*simp add: nth-append*)

apply (*simp (no-asm-simp) add: list-eq-iff*)

apply (*clarsimp simp: f-shrink-length f-expand-nth-if f-shrink-nth last-message- replicate-NoMsg nth-Cons'*)

done

thm *f-expand-f-shrink-id-take*[*of xs k*]

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

$\llbracket \text{length } xs \bmod k = 0;$

$\bigwedge i. \llbracket i < \text{length } xs; 0 < i \bmod k \rrbracket \implies xs ! i = \varepsilon \rrbracket \implies$

$xs \div_f k \odot_f k = xs$

by (*clarsimp simp: f-expand-f-shrink-id-take*)

lemma *f-shrink-take*:

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

by (*simp add: f-shrink-def f-aggregate-take*)

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

by (*simp add: f-shrink-def f-aggregate-take-mult*)

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

by (*simp add: f-shrink-def f-aggregate-drop-mult*)

lemma *f-shrink-drop-mod*:

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

by (*simp add: f-shrink-def f-aggregate-drop-mod*)

thm *last-message-conv-if*
thm *f-expand-eq-conv*
lemma *f-shrink-eq-conv*:
 $(xs \dot{\div}_f k1 = ys \dot{\div}_f k2) =$
 $(\text{length } xs \text{ div } k1 = \text{length } ys \text{ div } k2 \wedge$
 $(\forall i < \text{length } xs \text{ div } k1.$
 $\text{last-message } (xs \uparrow (i * k1) \downarrow k1) = \text{last-message } (ys \uparrow (i * k2) \downarrow k2)))$
apply (*case-tac* $k1 = 0$)
apply (*simp* *add*: *eq-commute*[*of* []] *length-0-conv*[*symmetric*] *f-shrink-length* *del*:
length-0-conv)
apply (*case-tac* $k2 = 0$)
apply (*fastsimp* *simp*: *f-shrink-empty-conv* *div-eq-0-conv*)
apply (*simp* *add*: *list-eq-iff* *f-shrink-length*)
apply (*rule* *conj-cong*, *simp*)
apply (*rule* *all-imp-eqI*, *simp*)
apply (*simp* *add*: *f-shrink-nth*)
done

thm *f-expand-eq-conv'*
lemma *f-shrink-eq-conv'*:
 $(xs' \dot{\div}_f k = xs) =$
 $(\text{length } xs' \text{ div } k = \text{length } xs \wedge$
 $(\forall i < \text{length } xs.$
 $\text{if } xs ! i = \varepsilon \text{ then } (\forall j < k. xs' ! (i * k + j) = \varepsilon)$
 $\text{else } (\exists n < k. xs' ! (i * k + n) = xs ! i \wedge$
 $(\forall j < k. n < j \longrightarrow xs' ! (i * k + j) = \varepsilon))))$
apply (*case-tac* $k = 0$, *fastsimp*)
apply (*simp* *add*: *list-eq-iff* *f-shrink-length* *split* *del*: *split-if*)
apply (*rule* *conj-cong*, *simp*)
apply (*rule* *all-imp-eqI*, *simp*)
apply (*cut-tac* $x=i$ **in** *less-div-imp-mult-add-divisor-le*[*of* - *length* $xs' k$], *simp*)
apply (*clarsimp* *simp*: *f-shrink-nth* *last-message-conv-if* *min-eqR*)
apply (*rule* *ex-imp-eqI*, *simp*)
apply *simp*
done

lemma *f-shrink-assoc*: $xs \dot{\div}_f a \dot{\div}_f b = xs \dot{\div}_f (a * b)$
by (*unfold* *f-shrink-def*, *rule* *f-aggregate-assoc*, *fold* *f-shrink-def*, *rule* *f-shrink-last-message*)

lemma *f-shrink-commute*: $xs \dot{\div}_f a \dot{\div}_f b = xs \dot{\div}_f b \dot{\div}_f a$
by (*simp* *add*: *f-shrink-assoc* *mult-commute*[*of* a])

lemma *i-shrink-0*[*simp*]: $f \dot{\div}_i 0 = (\lambda n. \varepsilon)$
by (*simp* *add*: *i-shrink-defs*)
lemma *i-shrink-1*[*simp*]: $f \dot{\div}_i \text{Suc } 0 = f$
by (*simp* *add*: *i-shrink-def* *i-aggregate-1*)
lemma *i-shrink-nth*: $(f \dot{\div}_i k) n = \text{last-message } (f \uparrow (n * k) \downarrow k)$
by (*simp* *add*: *i-shrink-defs*)

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

by (*simp add: i-shrink-def i-aggregate-nth-eq-map*)

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

by (*simp add: i-shrink-nth*)

lemma *i-shrink-i-append-mod*:

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

by (*simp add: f-shrink-def i-shrink-def i-aggregate-i-append-mod*)

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

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

by (*simp add: i-shrink-i-append-mod*)

lemma *i-shrink-Cons*:

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

by (*simp add: f-shrink-def i-shrink-def i-aggregate-Cons*)

lemma *i-shrink-take-nth*:

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

by (*simp add: f-shrink-def i-shrink-def i-aggregate-take-nth*)

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

by (*simp add: ilist-eq-iff i-shrink-nth last-message-replicate*)

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

by (*case-tac k = 0, simp+*)

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

by (*simp add: ilist-eq-iff i-shrink-nth i-expand-i-drop-mult i-expand-i-take-mod i-drop-i-take-1 last-message-replicate-NoMsg*)

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

by (*simp add: f-shrink-def i-shrink-def i-aggregate-i-take-mult*)

lemma *i-shrink-i-take*:

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

by (*simp add: f-shrink-def i-shrink-def i-aggregate-i-take*)

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

by (*simp add: f-shrink-def i-shrink-def i-aggregate-i-drop-mult*)

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

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

by (*simp add: f-shrink-def i-shrink-def i-aggregate-i-drop-mod*)

thm *f-shrink-eq-conv*

lemma *i-shrink-eq-conv*:

$(f \div_i k1 = g \div_i k2) =$

$(\forall i. \text{last-message } (f \Uparrow (i * k1) \Downarrow k1) =$

$\text{last-message } (g \Uparrow (i * k2) \Downarrow k2))$

by (*simp add: ilist-eq-iff i-shrink-nth*)

thm *f-shrink-eq-conv'*

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

$$(f' \dot{\div}_i k = f) = \\ (\forall i. \text{if } f \ i = \varepsilon \text{ then } \forall j < k. f' (i * k + j) = \varepsilon \\ \text{else } \exists n < k. f' (i * k + n) = f \ i \wedge \\ (\forall j < k. n < j \longrightarrow f' (i * k + j) = \varepsilon))$$

apply (*simp add: ilist-eq-iff*)

apply (*case-tac k = 0, fastsimp*)

apply (*rule all-eqI, rename-tac i*)

apply (*simp add: i-shrink-nth*)

apply (*case-tac f i = NoMsg*)

apply (*simp add: last-message-NoMsg-conv*)

apply (*force simp add: last-message-conv*)

done

thm *f-shrink-assoc*

lemma *i-shrink-assoc*: $f \dot{\div}_i a \dot{\div}_i b = f \dot{\div}_i (a * b)$

apply (*case-tac a = 0, simp*)

apply (*case-tac b = 0, simp*)

apply (*unfold i-shrink-def, rule i-aggregate-assoc, simp+*)

apply (*fold f-shrink-def, simp add: f-shrink-last-message*)

done

lemma *i-shrink-commute*: $f \dot{\div}_i a \dot{\div}_i b = f \dot{\div}_i b \dot{\div}_i a$

by (*simp add: i-shrink-assoc mult-commute[of a]*)

2.2.4 Holding last messages in every cycle of a stream

primrec

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

where

last-message-hold-init [] m = []

| *last-message-hold-init* (x # xs) m =

(if x = ε then m else x) #

(*last-message-hold-init* xs (if x = ε then m else x))

definition

last-message-hold :: 'a fstream-af \Rightarrow 'a fstream-af

where

last-message-hold xs \equiv *last-message-hold-init* xs ε

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

$\bigwedge m. \text{length} (\text{last-message-hold-init } xs \ m) = \text{length } xs$

by (*induct xs, simp+*)

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

$\bigwedge i \ m. i < \text{length } xs \Longrightarrow$

(*last-message-hold-init* xs m) ! i = *last-message* (m # xs \downarrow Suc i)

apply (*induct xs, simp*)

apply (*simp add: nth-Cons'*)
done

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*]

by (*simp add: list-eq-iff nth-append last-message-hold-init-nth last-message-append*)

lemma *last-message-hold-init-append[rule-format]*:

$\bigwedge xs m. \text{last-message-hold-init } (xs @ ys) m =$

last-message-hold-init xs m @ last-message-hold-init ys (last-message (m # xs))

apply (*induct ys, simp*)

apply (*rule-tac x=a in subst[OF append-eq-Cons, rule-format]*)

apply (*simp only: append-Cons[symmetric] append-assoc[symmetric]*)

apply (*simp add: last-message-hold-init-snoc last-message-append*)

done

lemma *last-message-hold-length[simp]*: *length (last-message-hold xs) = length xs*

by (*simp add: last-message-hold-def*)

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

by (*simp add: last-message-hold-def*)

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

by (*simp add: last-message-hold-def*)

lemma *last-message-hold-nth*:

i < length xs \implies last-message-hold xs ! i = last-message (xs ↓ Suc i)

by (*simp add: last-message-hold-def last-message-hold-init-nth*)

lemma *last-message-hold-last*:

xs ≠ [] \implies last (last-message-hold xs) = last-message xs

apply (*subgoal-tac last-message-hold xs ≠ []*)

prefer 2

apply (*simp add: length-greater-0-conv[symmetric] del: length-greater-0-conv*)

apply (*simp add: last-nth last-message-hold-nth length-greater-0-conv[symmetric]*)

del: length-greater-0-conv)

done

lemma *last-message-hold-take*:

last-message-hold xs ↓ n = last-message-hold (xs ↓ n)

apply (*case-tac length xs ≤ n, simp*)

apply (*simp add: list-eq-iff last-message-hold-nth min-eqL*)

done

lemma *last-message-hold-snoc*:

last-message-hold (xs @ [x]) =

last-message-hold xs @ [if x = ε then last-message xs else x]

by (*simp add: last-message-hold-def last-message-hold-init-snoc*)

lemma *last-message-hold-append*:

```

last-message-hold (xs @ ys) =
  last-message-hold xs @ last-message-hold-init ys (last-message xs)
by (simp add: last-message-hold-def last-message-hold-init-append)
lemma last-message-hold-append':
  last-message-hold (xs @ ys) =
    last-message-hold xs @ tl (last-message-hold (last-message xs # ys))
apply (simp add: last-message-hold-append)
apply (simp add: last-message-hold-def)
done

```

```

lemma last-message-last-message-hold[simp]:
  last-message (last-message-hold xs) = last-message xs
apply (induct xs rule: rev-induct, simp)
apply (simp add: last-message-hold-snoc last-message-append)
done

```

```

lemma last-message-hold-idem[simp]:
  last-message-hold (last-message-hold xs) = last-message-hold xs
by (simp add: list-eq-iff last-message-hold-nth last-message-hold-take)

```

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

definition

```

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

```

```

f-last-message-hold xs k  $\equiv$  concat (map last-message-hold (list-slice2 xs k))

```

definition

```

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

```

where

```

i-last-message-hold f k  $\equiv$   $\lambda n.$  last-message (f  $\uparrow$  (n - n mod k)  $\downarrow$  Suc (n mod k))

```

syntax (xsymbols)

```

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

```

```

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

```

translations

```

-f-last-message-hold xs n  $\equiv$  CONST f-last-message-hold xs n

```

```

-i-last-message-hold f n  $\equiv$  CONST i-last-message-hold f n

```

syntax (HTML output)

```

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

```

```

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

```

term $xs \mapsto_f k$

term $f \mapsto_i k$

```

lemma f-last-message-hold-0[simp]:  $xs \mapsto_f 0 =$  last-message-hold xs

```

```

by (simp add: f-last-message-hold-def list-slice2-0)

```

```

lemma f-last-message-hold-1[simp]:  $xs \mapsto_f (Suc 0) = xs$ 

```

```

apply (simp add: f-last-message-hold-def list-slice2-1)

```

```

apply (induct xs, simp+)

```

done

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

by (simp add: *f-last-message-hold-def list-slice2-Nil*)

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

apply (case-tac $k = 0$, simp)

apply (simp add: *f-last-message-hold-def*)

thm *append-constant-length-induct*[of k]

apply (induct xs rule: *append-constant-length-induct*[of k])

apply (simp add: *list-slice2-le*)

apply (simp add: *list-slice2-append-mod list-slice2-mod-0 list-slice-div-eq-1*)

done

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

by (simp add: *f-last-message-hold-def list-slice2-le*)

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

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

by (simp add: *f-last-message-hold-def list-slice2-append-mod*)

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

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

by (simp add: *f-last-message-hold-def list-slice2-append-mod*)

thm *f-shrink-nth*

lemma *f-last-message-hold-nth*[rule-format]:

$\forall n. n < \text{length } xs \implies xs \mapsto_f k ! n = \text{last-message } (xs \uparrow (n \text{ div } k * k) \downarrow \text{Suc } (n \bmod k))$

apply (case-tac $k = 0$)

apply (simp add: *last-message-hold-nth*)

apply (induct xs rule: *append-constant-length-induct*[of k])

apply (simp add: *f-last-message-hold-def list-slice2-le last-message-hold-nth*)

apply (simp add: *f-last-message-hold-append-mod nth-append*)

apply (intro allI conjI impI)

apply (simp add: *f-last-message-hold-def list-slice2-mod-0 list-slice-div-eq-1 last-message-hold-nth*)

apply (case-tac $n < k$, simp)

apply (simp add: *linorder-not-less last-message-append div-mult-cancel*)

apply (subgoal-tac $k + n \bmod k \leq n$)

prefer 2

apply (drule *div-le-mono*[of - - k], drule *mult-le-mono1*[of - - k])

apply (simp add: *div-mult-cancel*)

apply (simp add: *mod-diff-self2 add-commute*[of k])

done

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

by (clarsimp simp: *list-eq-iff f-last-message-hold-nth drop-take div-mult-cancel min-eqL*)

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

$xs \uparrow (n * k) \mapsto_f k = xs \mapsto_f k \uparrow (n * k)$

apply (rule subst[OF *append-take-drop-id*, of - $n * k$ xs])

apply (*case-tac* $\text{length } xs \leq n * k$, *simp*)
apply (*simp* *add*: *f-last-message-hold-append-mod min-eqR del*: *append-take-drop-id*)
done

lemma *f-last-message-hold-drop-mod*:
 $n \bmod k = 0 \implies xs \uparrow n \mapsto_f k = xs \mapsto_f k \uparrow n$
by (*clarsimp simp*: *mult-commute*[*of k*] *f-last-message-hold-drop-mult*)

lemma *f-last-message-hold-idem*: $xs \mapsto_f k \mapsto_f k = xs \mapsto_f k$
apply (*case-tac* $k = 0$, *simp*)
apply (*simp* *add*: *list-eq-iff-f-last-message-hold-nth f-last-message-hold-drop-mod*[*symmetric*]
f-last-message-hold-take[*symmetric*])
apply (*simp* *add*: *f-last-message-hold-le min-max.le-infl2 Suc-mod-le-divisor*)
done

thm *f-shrink-nth*

lemma *f-shrink-nth-eq-f-last-message-hold-last*:
 $n < \text{length } xs \text{ div } k \implies xs \div_f k ! n = \text{last } (xs \mapsto_f k \uparrow (n * k) \downarrow k)$
apply (*case-tac* $k = 0$, *simp*)
apply (*case-tac* $xs = []$, *simp*)
apply (*simp* *add*: *f-shrink-nth f-last-message-hold-drop-mult*[*symmetric*] *f-last-message-hold-take*[*symmetric*])
apply (*drule* *less-div-imp-mult-add-divisor-le*)
apply (*simp* *add*: *f-last-message-hold-le last-message-hold-last*)
done

lemma *f-shrink-nth-eq-f-last-message-hold-nth*:
 $n < \text{length } xs \text{ div } k \implies xs \div_f k ! n = xs \mapsto_f k ! (n * k + k - \text{Suc } 0)$
apply (*case-tac* $k = 0$, *simp*)
apply (*simp* *add*: *f-shrink-nth-eq-f-last-message-hold-last*)
apply (*frule* *less-div-imp-mult-add-divisor-le*)
apply (*simp* *add*: *last-nth min-eqR*)
done

lemma *last-message-f-last-message-hold*:
 $\text{last-message } (xs \mapsto_f k) = \text{last-message } xs$
apply (*case-tac* $k = 0$, *simp*)
apply (*induct* xs *rule*: *append-constant-length-induct*[*of k*])
apply (*simp* *add*: *f-last-message-hold-le*)
apply (*simp* *add*: *f-last-message-hold-append-mult last-message-append f-last-message-hold-le*)
done

lemma *i-last-message-hold-0*[*simp*]: $f \mapsto_i 0 = (\lambda n. \text{last-message } (f \downarrow \text{Suc } n))$
by (*simp* *add*: *i-last-message-hold-def*)
lemma *i-last-message-hold-1*[*simp*]: $f \mapsto_i \text{Suc } 0 = f$
by (*simp* *add*: *i-last-message-hold-def i-drop-i-take-1*)

lemma *i-last-message-hold-nth*:
 $(f \mapsto_i k) n = \text{last-message } (f \uparrow (n - n \bmod k) \downarrow \text{Suc } (n \bmod k))$
by (*simp* *add*: *i-last-message-hold-def*)

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

$length\ xs = m * k \implies (xs \frown f) \mapsto_i k = (xs \mapsto_f k) \frown (f \mapsto_i k)$
apply (*case-tac* $k = 0$, *simp*)
apply (*clarsimp simp: ilist-eq-iff i-last-message-hold-nth i-append-nth f-last-message-hold-nth div-mult-cancel linorder-not-less*)
apply (*subgoal-tac* $length\ xs \leq x - x \bmod k$)
prefer 2
apply (*drule div-le-mono*[*of - - k*])
apply (*simp add: div-mult-cancel*[*symmetric*])
apply (*simp add: mod-diff-mult-self1*)
apply (*drule-tac* $j=x - x \bmod k$ **and** $k=x \bmod k$ **in** *add-le-mono1*)
apply (*simp add: add-commute*[*of m * k*])
done

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

$length\ xs \bmod k = 0 \implies (xs \frown f) \mapsto_i k = (xs \mapsto_f k) \frown (f \mapsto_i k)$
by (*clarsimp simp: mult-commute*[*of k*], *rule i-last-message-hold-i-append-mult*)

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

by (*simp add: list-eq-iff f-last-message-hold-nth i-last-message-hold-nth div-mult-cancel i-take-drop min-eqR*)

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

$f \Uparrow (n * k) \mapsto_i k = f \mapsto_i k \Uparrow (n * k)$
by (*simp add: ilist-eq-iff i-last-message-hold-nth*)

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

$n \bmod k = 0 \implies f \Uparrow n \mapsto_i k = f \mapsto_i k \Uparrow n$
by (*clarsimp simp: mult-commute*[*of k*], *rule i-last-message-hold-i-drop-mult*)

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

by (*simp add: ilist-eq-iff i-last-message-hold-nth mult-div-cancel*[*symmetric*] *i-last-message-hold-i-drop-mod*[*symmetric*] *i-last-message-hold-i-take*[*symmetric*] *last-message-f-last-message-hold*)

thm *f-shrink-nth-eq-f-last-message-hold-nth*

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

$0 < k \implies (f \div_i k) n = (f \mapsto_i k) (n * k + k - Suc\ 0)$
apply (*simp add: i-shrink-nth i-last-message-hold-nth*)
apply (*simp add: diff-add-assoc del: add-diff-assoc*)
done

thm *f-shrink-nth-eq-f-last-message-hold-last*

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

$0 < k \implies (f \div_i k) n = last\ (f \mapsto_i k \Uparrow (n * k) \Downarrow k)$
by (*simp add: last-nth i-shrink-nth-eq-i-last-message-hold-nth*)

2.2.5 Compressing lists

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

definition

$f\text{-shrink-last} :: 'a \text{ list} \Rightarrow \text{nat} \Rightarrow 'a \text{ list}$ (**infixl** \div_f 100)

where

$f\text{-shrink-last } xs \ k \equiv f\text{-aggregate } xs \ k \ \text{last}$

definition

$i\text{-shrink-last} :: 'a \ \text{ilist} \Rightarrow \text{nat} \Rightarrow 'a \ \text{ilist}$ (**infixl** \div_i 100)

where

$i\text{-shrink-last } f \ k \equiv i\text{-aggregate } f \ k \ \text{last}$

syntax (*xsymbols*)

$-f\text{-shrink-last} :: 'a \ \text{list} \Rightarrow \text{nat} \Rightarrow 'a \ \text{list}$ (**infixl** \div_l 100)

$-i\text{-shrink-last} :: 'a \ \text{ilist} \Rightarrow \text{nat} \Rightarrow 'a \ \text{ilist}$ (**infixl** \div_l 100)

translations

$-f\text{-shrink-last } xs \ n \equiv \text{CONST } f\text{-shrink-last } xs \ n$

$-i\text{-shrink-last } xs \ n \equiv \text{CONST } i\text{-shrink-last } xs \ n$

syntax (*HTML output*)

$-f\text{-shrink-last} :: 'a \ \text{list} \Rightarrow \text{nat} \Rightarrow 'a \ \text{list}$ (**infixl** \div_l 100)

$-i\text{-shrink-last} :: 'a \ \text{ilist} \Rightarrow \text{nat} \Rightarrow 'a \ \text{ilist}$ (**infixl** \div_l 100)

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

by (simp add: $f\text{-shrink-last-def } f\text{-aggregate-def } \text{list-slice-0}$)

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

by (simp add: $f\text{-shrink-last-def } f\text{-aggregate-1}$)

lemma $f\text{-shrink-last-Nil}$ [simp]: $[] \div_f k = []$

by (simp add: $f\text{-shrink-last-def } f\text{-aggregate-def } \text{list-slice-Nil}$)

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

by (simp add: $f\text{-shrink-last-def}$)

lemma $f\text{-shrink-last-empty-conv}$:

$0 < k \implies (xs \div_f k = []) = (\text{length } xs < k)$

by (simp add: $f\text{-shrink-last-def } f\text{-aggregate-empty-conv}$)

lemma $f\text{-shrink-last-Cons}$:

$[0 < k;$

$\text{length } xs = k] \implies (xs @ ys) \div_f k = \text{last } xs \ \# \ (ys \div_f k)$

by (simp add: $f\text{-shrink-last-def } f\text{-aggregate-Cons}$)

lemma $f\text{-shrink-last-one}$:

$[0 < k; \text{length } xs = k] \implies xs \div_f k = [\text{last } xs]$

by (simp add: f-shrink-last-def f-aggregate-one)

lemma f-shrink-last-eq-f-shrink-last-take:

$$xs \downarrow (\text{length } xs \text{ div } k * k) \div_{\#} k = xs \div_{\#} k$$

by (simp add: f-shrink-last-def f-aggregate-eq-f-aggregate-take)

lemma f-shrink-last-nth:

$$n < \text{length } xs \text{ div } k \implies (xs \div_{\#} k) ! n = xs ! (n * k + k - \text{Suc } 0)$$

apply (case-tac k = 0, simp)

apply (frule less-div-imp-mult-add-divisor-le)

apply (simp add: f-shrink-last-def f-aggregate-nth last-take2)

done

corollary f-shrink-last-nth':

$$n < \text{length } xs \text{ div } k \implies (xs \div_{\#} k) ! n = xs ! (\text{Suc } n * k - \text{Suc } 0)$$

by (simp add: f-shrink-last-nth add-commute[of k])

lemma f-shrink-last-hd:

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

by (simp add: f-shrink-last-def f-aggregate-hd last-take2)

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

apply (case-tac k = 0, simp)

apply (clarsimp simp: list-eq-iff f-shrink-last-length)

apply (frule less-div-imp-mult-add-divisor-le)

apply (simp add: f-shrink-last-nth)

done

lemma f-shrink-last-append-mod:

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

by (simp add: f-shrink-last-def f-aggregate-append-mod)

lemma f-shrink-last-append-mult:

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

by (unfold f-shrink-last-def, rule f-aggregate-append-mult)

lemma f-shrink-last-snoc:

$$\llbracket 0 < k; \text{length } ys = k; \text{length } xs \bmod k = 0 \rrbracket \implies \\ (xs @ ys) \div_{\#} k = xs \div_{\#} k @ [\text{last } ys]$$

by (simp add: f-shrink-last-append-mod f-shrink-last-one)

lemma f-shrink-last-last:

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

apply (case-tac k = 0, simp)

apply (case-tac xs = [], simp)

apply (subgoal-tac k \leq length xs)

prefer 2

```

apply (rule ccontr, simp)
thm append-take-drop-id[of length xs - k xs]
apply (rule subst[OF append-take-drop-id[of length xs - k xs]])
apply (subst f-shrink-last-snoc)
apply (simp add: min-eqR mod-diff-self2)+
done

```

```

lemma f-shrink-last-replicate:  $m^n \div_{\mathbb{F}} k = m^n \text{ div } k$ 
apply (case-tac k = 0, simp)
apply (clar simp simp: list-eq-iff f-shrink-last-length)
apply (frule less-div-imp-mult-add-divisor-le)
apply (simp add: f-shrink-last-nth)
done

```

```

lemma f-shrink-last-take:
   $xs \downarrow n \div_{\mathbb{F}} k = xs \div_{\mathbb{F}} k \downarrow (n \text{ div } k)$ 
by (unfold f-shrink-last-def, rule f-aggregate-take)

```

```

lemma f-shrink-last-take-mult:  $xs \downarrow (n * k) \div_{\mathbb{F}} k = xs \div_{\mathbb{F}} k \downarrow n$ 
by (unfold f-shrink-last-def, rule f-aggregate-take-mult)

```

```

lemma f-shrink-last-drop-mult:  $xs \uparrow (n * k) \div_{\mathbb{F}} k = xs \div_{\mathbb{F}} k \uparrow n$ 
by (unfold f-shrink-last-def, rule f-aggregate-drop-mult)

```

```

lemma f-shrink-last-drop-mod:
   $n \text{ mod } k = 0 \implies xs \uparrow n \div_{\mathbb{F}} k = xs \div_{\mathbb{F}} k \uparrow (n \text{ div } k)$ 
by (unfold f-shrink-last-def, rule f-aggregate-drop-mod)

```

```

lemma f-shrink-last-assoc:  $xs \div_{\mathbb{F}} a \div_{\mathbb{F}} b = xs \div_{\mathbb{F}} (a * b)$ 
by (unfold f-shrink-last-def, rule f-aggregate-assoc, fold f-shrink-last-def, rule f-shrink-last-last)

```

```

lemma f-shrink-last-commute:  $xs \div_{\mathbb{F}} a \div_{\mathbb{F}} b = xs \div_{\mathbb{F}} b \div_{\mathbb{F}} a$ 
by (simp add: f-shrink-last-assoc mult-commute[of a])

```

```

lemma i-shrink-last-1[simp]:  $f \div_{\mathbb{I}} \text{Suc } 0 = f$ 
by (simp add: i-shrink-last-def i-aggregate-1)

```

```

thm f-shrink-last-nth

```

```

lemma i-shrink-last-nth:  $0 < k \implies (f \div_{\mathbb{I}} k) n = f (n * k + k - \text{Suc } 0)$ 
by (simp add: i-shrink-last-def i-aggregate-nth last-i-take2)

```

```

thm f-shrink-last-nth'

```

```

lemma i-shrink-last-nth':  $0 < k \implies (f \div_{\mathbb{I}} k) n = f (\text{Suc } n * k - \text{Suc } 0)$ 
by (simp add: i-shrink-last-nth add-commute[of k])

```

```

lemma i-shrink-last-hd:  $(f \div_{\mathbb{I}} k) 0 = \text{last } (f \downarrow k)$ 
apply (case-tac k = 0)
apply (simp add: i-shrink-last-def)
apply (simp add: i-shrink-last-nth last-i-take2)
done

```

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

by (*simp add: ilist-eq-iff i-shrink-last-nth*)

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

$length\ xs\ mod\ k = 0 \implies (xs \frown f) \div_{il} k = xs \div_{fl} k \frown (f \div_{il} k)$

by (*simp add: f-shrink-last-def i-shrink-last-def i-aggregate-i-append-mod*)

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

$length\ xs = m * k \implies (xs \frown f) \div_{il} k = xs \div_{fl} k \frown (f \div_{il} k)$

by (*simp add: i-shrink-last-i-append-mod*)

lemma *i-shrink-last-Cons*:

$\llbracket 0 < k; length\ xs = k \rrbracket \implies (xs \frown f) \div_{il} k = [last\ xs] \frown (f \div_{il} k)$

by (*simp add: f-shrink-last-def i-shrink-last-def i-aggregate-Cons*)

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

by (*simp add: ilist-eq-iff i-shrink-last-nth*)

lemma *i-shrink-last-i-take-mult*:

$0 < k \implies f \Downarrow (n * k) \div_{fl} k = f \div_{il} k \Downarrow n$

by (*simp add: f-shrink-last-def i-shrink-last-def i-aggregate-i-take-mult*)

lemma *i-shrink-last-i-take*:

$f \Downarrow n \div_{fl} k = f \div_{il} k \Downarrow (n\ div\ k)$

by (*simp add: f-shrink-last-def i-shrink-last-def i-aggregate-i-take*)

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

by (*simp add: f-shrink-last-def i-shrink-last-def i-aggregate-i-drop-mult*)

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

$n\ mod\ k = 0 \implies f \Uparrow n \div_{il} k = f \div_{il} k \Uparrow (n\ div\ k)$

by (*simp add: f-shrink-last-def i-shrink-last-def i-aggregate-i-drop-mod*)

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

apply (*unfold i-shrink-last-def*)

apply (*case-tac b = 0, simp*)

apply (*case-tac a = 0, simp add: i-aggregate-def*)

apply (*simp add: i-aggregate-assoc f-shrink-last-last[unfolded f-shrink-last-def]*)

done

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

by (*simp add: i-shrink-last-assoc mult-commute[of a]*)

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

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

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

by (*simp add: list-eq-iff f-shrink-length f-shrink-last-length f-shrink-nth-eq-f-last-message-hold-nth*)

f-shrink-last-nth)

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

$$0 < k \implies f \dot{\div}_i k = f \mapsto_i k \dot{\div}_i k$$

by (*simp add: ilist-eq-iff i-shrink-last-nth i-shrink-nth-eq-i-last-message-hold-nth*)

end

3 AF-Stream-Exec: Processing of message streams

theory *AF-Stream-Exec*

imports *AF-Stream ListInf-Prefix 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

$$('a, 'in) \text{ Port-Input-Value} = 'a \Rightarrow 'in \text{ message-af}$$

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

type-synonym

$$('comp, 'out) \text{ Port-Output-Value} = 'comp \Rightarrow 'out \text{ message-af}$$

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

type-synonym

$$('comp, 'state) \text{ Comp-Local-State} = 'comp \Rightarrow 'state$$

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

type-synonym

$$('comp, 'input) \text{ Comp-Trans-Fun} = 'input \Rightarrow 'comp \Rightarrow 'comp$$

primrec *f-Exec-Comp* ::

$$('comp, 'input) \text{ Comp-Trans-Fun} \Rightarrow 'input \text{ list} \Rightarrow 'comp \Rightarrow 'comp$$

where

$$f\text{-Exec-Nil}: f\text{-Exec-Comp} \text{ trans-fun } [] \ c = \ c$$

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

definition

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

where

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

primrec

f-Exec-Comp-Stream ::

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

where

i-Exec-Comp-Stream \equiv λ trans-fun input c n.

f-Exec-Comp trans-fun (input \downarrow Suc n) c

definition

i-Exec-Comp-Stream-Init :: ('comp, 'input) *Comp-Trans-Fun* \Rightarrow 'input ilist \Rightarrow 'comp \Rightarrow 'comp ilist

where

i-Exec-Comp-Stream-Init \equiv λ trans-fun input c n.

f-Exec-Comp trans-fun (input \downarrow n) c

3.1.2 Basic results

lemma *f-Exec-one*: *f-Exec-Comp trans-fun [m] c = trans-fun m c*
by *simp*

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

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

by (*induct xs, simp-all*)

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

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

by (*simp add: length-0-conv[symmetric] del: length-0-conv*)

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

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

by (*simp add: f-Exec-Stream-empty-conv*)

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

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

by (*induct xs, simp-all*)

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

$xs \neq [] \implies$

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

by (*simp add: f-Exec-eq-f-Exec-Stream-last f-Exec-Stream-empty-conv*[*symmetric, of xs trans-fun c*])

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

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

by (*simp add: f-Exec-eq-f-Exec-Stream-last2*)

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

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

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

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

by (*simp add: f-Exec-eq-f-Exec-Stream-last2 take-not-empty-conv*)

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

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

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

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

by (*simp add: f-Exec-Comp-N-def f-Exec-take-eq-last-f-Exec-Stream-take*)

lemma *f-Exec-Stream-nth*:

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

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

apply (*induct xs, simp*)

apply (*simp add: nth-Cons'*)

done

lemma *f-Exec-Stream-nth2*:

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

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

by (*simp add: nth-Cons' f-Exec-Stream-nth*)

lemma *f-Exec-N-all*:

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

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

by (*simp add: f-Exec-Comp-N-def*)

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

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

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

(*f-Exec-Comp-Stream trans-fun ys (f-Exec-Comp trans-fun xs c)*)
by (*induct xs, simp-all*)

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))))
by (*simp add: f-Exec-Stream-append f-Exec-eq-f-Exec-Stream-last*)

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)))
by (*simp add: f-Exec-Stream-append-last-Cons f-Exec-Stream-empty-conv*)

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)))
by (*simp add: f-Exec-Stream-append f-Exec-eq-f-Exec-Stream-last-if*)

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)
by (*simp add: f-Exec-eq-f-Exec-Stream-last f-Exec-Stream-append-if f-Exec-Stream-empty-conv*)

thm *f-Exec-Stream-Cons*

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

thm *f-Exec-Stream-Cons*

by (*subst f-Exec-Stream-Cons[symmetric], simp*)

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)]
by (*simp add: f-Exec-Stream-append*)

lemma *f-Exec-snoc*:

f-Exec-Comp trans-fun (xs @ [x]) c =
trans-fun x (f-Exec-Comp trans-fun xs c)
by (*simp add: f-Exec-append*)

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)
apply (*simp add: f-Exec-Comp-N-def f-Exec-append[symmetric]*)
apply (*simp add: take-drop add-commute[of b]*)
apply (*rule subst[of xs ↓ (a + b) ↓ a xs ↓ a], simp add: min-eqL*)
apply (*subst append-take-drop-id, simp*)
done

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)
thm *f-Exec-N-append*[of *trans-fun n Suc 0 xs c*]
by (*simp add: f-Exec-N-append[symmetric]*)

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)
by (*simp add: f-Exec-Comp-N-def take-Suc-conv-app-nth f-Exec-append*)

theorem *f-Exec-Stream-take*:

(f-Exec-Comp-Stream trans-fun xs c) ↓ n =
f-Exec-Comp-Stream trans-fun (xs ↓ n) c
apply (*case-tac length xs ≤ n, simp*)
apply (*rule subst[OF append-take-drop-id, of - n xs]*)
apply (*simp add: f-Exec-Stream-append del: append-take-drop-id*)
done

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)
apply (*case-tac length xs ≤ n, simp*)
apply (*rule subst[OF append-take-drop-id, of - n xs]*)
apply (*simp add: f-Exec-Stream-append del: append-take-drop-id*)
done

lemma *i-Exec-Stream-nth*:

i-Exec-Comp-Stream trans-fun input c n = f-Exec-Comp trans-fun (input ↓ Suc n) c
by (*simp add: i-Exec-Comp-Stream-def*)

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)
by (*simp add: i-Exec-Stream-nth i-take-Suc-conv-app-nth f-Exec-append*)

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

$i\text{-Exec-Comp-Stream trans-fun input } c \text{ (Suc } n) =$
 $(i\text{-Exec-Comp-Stream trans-fun (input } \uparrow \text{ Suc } 0) \text{ (trans-fun (input } 0) \text{ } c) \text{ } n)$
by (simp add: $i\text{-Exec-Stream-nth } i\text{-take-Suc}$)

thm $i\text{-Exec-Stream-nth}$

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

$n < n' \implies$

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

$i\text{-Exec-Comp-Stream trans-fun input } c \text{ } n$

by (simp add: $f\text{-Exec-Stream-nth } i\text{-Exec-Stream-nth min-eqR}$)

lemma $i\text{-Exec-Stream-append}$:

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

$f\text{-Exec-Comp-Stream trans-fun xs } c \frown$

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

by (simp add: $ilist\text{-eq-iff } i\text{-Exec-Stream-nth } f\text{-Exec-Stream-nth } f\text{-Exec-append } i\text{-append-nth } \text{Suc-diff-le}$)

lemma $i\text{-Exec-Stream-append-last-Cons}$:

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

$f\text{-Exec-Comp-Stream trans-fun xs } c \frown$

$i\text{-Exec-Comp-Stream trans-fun input ($

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

by (simp add: $f\text{-Exec-eq-}f\text{-Exec-Stream-last } i\text{-Exec-Stream-append}$)

lemma $i\text{-Exec-Stream-append-last}$:

$xs \neq [] \implies$

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

$f\text{-Exec-Comp-Stream trans-fun xs } c \frown$

$i\text{-Exec-Comp-Stream trans-fun input ($

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

by (simp add: $f\text{-Exec-Stream-empty-conv } i\text{-Exec-Stream-append-last-Cons}$)

lemma $i\text{-Exec-Stream-append-if}$:

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

$f\text{-Exec-Comp-Stream trans-fun xs } c \frown$

$i\text{-Exec-Comp-Stream trans-fun input ($

$\text{if } xs = [] \text{ then } c$

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

by (simp add: $i\text{-Exec-Stream-append-last}$)

corollary $i\text{-Exec-Stream-Cons}$:

$i\text{-Exec-Comp-Stream trans-fun ([x] } \frown \text{ input) } c =$

$[trans\text{-fun } x \text{ } c] \frown i\text{-Exec-Comp-Stream trans-fun input (trans-fun } x \text{ } c)$

by (simp add: $i\text{-Exec-Stream-append}$)

corollary $i\text{-Exec-Stream-Cons-rev}$:

$[trans\text{-fun (input } 0) \text{ } c] \frown$

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

i-Exec-Comp-Stream trans-fun input c
thm *i-Exec-Stream-append*[of *trans-fun [input 0] input ↑ Suc 0 c*]
apply (*insert i-Exec-Stream-append*[of *trans-fun [input 0] input ↑ Suc 0 c*])
apply (*simp add: i-drop-Suc-conv-tl*)
done

theorem *i-Exec-Stream-take*:
(i-Exec-Comp-Stream trans-fun input c) ↓ n =
f-Exec-Comp-Stream trans-fun (input ↓ n) c
by (*simp add: list-eq-iff f-Exec-Stream-nth i-Exec-Stream-nth min-eqR*)

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)
apply (*rule subst[OF i-append-i-take-i-drop-id, of - n input]*)
apply (*simp add: i-Exec-Stream-append i-drop-def del: i-append-i-take-i-drop-id*)
done

lemma *f-Exec-Stream-expand-aggregate-map-take*:
f-aggregate (map f (f-Exec-Comp-Stream trans-fun (xs ⊙_f k) c)) k ag ↓ n =
f-aggregate (map f (f-Exec-Comp-Stream trans-fun ((xs ↓ n) ⊙_f k) c)) k ag
by (*simp add: f-aggregate-take-mult[symmetric] take-map f-Exec-Stream-take f-expand-take-mult*)

corollary *f-Exec-Stream-expand-aggregate-take*:
f-aggregate (f-Exec-Comp-Stream trans-fun (xs ⊙_f k) c) k ag ↓ n =
f-aggregate (f-Exec-Comp-Stream trans-fun ((xs ↓ n) ⊙_f k) c) k ag
by (*insert f-Exec-Stream-expand-aggregate-map-take*[of *n id trans-fun xs k c ag*],
simp add: map-id)

lemma *i-Exec-Stream-expand-aggregate-map-take*:
 $0 < k \implies$
i-aggregate (f ∘ (i-Exec-Comp-Stream trans-fun (input ⊙_i k) c)) k ag ↓ n =
f-aggregate (map f (f-Exec-Comp-Stream trans-fun ((input ↓ n) ⊙_f k) c)) k ag
by (*simp add: i-aggregate-i-take-mult[symmetric] i-Exec-Stream-take i-expand-i-take-mult*)

corollary *i-Exec-Stream-expand-aggregate-take*:
 $0 < k \implies$
i-aggregate (i-Exec-Comp-Stream trans-fun (input ⊙_i k) c) k ag ↓ n =
f-aggregate (f-Exec-Comp-Stream trans-fun ((input ↓ n) ⊙_f k) c) k ag
by (*drule i-Exec-Stream-expand-aggregate-map-take*[of *k n id trans-fun input c ag*],
simp add: map-id)

thm *f-Exec-Stream-drop*

lemma *f-Exec-Stream-expand-aggregate-map-drop*:
f-aggregate (map f (f-Exec-Comp-Stream trans-fun (xs ⊙_f k) c)) k ag ↑ n =
f-aggregate (map f (f-Exec-Comp-Stream trans-fun ((xs ↑ n) ⊙_f k) (
f-Exec-Comp trans-fun ((xs ↓ n) ⊙_f k) c))) k ag
by (*simp add: f-aggregate-drop-mult[symmetric] drop-map f-Exec-Stream-drop f-expand-take-mult*)

f-expand-drop-mult)

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

by (*insert f-Exec-Stream-expand-aggregate-map-drop[of n id trans-fun xs k c ag],*
simp add: map-id)

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

$0 < k \implies$
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

by (*simp add: i-aggregate-i-drop-mult[symmetric] i-Exec-Stream-drop i-expand-i-take-mult*
i-expand-i-drop-mult)

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

$0 < k \implies$
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

by (*drule i-Exec-Stream-expand-aggregate-map-drop[of k n id trans-fun input c ag],*
simp)

thm *f-Exec-Stream-nth-eq-i-Exec-Stream-nth*

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

$\llbracket 0 < k; n < n' \rrbracket \implies$
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

apply (*simp add: f-aggregate-nth i-aggregate-nth f-Exec-Stream-take f-Exec-Stream-drop*
i-Exec-Stream-take i-Exec-Stream-drop drop-map take-map)

apply (*simp add: f-expand-take-mod i-expand-i-take-mod f-expand-drop-mod i-expand-i-drop-mod*
i-drop-i-take-1 drop-take-1 min-eqR)

done

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

$0 < k \implies$
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

by (*simp add: f-Exec-Stream-expand-aggregate-map-nth-eq-i-nth*)

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

$\llbracket 0 < k; n < n' \rrbracket \implies$
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

by (*drule f-Exec-Stream-expand-aggregate-map-nth-eq-i-nth[where f=id], simp-all*
add: map-id)

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

$0 < k \implies$

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*

by (*simp add: f-Exec-Stream-expand-aggregate-nth-eq-i-nth*)

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

$\llbracket 0 < k; n < \text{length } xs \rrbracket \implies$

map f (*f-Exec-Comp-Stream trans-fun* (*xs* \odot_f *k*) *c*) $\div_f k ! n =$
f (*f-Exec-Comp trans-fun* (*xs* \downarrow *Suc* *n*) \odot_f *k*) *c*)

thm *f-shrink-last-map*

apply (*simp add: f-shrink-last-map f-shrink-last-length f-shrink-last-nth*)

apply (*subgoal-tac* $n * k + k - \text{Suc } 0 < \text{length } xs * k$)

prefer 2

apply (*drule* *Suc-leI*[*of n*])

apply (*drule* *mult-le-mono1*[*of - - k*], *simp*)

apply (*simp add: f-Exec-Stream-nth add-commute*[*of k*] *f-expand-take-mult*[*symmetric*])

done

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

$\llbracket 0 < k; n < \text{length } xs \rrbracket \implies$

f-Exec-Comp-Stream trans-fun (*xs* \odot_f *k*) *c* $\div_f k ! n =$
f-Exec-Comp trans-fun (*xs* \downarrow *Suc* *n*) \odot_f *k*) *c*

by (*drule* *f-Exec-Stream-expand-shrink-last-map-nth-eq-f-Exec-Comp*[**where** *f=id*],
simp-all add: map-id)

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

$\llbracket 0 < k; n < \text{length } xs \rrbracket \implies$

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* $\# \varepsilon^k - \text{Suc } 0$)
(*f-Exec-Comp trans-fun* (*xs* \downarrow *n* \odot_f *k*) *c*)))

apply (*simp add: f-aggregate-nth take-map drop-map*)

apply (*simp add: take-map drop-map f-Exec-Stream-drop f-Exec-Stream-take f-expand-take-mod*
f-expand-drop-mod drop-take-1)

done

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

$\llbracket 0 < k; n < \text{length } xs \rrbracket \implies$

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* $\# \varepsilon^k - \text{Suc } 0$)
(*f-Exec-Comp trans-fun* (*xs* \downarrow *n* \odot_f *k*) *c*))

by (*drule* *f-Exec-Stream-expand-aggregate-map-nth*[**where** *f=id*], *simp-all add: map-id*)

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

$\llbracket 0 < k; n < \text{length } xs \rrbracket \implies$

(*map f* (*f-Exec-Comp-Stream trans-fun* (*xs* \odot_f *k*) *c*)) $\div_f k ! n =$
last-message (*map f* (*f-Exec-Comp-Stream trans-fun* (*xs* ! *n* $\# \varepsilon^k - \text{Suc } 0$)
(*f-Exec-Comp trans-fun* (*xs* \downarrow *n* \odot_f *k*) *c*)))

by (*simp add: f-shrink-def f-Exec-Stream-expand-aggregate-map-nth*)

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

$0 < k \implies$
 $i\text{-aggregate } (f \circ (i\text{-Exec-Comp-Stream trans-fun } (input \odot_i k) c)) k \text{ ag } n =$
 $ag \ (map \ f \ (f\text{-Exec-Comp-Stream trans-fun } (input \ \# \ \varepsilon^k - Suc \ 0)$
 $\ (f\text{-Exec-Comp trans-fun } ((input \ \Downarrow \ n) \odot_f k) c)))$
by (*simp add: i-aggregate-nth i-Exec-Stream-drop i-Exec-Stream-take i-expand-i-take-mod*
i-expand-i-drop-mod i-drop-i-take-1)

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

$0 < k \implies$
 $i\text{-aggregate } (i\text{-Exec-Comp-Stream trans-fun } (input \odot_i k) c) k \text{ ag } n =$
 $ag \ (f\text{-Exec-Comp-Stream trans-fun } (input \ \# \ \varepsilon^k - Suc \ 0)$
 $\ (f\text{-Exec-Comp trans-fun } ((input \ \Downarrow \ n) \odot_f k) c))$
by (*drule i-Exec-Stream-expand-aggregate-map-nth[where f=id], simp add: map-id*)

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

$0 < k \implies$
 $((f \circ (i\text{-Exec-Comp-Stream trans-fun } (input \odot_i k) c)) \div_i k) n =$
 $last\text{-message } (map \ f \ (f\text{-Exec-Comp-Stream trans-fun } (input \ \# \ \varepsilon^k - Suc \ 0)$
 $\ (f\text{-Exec-Comp trans-fun } (input \ \Downarrow \ n \odot_f k) c)))$
by (*simp add: i-shrink-def i-Exec-Stream-expand-aggregate-map-nth*)

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

$\llbracket 0 < k; n < length \ xs \rrbracket \implies$
 $f\text{-Exec-Comp-Stream trans-fun } (xs \odot_f k) c \ \uparrow \ (n * k) \ \downarrow \ k =$
 $f\text{-Exec-Comp-Stream trans-fun } (xs \ ! \ n \ \# \ \varepsilon^k - Suc \ 0)$
 $\ (f\text{-Exec-Comp trans-fun } (xs \ \downarrow \ n \odot_f k) c)$
by (*simp add: f-Exec-Stream-drop f-Exec-Stream-take f-expand-take-mod f-expand-drop-mod*
drop-take-1)

thm *f-aggregate-def*

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

$f\text{-aggregate } (map \ f \ (f\text{-Exec-Comp-Stream trans-fun } ((xs \ @ \ ys) \odot_f k) c)) k \text{ ag } =$
 $f\text{-aggregate } (map \ f \ (f\text{-Exec-Comp-Stream trans-fun } (xs \odot_f k) c)) k \text{ ag } \ @$
 $f\text{-aggregate } (map \ f \ (f\text{-Exec-Comp-Stream trans-fun } (ys \odot_f k) ($
 $\ f\text{-Exec-Comp trans-fun } (xs \odot_f k) c))) k \text{ ag}$
by (*simp add: f-Exec-Stream-append f-aggregate-append-mod*)

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

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

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

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

apply (*subst append-eq-Cons*[of x xs , *symmetric*])
apply (*subst f-Exec-Stream-expand-map-aggregate-append*)
apply (*simp add: f-aggregate-one*)
done
lemma *f-Exec-Stream-expand-map-aggregate-snoc*:
 $0 < k \implies$
 $f\text{-aggregate } (\text{map } f \text{ (} f\text{-Exec-Comp-Stream trans-fun } ((xs \text{ @ } [x]) \odot_f k) c)) k \text{ ag} =$
 $f\text{-aggregate } (\text{map } f \text{ (} f\text{-Exec-Comp-Stream trans-fun } (xs \odot_f k) c)) k \text{ ag @}$
 $[ag \text{ (} \text{map } f \text{ (} f\text{-Exec-Comp-Stream trans-fun } (x \# \varepsilon^k - \text{Suc } 0) (\text{f-Exec-Comp trans-fun } (xs \odot_f k) c)))]$
apply (*subst f-Exec-Stream-expand-map-aggregate-append*)
apply (*simp add: f-aggregate-one*)
done

lemma *i-Exec-Stream-expand-map-aggregate-Cons*:
 $0 < k \implies$
 $i\text{-aggregate } (f \circ (i\text{-Exec-Comp-Stream trans-fun } (([x] \frown \text{input}) \odot_i k) c)) k \text{ ag} =$
 $[ag \text{ (} \text{map } f \text{ (} f\text{-Exec-Comp-Stream trans-fun } (x \# \varepsilon^k - \text{Suc } 0) c))] \frown$
 $i\text{-aggregate } (f \circ (i\text{-Exec-Comp-Stream trans-fun } (\text{input} \odot_i k) (\text{f-Exec-Comp trans-fun } (x \# \varepsilon^k - \text{Suc } 0) c))) k \text{ ag}$
apply (*subst i-Exec-Stream-expand-map-aggregate-append*)
apply (*simp add: f-aggregate-one*)
done

thm *f-Exec-Stream-nth*

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

$n \leq \text{length } xs \implies$
 $f\text{-Exec-Comp-N trans-fun } n \text{ } xs \text{ } c = (c \# f\text{-Exec-Comp-Stream trans-fun } xs \text{ } c) ! n$
by (*simp add: f-Exec-Comp-N-def f-Exec-Stream-nth2*)

theorem *f-Exec-Stream-causal*:

$xs \downarrow n = ys \downarrow n \implies$
 $(f\text{-Exec-Comp-Stream trans-fun } xs \text{ } c) \downarrow n = (f\text{-Exec-Comp-Stream trans-fun } ys \text{ } c)$
 $\downarrow n$
by (*simp add: f-Exec-Stream-take*)

theorem *i-Exec-Stream-causal*:

$\text{input1} \downarrow n = \text{input2} \downarrow n \implies$
 $(i\text{-Exec-Comp-Stream trans-fun } \text{input1} \text{ } c) \downarrow n = (i\text{-Exec-Comp-Stream trans-fun } \text{input2} \text{ } c) \downarrow n$
by (*simp add: i-Exec-Stream-take*)

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*]:

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

by (*induct xs, simp-all*)

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

(*f-Exec-Comp-Stream-Init trans-fun xs c* ≠ [])

by (*simp add: length-0-conv[symmetric] del: length-0-conv*)

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)*

by (*induct xs, simp-all add: f-Exec-Stream-Init-not-empty*)

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*

by (*induct xs, simp-all*)

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)*

by (*simp add: f-Exec-Stream-Init-eq-f-Exec-Stream-Cons*)

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*

by (*simp add: f-Exec-Stream-Init-eq-f-Exec-Stream-Cons*)

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)

by (*simp add: f-Exec-Comp-N-def f-Exec-eq-f-Exec-Stream-Init-last*)

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*

apply (*subst f-Exec-Stream-Init-eq-f-Exec-Stream-Cons*)

apply (*case-tac n, simp*)

apply (*simp add: f-Exec-Stream-nth*)

done

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

by (*simp add: f-Exec-Stream-Init-nth*)

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

by (*simp add: hd-conv-nth f-Exec-Stream-Init-not-empty f-Exec-Stream-Init-nth-0*)

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*

by (*simp add: f-Exec-Stream-Init-eq-f-Exec-Stream-Cons*)

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))

by (*simp add: f-Exec-Stream-Init-eq-f-Exec-Stream-Cons f-Exec-Stream-append*)

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)))

by (*simp add: f-Exec-Stream-Init-append f-Exec-eq-f-Exec-Stream-Init-last*)

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))

by (*simp add: f-Exec-Stream-Init-eq-f-Exec-Stream-Cons f-Exec-Stream-append*)

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

by (*simp add: f-Exec-Stream-Init-eq-f-Exec-Stream-Cons f-Exec-Stream-take*)

thm *f-Exec-Stream-drop*

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)

apply (*case-tac n, simp*)

apply (*simp add: f-Exec-Stream-Init-eq-f-Exec-Stream-Cons f-Exec-Stream-drop*)

thm *take-Suc-conv-app-nth drop-Suc-conv-tl[symmetric]*

apply (*simp add: take-Suc-conv-app-nth f-Exec-append drop-Suc-conv-tl[symmetric]*)

done

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

by (*simp add: f-Exec-Stream-Init-not-empty[symmetric]*)

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

i-Exec-Comp-Stream-Init trans-fun input c n = f-Exec-Comp trans-fun (input ↓
n) c

by (*simp add: i-Exec-Comp-Stream-Init-def*)

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

i-Exec-Comp-Stream-Init trans-fun input c 0 = c

by (*simp add: i-Exec-Stream-Init-nth*)

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

by (*simp add: i-Exec-Stream-Init-nth i-Exec-Stream-nth*)

thm *f-Exec-Stream-Init-eq-f-Exec-Stream-Cons*

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

i-Exec-Comp-Stream-Init trans-fun input c = [c] \frown i-Exec-Comp-Stream trans-fun input c

by (*simp add: ilist-eq-iff i-Exec-Stream-Init-nth i-append-nth i-Exec-Stream-nth*)

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

output-fun c = ε \implies

output-fun \circ i-Exec-Comp-Stream-Init trans-fun input c =

[ε] \frown (output-fun \circ i-Exec-Comp-Stream trans-fun input c)

by (*simp add: i-Exec-Stream-Init-eq-i-Exec-Stream-Cons*)

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

i-Exec-Comp-Stream-Init trans-fun (input1 \frown input2) c =

(f-Exec-Comp-Stream-Init trans-fun input1 c) \frown

((i-Exec-Comp-Stream-Init trans-fun input2 (f-Exec-Comp trans-fun input1 c))

\uparrow Suc 0)

by (*simp add: f-Exec-Stream-Init-eq-f-Exec-Stream-Cons i-Exec-Stream-Init-eq-i-Exec-Stream-Cons i-Exec-Stream-append*)

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

i-Exec-Comp-Stream-Init trans-fun (input1 \frown input2) c =

(f-Exec-Comp-Stream-Init trans-fun input1 c) \frown

((i-Exec-Comp-Stream-Init trans-fun input2 (last (f-Exec-Comp-Stream-Init trans-fun input1 c))) \uparrow Suc 0)

by (*simp add: i-Exec-Stream-Init-append f-Exec-eq-f-Exec-Stream-Init-last*)

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

i-Exec-Comp-Stream-Init trans-fun (input1 \frown input2) c =

(f-Exec-Comp-Stream-Init trans-fun input1 c) \frown

(i-Exec-Comp-Stream trans-fun input2 (f-Exec-Comp trans-fun input1 c))

by (*simp add: f-Exec-Stream-Init-eq-f-Exec-Stream-Cons i-Exec-Stream-Init-eq-i-Exec-Stream-Cons i-Exec-Stream-append*)

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

(i-Exec-Comp-Stream-Init trans-fun input c) \Downarrow Suc n =

f-Exec-Comp-Stream-Init trans-fun (input \Downarrow n) c

by (*simp add: f-Exec-Stream-Init-eq-f-Exec-Stream-Cons i-Exec-Stream-Init-eq-i-Exec-Stream-Cons i-Exec-Stream-take*)

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

(i-Exec-Comp-Stream-Init trans-fun input c) \Uparrow n =

i-Exec-Comp-Stream-Init trans-fun (input \Uparrow n)

(f-Exec-Comp trans-fun (input \Downarrow n) c)

apply (*case-tac n, simp*)

apply (*simp add: i-Exec-Stream-Init-eq-i-Exec-Stream-Cons i-Exec-Stream-drop*)

apply (*simp add: ilist-eq-iff i-take-Suc-conv-app-nth f-Exec-append i-Exec-Stream-nth i-append-nth i-take-first i-take-drop-eq-map*)

thm *upt-conv-Cons*

apply (*simp add: upt-conv-Cons*)

done

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

$$xs \downarrow n = ys \downarrow n \implies$$

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

by (*simp add: f-Exec-Stream-Init-take*)

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

$$\text{input1} \Downarrow n = \text{input2} \Downarrow n \implies$$

$$(i\text{-Exec-Comp-Stream-Init trans-fun } \text{input1} \ c) \Downarrow \text{Suc } n = (i\text{-Exec-Comp-Stream-Init trans-fun } \text{input2} \ c) \Downarrow \text{Suc } n$$

by (*simp add: i-Exec-Stream-Init-take*)

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

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

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

by (*simp add: f-Exec-Stream-Init-eq-f-Exec-Stream-Cons f-Exec-N-eq-f-Exec-Stream-nth*)

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

$$\text{list-Previous} :: 'value \text{ list} \Rightarrow 'value \Rightarrow \text{nat} \Rightarrow 'value$$

where

$$\text{list-Previous } xs \ \text{init} \ n \equiv$$

case n of

$$0 \Rightarrow \text{init} \mid$$

$$\text{Suc } n' \Rightarrow xs \ ! \ n'$$

definition

$$\text{ilist-Previous} :: 'value \text{ ilist} \Rightarrow 'value \Rightarrow \text{nat} \Rightarrow 'value$$

where

$$\text{ilist-Previous } f \ \text{init} \ n \equiv$$

case n of

$$0 \Rightarrow \text{init} \mid$$

$$\text{Suc } n' \Rightarrow f \ n'$$

abbreviation (*xsymbols*)

$$\text{list-Previous}' :: 'value \text{ list} \Rightarrow 'value \Rightarrow \text{nat} \Rightarrow 'value \ (\text{-}^{\leftarrow} \text{-} \text{-} [1000, 10, 100] \ 100)$$

where

$$xs^{\leftarrow} \ \text{init} \ n \equiv \text{list-Previous } xs \ \text{init} \ n$$

abbreviation (*xsymbols*)

$$\text{ilist-Previous}' :: 'value \text{ ilist} \Rightarrow 'value \Rightarrow \text{nat} \Rightarrow 'value \ (\text{-}^{\leftarrow} \text{-} \text{-} [1000, 10, 100] \ 100)$$

where

$$f^{\leftarrow} \ \text{init} \ n \equiv \text{ilist-Previous } f \ \text{init} \ n$$

lemma *list-Previous-nth*: $xs^{\leftarrow'} \text{init } n = (\text{case } n \text{ of } 0 \Rightarrow \text{init} \mid \text{Suc } n' \Rightarrow xs ! n')$
by (*simp add: list-Previous-def*)
lemma *ilist-Previous-nth*: $f^{\leftarrow} \text{init } n = (\text{case } n \text{ of } 0 \Rightarrow \text{init} \mid \text{Suc } n' \Rightarrow f n')$
by (*simp add: ilist-Previous-def*)

lemma *list-Previous-nth-if*: $xs^{\leftarrow'} \text{init } n = (\text{if } n = 0 \text{ then } \text{init} \text{ else } xs ! (n - \text{Suc } 0))$
by (*case-tac n, simp-all add: list-Previous-nth*)
lemma *ilist-Previous-nth-if*: $f^{\leftarrow} \text{init } n = (\text{if } n = 0 \text{ then } \text{init} \text{ else } f (n - \text{Suc } 0))$
by (*case-tac n, simp-all add: ilist-Previous-nth*)

lemma *list-Previous-Cons*: $xs^{\leftarrow'} \text{init } n = (\text{init} \# xs) ! n$
by (*case-tac n, simp-all add: list-Previous-nth*)
lemma *ilist-Previous-Cons*: $f^{\leftarrow} \text{init } n = ([\text{init}] \frown f) n$
by (*case-tac n, simp-all add: ilist-Previous-nth*)

lemma *list-Previous-0*: $xs^{\leftarrow'} \text{init } 0 = \text{init}$
by (*simp add: list-Previous-def*)
lemma *ilist-Previous-0*: $f^{\leftarrow} \text{init } 0 = \text{init}$
by (*simp add: ilist-Previous-def*)

lemma *list-Previous-gr0*: $0 < n \Longrightarrow xs^{\leftarrow'} \text{init } n = xs ! (n - \text{Suc } 0)$
by (*case-tac n, simp-all add: list-Previous-nth*)
lemma *ilist-Previous-gr0*: $0 < n \Longrightarrow f^{\leftarrow} \text{init } n = f (n - \text{Suc } 0)$
by (*case-tac n, simp-all add: ilist-Previous-nth*)

lemma *list-Previous-Suc*: $xs^{\leftarrow'} \text{init } (\text{Suc } n) = xs ! n$
by (*simp add: list-Previous-def*)
lemma *ilist-Previous-Suc*: $f^{\leftarrow} \text{init } (\text{Suc } n) = f n$
by (*simp add: ilist-Previous-def*)

thm *list-Previous-def ilist-Previous-def*
lemma *f-Exec-Stream-Previous-f-Exec-Stream-Init*:
 $f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c ! \ n =$
 $(f\text{-Exec-Comp-Stream trans-fun } xs \ c)^{\leftarrow'} \ c \ n$
thm *f-Exec-Stream-Init-eq-f-Exec-Stream-Cons*
by (*simp add: f-Exec-Stream-Init-eq-f-Exec-Stream-Cons list-Previous-Cons*)
lemma *i-Exec-Stream-Previous-i-Exec-Stream-Init*:
 $i\text{-Exec-Comp-Stream-Init trans-fun input } c \ n =$
 $(i\text{-Exec-Comp-Stream trans-fun input } c)^{\leftarrow'} \ c \ n$
by (*simp add: i-Exec-Stream-Init-eq-i-Exec-Stream-Cons ilist-Previous-Cons*)

lemma *f-Exec-Stream-hd*:

$0 < \text{length } xs \implies \text{hd } (f\text{-Exec-Comp-Stream trans-fun } xs \ c) = \text{trans-fun } (\text{hd } xs) \ c$
by (*case-tac xs, simp+*)

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

$0 < \text{length } xs \implies (f\text{-Exec-Comp-Stream trans-fun } xs \ c) ! 0 = \text{trans-fun } (xs ! 0) \ c$
by (*case-tac xs, simp+*)

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

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

$\llbracket n < \text{length } xs; 0 < n \rrbracket \implies$
 $f\text{-Exec-Comp-Stream trans-fun } xs \ c ! n =$
 $\text{trans-fun } (xs ! n) (f\text{-Exec-Comp-Stream trans-fun } xs \ c ! (n - 1))$
by (*simp add: f-Exec-Stream-nth take-Suc-conv-app-nth f-Exec-append*)

thm *list-Previous-def*

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

$n < \text{length } xs \implies$
 $f\text{-Exec-Comp-Stream trans-fun } xs \ c ! n =$
 $\text{trans-fun } (xs ! n) ((f\text{-Exec-Comp-Stream trans-fun } xs \ c)^{\leftarrow' c} n)$
apply (*case-tac n*)
apply (*simp add: list-Previous-0 f-Exec-Stream-nth-0*)
apply (*simp add: list-Previous-def f-Exec-Stream-nth-gr0-calc*)
done

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

$(i\text{-Exec-Comp-Stream trans-fun input } c) 0 = \text{trans-fun } (\text{input } 0) \ c$
by (*simp add: i-Exec-Stream-nth i-take-first*)

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

$0 < n \implies$
 $(i\text{-Exec-Comp-Stream trans-fun input } c) n =$
 $\text{trans-fun } (\text{input } n) ((i\text{-Exec-Comp-Stream trans-fun input } c) (n - 1))$
by (*simp add: i-Exec-Stream-nth i-take-Suc-conv-app-nth f-Exec-append*)

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 .

thm *ilist-Previous-def*

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

$i\text{-Exec-Comp-Stream trans-fun input } c \ n =$
 $\text{trans-fun } (\text{input } n) ((i\text{-Exec-Comp-Stream trans-fun input } c)^{\leftarrow c} n)$
by (*simp add: i-Exec-Stream-nth ilist-Previous-nth-if i-take-first i-take-Suc-conv-app-nth f-Exec-append*)

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

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

by (*simp add: f-Exec-Stream-Init-eq-f-Exec-Stream-Cons f-Exec-Stream-nth nth-Cons'*
length-greater-0-conv[THEN iffD1, OF gr-implies-gr0] take-Suc-conv-app-nth f-Exec-append)

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

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

$f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c \ ! \ (n + 1) =$
 $\text{trans-fun } (xs \ ! \ n) \ (f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c \ ! \ n)$

by (*simp add: f-Exec-Stream-Init-nth-Suc-calc*)

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

$\llbracket n \leq \text{length } xs; 0 < n \rrbracket \implies$

$f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c \ ! \ n =$
 $\text{trans-fun } (xs \ ! \ (n - 1)) \ (f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c \ ! \ (n - 1))$

by (*clarsimp simp: gr0-conv-Suc f-Exec-Stream-Init-nth-Suc-calc*)

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.

thm *i-Exec-Stream-Init-nth-0*

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:*

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

by (*simp add: i-Exec-Stream-Init-nth i-take-Suc-conv-app-nth f-Exec-append*)

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

$i\text{-Exec-Comp-Stream-Init trans-fun } \text{input } c \ (n + 1) =$
 $\text{trans-fun } (\text{input } n) \ (i\text{-Exec-Comp-Stream-Init trans-fun } \text{input } c \ n)$

by (*simp add: i-Exec-Stream-Init-nth-Suc-calc*)

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

$0 < n \implies$

$i\text{-Exec-Comp-Stream-Init trans-fun } \text{input } c \ n =$
 $\text{trans-fun } (\text{input } (n - 1)) \ (i\text{-Exec-Comp-Stream-Init trans-fun } \text{input } c \ (n - 1))$

by (*clarsimp simp: gr0-conv-Suc i-Exec-Stream-Init-nth-Suc-calc*)

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

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

$\llbracket n < \text{length } xs;$

$c\text{-}n = (f\text{-Exec-Comp-Stream trans-fun } xs \ c) \leftarrow' c \ n; x\text{-}n = xs \ ! \ n \rrbracket \implies$

$(P1 \ x\text{-}n \ \wedge \ P2 \ c\text{-}n \ \longrightarrow \ Q \ (f\text{-Exec-Comp-Stream trans-fun } xs \ c \ ! \ n)) =$

$(P1 \ x\text{-}n \ \wedge \ P2 \ c\text{-}n \ \longrightarrow \ Q \ (\text{trans-fun } x\text{-}n \ c\text{-}n))$

by (*simp add: f-Exec-Stream-nth-calc-Previous*)

thm *f-Exec-Stream-Pre-Post1*

thm *f-Exec-Stream-Pre-Post1[OF - refl refl]*

Direct relation between input and result after transition

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

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

by (*simp add: f-Exec-Stream-nth-calc-Previous*)

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

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

by (*simp add: f-Exec-Stream-nth-gr0-calc*)

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

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

thm *f-Exec-Stream-Init-nth-Suc-calc*

by (*simp add: f-Exec-Stream-Init-nth-Suc-calc*)

Direct relation between input and state before transition

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

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

by (*simp add: f-Exec-Stream-Init-nth-Suc-calc*)

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

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

by (*simp add: i-Exec-Stream-nth-calc-Previous*)

thm *i-Exec-Stream-Pre-Post1*

thm *i-Exec-Stream-Pre-Post1[OF refl refl]*

Direct relation between input and result after transition

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

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

by (*simp add: i-Exec-Stream-nth-calc-Previous*)

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

$$\begin{aligned} & \llbracket c\text{-}n = i\text{-Exec-Comp-Stream trans-fun input } c \text{ } n; x\text{-}n1 = \text{input } (Suc \text{ } n) \rrbracket \implies \\ & (P \text{ } c\text{-}n \longrightarrow Q (\text{input } (Suc \text{ } n)) (i\text{-Exec-Comp-Stream trans-fun input } c \text{ } (Suc \text{ } n))) \\ & = \\ & (P \text{ } c\text{-}n \longrightarrow Q \text{ } x\text{-}n1 (\text{trans-fun } x\text{-}n1 \text{ } c\text{-}n)) \\ & \text{by (simp add: } i\text{-Exec-Stream-nth-gr0-calc)} \end{aligned}$$

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

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

Direct relation between input and state before transition

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

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

Basic results for stream prefixes

lemma *f-Exec-Stream-prefix*:

$$\begin{aligned} & xs \leq ys \implies \\ & f\text{-Exec-Comp-Stream trans-fun } xs \text{ } c \leq \\ & f\text{-Exec-Comp-Stream trans-fun } ys \text{ } c \\ & \text{by (clarsimp simp: prefix-def f-Exec-Stream-append)} \end{aligned}$$

lemma *i-Exec-Stream-prefix*:

$$\begin{aligned} & xs \sqsubseteq \text{input} \implies \\ & f\text{-Exec-Comp-Stream trans-fun } xs \text{ } c \sqsubseteq \\ & i\text{-Exec-Comp-Stream trans-fun input } c \\ & \text{by (simp add: prefix-def iprefix-eq-iprefix-take i-Exec-Stream-take)} \end{aligned}$$

lemma *f-Exec-N-prefix*:

$$\begin{aligned} & \llbracket n \leq \text{length } xs; xs \leq ys \rrbracket \implies \\ & f\text{-Exec-Comp-N trans-fun } n \text{ } xs \text{ } c = \\ & f\text{-Exec-Comp-N trans-fun } n \text{ } ys \text{ } c \\ & \text{by (simp add: f-Exec-Comp-N-def prefix-imp-take-eq)} \end{aligned}$$

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

$$\begin{aligned} & n \leq \text{length } (xs \sqcap ys) \implies \\ & f\text{-Exec-Comp-Stream trans-fun } xs \text{ } c \downarrow n = \\ & f\text{-Exec-Comp-Stream trans-fun } ys \text{ } c \downarrow n \\ & \text{by (rule f-Exec-Stream-causal, rule inf-prefix-take-correct)} \\ & \text{thm } f\text{-Exec-Stream-prefix-causal} \\ & \text{thm } f\text{-Exec-Stream-prefix-causal}[OF \text{ order-refl}] \end{aligned}$$

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

$xs \leq ys \implies$

f-Exec-Comp-Stream-Init trans-fun $xs\ c \leq$

f-Exec-Comp-Stream-Init trans-fun $ys\ c$

by (*clarsimp simp: prefix-def f-Exec-Stream-Init-append*)

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

$xs \sqsubseteq input \implies$

f-Exec-Comp-Stream-Init trans-fun $xs\ c \sqsubseteq$

i-Exec-Comp-Stream-Init trans-fun $input\ c$

by (*simp add: prefix-def iprefix-eq-iprefix-take i-Exec-Stream-Init-take*)

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

$n \leq \text{length } (xs \sqcap ys) \implies$

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

f-Exec-Comp-Stream-Init trans-fun $ys\ c \downarrow \text{Suc } n$

by (*rule f-Exec-Stream-Init-strictly-causal, rule inf-prefix-take-correct*)

thm *f-Exec-Stream-prefix-causal*

thm *f-Exec-Stream-prefix-causal*[*OF order-refl*]

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 ::

$(\text{'comp}, \text{'input}) \text{ Comp-Trans-Fun} \Rightarrow (\text{'comp}, \text{'state}) \text{ Comp-Local-State} \Rightarrow \text{bool}$

where

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

$\forall c1\ c2\ x. \text{localState } c1 = \text{localState } c2 \implies \text{trans-fun } x\ c1 = \text{trans-fun } x\ c2$

thm *Deterministic-Trans-Fun-def*

lemma *Deterministic-f-Exec*:

$\llbracket \text{Deterministic-Trans-Fun trans-fun } localState; \text{localState } c1 = \text{localState } c2; xs \neq [] \rrbracket \implies$

f-Exec-Comp trans-fun $xs\ c1 = \text{f-Exec-Comp trans-fun } xs\ c2$

apply (*unfold Deterministic-Trans-Fun-def*)

apply (*case-tac xs, simp*)

apply (*rename-tac y ys*)

apply (*drule-tac x=c1 in spec*)

apply (*drule-tac x=c2 in spec*)

apply *simp*

done

lemma *Deterministic-f-Exec-Stream*:

$\llbracket \text{Deterministic-Trans-Fun trans-fun } localState; \text{localState } c1 = \text{localState } c2 \rrbracket$

\implies

f-Exec-Comp-Stream trans-fun $xs\ c1 = \text{f-Exec-Comp-Stream trans-fun } xs\ c2$

apply (*clarsimp simp: list-eq-iff f-Exec-Stream-nth*)

thm *Deterministic-f-Exec*

apply (*rule Deterministic-f-Exec*)

apply (*simp add: length-greater-0-conv[THEN iffD1, OF gr-implies-gr0]*)

done

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
  apply (clarsimp simp: ilist-eq-iff i-Exec-Stream-nth)
  apply (rule Deterministic-f-Exec)
  apply simp+
  done

```

3.1.3 Connected streams

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

definition

f-Streams-Connected :: 'a fstream-af \Rightarrow 'a fstream-af \Rightarrow bool

where

f-Streams-Connected outS inS \equiv inS = ε # outS

definition

i-Streams-Connected :: 'a istream-af \Rightarrow 'a istream-af \Rightarrow bool

where

i-Streams-Connected outS inS \equiv inS = $[\varepsilon] \frown$ outS

lemmas *Streams-Connected-defs* =

f-Streams-Connected-def

i-Streams-Connected-def

thm *nth-Cons*

lemma *f-Streams-Connected-imp-not-empty*:

f-Streams-Connected outS inS \implies inS \neq []

by (simp add: *f-Streams-Connected-def*)

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

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

by (simp add: *f-Streams-Connected-def* list-eq-iff *nth-Cons*)

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

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

apply (subst *f-Streams-Connected-nth-conv*)

apply (rule conj-cong, simp)

apply (rule all-imp-eqI, simp)

apply (rename-tac i, case-tac i, simp+)

done

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

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

by (simp add: *i-Streams-Connected-def* ilist-eq-iff *i-append-nth-Cons*)

lemma *i-Streams-Connected-nth-conv-if*:
i-Streams-Connected *outS inS* =
 $(\forall i. \text{inS } i = (\text{if } i = 0 \text{ then } \varepsilon \text{ else } \text{outS } (i - \text{Suc } 0)))$
apply (*subst i-Streams-Connected-nth-conv*)
apply (*rule all-eqI*)
apply (*rename-tac i, case-tac i, simp+*)
done

lemma *f-Exec-Stream-Init-eq-output-channel*:
 $\llbracket \text{output-fun } c = \varepsilon;$
f-Streams-Connected
 $(\text{map output-fun } (f\text{-Exec-Comp-Stream trans-fun } xs \ c))$
 $\text{channel} \rrbracket \implies$
 $\text{map output-fun } (f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c) = \text{channel}$
by (*simp add: f-Streams-Connected-def f-Exec-Stream-Init-eq-f-Exec-Stream-Cons*)

lemma *i-Exec-Stream-Init-eq-output-channel*:
 $\llbracket \text{output-fun } c = \varepsilon;$
i-Streams-Connected
 $(\text{output-fun} \circ (i\text{-Exec-Comp-Stream trans-fun } \text{input } c))$
 $\text{channel} \rrbracket \implies$
 $\text{output-fun} \circ (i\text{-Exec-Comp-Stream-Init trans-fun } \text{input } c) = \text{channel}$
by (*simp add: i-Streams-Connected-def i-Exec-Stream-Init-eq-i-Exec-Stream-Cons*)

lemma *f-Exec-Stream-output-causal*:
 $\llbracket xs \downarrow n = ys \downarrow n;$
 $\text{output1} = \text{map output-fun } (f\text{-Exec-Comp-Stream trans-fun } xs \ c);$
 $\text{output2} = \text{map output-fun } (f\text{-Exec-Comp-Stream trans-fun } ys \ c) \rrbracket \implies$
 $\text{output1} \downarrow n = \text{output2} \downarrow n$
by (*simp add: take-map f-Exec-Stream-causal[of n xs]*)
thm *f-Exec-Stream-output-causal*
thm *f-Exec-Stream-output-causal[OF - refl refl]*

lemma *f-Exec-Stream-Init-output-strictly-causal*:
 $\llbracket xs \downarrow n = ys \downarrow n;$
 $\text{output1} = \text{map output-fun } (f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c);$
 $\text{output2} = \text{map output-fun } (f\text{-Exec-Comp-Stream-Init trans-fun } ys \ c) \rrbracket \implies$
 $\text{output1} \downarrow \text{Suc } n = \text{output2} \downarrow \text{Suc } n$
by (*simp add: take-map f-Exec-Stream-Init-strictly-causal[of n xs]*)

lemma *i-Exec-Stream-output-causal*:
 $\llbracket \text{input1} \Downarrow n = \text{input2} \Downarrow n;$
 $\text{output1} = \text{output-fun} \circ i\text{-Exec-Comp-Stream trans-fun } \text{input1 } c;$
 $\text{output2} = \text{output-fun} \circ i\text{-Exec-Comp-Stream trans-fun } \text{input2 } c \rrbracket \implies$
 $\text{output1} \Downarrow n = \text{output2} \Downarrow n$
by (*simp add: i-Exec-Stream-causal[of n input1]*)

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

\llbracket $input1 \Downarrow n = input2 \Downarrow n$;
 $output1 = output\text{-}fun \circ i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\text{-}trans\text{-}fun\ input1\ c$;
 $output2 = output\text{-}fun \circ i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\text{-}trans\text{-}fun\ input2\ c \rrbracket \implies$
 $output1 \Downarrow Suc\ n = output2 \Downarrow Suc\ n$

by (*simp add: i-Exec-Stream-Init-strictly-causal[of n input1]*)

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

\llbracket $xs \Downarrow n = ys \Downarrow n$;
 $f\text{-Streams-Connected}$
 $(map\ output\text{-}fun\ (f\text{-}Exec\text{-}Comp\text{-}Stream\text{-}trans\text{-}fun\ xs\ c))$
 $channel1$;
 $f\text{-Streams-Connected}$
 $(map\ output\text{-}fun\ (f\text{-}Exec\text{-}Comp\text{-}Stream\text{-}trans\text{-}fun\ ys\ c))$
 $channel2 \rrbracket \implies$
 $channel1 \Downarrow Suc\ n = channel2 \Downarrow Suc\ n$

by (*simp add: f-Streams-Connected-def take-map f-Exec-Stream-take*)

thm *i-Exec-Stream-causal*

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

\llbracket $input1 \Downarrow n = input2 \Downarrow n$;
 $i\text{-Streams-Connected}$
 $(portOutput \circ (i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}trans\text{-}fun\ input1\ c))$
 $channel1$;
 $i\text{-Streams-Connected}$
 $(portOutput \circ (i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}trans\text{-}fun\ input2\ c))$
 $channel2 \rrbracket \implies$
 $channel1 \Downarrow Suc\ n = channel2 \Downarrow Suc\ n$

by (*simp add: i-Streams-Connected-def i-take-Suc-Cons i-Exec-Stream-take*)

thm

f-Exec-Stream-Connected-strictly-causal

i-Exec-Stream-Connected-strictly-causal

A predicate for the semantics with initial state in result stream indicating for two message streams that the ports, they correspond to, are connected.

definition

f-Streams-Connected-Init :: 'a fstream-af \Rightarrow 'a fstream-af \Rightarrow bool

where

f-Streams-Connected-Init outS inS \equiv inS = outS

definition

i-Streams-Connected-Init :: 'a istream-af \Rightarrow 'a istream-af \Rightarrow bool

where

i-Streams-Connected-Init outS inS \equiv inS = outS

lemmas *Streams-Connected-Init-defs* =

f-Streams-Connected-Init-def

i-Streams-Connected-Init-def

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

f-Streams-Connected-Init *outS* *inS* =
 (length *inS* = length *outS* \wedge ($\forall i < \text{length } inS. inS ! i = outS ! i$))

by (*simp* *add*: *f-Streams-Connected-Init-def* *list-eq-iff*)

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

i-Streams-Connected-Init *outS* *inS* =
 ($\forall i. inS i = outS i$)

by (*simp* *add*: *i-Streams-Connected-Init-def* *ilist-eq-iff*)

lemma *f-Exec-Stream-Init-eq-output-channel2*:

\llbracket *output-fun* *c* = ε ;
f-Streams-Connected-Init
 (map *output-fun* (*f-Exec-Comp-Stream-Init* *trans-fun* *xs* *c*))
channel $\rrbracket \implies$
 map *output-fun* (*f-Exec-Comp-Stream-Init* *trans-fun* *xs* *c*) = *channel*

by (*simp* *add*: *f-Streams-Connected-Init-def* *f-Exec-Stream-Init-eq-f-Exec-Stream-Cons*)

lemma *i-Exec-Stream-Init-eq-output-channel2*:

\llbracket *output-fun* *c* = ε ;
i-Streams-Connected-Init
 (*output-fun* \circ (*i-Exec-Comp-Stream-Init* *trans-fun* *input* *c*))
channel $\rrbracket \implies$
output-fun \circ (*i-Exec-Comp-Stream-Init* *trans-fun* *input* *c*) = *channel*

by (*simp* *add*: *i-Streams-Connected-Init-def* *i-Exec-Stream-Init-eq-i-Exec-Stream-Cons*)

thm

f-Exec-Stream-output-causal
f-Exec-Stream-Init-output-strictly-causal
i-Exec-Stream-output-causal
i-Exec-Stream-Init-output-strictly-causal

thm

f-Exec-Stream-Connected-strictly-causal
i-Exec-Stream-Connected-strictly-causal

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

\llbracket *xs* \downarrow *n* = *ys* \downarrow *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 $\rrbracket \implies$
channel1 \downarrow *Suc* *n* = *channel2* \downarrow *Suc* *n*

by (*simp* *add*: *f-Streams-Connected-Init-def* *f-Exec-Stream-Init-eq-f-Exec-Stream-Cons*

take-map *f-Exec-Stream-take*)

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

\llbracket *input1* \Downarrow *n* = *input2* \Downarrow *n*;

$i\text{-Streams-Connected-Init}$
 $(\text{portOutput} \circ (i\text{-Exec-Comp-Stream-Init trans-fun input1 } c))$
 $\text{channel1};$
 $i\text{-Streams-Connected-Init}$
 $(\text{portOutput} \circ (i\text{-Exec-Comp-Stream-Init trans-fun input2 } c))$
 $\text{channel2} \parallel \implies$
 $\text{channel1} \Downarrow \text{Suc } n = \text{channel2} \Downarrow \text{Suc } n$
by (*simp add: i-Streams-Connected-Init-def i-Exec-Stream-Init-eq-i-Exec-Stream-Cons i-Exec-Stream-take*)

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*:

$\llbracket n1 \leq n2; n1 \in I; n2 \in I;$
 $\neg P (f\text{-Exec-Comp trans-fun (input } \downarrow n1) c);$
 $P (f\text{-Exec-Comp trans-fun (input } \downarrow n2) c) \rrbracket \implies$
 $\exists n \in I. n1 \leq n \wedge n < n2 \wedge$
 $\neg P (f\text{-Exec-Comp trans-fun (input } \downarrow n) c) \wedge$
 $P (f\text{-Exec-Comp trans-fun (input } \downarrow (\text{inext } n I)) c)$

by (*rule inext-predicate-change-exists*)

lemma *f-State-Change-exists*:

$\llbracket n1 \leq n2;$
 $\neg P (f\text{-Exec-Comp trans-fun (input } \downarrow n1) c);$
 $P (f\text{-Exec-Comp trans-fun (input } \downarrow n2) c) \rrbracket \implies$
 $\exists n \geq n1. n < n2 \wedge$
 $\neg P (f\text{-Exec-Comp trans-fun (input } \downarrow n) c) \wedge$
 $P (f\text{-Exec-Comp trans-fun (input } \downarrow (\text{Suc } n)) c)$

by (*rule nat-Suc-predicate-change-exists*)

lemma *i-State-Change-exists-set*:

$\llbracket n1 \leq n2; n1 \in I; n2 \in I;$
 $\neg P (i\text{-Exec-Comp-Stream trans-fun input } c n1);$
 $P (i\text{-Exec-Comp-Stream trans-fun input } c n2) \rrbracket \implies$
 $\exists n \in I. n1 \leq n \wedge n < n2 \wedge$
 $\neg P (i\text{-Exec-Comp-Stream trans-fun input } c n) \wedge$
 $P (i\text{-Exec-Comp-Stream trans-fun input } c (\text{inext } n I))$

by (*rule inext-predicate-change-exists*)

lemma *i-State-Change-exists*:

$\llbracket n1 \leq n2;$
 $\neg P (i\text{-Exec-Comp-Stream trans-fun input } c n1);$
 $P (i\text{-Exec-Comp-Stream trans-fun input } c n2) \rrbracket \implies$
 $\exists n \geq n1. n < n2 \wedge$
 $\neg P (i\text{-Exec-Comp-Stream trans-fun input } c n) \wedge$
 $P (i\text{-Exec-Comp-Stream trans-fun input } c (\text{Suc } n))$

by (*rule nat-Suc-predicate-change-exists*)

lemma *i-State-Change-Init-exists-set*:

$$\begin{aligned} & \llbracket n1 \leq n2; n1 \in I; n2 \in I; \\ & \quad \neg P (i\text{-Exec-Comp-Stream-Init trans-fun input } c \ n1); \\ & \quad P (i\text{-Exec-Comp-Stream-Init trans-fun input } c \ n2) \rrbracket \implies \\ & \exists n \in I. n1 \leq n \wedge n < n2 \wedge \\ & \quad \neg P (i\text{-Exec-Comp-Stream-Init trans-fun input } c \ n) \wedge \\ & \quad P (i\text{-Exec-Comp-Stream-Init trans-fun input } c \ (inext \ n \ I)) \\ & \text{by (rule inext-predicate-change-exists)} \end{aligned}$$

lemma *i-State-Change-Init-exists*:

$$\begin{aligned} & \llbracket n1 \leq n2; \\ & \quad \neg P (i\text{-Exec-Comp-Stream-Init trans-fun input } c \ n1); \\ & \quad P (i\text{-Exec-Comp-Stream-Init trans-fun input } c \ n2) \rrbracket \implies \\ & \exists n \geq n1. n < n2 \wedge \\ & \quad \neg P (i\text{-Exec-Comp-Stream-Init trans-fun input } c \ n) \wedge \\ & \quad P (i\text{-Exec-Comp-Stream-Init trans-fun input } c \ (Suc \ n)) \\ & \text{by (rule nat-Suc-predicate-change-exists)} \end{aligned}$$

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.

typ (*'comp1, 'input*) *Comp-Trans-Fun*
typ (*'input, 'input1*) *Port-Input-Value*
typ (*'comp1, 'output*) *Port-Output-Value*

definition

$$\begin{aligned} & \text{Equiv-Exec} :: \\ & \quad 'input \Rightarrow \\ & \quad ('state1 \Rightarrow 'state2 \Rightarrow \text{bool}) \Rightarrow (* \text{Equivalence predicate for local states} *) \\ & \quad ('comp1, 'state1) \text{Comp-Local-State} \Rightarrow \\ & \quad ('comp2, 'state2) \text{Comp-Local-State} \Rightarrow \\ & \quad ('input, 'input1) \text{Port-Input-Value} \Rightarrow (* \text{Input adaptor for first component} *) \\ & \quad ('input, 'input2) \text{Port-Input-Value} \Rightarrow (* \text{Input adaptor for second component} *) \\ & \quad ('comp1, 'output) \text{Port-Output-Value} \Rightarrow \\ & \quad ('comp2, 'output) \text{Port-Output-Value} \Rightarrow \\ & \quad ('comp1, 'input1 \text{message-af}) \text{Comp-Trans-Fun} \Rightarrow \\ & \quad ('comp2, 'input2 \text{message-af}) \text{Comp-Trans-Fun} \Rightarrow \end{aligned}$$

$$\text{nat} \Rightarrow \text{nat} \Rightarrow 'comp1 \Rightarrow 'comp2 \Rightarrow \text{bool}$$
where*Equiv-Exec**m equiv-states*

$$\text{localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2}$$

$$\text{trans-fun1 trans-fun2 k1 k2 c1 c2} \equiv$$

$$\text{equiv-states (localState1 c1) (localState2 c2)} \longrightarrow ($$

$$\text{last-message (map output-fun1 (}$$

$$\text{f-Exec-Comp-Stream trans-fun1 (input-fun1 m \# } \varepsilon^{k1} - \text{Suc 0) c1)) =}$$

$$\text{last-message (map output-fun2 (}$$

$$\text{f-Exec-Comp-Stream trans-fun2 (input-fun2 m \# } \varepsilon^{k2} - \text{Suc 0) c2))} \wedge$$

$$\text{equiv-states}$$

$$\text{(localState1 (f-Exec-Comp trans-fun1 (input-fun1 m \# } \varepsilon^{k1} - \text{Suc 0) c1))}$$

$$\text{(localState2 (f-Exec-Comp trans-fun2 (input-fun2 m \# } \varepsilon^{k2} - \text{Suc 0) c2))})$$
thm *Equiv-Exec-def***term** *Equiv-Exec m equiv-states equiv-input*

Predicate indicating for two components together with transition functions and a given equivalence predicate for their local states, that the equivalence predicate is stable with respect to component execution, i.e., it determines the equivalence of components' local states both for the initial states and after the components have processed an arbitrary input. The restricting version *Equiv-Exec-stable-set* guarantees stability only for inputs from a given restriction set, the not-restricting version guarantees stability for all inputs.

definition*Equiv-Exec-stable-set* ::*'input set* \Rightarrow

$$('state1 \Rightarrow 'state2 \Rightarrow \text{bool}) \Rightarrow (* \text{ Equivalence predicate for local states } *)$$

$$('comp1, 'state1) \text{ Comp-Local-State} \Rightarrow$$

$$('comp2, 'state2) \text{ Comp-Local-State} \Rightarrow$$

$$('input, 'input1) \text{ Port-Input-Value} \Rightarrow (* \text{ Input adaptor for first component } *)$$

$$('input, 'input2) \text{ Port-Input-Value} \Rightarrow (* \text{ Input adaptor for second component } *)$$

$$('comp1, 'output) \text{ Port-Output-Value} \Rightarrow$$

$$('comp2, 'output) \text{ Port-Output-Value} \Rightarrow$$

$$('comp1, 'input1 \text{ message-af}) \text{ Comp-Trans-Fun} \Rightarrow$$

$$('comp2, 'input2 \text{ message-af}) \text{ Comp-Trans-Fun} \Rightarrow$$

$$\text{nat} \Rightarrow \text{nat} \Rightarrow 'comp1 \Rightarrow 'comp2 \Rightarrow \text{bool}$$
where*Equiv-Exec-stable-set A*

$$\text{equiv-states localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2}$$

$$\text{trans-fun1 trans-fun2 k1 k2 c1 c2} \equiv$$

$$\forall \text{input m. set input} \subseteq A \wedge m \in A \longrightarrow$$
Equiv-Exec m

$$\text{equiv-states localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2}$$

$$\text{trans-fun1 trans-fun2 k1 k2}$$

$$(\text{f-Exec-Comp trans-fun1 (map input-fun1 input } \odot_f k1) c1)$$

$$(\text{f-Exec-Comp trans-fun2 (map input-fun2 input } \odot_f k2) c2)$$

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 # ε^{k1} - Suc 0) c1))
 (localState2 (f-Exec-Comp trans-fun2 (input-fun2 m # ε^{k2} - Suc 0) c2))

by (*simp* add: *Equiv-Exec-def*)**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 # ε^{k1} - Suc 0) c1)) =

last-message (map output-fun2 (

f-Exec-Comp-Stream trans-fun2 (input-fun2 m # ε^{k2} - Suc 0) c2))

by (*simp* add: *Equiv-Exec-def*)**lemma** *Equiv-Exec-equiv-statesI'*:[[*equiv-states* (localState1 c1) (localState2 c2);*Equiv-Exec*m *equiv-states*

$$\begin{array}{l} \text{localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2} \\ \text{trans-fun1 trans-fun2 k1 k2 c1 c2} \parallel \implies \\ \text{equiv-states} \\ (\text{localState1 } (f\text{-Exec-Comp trans-fun1 NoMsg}^{k1} - \text{Suc } 0 \text{ (trans-fun1 (input-fun1} \\ m) \text{ c1))))} \\ (\text{localState2 } (f\text{-Exec-Comp trans-fun2 NoMsg}^{k2} - \text{Suc } 0 \text{ (trans-fun2 (input-fun2} \\ m) \text{ c2))))} \\ \text{by (simp add: Equiv-Exec-def)} \end{array}$$

lemma *Equiv-Exec-le1*:

$$\begin{array}{l} \llbracket k1 \leq \text{Suc } 0; k2 \leq \text{Suc } 0; \\ \text{equiv-states (localState1 c1) (localState2 c2);} \\ \text{Equiv-Exec m} \\ \text{equiv-states localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2} \\ \text{trans-fun1 trans-fun2 k1 k2 c1 c2} \parallel \implies \\ \text{output-fun1 (trans-fun1 (input-fun1 m) c1) =} \\ \text{output-fun2 (trans-fun2 (input-fun2 m) c2)} \wedge \\ \text{equiv-states} \\ (\text{localState1 (trans-fun1 (input-fun1 m) c1)}) \\ (\text{localState2 (trans-fun2 (input-fun2 m) c2)}) \\ \text{by (simp add: Equiv-Exec-def)} \\ \text{thm Equiv-Exec-le1 [THEN conjunct1]} \\ \text{thm Equiv-Exec-le1 [THEN conjunct2]} \end{array}$$

lemma *Equiv-Exec-stable-set-UNIV*:

$$\begin{array}{l} \text{Equiv-Exec-stable-set} \\ \text{UNIV equiv-states} \\ \text{localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2} \\ \text{trans-fun1 trans-fun2 k1 k2 c1 c2} = \\ \text{Equiv-Exec-stable} \\ \text{equiv-states} \\ \text{localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2} \\ \text{trans-fun1 trans-fun2 k1 k2 c1 c2} \\ \text{by (simp add: Equiv-Exec-stable-set-def Equiv-Exec-stable-def)} \end{array}$$

lemma *Equiv-Exec-stable-setI*:

$$\begin{array}{l} \llbracket \text{Equiv-Exec-stable-set } A \\ \text{equiv-states localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2} \\ \text{trans-fun1 trans-fun2 k1 k2 c1 c2;} \\ \text{set input} \subseteq A; m \in A \parallel \implies \\ \text{Equiv-Exec} \\ m \text{ equiv-states} \\ \text{localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2} \\ \text{trans-fun1 trans-fun2 k1 k2} \\ (f\text{-Exec-Comp trans-fun1 (map input-fun1 input} \odot_f k1) \text{ c1)} \\ (f\text{-Exec-Comp trans-fun2 (map input-fun2 input} \odot_f k2) \text{ c2)} \\ \text{by (simp add: Equiv-Exec-stable-set-def)} \end{array}$$

lemma *Equiv-Exec-stableI*:

Equiv-Exec-stable

equiv-states localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2 \implies

Equiv-Exec m

equiv-states localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2

(f-Exec-Comp trans-fun1 (map input-fun1 input \odot_f k1) c1)

(f-Exec-Comp trans-fun2 (map input-fun2 input \odot_f k2) c2)

by (*simp add: Equiv-Exec-stable-def*)

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

lemma *Equiv-Exec-refl*:

$\llbracket \bigwedge c. \text{equiv-states } (localState\ c) (localState\ c) \rrbracket \implies$

Equiv-Exec

m equiv-states

localState localState input-fun input-fun output-fun output-fun

trans-fun trans-fun k k c c

by (*simp add: Equiv-Exec-def*)

lemma *Equiv-Exec-sym[rule-format]*:

$\llbracket \forall c1\ c2.$

equiv-states (localState1 c1) (localState2 c2) =

equiv-states (localState2 c2) (localState1 c1) \rrbracket \implies

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

by (*fastsimp simp: Equiv-Exec-def*)

lemma *Equiv-Exec-sym2*:

$\llbracket \text{equiv-states-sym} = (\lambda s1\ s2. \text{equiv-states } s2\ s1) \rrbracket \implies$

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

by (*fastsimp simp: Equiv-Exec-def*)

lemma *Equiv-Exec-sym2-ex*:

$\exists \text{equiv-states-sym.}$

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
 $localState2$ $localState1$ $input-fun2$ $input-fun1$ $output-fun2$ $output-fun1$
 $trans-fun2$ $trans-fun1$ $k2$ $k1$ $c2$ $c1$
by (rule exI , rule $Equiv-Exec-sym2$, $simp$)

lemma $Equiv-Exec-trans$:

\llbracket 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
 $localState2$ $localState3$ $input-fun2$ $input-fun3$ $output-fun2$ $output-fun3$
 $trans-fun2$ $trans-fun3$ $k2$ $k3$ $c2$ $c3$;
 $equiv-states13 = (\lambda s1\ s3.$ (
 if $s1 = localState1\ c1 \wedge s3 = localState3\ c3$ then
 $equiv-states12\ s1\ (localState2\ c2) \wedge$
 $equiv-states23\ (localState2\ c2)\ s3$
 else
 $equiv-states12\ s1\ ($
 $localState2\ (f-Exec-Comp\ trans-fun2\ (input-fun2\ m\ \# \ \epsilon^{k2} - Suc\ 0)\ c2))$
 \wedge
 $equiv-states23\ ($
 $localState2\ (f-Exec-Comp\ trans-fun2\ (input-fun2\ m\ \# \ \epsilon^{k2} - Suc\ 0)\ c2))$
 $s3)$ $\rrbracket \implies$
 Equiv-Exec
 m equiv-states13
 $localState1$ $localState3$ $input-fun1$ $input-fun3$ $output-fun1$ $output-fun3$
 $trans-fun1$ $trans-fun3$ $k1$ $k3$ $c1$ $c3$
by ($fastsimp$ $simp$: $Equiv-Exec-def$)

lemma $Equiv-Exec-trans-ex$:

\llbracket 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
 $localState2$ $localState3$ $input-fun2$ $input-fun3$ $output-fun2$ $output-fun3$
 $trans-fun2$ $trans-fun3$ $k2$ $k3$ $c2$ $c3$ $\rrbracket \implies$
 \exists $equiv-states13.$ Equiv-Exec
 m equiv-states13
 $localState1$ $localState3$ $input-fun1$ $input-fun3$ $output-fun1$ $output-fun3$
 $trans-fun1$ $trans-fun3$ $k1$ $k3$ $c1$ $c3$
by ($blast$ $intro$: $Equiv-Exec-trans$)

A predicate indicating for a given local state extraction function and a given transition function, that components, whose states are equal with regard to the local state extraction function, are transformed into equal components, when the transition function is applied with the same input.

definition *Exec-Equal-State* ::

(*'comp*, *'state*) *Comp-Local-State* \Rightarrow
(*'comp*, *'input message-af*) *Comp-Trans-Fun* \Rightarrow *bool*

where

Exec-Equal-State localState trans-fun \equiv
 $\forall c1\ c2\ m.\ localState\ c1 = localState\ c2 \longrightarrow trans-fun\ m\ c1 = trans-fun\ m\ c2$

lemma *Exec-Equal-StateD*:

$\llbracket\ Exec-Equal-State\ localState\ trans-fun;$
 $localState\ c1 = localState\ c2 \rrbracket \Longrightarrow$
 $trans-fun\ m\ c1 = trans-fun\ m\ c2$

by (*unfold Exec-Equal-State-def*, *blast*)

lemma *Exec-Equal-StateD'*:

Exec-Equal-State localState trans-fun \Longrightarrow
 $\forall c1\ c2\ m.\ localState\ c1 = localState\ c2 \longrightarrow trans-fun\ m\ c1 = trans-fun\ m\ c2$

by (*unfold Exec-Equal-State-def*, *blast*)

lemma *Exec-Equal-StateI*:

$(\bigwedge c1\ c2\ m.\ localState\ c1 = localState\ c2 \Longrightarrow trans-fun\ m\ c1 = trans-fun\ m\ c2)$
 $\Longrightarrow Exec-Equal-State\ localState\ trans-fun$

by (*unfold Exec-Equal-State-def*, *blast*)

lemma *f-Exec-Equal-State*: $\bigwedge c1\ c2.$

$\llbracket\ Exec-Equal-State\ localState\ trans-fun;$
 $localState\ c1 = localState\ c2; xs \neq [] \rrbracket \Longrightarrow$
 $f-Exec-Comp\ trans-fun\ xs\ c1 = f-Exec-Comp\ trans-fun\ xs\ c2$

apply (*induct xs*, *simp*)

apply (*case-tac xs = []*)

apply *simp*

apply (*rule Exec-Equal-StateD*, *assumption+*)

apply (*drule-tac x=trans-fun a c1 in meta-spec*)

apply (*drule-tac x=trans-fun a c2 in meta-spec*)

thm *Exec-Equal-StateD*

apply (*drule-tac ?c1.0=c1 and ?c2.0=c2 and m=a in Exec-Equal-StateD*, *assumption*)

apply *simp*

done

lemma *f-Exec-Stream-Equal-State*:

$\llbracket\ Exec-Equal-State\ localState\ trans-fun;$
 $localState\ c1 = localState\ c2 \rrbracket \Longrightarrow$
 $f-Exec-Comp-Stream\ trans-fun\ xs\ c1 =$
 $f-Exec-Comp-Stream\ trans-fun\ xs\ c2$

apply (*clarsimp simp: list-eq-iff f-Exec-Stream-nth*)

apply (*drule gr-implies-gr0*)

apply (*rule f-Exec-Equal-State*)

apply *simp+*

done

lemma *i-Exec-Stream-Equal-State*:
 $\llbracket \text{Exec-Equal-State localState trans-fun};$
 $\text{localState } c1 = \text{localState } c2 \rrbracket \implies$
 $i\text{-Exec-Comp-Stream trans-fun input } c1 =$
 $i\text{-Exec-Comp-Stream trans-fun input } c2$
apply (*clarsimp simp: ilist-eq-iff i-Exec-Stream-nth*)
apply (*rule f-Exec-Equal-State*)
apply *simp+*
done

3.2.2 Idle states

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

thm *State-Idle-def*[*THEN meta-eq-to-obj-eq, THEN iffD1*]

lemma *State-IdleD*:
 $\llbracket \text{State-Idle localState output-fun trans-fun state};$
 $\text{localState } c = \text{state} \rrbracket \implies$
 $\text{localState } (\text{trans-fun } \varepsilon \ c) = \text{state} \wedge$
 $\text{output-fun } (\text{trans-fun } \varepsilon \ c) = \varepsilon$
by (*unfold State-Idle-def, blast*)

lemma *State-IdleD'*:
 $\text{State-Idle localState output-fun trans-fun state} \implies$
 $\forall c. \text{localState } c = \text{state} \longrightarrow$
 $\text{localState } (\text{trans-fun } \varepsilon \ c) = \text{state} \wedge$
 $\text{output-fun } (\text{trans-fun } \varepsilon \ c) = \varepsilon$
by (*unfold State-Idle-def, blast*)

lemma *State-IdleI*:
 $\llbracket \bigwedge c. \text{localState } c = \text{state} \implies$
 $\text{localState } (\text{trans-fun } \varepsilon \ c) = \text{state} \wedge$
 $\text{output-fun } (\text{trans-fun } \varepsilon \ c) = \varepsilon \rrbracket \implies$
 $\text{State-Idle localState output-fun trans-fun state}$
by (*unfold State-Idle-def, blast*)

lemma *State-Idle-step*[*rule-format*]:
 $\llbracket \text{State-Idle localState output-fun trans-fun } (\text{localState } c) \rrbracket \implies$
 $\text{State-Idle localState output-fun trans-fun } (\text{localState } (\text{trans-fun } \varepsilon \ c))$
apply (*frule State-IdleD[OF - refl], erule conjE*)
apply (*rule State-IdleI, rename-tac c0*)
apply (*drule-tac c=c0 in State-IdleD*)
apply *simp+*

done

lemma *f-Exec-State-Idle-replicate-NoMsg-state*[*rule-format*]:
 $\bigwedge c. \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\text{localState } c) \implies$
 $\text{localState } (\text{f-Exec-Comp } \text{trans-fun } \varepsilon^n c) = \text{localState } c$
apply (*induct n, simp*)
apply (*frule State-Idle-step*)
apply (*drule-tac c=c in State-IdleD, rule refl*)
apply *simp*
done

lemma *f-Exec-State-Idle-replicate-NoMsg-gr0-output*[*rule-format*]: $\bigwedge c.$
 $\llbracket \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\text{localState } c); 0 < n \rrbracket \implies$
 $\text{output-fun } (\text{f-Exec-Comp } \text{trans-fun } \varepsilon^n c) = \varepsilon$
apply (*induct n, simp*)
apply (*case-tac n = 0*)
apply *simp*
apply (*rule State-IdleD[THEN conjunct2], assumption, simp*)
apply (*drule State-Idle-step*)
apply *simp*
done

lemma *f-Exec-State-Idle-replicate-NoMsg-output*[*rule-format*]:
 $\llbracket \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\text{localState } c);$
 $\text{output-fun } c = \varepsilon \rrbracket \implies$
 $\text{output-fun } (\text{f-Exec-Comp } \text{trans-fun } \varepsilon^n c) = \varepsilon$
apply (*case-tac n = 0, simp*)
apply (*simp add: f-Exec-State-Idle-replicate-NoMsg-gr0-output*)
done

lemma *f-Exec-Stream-State-Idle-replicate-NoMsg-output*[*rule-format*]:
 $\llbracket \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } (\text{localState } c) \rrbracket \implies$
 $\text{map } \text{output-fun } (\text{f-Exec-Comp-Stream } \text{trans-fun } \varepsilon^n c) = \varepsilon^n$
thm *f-Exec-State-Idle-replicate-NoMsg-gr0-output*
by (*simp add: list-eq-iff f-Exec-Stream-nth min-eqL f-Exec-State-Idle-replicate-NoMsg-gr0-output*
del: replicate.simps)

corollary *f-Exec-State-Idle-append-replicate-NoMsg-state*:
 $\llbracket \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } ($
 $\text{localState } (\text{f-Exec-Comp } \text{trans-fun } xs c)) \rrbracket \implies$
 $\text{localState } (\text{f-Exec-Comp } \text{trans-fun } (xs @ \varepsilon^n) c) =$
 $\text{localState } (\text{f-Exec-Comp } \text{trans-fun } xs c)$
by (*simp add: f-Exec-append f-Exec-State-Idle-replicate-NoMsg-state*)
corollary *f-Exec-State-Idle-append-replicate-NoMsg-ge-state*:
 $\llbracket \text{State-Idle } \text{localState } \text{output-fun } \text{trans-fun } ($
 $\text{localState } (\text{f-Exec-Comp } \text{trans-fun } (xs @ \varepsilon^m) c));$
 $m \leq n \rrbracket \implies$

$localState (f-Exec-Comp \text{ trans-fun } (xs @ \varepsilon^n) c) =$
 $localState (f-Exec-Comp \text{ trans-fun } (xs @ \varepsilon^m) c)$
apply (rule-tac $t=n$ and $s=m + (n - m)$ in subst, simp)
apply (simp only: replicate-add append-assoc[symmetric])
apply (rule $f-Exec-State-Idle-append-replicate-NoMsg-state$, simp)
done
corollary $f-Exec-State-Idle-replicate-NoMsg-ge-state$:
 $\llbracket State-Idle \text{ localState output-fun trans-fun } ($
 $localState (f-Exec-Comp \text{ trans-fun } \varepsilon^m c));$
 $m \leq n \rrbracket \implies$
 $localState (f-Exec-Comp \text{ trans-fun } \varepsilon^n c) =$
 $localState (f-Exec-Comp \text{ trans-fun } \varepsilon^m c)$
by (cut-tac $f-Exec-State-Idle-append-replicate-NoMsg-ge-state$ [where $xs=[]$], simp+)
corollary $f-Exec-State-Idle-append-replicate-NoMsg-gr0-output$:
 $\llbracket State-Idle \text{ localState output-fun trans-fun } ($
 $localState (f-Exec-Comp \text{ trans-fun } xs c));$
 $0 < n \rrbracket \implies$
 $output-fun (f-Exec-Comp \text{ trans-fun } (xs @ \varepsilon^n) c) = \varepsilon$
by (simp add: $f-Exec-append f-Exec-State-Idle-replicate-NoMsg-gr0-output$)
corollary $f-Exec-Stream-State-Idle-append-replicate-NoMsg-gr0-output$:
 $\llbracket State-Idle \text{ localState output-fun trans-fun } ($
 $localState (f-Exec-Comp \text{ trans-fun } xs c)) \rrbracket \implies$
 $map \text{ output-fun } (f-Exec-Comp-Stream \text{ trans-fun } (xs @ \varepsilon^n) c) =$
 $map \text{ output-fun } (f-Exec-Comp-Stream \text{ trans-fun } xs c) @ \varepsilon^n$
by (simp add: $f-Exec-Stream-append f-Exec-Stream-State-Idle-replicate-NoMsg-output$)
corollary $f-Exec-State-Idle-append-replicate-NoMsg-gr-output$:
 $\llbracket State-Idle \text{ localState output-fun trans-fun } ($
 $localState (f-Exec-Comp \text{ trans-fun } (xs @ \varepsilon^m) c));$
 $m < n \rrbracket \implies$
 $output-fun (f-Exec-Comp \text{ trans-fun } (xs @ \varepsilon^n) c) = \varepsilon$
apply (rule-tac $t=n$ and $s=m + (n - m)$ in subst, simp)
apply (simp only: replicate-add append-assoc[symmetric])
apply (rule $f-Exec-State-Idle-append-replicate-NoMsg-gr0-output$, simp+)
done
corollary $f-Exec-State-Idle-append-replicate-NoMsg-ge-output$:
 $\llbracket State-Idle \text{ localState output-fun trans-fun } ($
 $localState (f-Exec-Comp \text{ trans-fun } (xs @ \varepsilon^m) c));$
 $output-fun (f-Exec-Comp \text{ trans-fun } (xs @ \varepsilon^m) c) = \varepsilon; m \leq n \rrbracket \implies$
 $output-fun (f-Exec-Comp \text{ trans-fun } (xs @ \varepsilon^n) c) = \varepsilon$
by (fastsimp simp: order-le-less $f-Exec-State-Idle-append-replicate-NoMsg-gr-output$)
corollary $f-Exec-State-Idle-replicate-NoMsg-gr-output$:
 $\llbracket State-Idle \text{ localState output-fun trans-fun } ($
 $localState (f-Exec-Comp \text{ trans-fun } \varepsilon^m c));$
 $m < n \rrbracket \implies$
 $output-fun (f-Exec-Comp \text{ trans-fun } \varepsilon^n c) = \varepsilon$
thm $f-Exec-State-Idle-append-replicate-NoMsg-gr-output$
by (cut-tac $xs=[]$ in $f-Exec-State-Idle-append-replicate-NoMsg-gr-output$, simp+)
corollary $f-Exec-State-Idle-replicate-NoMsg-ge-output$:
 $\llbracket State-Idle \text{ localState output-fun trans-fun } ($

$localState (f-Exec-Comp \text{ trans-fun } \varepsilon^m c);$
 $output-fun (f-Exec-Comp \text{ trans-fun } \varepsilon^m c) = \varepsilon; m \leq n \implies$
 $output-fun (f-Exec-Comp \text{ trans-fun } \varepsilon^n c) = \varepsilon$
by (*fastsimp simp: order-le-less f-Exec-State-Idle- replicate- NoMsg-gr-output*)

lemma *State-Idle-append-replicate-NoMsg-output-last-message:*

$\llbracket State-Idle \text{ localState } output-fun \text{ trans-fun } ($
 $localState (f-Exec-Comp \text{ trans-fun } xs c)) \rrbracket \implies$
 $last-message (map \text{ output-fun } (f-Exec-Comp-Stream \text{ trans-fun } (xs @ \varepsilon^n) c)) =$
 $last-message (map \text{ output-fun } (f-Exec-Comp-Stream \text{ trans-fun } xs c))$
by (*simp add: f-Exec-Stream-State-Idle-append-replicate-NoMsg-gr0-output last-message-append-replicate-NoM*

lemma *State-Idle-append-replicate-NoMsg-output-Msg-eq-last-message:*

$\llbracket State-Idle \text{ localState } output-fun \text{ trans-fun } ($
 $localState (f-Exec-Comp \text{ trans-fun } xs c);$
 $output-fun (f-Exec-Comp \text{ trans-fun } xs c) \neq \varepsilon;$
 $xs \neq [] \rrbracket \implies$
 $last-message (map \text{ output-fun } (f-Exec-Comp-Stream \text{ trans-fun } (xs @ \varepsilon^n) c)) =$
 $output-fun (f-Exec-Comp \text{ trans-fun } xs c)$

apply (*simp add: State-Idle-append-replicate-NoMsg-output-last-message f-Exec-eq-f-Exec-Stream-last2*)

thm *last-message-Msg-eq-last*

apply (*subst last-message-Msg-eq-last*)

apply (*simp add: map-last f-Exec-Stream-not-empty-conv*)+

done

corollary *State-Idle-output-Msg-eq-last-message:*

$\llbracket State-Idle \text{ localState } output-fun \text{ trans-fun } ($
 $localState (f-Exec-Comp \text{ trans-fun } xs c);$
 $output-fun (f-Exec-Comp \text{ trans-fun } xs c) \neq \varepsilon;$
 $xs \neq [] \rrbracket \implies$
 $last-message (map \text{ output-fun } (f-Exec-Comp-Stream \text{ trans-fun } xs c)) =$
 $output-fun (f-Exec-Comp \text{ trans-fun } xs c)$

thm *subst[OF State-Idle-append-replicate-NoMsg-output-Msg-eq-last-message, rule-format]*

by (*rule-tac n=0 in subst[OF State-Idle-append-replicate-NoMsg-output-Msg-eq-last-message, rule-format], simp+*)

lemma *State-Idle-imp-exists-state-change:*

$\llbracket \neg State-Idle \text{ localState } output-fun \text{ trans-fun } (localState c);$
 $State-Idle \text{ localState } output-fun \text{ trans-fun } (localState (f-Exec-Comp \text{ trans-fun } \varepsilon^n$
 $c)) \rrbracket \implies$
 $\exists i < n. ($
 $\neg State-Idle \text{ localState } output-fun \text{ trans-fun } (localState (f-Exec-Comp \text{ trans-fun } \varepsilon^i$
 $c)) \wedge ($
 $\forall j \leq n. i < j \longrightarrow State-Idle \text{ localState } output-fun \text{ trans-fun } (localState (f-Exec-Comp$
 $\text{ trans-fun } \varepsilon^j c))))$

apply (*cut-tac*

a=0 and b=n and

P= $\lambda x. State-Idle \text{ localState } output-fun \text{ trans-fun } (localState (f-Exec-Comp \text{ trans-fun } \varepsilon^n$

```

NoMsgx c))
  in nat-Suc-predicate-change-exists, simp+)
apply (clarify, rename-tac n1)
apply (rule-tac x=n1 in exI)
apply clarsimp
apply (rule-tac t=j and s=Suc n1 + (j - Suc n1) in subst, simp)
apply (subst replicate-add)
apply (simp add: replicate-add f-Exec-State-Idle-append-replicate-NoMsg-state)
done

```

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 εn
c)) ]] ⇒
  ∃ i < n. (
    (∀ j ≤ i. ¬ State-Idle localState output-fun trans-fun (localState (f-Exec-Comp
trans-fun εi c))) ∧
    (∀ j ≤ n. i < j → State-Idle localState output-fun trans-fun (localState (f-Exec-Comp
trans-fun εj c))))
apply (frule State-Idle-imp-exists-state-change, assumption)
apply (clarify, rename-tac i)
apply (rule-tac x=i in exI)
apply simp
done

```

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)) ÷f k

```

definition

i-Exec-Comp-Stream-Acc-Output ::
 nat \Rightarrow (* Acceleration factor *)
 ('comp \Rightarrow 'output message-af) \Rightarrow (* Output extraction function *)
 ('comp, 'input message-af) *Comp-Trans-Fun* \Rightarrow
 'input istream-af \Rightarrow 'comp \Rightarrow
 'output istream-af

where

i-Exec-Comp-Stream-Acc-Output *k* *output-fun* *trans-fun* *input* *c* \equiv
 (*output-fun* \circ (*i-Exec-Comp-Stream* *trans-fun* (*input* \odot_i *k*) *c*)) \div_i *k*

definition

i-Exec-Comp-Stream-Acc-LocalState ::
 nat \Rightarrow (* Acceleration factor *)
 ('comp \Rightarrow 'state) \Rightarrow (* Local state extraction function *)
 ('comp, 'input message-af) *Comp-Trans-Fun* \Rightarrow
 'input istream-af \Rightarrow 'comp \Rightarrow
 'state ilist

where

i-Exec-Comp-Stream-Acc-LocalState *k* *localState* *trans-fun* *input* *c* \equiv
 (*localState* \circ (*i-Exec-Comp-Stream* *trans-fun* (*input* \odot_i *k*) *c*)) \div_{il} *k*

definition

f-Exec-Comp-Stream-Acc-Output-Init ::
 nat \Rightarrow (* Acceleration factor *)
 ('comp \Rightarrow 'output message-af) \Rightarrow (* Output extraction function *)
 ('comp, 'input message-af) *Comp-Trans-Fun* \Rightarrow
 'input fstream-af \Rightarrow 'comp \Rightarrow
 'output fstream-af

where

f-Exec-Comp-Stream-Acc-Output-Init *k* *output-fun* *trans-fun* *xs* *c* \equiv
 (*output-fun* *c*) # *f-Exec-Comp-Stream-Acc-Output* *k* *output-fun* *trans-fun* *xs* *c*

definition

f-Exec-Comp-Stream-Acc-LocalState-Init ::
 nat \Rightarrow (* Acceleration factor *)
 ('comp \Rightarrow 'state) \Rightarrow (* Local state extraction function *)
 ('comp, 'input message-af) *Comp-Trans-Fun* \Rightarrow 'input fstream-af \Rightarrow 'comp \Rightarrow
 'state list

where

f-Exec-Comp-Stream-Acc-LocalState-Init *k* *localState* *trans-fun* *xs* *c* \equiv
 (*localState* *c*) # *f-Exec-Comp-Stream-Acc-LocalState* *k* *localState* *trans-fun* *xs* *c*

definition

i-Exec-Comp-Stream-Acc-Output-Init ::
 nat \Rightarrow (* Acceleration factor *)
 ('comp \Rightarrow 'output message-af) \Rightarrow (* Output extraction function *)
 ('comp, 'input message-af) *Comp-Trans-Fun* \Rightarrow
 'input istream-af \Rightarrow 'comp \Rightarrow
 'output istream-af

where

$i\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun input } c \equiv$
 $[output\text{-fun } c] \curvearrowright (i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c)$

definition

$i\text{-Exec-Comp-Stream-Acc-LocalState-Init} ::$
 $nat \Rightarrow (* \text{ Acceleration factor } *)$
 $('comp \Rightarrow 'state) \Rightarrow (* \text{ Local state extraction function } *)$
 $('comp, 'input \text{ message-af}) \text{ Comp-Trans-Fun} \Rightarrow$
 $'input \text{ istream-af} \Rightarrow 'comp \Rightarrow$
 $'state \text{ ilist}$

where

$i\text{-Exec-Comp-Stream-Acc-LocalState-Init } k \text{ localState trans-fun input } c \equiv$
 $[localState } c] \curvearrowright (i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input } c)$

lemma $f\text{-Exec-Stream-Acc-Output-length}[simp]:$

$0 < k \implies$

$length (f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } xs \ c) = length \ xs$

by ($simp \text{ add: } f\text{-Exec-Comp-Stream-Acc-Output-def } f\text{-shrink-length}$)

lemma $f\text{-Exec-Stream-Acc-LocalState-length}[simp]:$

$0 < k \implies$

$length (f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } xs \ c) = length \ xs$

by ($simp \text{ add: } f\text{-Exec-Comp-Stream-Acc-LocalState-def } f\text{-shrink-last-length}$)

lemmas $f\text{-Exec-Stream-Acc-length} =$

$f\text{-Exec-Stream-Acc-LocalState-length}$

$f\text{-Exec-Stream-Acc-Output-length}$

3.2.4 Basic results for accelerated execution

lemma $f\text{-Exec-Stream-Acc-Output-Nil}[simp]:$

$f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } [] \ c = []$

by ($simp \text{ add: } f\text{-Exec-Comp-Stream-Acc-Output-def}$)

lemma $f\text{-Exec-Stream-Acc-LocalState-Nil}[simp]:$

$f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } [] \ c = []$

by ($simp \text{ add: } f\text{-Exec-Comp-Stream-Acc-LocalState-def}$)

lemmas $f\text{-Exec-Stream-Acc-Nil} =$

$f\text{-Exec-Stream-Acc-LocalState-Nil}$

$f\text{-Exec-Stream-Acc-Output-Nil}$

lemma $f\text{-Exec-Stream-Acc-Output-0}[simp]:$

$f\text{-Exec-Comp-Stream-Acc-Output } 0 \text{ output-fun trans-fun } xs \ c = []$

by ($simp \text{ add: } f\text{-Exec-Comp-Stream-Acc-Output-def}$)

lemma $f\text{-Exec-Stream-Acc-LocalState-0}[simp]:$

$f\text{-Exec-Comp-Stream-Acc-LocalState } 0 \text{ localState trans-fun } xs \ c = []$

by ($simp \text{ add: } f\text{-Exec-Comp-Stream-Acc-LocalState-def}$)

lemmas $f\text{-Exec-Stream-Acc-0} =$

$f\text{-Exec-Stream-Acc-LocalState-0}$

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)

by (simp add: f-Exec-Comp-Stream-Acc-Output-def)

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)

by (simp add: f-Exec-Comp-Stream-Acc-LocalState-def)

lemma *i-Exec-Stream-Acc-Output-1* [simp]:

i-Exec-Comp-Stream-Acc-Output (Suc 0) output-fun trans-fun input c =
output-fun \circ (i-Exec-Comp-Stream trans-fun input c)

by (simp add: i-Exec-Comp-Stream-Acc-Output-def)

lemma *i-Exec-Stream-Acc-LocalState-1* [simp]:

i-Exec-Comp-Stream-Acc-LocalState (Suc 0) localState trans-fun input c =
localState \circ (i-Exec-Comp-Stream trans-fun input c)

by (simp add: i-Exec-Comp-Stream-Acc-LocalState-def)

lemma *f-Exec-Stream-Acc-Output-eq-last-message-hold*:

f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c =
(map output-fun (f-Exec-Comp-Stream trans-fun (xs \odot_f k) c)) \mapsto_f k \div_{fl} k

by (simp add: f-Exec-Comp-Stream-Acc-Output-def f-shrink-eq-f-last-message-hold-shrink-last)

lemma *i-Exec-Stream-Acc-Output-eq-last-message-hold*: $0 < k \implies$

i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c =
(output-fun \circ (i-Exec-Comp-Stream trans-fun (input \odot_i k) c)) \mapsto_i k \div_{il} k

by (simp add: i-Exec-Comp-Stream-Acc-Output-def i-shrink-eq-i-last-message-hold-shrink-last)

thm *f-Exec-Stream-take*

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

by (simp add: f-Exec-Comp-Stream-Acc-Output-def f-shrink-def f-Exec-Stream-expand-aggregate-map-take)

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)

by (simp add: f-Exec-Comp-Stream-Acc-Output-def f-shrink-def f-Exec-Stream-expand-aggregate-map-drop)

lemma *i-Exec-Stream-Acc-Output-take*:

$0 < k \implies$

i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c \Downarrow n =
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun (input \Downarrow n) c

by (simp add: f-Exec-Comp-Stream-Acc-Output-def i-Exec-Comp-Stream-Acc-Output-def
f-shrink-def i-shrink-def i-Exec-Stream-expand-aggregate-map-take)

lemma *i-Exec-Stream-Acc-Output-drop*:

$0 < k \implies$
 $i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c \uparrow n =$
 $i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun (input } \uparrow n) (\$
 $\quad f\text{-Exec-Comp trans-fun (input } \downarrow n \odot_f k) c)$
by (*simp add: i-Exec-Comp-Stream-Acc-Output-def i-shrink-def i-Exec-Stream-expand-aggregate-map-drop*)

lemma *i-Exec-Stream-Acc-LocalState-take*:

$0 < k \implies$
 $i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input } c \downarrow n =$
 $f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun (input } \downarrow n) c$
by (*simp add: f-Exec-Comp-Stream-Acc-LocalState-def i-Exec-Comp-Stream-Acc-LocalState-def*
f-shrink-last-def i-shrink-last-def i-Exec-Stream-expand-aggregate-map-take)

lemma *i-Exec-Stream-Acc-LocalState-drop*:

$0 < k \implies$
 $i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input } c \uparrow n =$
 $i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun (input } \uparrow n) (\$
 $\quad f\text{-Exec-Comp trans-fun (input } \downarrow n \odot_f k) c)$
by (*simp add: i-Exec-Comp-Stream-Acc-LocalState-def i-shrink-last-def i-Exec-Stream-expand-aggregate-map-drop*)

thm *f-Exec-Stream-expand-map-aggregate-append*

lemma *f-Exec-Stream-Acc-Output-append*:

$f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun (xs } @ \text{ ys) } c =$
 $f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun xs } c @$
 $f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun ys } (\$
 $\quad f\text{-Exec-Comp trans-fun (xs } \odot_f k) c)$
by (*simp only: f-Exec-Comp-Stream-Acc-Output-def f-shrink-def f-Exec-Stream-expand-map-aggregate-append*)

lemma *f-Exec-Stream-Acc-Output-Cons*:

$0 < k \implies$
 $f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun (x } \# \text{ xs) } c =$
 $\text{last-message (map output-fun (f-Exec-Comp-Stream trans-fun (x } \# \text{ } \varepsilon^k - \text{Suc } 0)$
 $\text{c)) } \#$
 $f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun xs } (\$
 $\quad f\text{-Exec-Comp trans-fun (x } \# \text{ } \varepsilon^k - \text{Suc } 0) c)$
by (*simp only: f-Exec-Comp-Stream-Acc-Output-def f-shrink-def f-Exec-Stream-expand-map-aggregate-Cons*)

lemma *f-Exec-Stream-Acc-Output-one*:

$0 < k \implies$
 $f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun [x] } c =$
 $[\text{last-message (map output-fun (f-Exec-Comp-Stream trans-fun (x } \# \text{ } \varepsilon^k - \text{Suc } 0)$
 $\text{c))}]$

by (*simp add: f-Exec-Stream-Acc-Output-Cons*)

lemma *f-Exec-Stream-Acc-Output-snoc*:

$0 < k \implies$
 $f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun (xs } @ \text{ [x]) } c =$
 $f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun xs } c @$
 $[\text{last-message (map output-fun (f-Exec-Comp-Stream trans-fun (x } \# \text{ } \varepsilon^k - \text{Suc } 0)$
 $\text{c))}]$
 $($
 $\quad f\text{-Exec-Comp trans-fun (xs } \odot_f k) c)))]$

by (*simp add*: *f-Exec-Stream-Acc-Output-append f-Exec-Stream-Acc-Output-one*)

lemma *i-Exec-Stream-Acc-Output-append*:

$$\begin{aligned} & i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } (xs \frown \text{input}) \ c = \\ & f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } xs \ c \frown \\ & i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } \text{input } (\\ & \quad f\text{-Exec-Comp trans-fun } (xs \odot_f k) \ c) \end{aligned}$$

by (*simp add*: *f-Exec-Comp-Stream-Acc-Output-def i-Exec-Comp-Stream-Acc-Output-def f-shrink-def i-shrink-def i-Exec-Stream-expand-map-aggregate-append*)

lemma *i-Exec-Stream-Acc-Output-Cons*:

$$\begin{aligned} & 0 < k \implies \\ & i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } ([x] \frown \text{input}) \ c = \\ & [\text{last-message } (\text{map output-fun } (f\text{-Exec-Comp-Stream trans-fun } (x \# \varepsilon^k - \text{Suc } 0) \\ & \quad c))] \frown \\ & i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } \text{input } (\\ & \quad f\text{-Exec-Comp trans-fun } (x \# \varepsilon^k - \text{Suc } 0) \ c) \end{aligned}$$

by (*simp add*: *i-Exec-Stream-Acc-Output-append f-Exec-Stream-Acc-Output-one*)

lemma *f-Exec-Stream-Acc-LocalState-append*:

$$\begin{aligned} & f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } (xs @ ys) \ c = \\ & f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } xs \ c @ \\ & f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } ys \ (\\ & \quad f\text{-Exec-Comp trans-fun } (xs \odot_f k) \ c) \end{aligned}$$

by (*simp only*: *f-Exec-Comp-Stream-Acc-LocalState-def f-shrink-last-def f-Exec-Stream-expand-map-aggregate-a*)

lemma *f-Exec-Stream-Acc-LocalState-Cons*:

$$\begin{aligned} & 0 < k \implies \\ & f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } (x \# xs) \ c = \\ & \text{localState } (f\text{-Exec-Comp trans-fun } (x \# \varepsilon^k - \text{Suc } 0) \ c) \# \\ & f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } xs \ (\\ & \quad f\text{-Exec-Comp trans-fun } (x \# \varepsilon^k - \text{Suc } 0) \ c) \end{aligned}$$

apply (*unfold f-Exec-Comp-Stream-Acc-LocalState-def*)

apply (*simp only*: *f-shrink-last-map f-expand-Cons append-Cons[symmetric]*)

apply (*simp add*: *f-Exec-Stream-append replicate-pred-Cons-length f-shrink-last-Cons del: f-Exec-Stream-Cons append-Cons*)

apply (*simp add*: *f-Exec-eq-f-Exec-Stream-last2[symmetric] f-Exec-Stream-empty-conv*)

done

lemma *f-Exec-Stream-Acc-LocalState-one*:

$$\begin{aligned} & 0 < k \implies \\ & f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } [x] \ c = \\ & [\text{localState } (f\text{-Exec-Comp trans-fun } (x \# \varepsilon^k - \text{Suc } 0) \ c)] \end{aligned}$$

by (*simp add*: *f-Exec-Stream-Acc-LocalState-Cons*)

lemma *f-Exec-Stream-Acc-LocalState-snoc*:

$$\begin{aligned} & 0 < k \implies \\ & f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } (xs @ [x]) \ c = \\ & f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } xs \ c @ \\ & [\text{localState } (f\text{-Exec-Comp trans-fun } ((xs @ [x]) \odot_f k) \ c)] \end{aligned}$$

by (*simp add*: *f-Exec-Stream-Acc-LocalState-append f-Exec-Stream-Acc-LocalState-Cons f-Exec-append*)

lemma *i-Exec-Stream-Acc-LocalState-append*:

$$\begin{aligned} & i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } (xs \frown \text{input}) \ c = \\ & f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } xs \ c \frown \\ & i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } \text{input} \ (\\ & \quad f\text{-Exec-Comp trans-fun } (xs \odot_f k) \ c) \end{aligned}$$

by (*simp add: f-Exec-Comp-Stream-Acc-LocalState-def i-Exec-Comp-Stream-Acc-LocalState-def f-shrink-last-def i-shrink-last-def i-Exec-Stream-expand-map-aggregate-append*)

lemma *i-Exec-Stream-Acc-LocalState-Cons*:

$$\begin{aligned} & 0 < k \implies \\ & i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } ([x] \frown \text{input}) \ c = \\ & [\text{localState } (f\text{-Exec-Comp trans-fun } (x \# \varepsilon^k - \text{Suc } 0) \ c)] \frown \\ & i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } \text{input} \ (\\ & \quad f\text{-Exec-Comp trans-fun } (x \# \varepsilon^k - \text{Suc } 0) \ c) \end{aligned}$$

by (*simp add: i-Exec-Stream-Acc-LocalState-append f-Exec-Stream-Acc-LocalState-one f-expand-one*)

thm *f-Exec-Stream-expand-aggregate-map-nth*

lemma *f-Exec-Stream-Acc-Output-nth*:

$$\begin{aligned} & \llbracket 0 < k; n < \text{length } xs \rrbracket \implies \\ & f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } xs \ c \ ! \ n = \\ & \text{last-message } (\text{map output-fun } (\\ & \quad f\text{-Exec-Comp-Stream trans-fun } (xs \ ! \ n \# \varepsilon^k - \text{Suc } 0) \ (\\ & \quad \quad f\text{-Exec-Comp trans-fun } (xs \ \downarrow \ n \odot_f k) \ c))) \end{aligned}$$

by (*unfold f-Exec-Comp-Stream-Acc-Output-def f-shrink-def, rule f-Exec-Stream-expand-aggregate-map-nth*)

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

$$\begin{aligned} & \llbracket 0 < k; n < n' \rrbracket \implies \\ & f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } (\text{input} \ \downarrow \ n') \ c \ ! \ n = \\ & i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } \text{input} \ c \ n \end{aligned}$$

by (*unfold f-Exec-Comp-Stream-Acc-Output-def i-Exec-Comp-Stream-Acc-Output-def f-shrink-def i-shrink-def, rule f-Exec-Stream-expand-aggregate-map-nth-eq-i-nth*)

lemma *i-Exec-Stream-Acc-Output-nth*:

$$\begin{aligned} & 0 < k \implies \\ & i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } \text{input} \ c \ n = \\ & \text{last-message } (\text{map output-fun } (\\ & \quad f\text{-Exec-Comp-Stream trans-fun } (\text{input} \ n \# \varepsilon^k - \text{Suc } 0) \ (\\ & \quad \quad f\text{-Exec-Comp trans-fun } (\text{input} \ \downarrow \ n \odot_f k) \ c))) \end{aligned}$$

by (*unfold i-Exec-Comp-Stream-Acc-Output-def i-shrink-def, rule i-Exec-Stream-expand-aggregate-map-nth*)

corollary *i-Exec-Stream-Acc-Output-nth-f-nth*:

$$\begin{aligned} & 0 < k \implies \\ & i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } \text{input} \ c \ n = \\ & f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } (\text{input} \ \downarrow \ \text{Suc } n) \ c \ ! \ n \end{aligned}$$

by (*simp add: f-Exec-Stream-Acc-Output-nth-eq-i-nth*)

corollary *i-Exec-Stream-Acc-Output-nth-f-last*:

$$\begin{aligned} & 0 < k \implies \\ & i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } \text{input} \ c \ n = \\ & \text{last } (f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } (\text{input} \ \downarrow \ \text{Suc } n) \ c) \end{aligned}$$

by (*simp add: i-Exec-Stream-Acc-Output-nth-f-nth last-nth length-greater-0-conv[THEN iffD1]*)

lemma *f-Exec-Stream-Acc-LocalState-nth*:

$\llbracket 0 < k; n < \text{length } xs \rrbracket \implies$
 $f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } xs \ c \ ! \ n =$
 $\text{localState } (f\text{-Exec-Comp trans-fun } (xs \downarrow \text{Suc } n \odot_f k) \ c)$
apply (*simp add: f-Exec-Comp-Stream-Acc-LocalState-def f-shrink-last-map*)
apply (*simp add: f-shrink-last-nth' f-shrink-last-length del: mult-Suc*)
apply (*simp add: f-Exec-Stream-nth less-imp-Suc-mult-pred-less f-expand-take-mod*
del: mult-Suc)
done

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

$\llbracket 0 < k; n < n' \rrbracket \implies$
 $f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } (input \downarrow n') \ c \ ! \ n =$
 $i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } input \ c \ n$
by (*unfold f-Exec-Comp-Stream-Acc-LocalState-def i-Exec-Comp-Stream-Acc-LocalState-def*
f-shrink-last-def i-shrink-last-def, rule f-Exec-Stream-expand-aggregate-map-nth-eq-i-nth)

corollary *i-Exec-Stream-Acc-LocalState-nth-f-nth*:

$0 < k \implies$
 $i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ output-fun trans-fun } input \ c \ n =$
 $f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ output-fun trans-fun } (input \downarrow \text{Suc } n) \ c \ ! \ n$
by (*simp add: f-Exec-Stream-Acc-LocalState-nth-eq-i-nth*)

corollary *i-Exec-Stream-Acc-LocalState-nth-f-last*:

$0 < k \implies$
 $i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } input \ c \ n =$
 $\text{last } (f\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } (input \downarrow \text{Suc } n)$
 $c)$
by (*simp add: i-Exec-Stream-Acc-LocalState-nth-f-nth last-nth length-greater-0-conv[THEN*
iffD1])

lemma *i-Exec-Stream-Acc-LocalState-nth*:

$0 < k \implies$
 $i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } input \ c \ n =$
 $\text{localState } (f\text{-Exec-Comp trans-fun } (input \downarrow \text{Suc } n \odot_f k) \ c)$
by (*simp add: i-Exec-Stream-Acc-LocalState-nth-f-nth f-Exec-Stream-Acc-LocalState-nth*)

lemma *f-Exec-Stream-Acc-Output-causal*:

$xs \downarrow n = ys \downarrow n \implies$
 $f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } xs \ c \downarrow n =$
 $f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } ys \ c \downarrow n$
by (*simp add: f-Exec-Stream-Acc-Output-take*)

lemma *i-Exec-Stream-Acc-Output-causal*:

$input1 \downarrow n = input2 \downarrow n \implies$
 $i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } input1 \ c \downarrow n =$
 $i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } input2 \ c \downarrow n$
apply (*case-tac k = 0*)
apply (*simp add: i-Exec-Comp-Stream-Acc-Output-def*)
apply (*simp add: i-Exec-Stream-Acc-Output-take*)
done

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

$\llbracket xs \downarrow n = ys \downarrow n;$
f-Streams-Connected
 (*f-Exec-Comp-Stream-Acc-Output* *k output-fun trans-fun xs c*)
channel1;
f-Streams-Connected
 (*f-Exec-Comp-Stream-Acc-Output* *k output-fun trans-fun ys c*)
channel2 $\rrbracket \implies$
channel1 \downarrow *Suc n* = *channel2* \downarrow *Suc n*

by (*simp add: f-Streams-Connected-def f-Exec-Stream-Acc-Output-take*)

lemma *i-Exec-Stream-Acc-Output-Connected-strictly-causal*:

$\llbracket input1 \Downarrow n = input2 \Downarrow n;$
i-Streams-Connected
 (*i-Exec-Comp-Stream-Acc-Output* *k output-fun trans-fun input1 c*)
channel1;
i-Streams-Connected
 (*i-Exec-Comp-Stream-Acc-Output* *k output-fun trans-fun input2 c*)
channel2 $\rrbracket \implies$
channel1 \Downarrow *Suc n* = *channel2* \Downarrow *Suc n*

apply (*unfold i-Streams-Connected-def*)

apply (*case-tac k = 0*)

apply (*simp add: i-Exec-Comp-Stream-Acc-Output-def*)

apply (*simp add: i-Exec-Stream-Acc-Output-take*)

done

Complete execution cycles/steps of accelerated execution

definition

Acc-Trans-Fun-Step ::
nat \Rightarrow (* *Acceleration factor* *)
 (*'comp*, *'input message-af*) *Comp-Trans-Fun* \Rightarrow
 (*'comp list* \Rightarrow *'comp*) \Rightarrow (* *Pointwise output shrink function* *)
'input message-af \Rightarrow *'comp* \Rightarrow
'comp

where

Acc-Trans-Fun-Step *k trans-fun pointwise-shrink x c* \equiv
pointwise-shrink (f-Exec-Comp-Stream trans-fun (x # $\varepsilon^k - Suc 0$) c)

definition *is-Pointwise-Output-Shrink* ::

(*'comp list* \Rightarrow *'comp*) \Rightarrow (* *Pointwise output shrink function* *)
 (*'comp* \Rightarrow *'output message-af*) \Rightarrow (* *Output extraction function for consideration* *)
 *)
bool

where

is-Pointwise-Output-Shrink *pointwise-shrink output-fun* \equiv
 $\forall cs.$ *output-fun (pointwise-shrink cs)* = *last-message (map output-fun cs)*

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

(*'comp list* \Rightarrow *'comp*) \Rightarrow (* *Pointwise output shrink function* *)
 (*'comp* \Rightarrow *'output message-af*) *list* \Rightarrow (* *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 ≡
 $\forall cs. cs \neq [] \longrightarrow localState (pointwise-shrink cs) = localState (last cs)$

thm *Deterministic-Trans-Fun-def*

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

apply (*simp (no-asm) only: Deterministic-Trans-Fun-def Acc-Trans-Fun-Step-def*)

apply *clarify*

thm *Deterministic-f-Exec-Stream*

apply (*subst Deterministic-f-Exec-Stream, simp+*)

done

lemma *is-Pointwise-Output-Shrink-list-imp-is-Pointwise-Output-Shrink:*

$\llbracket is-Pointwise-Output-Shrink-list pointwise-shrink fs; output-fun \in set fs \rrbracket \implies$

is-Pointwise-Output-Shrink pointwise-shrink output-fun

apply (*induct fs, simp*)

apply *fastsimp*

done

lemma *is-Pointwise-Output-Shrink-list-eq-is-Pointwise-Output-Shrink-all:*

(is-Pointwise-Output-Shrink-list pointwise-shrink fs) =

($\forall output-fun \in set fs. is-Pointwise-Output-Shrink pointwise-shrink output-fun$)

apply (*rule iffI*)

apply (*rule ballI*)

apply (*rule is-Pointwise-Output-Shrink-list-imp-is-Pointwise-Output-Shrink*)

apply (*simp add: member-def*)⁺

apply (*induct fs, simp*)

apply *simp*

done

lemma *is-Pointwise-Output-Shrink-subset:*

$\llbracket is-Pointwise-Output-Shrink-list pointwise-shrink fs; set fs' \subseteq set fs \rrbracket \implies$

is-Pointwise-Output-Shrink-list pointwise-shrink fs'

by (*fastsimp simp: is-Pointwise-Output-Shrink-list-eq-is-Pointwise-Output-Shrink-all*)

thm *Acc-Trans-Fun-Step-def*

lemma *f-Exec-Stream-Acc-LocalState-eq-Acc-Trans-Fun-Step-LocalState*: $\bigwedge c.$

$\llbracket 0 < k;$

Deterministic-Trans-Fun trans-fun localState;

is-correct-localState-Pointwise-Output-Shrink pointwise-shrink localState $\rrbracket \implies$

f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c =

map localState (f-Exec-Comp-Stream (Acc-Trans-Fun-Step k trans-fun pointwise-shrink) xs c)

apply (*drule Deterministic-trans-fun-imp-acc-trans-fun[of trans-fun localState k pointwise-shrink]*)

apply (*clarsimp simp: list-eq-iff*)

apply (*simp add: f-Exec-Stream-Acc-LocalState-nth f-Exec-Stream-nth*)

apply (*induct xs, simp*)

apply (*rename-tac x xs c i*)

apply (*simp add: Acc-Trans-Fun-Step-def f-expand-Cons f-Exec-append*)

apply (*case-tac i*)

apply *simp*

apply (*simp only: is-correct-localState-Pointwise-Output-Shrink-def*)

apply (*drule-tac x=f-Exec-Comp-Stream trans-fun (x # NoMsg^k - Suc 0) c in spec*)

apply (*simp add: f-Exec-Stream-not-empty-conv f-Exec-eq-f-Exec-Stream-last*)

apply (*rename-tac i2*)

apply (*drule-tac x=f-Exec-Comp trans-fun $\varepsilon^k - Suc 0$ (trans-fun x c) in meta-spec*)

apply (*drule-tac x=i2 in meta-spec*)

apply (*simp add: is-correct-localState-Pointwise-Output-Shrink-def*)

apply (*drule-tac x=f-Exec-Comp-Stream trans-fun (x # NoMsg^k - Suc 0) c in spec*)

apply (*simp add: f-Exec-Stream-not-empty-conv*)

apply (*rule arg-cong[where f=localState]*)

thm *Deterministic-f-Exec*

apply (*rule Deterministic-f-Exec*)

apply *assumption*

apply (*simp add: f-Exec-eq-f-Exec-Stream-last*)

apply (*simp add: length-greater-0-conv[symmetric] del: length-greater-0-conv*)

done

lemma *f-Exec-Stream-Acc-Output-eq-Acc-Trans-Fun-Step-Output*: $\bigwedge c.$

$\llbracket 0 < k;$

Deterministic-Trans-Fun trans-fun localState;

is-correct-localState-Pointwise-Output-Shrink pointwise-shrink localState;

is-Pointwise-Output-Shrink pointwise-shrink output-fun $\rrbracket \implies$

f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c =

map output-fun (f-Exec-Comp-Stream (Acc-Trans-Fun-Step k trans-fun pointwise-shrink) xs c)

apply (*drule Deterministic-trans-fun-imp-acc-trans-fun[of trans-fun localState k pointwise-shrink]*)

apply (*clarsimp simp: list-eq-iff*)

apply (*simp add: f-Exec-Stream-Acc-Output-nth f-Exec-Stream-nth del: f-Exec-Stream-Cons*)

apply (*induct xs, simp*)

```

apply (rename-tac x xs c i)
apply (simp add: Acc-Trans-Fun-Step-def del: f-Exec-Stream-Cons)
apply (case-tac i)
  apply (simp add: is-Pointwise-Output-Shrink-def)
apply (rename-tac i2)
apply (simp add: f-Exec-append)
apply (drule-tac x=f-Exec-Comp trans-fun  $\varepsilon^k - \text{Suc } 0$  (trans-fun x c) in meta-spec)
apply (drule-tac x=i2 in meta-spec)
apply (simp add: is-correct-localState-Pointwise-Output-Shrink-def)
apply (drule-tac x=f-Exec-Comp-Stream trans-fun (x # NoMsgk - Suc 0) c in
spec)
apply (simp add: f-Exec-Stream-not-empty-conv)
apply (rule arg-cong[where f=output-fun])
thm Deterministic-f-Exec
apply (rule Deterministic-f-Exec)
  apply assumption
  apply (simp add: f-Exec-eq-f-Exec-Stream-last)
apply (simp add: length-greater-0-conv[symmetric] del: length-greater-0-conv)
done

```

thm f-Exec-Stream-Acc-LocalState-eq-Acc-Trans-Fun-Step-LocalState

thm f-Exec-Stream-Acc-Output-eq-Acc-Trans-Fun-Step-Output

lemma i-Exec-Stream-Acc-LocalState-eq-Acc-Trans-Fun-Step-LocalState: $\bigwedge c.$

$\llbracket 0 < k;$

Deterministic-Trans-Fun trans-fun localState;

is-correct-localState-Pointwise-Output-Shrink pointwise-shrink localState $\rrbracket \implies$

i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input c =

localState \circ (i-Exec-Comp-Stream (Acc-Trans-Fun-Step k trans-fun pointwise-shrink) input c)

apply (rule ilist-i-take-eq-conv[THEN iffD2], rule allI)

apply (simp add: i-Exec-Stream-Acc-LocalState-take i-Exec-Stream-take f-Exec-Stream-Acc-LocalState-eq-Acc-LocalState)

done

lemma i-Exec-Stream-Acc-Output-eq-Acc-Trans-Fun-Step-Output: $\bigwedge c.$

$\llbracket 0 < k;$

Deterministic-Trans-Fun trans-fun localState;

is-correct-localState-Pointwise-Output-Shrink pointwise-shrink localState;

is-Pointwise-Output-Shrink pointwise-shrink output-fun $\rrbracket \implies$

i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c =

output-fun \circ (i-Exec-Comp-Stream (Acc-Trans-Fun-Step k trans-fun pointwise-shrink) input c)

apply (rule ilist-i-take-eq-conv[THEN iffD2], rule allI)

apply (simp add: i-Exec-Stream-Acc-Output-take i-Exec-Stream-take f-Exec-Stream-Acc-Output-eq-Acc-Trans-Fun-Step-Output)

done

3.2.5 Basic results for accelerated execution with initial state in the resulting stream

lemma *f-Exec-Stream-Acc-Output-Init-length*:

$$0 < k \implies$$

$$\text{length } (f\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } xs \ c) = \text{Suc } (\text{length } xs)$$

by (*simp add: f-Exec-Comp-Stream-Acc-Output-Init-def*)

lemma *f-Exec-Stream-Acc-LocalState-Init-length*:

$$0 < k \implies$$

$$\text{length } (f\text{-Exec-Comp-Stream-Acc-LocalState-Init } k \text{ localState trans-fun } xs \ c) = \text{Suc } (\text{length } xs)$$

by (*simp add: f-Exec-Comp-Stream-Acc-LocalState-Init-def*)

lemma *f-Exec-Stream-Acc-Output-Init-Nil*:

$$f\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } [] \ c = [output\text{-fun } c]$$

by (*simp add: f-Exec-Comp-Stream-Acc-Output-Init-def*)

lemma *f-Exec-Stream-Acc-LocalState-Init-Nil*:

$$f\text{-Exec-Comp-Stream-Acc-LocalState-Init } k \text{ localState trans-fun } [] \ c = [localState \ c]$$

by (*simp add: f-Exec-Comp-Stream-Acc-LocalState-Init-def*)

lemma *f-Exec-Stream-Acc-Output-Init-1*:

$$f\text{-Exec-Comp-Stream-Acc-Output-Init } (\text{Suc } 0) \text{ output-fun trans-fun } xs \ c = \text{map } output\text{-fun } (f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c)$$

by (*simp add: f-Exec-Comp-Stream-Acc-Output-Init-def f-Exec-Stream-Init-eq-f-Exec-Stream-Cons*)

lemma *f-Exec-Stream-Acc-LocalState-Init-1*:

$$f\text{-Exec-Comp-Stream-Acc-LocalState-Init } (\text{Suc } 0) \text{ localState trans-fun } xs \ c = \text{map } localState \ (f\text{-Exec-Comp-Stream-Init trans-fun } xs \ c)$$

by (*simp add: f-Exec-Comp-Stream-Acc-LocalState-Init-def f-Exec-Stream-Init-eq-f-Exec-Stream-Cons*)

lemma *i-Exec-Stream-Acc-Output-Init-1*:

$$i\text{-Exec-Comp-Stream-Acc-Output-Init } (\text{Suc } 0) \text{ output-fun trans-fun } input \ c = \text{output-fun } \circ \ (i\text{-Exec-Comp-Stream-Init trans-fun } input \ c)$$

by (*simp add: i-Exec-Comp-Stream-Acc-Output-Init-def i-Exec-Stream-Init-eq-i-Exec-Stream-Cons*)

lemma *i-Exec-Stream-Acc-LocalState-Init-1*:

$$i\text{-Exec-Comp-Stream-Acc-LocalState-Init } (\text{Suc } 0) \text{ localState trans-fun } input \ c = \text{localState } \circ \ (i\text{-Exec-Comp-Stream-Init trans-fun } input \ c)$$

by (*simp add: i-Exec-Comp-Stream-Acc-LocalState-Init-def i-Exec-Stream-Init-eq-i-Exec-Stream-Cons*)

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

$$f\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } xs \ c \downarrow (\text{Suc } n) =$$

$$f\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } (xs \downarrow n) \ c$$

by (*simp add: f-Exec-Comp-Stream-Acc-Output-Init-def f-Exec-Stream-Acc-Output-take*)

thm *f-Exec-Stream-Init-drop*

lemma *f-Exec-Stream-Acc-Output-Init-drop'*:

$$[0 < k; n < \text{length } xs] \implies$$

$$f\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } xs \ c \uparrow \text{Suc } n =$$

$$f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } xs \ c \uparrow n$$

by (*simp add: f-Exec-Comp-Stream-Acc-Output-Init-def*)

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

$$0 < k \implies$$

$$i\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun input } c \Downarrow (\text{Suc } n) = \\ f\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun (input } \Downarrow n) c$$

by (*simp add: f-Exec-Comp-Stream-Acc-Output-Init-def i-Exec-Comp-Stream-Acc-Output-Init-def i-Exec-Stream-Acc-Output-take*)

lemma *i-Exec-Stream-Acc-Output-Init-drop'*:

$$0 < k \implies$$

$$i\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } xs \ c \Uparrow \text{Suc } n = \\ i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } xs \ c \Uparrow n$$

by (*simp add: i-Exec-Comp-Stream-Acc-Output-Init-def*)

thm *f-Exec-Stream-Init-strictly-causal*

lemma *f-Exec-Stream-Acc-Output-Init-strictly-causal*:

$$xs \Downarrow n = ys \Downarrow n \implies$$

$$f\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } xs \ c \Downarrow \text{Suc } n = \\ f\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } ys \ c \Downarrow \text{Suc } n$$

by (*simp add: f-Exec-Comp-Stream-Acc-Output-Init-def, rule f-Exec-Stream-Acc-Output-causal*)

thm *i-Exec-Stream-Init-strictly-causal*

lemma *i-Exec-Stream-Acc-Output-Init-strictly-causal*:

$$\text{input1 } \Downarrow n = \text{input2 } \Downarrow n \implies$$

$$i\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun input1 } c \Downarrow \text{Suc } n = \\ i\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun input2 } c \Downarrow \text{Suc } n$$

by (*simp add: i-Exec-Comp-Stream-Acc-Output-Init-def, rule i-Exec-Stream-Acc-Output-causal*)

thm *f-Exec-Stream-Init-eq-f-Exec-Stream-Cons*

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

$$f\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } xs \ c = \\ \text{output-fun } c \ \# \ f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } xs \ c$$

by (*simp add: f-Exec-Comp-Stream-Acc-Output-def f-Exec-Comp-Stream-Acc-Output-Init-def*)

thm *f-Exec-Stream-Init-eq-f-Exec-Stream-Cons-output*

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

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

$$f\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } xs \ c = \\ \varepsilon \ \# \ f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } xs \ c$$

by (*simp add: f-Exec-Stream-Acc-Output-Init-eq-f-Exec-Stream-Acc-Output-Cons*)

thm *f-Exec-Stream-Init-tl-eq-f-Exec-Stream*

lemma *f-Exec-Stream--Acc-OutputInit-tl-eq-f-Exec-Stream-Acc-Output*:

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

by (*simp add: f-Exec-Stream-Acc-Output-Init-eq-f-Exec-Stream-Acc-Output-Cons*)

thm *f-Exec-Stream-Previous-f-Exec-Stream-Init*

lemma *f-Exec-Stream-Previous-f-Exec-Stream-Acc-Output-Init*:

$$f\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } xs \ c \ ! \ n = \\ (f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } xs \ c) \leftarrow' \text{ output-fun } c \ n$$

by (*simp add: f-Exec-Stream-Acc-Output-Init-eq-f-Exec-Stream-Acc-Output-Cons list-Previous-nth-if nth-Cons*)

thm *f-Exec-Stream-Init-eq-output-channel*

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

$$\llbracket \text{ output-fun } c = \varepsilon; \\ f\text{-Streams-Connected} \\ (f\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } xs \ c) \\ \text{channel} \rrbracket \implies$$

$$f\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun } xs \ c = \text{channel}$$

by (*simp add: f-Streams-Connected-def f-Exec-Stream-Acc-Output-Init-eq-f-Exec-Stream-Acc-Output-Cons-outp*)

thm *i-Exec-Stream-Init-eq-i-Exec-Stream-Cons*

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

$$i\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun input } c = \\ [\varepsilon] \frown i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c$$

by (*simp add: i-Exec-Comp-Stream-Acc-Output-def i-Exec-Comp-Stream-Acc-Output-Init-def*)

thm *i-Exec-Stream-Init-eq-i-Exec-Stream-Cons-output*

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

$$\text{output-fun } c = \varepsilon \implies \\ i\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun input } c = \\ [\varepsilon] \frown i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c$$

by (*simp add: i-Exec-Stream-Acc-Output-Init-eq-i-Exec-Stream-Acc-Output-Cons*)

thm *i-Exec-Stream-Previous-i-Exec-Stream-Init*

lemma *i-Exec-Stream-Previous-i-Exec-Stream-Acc-Output-Init*:

$$i\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun input } c \ n = \\ (i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) \leftarrow \text{ output-fun } c \ n$$

by (*simp add: i-Exec-Stream-Acc-Output-Init-eq-i-Exec-Stream-Acc-Output-Cons ilist-Previous-nth-if*)

thm *i-Exec-Stream-Init-eq-output-channel*

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

$$\llbracket \text{ output-fun } c = \varepsilon; \\ i\text{-Streams-Connected} \\ (i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) \\ \text{channel} \rrbracket \implies$$

$$i\text{-Exec-Comp-Stream-Acc-Output-Init } k \text{ output-fun trans-fun input } c = \text{channel}$$

by (*simp add: i-Streams-Connected-def i-Exec-Stream-Acc-Output-Init-eq-i-Exec-Stream-Acc-Output-Cons-outp*)

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*]:

```

 $\wedge c1\ c2\ i.\ \llbracket$ 
   $0 < k1; 0 < k2;$ 
   $equiv\text{-}states\ (localState1\ c1)\ (localState2\ c2);$ 
   $\forall input0.\ set\ input0 \subseteq A \longrightarrow (\forall m \in A.$ 
     $Equiv\text{-}Exec\ m\ equiv\text{-}states$ 
     $localState1\ localState2\ input\text{-}fun1\ input\text{-}fun2\ output\text{-}fun1\ output\text{-}fun2$ 
     $trans\text{-}fun1\ trans\text{-}fun2\ k1\ k2$ 
     $(f\text{-}Exec\text{-}Comp\ trans\text{-}fun1\ (map\ input\text{-}fun1\ input0\ \odot_f\ k1)\ c1)$ 
     $(f\text{-}Exec\text{-}Comp\ trans\text{-}fun2\ (map\ input\text{-}fun2\ input0\ \odot_f\ k2)\ c2));$ 
     $(*\ equiv\text{-}states\ relation\ implies\ equivalent\ executions$ 
       $not\ only\ at\ the\ beginning\ but\ also\ after\ processing\ an\ input\ *)$ 
   $set\ input \subseteq A; i < length\ input\ \rrbracket \implies$ 
   $equiv\text{-}states$ 
   $(localState1\ ((f\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun1\ (map\ input\text{-}fun1\ input\ \odot_f\ k1)$ 
   $c1) \div_{\#} k1\ !\ i))$ 
   $(localState2\ ((f\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun2\ (map\ input\text{-}fun2\ input\ \odot_f\ k2)$ 
   $c2) \div_{\#} k2\ !\ i))$ 
apply  $(induct\ input,\ simp)$ 
apply  $(clarsimp\ simp:\ append\text{-}Cons[symmetric]\ f\text{-}Exec\text{-}Stream\text{-}append\text{-}if\ f\text{-}shrink\text{-}last\text{-}Cons$ 
 $nth\text{-}Cons\ simp\ del:\ last.\text{simps}\ f\text{-}Exec\text{-}Stream\text{-}Cons\ append\text{-}Cons)$ 
apply  $(case\text{-}tac\ i)$ 
apply  $(drule\text{-}tac\ x=[]\ in\ spec)$ 
apply  $(drule\ mp,\ simp)$ 
apply  $(drule\text{-}tac\ x=a\ in\ bspec,\ assumption)$ 
apply  $(simp\ del:\ last.\text{simps}\ f\text{-}Exec\text{-}Stream\text{-}Cons)$ 
apply  $(subst\ f\text{-}Exec\text{-}eq\text{-}f\text{-}Exec\text{-}Stream\text{-}last2[symmetric],\ simp)+$ 
apply  $(rule\ Equiv\text{-}Exec\text{-}equiv\text{-}statesI[of\ equiv\text{-}states\ localState1\ -\ localState2\ -\ -$ 
 $input\text{-}fun1],\ assumption+)$ 
apply  $(rename\text{-}tac\ i')$ 
thm  $f\text{-}Exec\text{-}eq\text{-}f\text{-}Exec\text{-}Stream\text{-}last2[symmetric]$ 
apply  $(subst\ f\text{-}Exec\text{-}eq\text{-}f\text{-}Exec\text{-}Stream\text{-}last2[symmetric],\ simp)+$ 
apply  $(drule\text{-}tac\ x=f\text{-}Exec\text{-}Comp\ trans\text{-}fun1\ (input\text{-}fun1\ a\ \# \ \epsilon^{k1} - Suc\ 0)\ c1\ in$ 
 $meta\text{-}spec)$ 
apply  $(drule\text{-}tac\ x=f\text{-}Exec\text{-}Comp\ trans\text{-}fun2\ (input\text{-}fun2\ a\ \# \ \epsilon^{k2} - Suc\ 0)\ c2\ in$ 
 $meta\text{-}spec)$ 
apply  $(drule\text{-}tac\ x=i'\ in\ meta\text{-}spec)$ 
apply  $(drule\ meta\text{-}mp,\ simp)+$ 
apply  $(drule\text{-}tac\ x=[]\ in\ spec,\ simp)$ 
apply  $(drule\text{-}tac\ x=a\ in\ bspec,\ assumption)$ 
thm  $Equiv\text{-}Exec\text{-}equiv\text{-}statesI'$ 
apply  $(rule\ Equiv\text{-}Exec\text{-}equiv\text{-}statesI'[of\ equiv\text{-}states\ localState1\ -\ localState2\ -\ -$ 
 $input\text{-}fun1],\ simp+)$ 
apply  $clarsimp$ 
apply  $(drule\ meta\text{-}mp)$ 
apply  $clarify$ 
apply  $(drule\text{-}tac\ x=a\ \# \ input0\ in\ spec)$ 
apply  $(simp\ add:\ f\text{-}Exec\text{-}append)$ 
apply  $simp$ 
done

```

corollary *f-Equiv-Exec-Stream-expand-shrink-equiv-state*:

[[$0 < k1$; $0 < k2$;
equiv-states (*localState1* *c1*) (*localState2* *c2*);
 \wedge *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*);
 $i < \text{length } \text{input}$]] \implies
equiv-states
(*localState1* ((*f-Exec-Comp-Stream* *trans-fun1* (*map* *input-fun1* *input* \odot_f *k1*)
c1) $\div_{\#}$ *k1* ! *i*))
(*localState2* ((*f-Exec-Comp-Stream* *trans-fun2* (*map* *input-fun2* *input* \odot_f *k2*)
c2) $\div_{\#}$ *k2* ! *i*))
by (*rule f-Equiv-Exec-Stream-expand-shrink-equiv-state-set*[*of k1 k2 equiv-states localState1 c1 localState2 c2 UNIV input-fun1 input-fun2 output-fun1 output-fun2*],
simp+)
thm *f-Equiv-Exec-Stream-expand-shrink-equiv-state-set*[*no-vars*]
thm *f-Equiv-Exec-Stream-expand-shrink-equiv-state*[*no-vars*]

thm *f-Equiv-Exec-Stream-expand-shrink-equiv-state-set*[*rule-format, no-vars*]

lemma *f-Equiv-Exec-expand-shrink-equiv-state-set*:

[[$0 < k1$; $0 < k2$; *equiv-states* (*localState1* *c1*) (*localState2* *c2*);
 \wedge *input0* *m*. [[*set* *input0* \subseteq *A*; *m* \in *A*]] \implies
Equiv-Exec
m *equiv-states* *localState1* *localState2*
input-fun1 *input-fun2* *output-fun1* *output-fun2* *trans-fun1* *trans-fun2* *k1* *k2*
(*f-Exec-Comp* *trans-fun1* (*map* *input-fun1* *input0* \odot_f *k1*) *c1*)
(*f-Exec-Comp* *trans-fun2* (*map* *input-fun2* *input0* \odot_f *k2*) *c2*);
 $\text{set } \text{input} \subseteq A$]] \implies
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*))
apply (*case-tac* *input* = [], *simp*)
apply (*subgoal-tac* *map* *input-fun1* *input* \odot_f *k1* \neq [] \wedge *map* *input-fun2* *input* \odot_f *k2* \neq [])
prefer 2
apply (*simp* *add*: *length-greater-0-conv*[*symmetric*] *del*: *length-greater-0-conv*)
apply (*simp* *add*: *f-Exec-eq-f-Exec-Stream-last2* *last-nth* *f-Exec-Stream-not-empty-conv*)
thm *f-shrink-last-nth*[*of length input - Suc 0 f-Exec-Comp-Stream trans-fun1 (map*
input-fun1 *input* \odot_f *k1*) *c1* *k1*, *symmetric*]
apply (*insert* *f-shrink-last-nth*[*of length input - Suc 0 f-Exec-Comp-Stream trans-fun1*
(*map* *input-fun1* *input* \odot_f *k1*) *c1* *k1*, *symmetric*])
apply (*insert* *f-shrink-last-nth*[*of length input - Suc 0 f-Exec-Comp-Stream trans-fun2*
(*map* *input-fun2* *input* \odot_f *k2*) *c2* *k2*, *symmetric*])
apply (*simp* *add*: *diff-mult-distrib* *gr0-imp-self-le-mult2*)
thm *f-Equiv-Exec-Stream-expand-shrink-equiv-state-set*[*rule-format, no-vars*]
thm *f-Equiv-Exec-Stream-expand-shrink-equiv-state-set*[*of k1 k2 equiv-states localState1 - localState2 - A input-fun1 input-fun2 output-fun1 output-fun2*]

apply (rule *f-Equiv-Exec-Stream-expand-shrink-equiv-state-set*[of *k1 k2 equiv-states localState1 - localState2 - A input-fun1 input-fun2 output-fun1 output-fun2*])

apply *simp+*

done

lemma *f-Equiv-Exec-expand-shrink-equiv-state*:

$\llbracket 0 < k1; 0 < k2; \text{equiv-states } (localState1\ c1)\ (localState2\ c2);$

$\wedge \text{input0}\ m.$

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) $\rrbracket \implies$

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))

thm *f-Equiv-Exec-expand-shrink-equiv-state-set*

by (rule *f-Equiv-Exec-expand-shrink-equiv-state-set*[of *k1 k2 equiv-states localState1 - localState2 - UNIV input-fun1 input-fun2 output-fun1 output-fun2*], *simp+*)

lemma *i-Equiv-Exec-Stream-expand-shrink-equiv-state-set*[*rule-format*]:

$\llbracket 0 < k1; 0 < k2; \text{equiv-states } (localState1\ c1)\ (localState2\ c2);$

$\wedge \text{input0}\ m. \llbracket \text{set input0} \subseteq A; m \in A \rrbracket \implies$

Equiv-Exec

m 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 $\subseteq A$ $\rrbracket \implies$

equiv-states

(localState1 ((i-Exec-Comp-Stream trans-fun1 ((input-fun1 \circ input) \odot_i k1) c1

\div_{il} k1) i))

(localState2 ((i-Exec-Comp-Stream trans-fun2 ((input-fun2 \circ input) \odot_i k2) c2

\div_{il} k2) i))

apply (*simp add: i-shrink-last-nth i-Exec-Stream-nth i-expand-i-take-mod*)

apply (rule *f-Equiv-Exec-expand-shrink-equiv-state-set*[of

k1 k2 equiv-states localState1 c1 localState2 c2 A input-fun1 input-fun2 output-fun1 output-fun2])

apply (*simp add: subset-trans[OF set-i-take-subset]*)**+**

done

lemma *i-Equiv-Exec-Stream-expand-shrink-equiv-state*:

$\llbracket 0 < k1; 0 < k2; \text{equiv-states } (localState1\ c1)\ (localState2\ c2);$

$\wedge \text{input0}\ m.$

Equiv-Exec

m 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) $\rrbracket \implies$

equiv-states

```

  (localState1 ((i-Exec-Comp-Stream trans-fun1 ((input-fun1 o input) oi k1) c1
  ÷i k1) i))
  (localState2 ((i-Exec-Comp-Stream trans-fun2 ((input-fun2 o input) oi k2) c2
  ÷i k2) i))
by (rule i-Equiv-Exec-Stream-expand-shrink-equiv-state-set[of k1 k2 equiv-states lo-
calState1 c1 localState2 c2 UNIV input-fun1 input-fun2 output-fun1 output-fun2],
simp+)

```

thm *f-Equiv-Exec-Stream-expand-shrink-equiv-state-set*[no-vars]

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 of k1) c1)
  (f-Exec-Comp trans-fun2 (map input-fun2 input0 of k2) c2);
  set input ⊆ A |] ⇒
  (map output-fun1 (
  f-Exec-Comp-Stream trans-fun1 (map input-fun1 input of k1) c1)) ÷f k1 =
  (map output-fun2 (
  f-Exec-Comp-Stream trans-fun2 (map input-fun2 input of k2) c2)) ÷f k2
apply (subst list-eq-iff)
apply (clarsimp simp: f-shrink-length)
apply (simp del: last.simps f-Exec-Stream-Cons add: f-shrink-nth take-map drop-map
f-Exec-Stream-take f-Exec-Stream-drop f-expand-take-mod f-expand-drop-mod take-first)
apply (frule-tac n=i in subset-trans[OF set-take-subset, rule-format])
apply (unfold atomize-all atomize-imp, intro allI impI)
apply (frule-tac x=take i input in spec)
apply (drule-tac x=input ! i in spec)
apply (erule impE, assumption)
apply (erule impE)
  apply (blast intro: nth-mem)
apply (simp del: last.simps f-Exec-Stream-Cons)
apply (rule Equiv-Exec-output-eqI[of equiv-states localState1 - localState2 - - input-fun1
input-fun2])
apply (case-tac i, simp)
thm f-Exec-Stream-expand-shrink-last-nth-eq-f-Exec-Comp[symmetric]
apply (simp add: take-map[symmetric] f-Exec-Stream-expand-shrink-last-nth-eq-f-Exec-Comp[symmetric])
apply (frule Suc-lessD)
thm f-Equiv-Exec-Stream-expand-shrink-equiv-state-set[of k1 k2 equiv-states local-
State1 - localState2 - A input-fun1 input-fun2 output-fun1 output-fun2]
apply (simp add: f-Equiv-Exec-Stream-expand-shrink-equiv-state-set[of k1 k2 equiv-states
localState1 - localState2 - A input-fun1 input-fun2 output-fun1 output-fun2])
apply simp
done

```

thm *f-Equiv-Exec-Stream-expand-shrink-output-set-eq*[no-vars]

lemma *f-Equiv-Exec-Stream-expand-shrink-output-eq*:

$\llbracket 0 < k1; 0 < k2;$
 $\text{equiv-states } (\text{localState1 } c1) (\text{localState2 } c2);$
 $\bigwedge \text{input0 } m.$
 Equiv-Exec
 $m \text{ equiv-states } \text{localState1 } \text{localState2}$
 $\text{input-fun1 } \text{input-fun2 } \text{output-fun1 } \text{output-fun2}$
 $\text{trans-fun1 } \text{trans-fun2 } k1 \ k2$
 $(f\text{-Exec-Comp } \text{trans-fun1 } (\text{map } \text{input-fun1 } \text{input0 } \odot_f k1) c1)$
 $(f\text{-Exec-Comp } \text{trans-fun2 } (\text{map } \text{input-fun2 } \text{input0 } \odot_f k2) c2) \rrbracket \implies$
 $(\text{map } \text{output-fun1 } ($
 $f\text{-Exec-Comp-Stream } \text{trans-fun1 } (\text{map } \text{input-fun1 } \text{input } \odot_f k1) c1)) \div_f k1 =$
 $(\text{map } \text{output-fun2 } ($
 $f\text{-Exec-Comp-Stream } \text{trans-fun2 } (\text{map } \text{input-fun2 } \text{input } \odot_f k2) c2)) \div_f k2$
by (*rule f-Equiv-Exec-Stream-expand-shrink-output-set-eq*[of *k1 k2 equiv-states localState1 - localState2 - UNIV*], *simp+*)
thm *f-Equiv-Exec-Stream-expand-shrink-output-eq*[*no-vars*]

lemma *i-Equiv-Exec-Stream-expand-shrink-output-set-eq*:

$\llbracket 0 < k1; 0 < k2;$
 $\text{equiv-states } (\text{localState1 } c1) (\text{localState2 } c2);$
 $\bigwedge \text{input0 } m. \llbracket \text{set } \text{input0 } \subseteq A; m \in A \rrbracket \implies$
 Equiv-Exec
 $m \text{ equiv-states } \text{localState1 } \text{localState2}$
 $\text{input-fun1 } \text{input-fun2 } \text{output-fun1 } \text{output-fun2}$
 $\text{trans-fun1 } \text{trans-fun2 } k1 \ k2$
 $(f\text{-Exec-Comp } \text{trans-fun1 } (\text{map } \text{input-fun1 } \text{input0 } \odot_f k1) c1)$
 $(f\text{-Exec-Comp } \text{trans-fun2 } (\text{map } \text{input-fun2 } \text{input0 } \odot_f k2) c2);$
 $\text{range } \text{input} \subseteq A \rrbracket \implies$
 $(\text{output-fun1 } \circ$
 $i\text{-Exec-Comp-Stream } \text{trans-fun1 } ((\text{input-fun1 } \circ \text{input}) \odot_i k1) c1) \div_i k1 =$
 $(\text{output-fun2 } \circ$
 $i\text{-Exec-Comp-Stream } \text{trans-fun2 } ((\text{input-fun2 } \circ \text{input}) \odot_i k2) c2) \div_i k2$
apply (*clarsimp simp: ilist-eq-iff, rename-tac i*)
apply (*simp del: last.simps f-Exec-Stream-Cons add: i-shrink-nth i-Exec-Stream-take*
i-Exec-Stream-drop i-expand-i-take-mod i-expand-i-drop-mod i-take-first map-one
f-expand-one)
apply (*rule Equiv-Exec-output-eqI*[of
 $\text{equiv-states } \text{localState1 } - \text{localState2 } - -$
 $\text{input-fun1 } \text{input-fun2 } \text{output-fun1 } \text{output-fun2 } \text{trans-fun1 } \text{trans-fun2 } k1 \ k2$])
apply (*rule f-Equiv-Exec-expand-shrink-equiv-state-set*[of
 $k1 \ k2 \ \text{equiv-states } \text{localState1 } - \text{localState2 } - A$
 $\text{input-fun1 } \text{input-fun2 } \text{output-fun1 } \text{output-fun2 } \text{trans-fun1 } \text{trans-fun2}$])
apply (*simp add: subset-trans[OF set-i-take-subset] subsetD[OF - rangeI]*)+
done

lemma *i-Equiv-Exec-Stream-expand-shrink-output-eq*:

$\llbracket 0 < k1; 0 < k2;$
 $\text{equiv-states } (\text{localState1 } c1) (\text{localState2 } c2);$

```

 $\wedge$  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) ]  $\implies$ 
  (output-fun1  $\circ$ 
    i-Exec-Comp-Stream trans-fun1 ((input-fun1  $\circ$  input)  $\odot_i$  k1) c1)  $\div_i$  k1 =
  (output-fun2  $\circ$ 
    i-Exec-Comp-Stream trans-fun2 ((input-fun2  $\circ$  input)  $\odot_i$  k2) c2)  $\div_i$  k2
apply (rule i-Equiv-Exec-Stream-expand-shrink-output-set-eq[of
  k1 k2 equiv-states localState1 c1 localState2 c2 UNIV
  input-fun1 input-fun2 output-fun1 output-fun2 trans-fun1 trans-fun2])
apply simp+
done

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  $\subseteq$  A;
    i < length input ]  $\implies$ 
  equiv-states
    (f-Exec-Comp-Stream-Acc-LocalState k1 localState1 trans-fun1 (map input-fun1
  input) c1 ! i)
    (f-Exec-Comp-Stream-Acc-LocalState k2 localState2 trans-fun2 (map input-fun2
  input) c2 ! i)
apply (unfold f-Exec-Comp-Stream-Acc-LocalState-def Equiv-Exec-stable-set-def)
apply (simp add: f-shrink-last-map f-shrink-last-length)
thm f-Equiv-Exec-Stream-expand-shrink-equiv-state-set[of
  k1 k2 equiv-states localState1 c1 localState2 c2 A
  input-fun1 input-fun2 output-fun1 output-fun2 trans-fun1 trans-fun2 input]
apply (rule f-Equiv-Exec-Stream-expand-shrink-equiv-state-set[of
  k1 k2 equiv-states localState1 c1 localState2 c2 A
  input-fun1 input-fun2 output-fun1 output-fun2 trans-fun1 trans-fun2 input, rule-format])
apply simp+
done

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)
apply (rule f-Equiv-Exec-Stream-Acc-LocalState-set[where A=UNIV])
apply (simp add: Equiv-Exec-stable-set-UNIV)+
done
thm f-Equiv-Exec-Stream-Acc-LocalState[no-vars]

```

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 in-
put) c1 =
   f-Exec-Comp-Stream-Acc-Output k2 output-fun2 trans-fun2 (map input-fun2 in-
put) c2
apply (unfold f-Exec-Comp-Stream-Acc-Output-def Equiv-Exec-stable-set-def)
apply (rule f-Equiv-Exec-Stream-expand-shrink-output-set-eq[of
k1 k2 equiv-states localState1 c1 localState2 c2
A input-fun1 input-fun2 output-fun1 output-fun2 trans-fun1 trans-fun2 input])
apply simp+
done

```

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 in-
put) c1 =
   f-Exec-Comp-Stream-Acc-Output k2 output-fun2 trans-fun2 (map input-fun2 in-
put) c2
apply (rule f-Equiv-Exec-Stream-Acc-Output-set-eq[of k1 k2 equiv-states localState1
c1 localState2 c2 UNIV])
apply (simp add: Equiv-Exec-stable-set-UNIV)+
done
thm f-Equiv-Exec-Stream-Acc-Output-eq[no-vars]

```

lemma *i-Equiv-Exec-Stream-Acc-LocalState-set*:
 $\llbracket 0 < k1; 0 < k2;$
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* \subseteq *A* $\rrbracket \implies$
equiv-states
(*i-Exec-Comp-Stream-Acc-LocalState* *k1* *localState1* *trans-fun1* (*input-fun1* \circ
input) *c1* *i*)
(*i-Exec-Comp-Stream-Acc-LocalState* *k2* *localState2* *trans-fun2* (*input-fun2* \circ
input) *c2* *i*)
apply (*simp* *add*: *i-Exec-Stream-Acc-LocalState-nth-f-nth*)
apply (*rule* *f-Equiv-Exec-Stream-Acc-LocalState-set*)
apply (*simp* *add*: *subset-trans[OF set-i-take-subset]*)+
done

lemma *i-Equiv-Exec-Stream-Acc-LocalState*:
 $\llbracket 0 < k1; 0 < k2;$
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* $\rrbracket \implies$
equiv-states
(*i-Exec-Comp-Stream-Acc-LocalState* *k1* *localState1* *trans-fun1* (*input-fun1* \circ
input) *c1* *i*)
(*i-Exec-Comp-Stream-Acc-LocalState* *k2* *localState2* *trans-fun2* (*input-fun2* \circ
input) *c2* *i*)
apply (*rule* *i-Equiv-Exec-Stream-Acc-LocalState-set*[**where** *A*=*UNIV*])
apply (*simp* *add*: *Equiv-Exec-stable-set-UNIV*)+
done

lemma *i-Equiv-Exec-Stream-Acc-Output-set-eq*:
 $\llbracket 0 < k1; 0 < k2;$
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* \subseteq *A* $\rrbracket \implies$
i-Exec-Comp-Stream-Acc-Output *k1* *output-fun1* *trans-fun1* (*input-fun1* \circ *input*)
c1 =
i-Exec-Comp-Stream-Acc-Output *k2* *output-fun2* *trans-fun2* (*input-fun2* \circ *input*)
c2
apply (*clarsimp* *simp*: *ilist-eq-iff i-Exec-Stream-Acc-Output-nth-f-nth*, *rename-tac*
i)

```

apply (drule-tac n=Suc i in subset-trans[OF set-i-take-subset, rule-format])
apply (simp add: f-Equiv-Exec-Stream-Acc-Output-set-eq[where equiv-states=equiv-states])
done

```

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 o input)
c1 =
  i-Exec-Comp-Stream-Acc-Output k2 output-fun2 trans-fun2 (input-fun2 o input)
c2
apply (rule i-Equiv-Exec-Stream-Acc-Output-set-eq[of k1 k2 equiv-states localState1
c1 localState2 c2 UNIV])
apply (simp add: Equiv-Exec-stable-set-UNIV)+
done

```

3.2.7 Idle states and accelerated execution

lemma *f-Exec-Stream-Acc-LocalState--State-Idle-nth*[rule-format]:

```

 $\bigwedge^c i.$ 
[[ 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
apply (frule length-greater-0-conv[THEN iffD1, OF gr-implies-gr0])
apply (simp only: f-Exec-Stream-Acc-LocalState-nth take-Suc-conv-app-nth)
apply (simp only: f-expand-snoc f-Exec-append)
apply (rule-tac s=εl - Suc 0 @ εk-l and t=εk - Suc 0 in subst)
apply (simp add: replicate-le-diff2)
apply (subst append-Cons[symmetric])
apply (induct xs, simp)
apply (case-tac i)
thm f-Exec-State-Idle-append-replicate-NoMsg-state
apply (simp add: f-Exec-Stream-Acc-LocalState-Cons f-Exec-State-Idle-append-replicate-NoMsg-state)
apply (rename-tac n)
apply (drule-tac x=f-Exec-Comp trans-fun (a # εl - Suc 0) c in meta-spec)
apply (drule-tac x=n in meta-spec)
apply (simp del: f-Exec-Cons)
apply (frule length-greater-imp-not-empty)
apply (drule meta-mp)
thm f-Exec-Stream-Acc-LocalState-nth
apply (simp add: f-Exec-Stream-Acc-LocalState-nth f-Exec-append)
apply (simp add: append-Cons[symmetric] f-expand-Cons f-Exec-append del: append-Cons)

```

apply (*subgoal-tac*
localState (f-Exec-Comp trans-fun (a # NoMsg^k - Suc 0) c) =
localState (f-Exec-Comp trans-fun (a # NoMsg^l - Suc 0) c))
prefer 2
apply (*drule-tac x=0 in spec*)
apply (*simp add: f-Exec-Stream-Acc-LocalState-Cons*)
apply (*subst replicate-le-diff2[OF Suc-leI, symmetric], assumption+*)
apply (*simp add: append-Cons[symmetric] f-Exec-append del: append-Cons*)
apply (*rule f-Exec-State-Idle-replicate-NoMsg-state, assumption*)
apply (*case-tac n = 0*)
apply (*frule-tac*
?c1.0=f-Exec-Comp trans-fun (a # NoMsg^k - Suc 0) c and
xs = xs ! 0 # NoMsg^l - Suc 0 in f-Exec-Equal-State)
apply *simp+*
apply (*frule-tac*
?c1.0=f-Exec-Comp trans-fun (a # NoMsg^k - Suc 0) c and
xs = xs ↓ n ⊙_f k in f-Exec-Equal-State)
apply (*simp add: f-expand-not-empty-conv*)+
done
thm *f-Exec-Stream-Acc-LocalState--State-Idle-nth[no-vars]*

corollary *f-Exec-Stream-Acc-LocalState--State-Idle-eq[rule-format]:*
 $\llbracket 0 < l; l \leq k; \text{Exec-Equal-State localState trans-fun};$
 $\forall n < \text{length } xs. \text{State-Idle localState output-fun trans-fun (}$
 $\text{f-Exec-Comp-Stream-Acc-LocalState } l \text{ localState trans-fun } xs \text{ c ! } n) \rrbracket \implies$
 $\text{f-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun } xs \text{ c} =$
 $\text{f-Exec-Comp-Stream-Acc-LocalState } l \text{ localState trans-fun } xs \text{ c}$
apply (*clarsimp simp: list-eq-iff*)
apply (*rule f-Exec-Stream-Acc-LocalState--State-Idle-nth*)
apply *simp-all*
apply (*drule-tac x=n in spec*)
apply *simp*
done

lemma *i-Exec-Stream-Acc-LocalState--State-Idle-nth[rule-format]:*
 $\llbracket 0 < l; l \leq k; \text{Exec-Equal-State localState trans-fun};$
 $\forall n \leq i. \text{State-Idle localState output-fun trans-fun (}$
 $\text{i-Exec-Comp-Stream-Acc-LocalState } l \text{ localState trans-fun input c } n) \rrbracket \implies$
 $\text{i-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input c } i =$
 $\text{i-Exec-Comp-Stream-Acc-LocalState } l \text{ localState trans-fun input c } i$
apply (*simp only: f-Exec-Stream-Acc-LocalState-nth-eq-i-nth[of - - Suc i, symmetric]*)
apply (*rule f-Exec-Stream-Acc-LocalState--State-Idle-nth*)
apply *simp-all*
apply (*drule-tac x=n in spec*)
apply (*simp add: f-Exec-Stream-Acc-LocalState-nth-eq-i-nth*)
done

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
apply (clarsimp simp: ilist-eq-iff)
apply (rule i-Exec-Stream-Acc-LocalState--State-Idle-nth)
apply simp-all
apply (drule-tac x=n in spec)
apply simp
done

```

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
apply (drule order-le-less[THEN iffD1], erule disjE)
prefer 2
apply simp
apply (frule zero-less-diff[of k l, THEN iffD2])
apply (frule length-greater-imp-not-empty)
apply (simp add: f-Exec-Stream-Acc-Output-nth del: f-Exec-Stream-Cons)
apply (subst replicate-le-diff2[OF Suc-leI, symmetric])
  apply (simp del: f-Exec-Stream-Cons)+
apply (subst append-Cons[symmetric])
apply (case-tac i)
  apply (drule-tac x=0 in spec)
  apply (simp add: f-Exec-Stream-Acc-LocalState-nth take-first f-expand-one del:
last.simps f-Exec-Cons f-Exec-Stream-Cons append-Cons replicate.simps)
  apply (simp only: f-Exec-Stream-append map-append last-message-append)
  apply (rule if-P')
    thm last-message-NoMsg-conv
  apply (clarsimp simp: last-message-NoMsg-conv f-Exec-Stream-nth min-eqL simp
del: last.simps f-Exec-Comp.simps append-Cons replicate.simps)
    thm f-Exec-State-Idle-replicate-NoMsg-gr0-output
  apply (rule f-Exec-State-Idle-replicate-NoMsg-gr0-output)
  apply (simp del: last.simps f-Exec-Comp-Stream.simps append-Cons)+
apply (rename-tac n)
apply (simp only: f-Exec-Stream-append map-append last-message-append)
apply (subgoal-tac
  localState (f-Exec-Comp trans-fun (xs ↓ Suc n ⊙f k) c) =
  localState (f-Exec-Comp trans-fun (xs ↓ Suc n ⊙f l) c))
prefer 2
thm f-Exec-Stream-Acc-LocalState-nth[of k - xs, symmetric]
apply (simp add: f-Exec-Stream-Acc-LocalState-nth[symmetric])
apply (rule f-Exec-Stream-Acc-LocalState--State-Idle-nth)

```

```

apply simp+
apply (rename-tac n, drule-tac x=n in spec, simp)
apply simp
apply (rule if-P')
apply (simp add: last-message-NoMsg-conv f-Exec-Stream-nth min-eqL del: f-Exec-Comp.simps
replicate.simps)
apply (clarify, rename-tac j)
thm f-Exec-Stream-Acc-LocalState-snoc
apply (frule-tac x=Suc n in spec)
apply (simp only: f-Exec-Stream-Acc-LocalState-nth)
thm f-Exec-Equal-State
apply (rule-tac
  ?c1.0=f-Exec-Comp trans-fun (xs ↓ Suc n ⊙f l) c
  and ?c2.0=f-Exec-Comp trans-fun (xs ↓ Suc n ⊙f k) c
  in subst[OF f-Exec-Equal-State, rule-format])
apply (simp del: f-Exec-Comp.simps replicate.simps)+
apply (simp only: take-Suc-conv-app-nth f-expand-snoc f-Exec-append)
thm f-Exec-State-Idle-replicate-NoMsg-gr0-output
apply (rule f-Exec-State-Idle-replicate-NoMsg-gr0-output, assumption)
apply simp
apply (rule arg-cong[where f=λx. last-message (map output-fun x)])
thm f-Exec-Stream-Equal-State
apply (rule f-Exec-Stream-Equal-State, assumption+)
done

```

```

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
apply (clarsimp simp: list-eq-iff)
apply (rule f-Exec-Stream-Acc-Output--State-Idle-nth)
apply simp-all
apply (drule-tac x=n in spec)
apply simp
done

```

```

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
apply (simp only: i-Exec-Stream-Acc-Output-nth-f-nth)
apply (rule f-Exec-Stream-Acc-Output--State-Idle-nth)
apply simp-all
apply (drule-tac x=n in spec)
apply (simp add: f-Exec-Stream-Acc-LocalState-nth-eq-i-nth)

```

done

lemma *i-Exec-Stream-Acc-Output--State-Idle-eq*[rule-format]:

[[$0 < l; l \leq k$; *Exec-Equal-State* localState trans-fun;
 $\forall n. \text{State-Idle localState output-fun trans-fun (}$
 $\quad i\text{-Exec-Comp-Stream-Acc-LocalState } l \text{ localState trans-fun input } c \text{ } n) \text{]} \implies$
 $i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c =$
 $i\text{-Exec-Comp-Stream-Acc-Output } l \text{ output-fun trans-fun input } c$
apply (*clarsimp simp: ilist-eq-iff*)
apply (*rule i-Exec-Stream-Acc-Output--State-Idle-nth*)
apply *simp-all*
apply (*drule-tac x=n in spec*)
apply *simp*
done

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]:

$\bigwedge c \text{ xs. } \llbracket 0 < l; l \leq k;$
 $\text{State-Idle localState output-fun trans-fun (localState } c);$
 $\forall c \text{ m. State-Idle localState output-fun trans-fun (localState } c) \longrightarrow$
 $\text{State-Idle localState output-fun trans-fun (}$
 $\quad \text{localState (f-Exec-Comp trans-fun (m \# } \varepsilon^l - \text{Suc } 0) \text{ } c));$
 $i < \text{length xs} \rrbracket \implies$
 $\text{State-Idle localState output-fun trans-fun (}$
 $\quad \text{f-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun xs } c \text{ ! } i)$
apply (*frule length-greater-imp-not-empty*)
apply (*subgoal-tac*
 $\text{State-Idle localState output-fun trans-fun (}$
 $\quad \text{localState (f-Exec-Comp trans-fun (hd xs \# NoMsg}^k - \text{Suc } 0) \text{ } c))$)
prefer 2
apply (*drule-tac x=c in spec, drule-tac x=hd xs in spec*)
apply (*rule subst[OF replicate-le-diff2[OF Suc-leI], of 0 l k], assumption+*)
thm *f-Exec-State-Idle-replicate-NoMsg-state*
apply (*simp add: f-Exec-append f-Exec-State-Idle-replicate-NoMsg-state*)
apply (*induct i*)
apply (*simp add: f-Exec-Stream-Acc-LocalState-nth take-first hd-eq-first*)
apply (*drule-tac x=f-Exec-Comp trans-fun (hd xs \# NoMsg}^k - \text{Suc } 0) c in meta-spec*)
apply (*drule-tac x=tl xs in meta-spec*)
apply (*subgoal-tac i < length (tl xs) \wedge tl xs \neq [], elim conjE*)
prefer 2
apply (*simp add: length-greater-0-conv[symmetric] del: length-greater-0-conv*)
apply (*simp add: f-Exec-Stream-Acc-LocalState-nth*)
apply (*rule-tac n=Suc i in ssubst[OF take-Suc, rule-format], assumption*)
apply (*simp add: append-Cons[symmetric] f-Exec-append del: append-Cons*)
apply (*drule meta-mp*)
apply (*drule-tac x=f-Exec-Comp trans-fun (hd xs \# NoMsg}^k - \text{Suc } 0) c in spec*)
apply (*drule mp, simp*)

```

apply (drule-tac x=hd (tl xs) in spec)
apply (subst replicate-le-diff2[OF Suc-leI, of 0 l k, symmetric], simp+)
apply (simp add: f-Exec-append f-Exec-State-Idle-replicate-NoMsg-state)
thm f-Exec-Stream-Acc-LocalState-nth
apply (simp add: f-Exec-Stream-Acc-LocalState-nth)
done
thm f-Exec-Stream-Acc-LocalState--State-Idle-all[no-vars]

```

```

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 # εl - Suc 0) c)) |] ⟹
    State-Idle localState output-fun trans-fun (
      i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c i)
apply (simp only: i-Exec-Stream-Acc-LocalState-nth-f-nth)
apply (rule f-Exec-Stream-Acc-LocalState--State-Idle-all)
apply simp-all
apply (rename-tac c' m, drule-tac x=c' in spec)
apply simp
done

```

```

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 # εl - 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
apply (rule f-Exec-Stream-Acc-Output--State-Idle-eq, assumption+)
apply (simp add: f-Exec-Stream-Acc-LocalState--State-Idle-all)
done

```

```

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 # εl - 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
apply (rule i-Exec-Stream-Acc-Output--State-Idle-eq, assumption+)
apply (simp add: i-Exec-Stream-Acc-LocalState--State-Idle-all)
done

```

```

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);
     $\forall c m.$  State-Idle localState output-fun trans-fun (localState c)  $\longrightarrow$ 
      State-Idle localState output-fun trans-fun (
        localState (f-Exec-Comp trans-fun (m #  $\varepsilon^l$  - Suc 0) c))  $\parallel \Longrightarrow$ 
        f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c =
        f-Exec-Comp-Stream-Acc-LocalState l localState trans-fun xs c
apply (rule f-Exec-Stream-Acc-LocalState--State-Idle-eq, assumption+)
apply (rule f-Exec-Stream-Acc-LocalState--State-Idle-all)
apply simp+
done

```

```

lemma i-Exec-Stream-Acc-LocalState--State-Idle-all-imp-eq[rule-format]:
  [ 0 < l; l  $\leq$  k; Exec-Equal-State localState trans-fun;
    State-Idle localState output-fun trans-fun (localState c);
     $\forall c m.$  State-Idle localState output-fun trans-fun (localState c)  $\longrightarrow$ 
      State-Idle localState output-fun trans-fun (
        localState (f-Exec-Comp trans-fun (m #  $\varepsilon^l$  - Suc 0) c))  $\parallel \Longrightarrow$ 
        i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c =
        i-Exec-Comp-Stream-Acc-LocalState l localState trans-fun xs c
apply (rule i-Exec-Stream-Acc-LocalState--State-Idle-eq, assumption+)
apply (rule i-Exec-Stream-Acc-LocalState--State-Idle-all)
apply simp+
done

```

Converting inputs

```

lemma f-Exec-input-map:  $\bigwedge c.$ 
  f-Exec-Comp trans-fun (map f xs) c = f-Exec-Comp (trans-fun  $\circ$  f) xs c
by (induct xs, simp+)
lemma f-Exec-Stream-input-map:
  f-Exec-Comp-Stream trans-fun (map f xs) c =
  f-Exec-Comp-Stream (trans-fun  $\circ$  f) xs c
by (simp add: list-eq-iff f-Exec-Stream-nth take-map f-Exec-input-map)
lemma i-Exec-Stream-input-map:
  i-Exec-Comp-Stream trans-fun (f  $\circ$  input) c =
  i-Exec-Comp-Stream (trans-fun  $\circ$  f) input c
by (simp add: ilist-eq-iff i-Exec-Stream-nth f-Exec-input-map)

```

end

4 IL-AF-Stream: AutoFocus message streams and temporal logic on intervals

```

theory IL-AF-Stream
imports Main IL-TemporalOperators AF-Stream
begin

```

4.1 Stream views – joining streams and intervals

4.1.1 Basic definitions

The functions $f\text{-join}$ and $i\text{-join}$ deliver views of finite and infinite streams through intervals (more exactly: arbitrary natural sets). A stream view contains only the elements of the original stream at positions, which are contained in the interval. For instance, $f\text{-join } [0,10,20,30,40] \{1,4\} = [10,40]$

consts

$f\text{-join} :: 'a \text{ list} \Rightarrow iT \Rightarrow 'a \text{ list} \quad (\mathbf{infixl} \ \bowtie_f \ 100)$
 $i\text{-join} :: 'a \text{ ilist} \Rightarrow iT \Rightarrow 'a \text{ ilist} \quad (\mathbf{infixl} \ \bowtie_i \ 100)$

syntax (xsymbols)

$-f\text{-join} :: 'a \text{ list} \Rightarrow iT \Rightarrow 'a \text{ list} \quad (\mathbf{infixl} \ \bowtie \ 100)$
 $-i\text{-join} :: 'a \text{ ilist} \Rightarrow iT \Rightarrow 'a \text{ ilist} \quad (\mathbf{infixl} \ \bowtie \ 100)$

syntax (HTML output)

$-f\text{-join} :: 'a \text{ list} \Rightarrow iT \Rightarrow 'a \text{ list} \quad (\mathbf{infixl} \ \bowtie \ 100)$
 $-i\text{-join} :: 'a \text{ ilist} \Rightarrow iT \Rightarrow 'a \text{ ilist} \quad (\mathbf{infixl} \ \bowtie \ 100)$

translations

$-f\text{-join } xs \ n \equiv \mathbf{CONST} \ f\text{-join } xs \ n$
 $-i\text{-join } f \ n \equiv \mathbf{CONST} \ i\text{-join } f \ n$

The function $i\text{-f-join}$ can be used for the case, when an infinite stream is joined with a finite interval. The function $i\text{-join}$ would then deliver an infinite stream, whose elements after position $\mathit{card} \ I$ are equal to initial stream’s element at position $\mathit{Max} \ I$. The function $i\text{-f-join}$ in contrast cuts the resulting stream at this position and returns a finite stream.

consts

$i\text{-f-join} :: 'a \text{ ilist} \Rightarrow iT \Rightarrow 'a \text{ list} \quad (\mathbf{infixl} \ \bowtie_{i-f} \ 100)$

syntax (xsymbols)

$-i\text{-f-join} :: 'a \text{ ilist} \Rightarrow iT \Rightarrow 'a \text{ list} \quad (\mathbf{infixl} \ \bowtie \ 100)$

syntax (HTML output)

$-i\text{-f-join} :: 'a \text{ ilist} \Rightarrow iT \Rightarrow 'a \text{ list} \quad (\mathbf{infixl} \ \bowtie \ 100)$

translations

$-i\text{-f-join } f \ n \equiv \mathbf{CONST} \ i\text{-f-join } f \ n$

The function $i\text{-f-join}$ should be used only for finite sets in order to deliver well-defined results. The function $i\text{-join}$ should be used for infinite sets, because joining an infinite stream s and a finite set I using $i\text{-join}$ would deliver an infinite stream, ending with an infinite sequence of elements equal to s ($\mathit{Max} \ I$).

primrec

$f\text{-join-aux} :: 'a \text{ list} \Rightarrow \mathit{nat} \Rightarrow iT \Rightarrow 'a \text{ list}$

where

$f\text{-join-aux} \ [] \ n \ I = []$
 $| f\text{-join-aux} \ (x \ \# \ xs) \ n \ I =$
 $\quad (\mathit{if} \ n \in I \ \mathit{then} \ [x] \ \mathit{else} \ []) \ @ \ f\text{-join-aux} \ xs \ (\mathit{Suc} \ n) \ I$

defs

f-join-def : $xs \bowtie_f I \equiv f\text{-join-aux } xs \ 0 \ I$
i-join-def : $f \bowtie_i I \equiv \lambda n. (f (I \rightarrow n))$
i-f-join-def : $f \bowtie_{i-f} I \equiv f \Downarrow \text{Suc } (\text{Max } I) \bowtie_f I$

4.1.2 Basic results

lemma *f-join-aux-length*:

$\bigwedge n. \text{length } (f\text{-join-aux } xs \ n \ I) = \text{card } (I \cap \{n..<n + \text{length } xs\})$
apply (*induct xs, simp*)
apply (*simp add: atLeastLessThan-def*)
apply (*rule-tac t={n..} and s=insert n {Suc n..} in subst, fastsimp*)
apply *simp*
done

lemma *f-join-aux-nth*[*rule-format*]:

$\forall n \ i. \ i < \text{card } (I \cap \{n..<n + \text{length } xs\}) \longrightarrow$
 $(f\text{-join-aux } xs \ n \ I) ! i = xs ! (((I \cap \{n..<n + \text{length } xs\}) \rightarrow i) - n)$
apply (*induct xs, simp*)
apply (*clarsimp split del: split-if*)
apply (*subgoal-tac {n..<Suc (n + length xs)} = insert n {Suc n..<Suc (n + length xs)}*)
prefer 2
apply *fastsimp*
apply (*frule card-gr0-imp-not-empty[OF gr-implies-gr0]*)
apply (*case-tac n ∈ I*)
prefer 2
apply (*simp add: nth-Cons'*)
apply (*subgoal-tac Suc n ≤ (I ∩ {Suc n..<Suc (n + length xs)}) → i, simp*)
thm *order-trans[OF - iMin-le[OF inext-nth-closed]]*
apply (*rule order-trans[OF - iMin-le[OF inext-nth-closed]]*)
thm *order-trans[OF - iMin-Int-ge2]*
apply (*rule order-trans[OF - iMin-Int-ge2]*)
apply (*subgoal-tac n < n + length xs*)
prefer 2
apply (*rule ccontr, simp*)
apply (*simp add: iMin-atLeastLessThan*)
apply *assumption+*
apply *simp*
apply (*case-tac I ∩ {Suc n..<Suc (n + length xs)} = {}, simp*)
apply (*case-tac i*)
thm *iMin-insert*
apply (*simp add: iMin-insert*)
apply (*subgoal-tac Suc n ≤ iMin {Suc n..<Suc (n + length xs)}*)
prefer 2
apply (*subgoal-tac n < n + length xs*)
prefer 2
apply (*rule ccontr, simp*)

```

apply (simp add: iMin-atLeastLessThan)
apply (rename-tac i1)
apply (simp del: inext-nth.simps)
thm inext-nth-insert-Suc
apply (subst inext-nth-insert-Suc)
  apply simp
  apply (rule Suc-le-lessD)
  thm order-trans[OF - iMin-Int-ge2]
  apply (rule order-trans[OF - iMin-Int-ge2])
  apply assumption+
apply (simp add: nth-Cons^)
apply (subgoal-tac Suc n ≤ (I ∩ {Suc n..<Suc (n + length xs)}) → i1, simp)
  thm order-trans[OF - iMin-le[OF inext-nth-closed]]
apply (rule order-trans[OF - iMin-le[OF inext-nth-closed]])
  thm order-trans[OF - iMin-Int-ge2]
  apply (rule order-trans[OF - iMin-Int-ge2])
  apply assumption+
done

```

Joining finite streams and intervals

```

lemma f-join-length: length (xs ⋈f I) = card (I ↓< length xs)
by (simp add: f-join-def f-join-aux-length atLeast0LessThan cut-less-Int-conv)

```

```

lemma f-join-nth: n < length (xs ⋈f I) ⇒ (xs ⋈f I) ! n = xs ! (I → n)
apply (simp add: f-join-length)
apply (unfold f-join-def)
thm f-join-aux-nth[of n I 0 xs]
thm back-subst[OF - cut-less-Int-conv]
apply (drule back-subst[OF - cut-less-Int-conv])
apply (simp add: f-join-aux-nth atLeast0LessThan cut-less-Int-conv[symmetric]
  inext-nth-cut-less-eq)
done
lemma f-join-nth2: n < card (I ↓< length xs) ⇒ (xs ⋈f I) ! n = xs ! (I → n)
by (simp add: f-join-nth f-join-length)

```

```

lemma f-join-empty: xs ⋈f {} = []
by (simp add: length-0-conv[symmetric] f-join-length cut-less-empty del: length-0-conv)
lemma f-join-Nil: [] ⋈f I = []
by (simp add: length-0-conv[symmetric] f-join-length cut-less-0-empty del: length-0-conv)
lemma f-join-Nil-conv: (xs ⋈f I = []) = (I ↓< length xs = {})
by (simp add: length-0-conv[symmetric] f-join-length card-0-eq[OF nat-cut-less-finite]
  del: length-0-conv)
lemma f-join-Nil-conv': (xs ⋈f I = []) = (∀ i < length xs. i ∉ I)
by (fastsimp simp: f-join-Nil-conv)
lemma f-join-all-conv: (xs ⋈f I = xs) = ({..<length xs} ⊆ I)
apply (case-tac length xs = 0, simp add: f-join-Nil)
apply (rule iffI)

```

```

apply (rule subsetI, rename-tac t)
apply (clarsimp simp: list-eq-iff[of - xs] f-join-length)
apply (rule ccontr)
apply (subgoal-tac I  $\downarrow < \text{length } xs \subseteq \{..<\text{length } xs\}$ )
  prefer 2
  apply blast
thm psubset-card-mono[OF finite-lessThan]
apply (drule psubset-card-mono[OF finite-lessThan])
apply simp
apply (subgoal-tac length (xs  $\bowtie_f$  I) = length xs)
  prefer 2
  apply (simp add: f-join-length cut-less-Int-conv Int-absorb1)
apply (clarsimp simp: list-eq-iff[of - xs] f-join-nth)
apply (rule arg-cong[where f=op ! xs])
apply (subgoal-tac I  $\downarrow < \text{length } xs = \{..<\text{length } xs\}$ )
  prefer 2
  apply fastsimp
thm inext-nth-cut-less-eq
apply (subst inext-nth-cut-less-eq[where t=length xs, symmetric], simp)
apply (simp add: inext-nth-lessThan)
done

```

```

lemma f-join-all:  $\{..<\text{length } xs\} \subseteq I \implies xs \bowtie_f I = xs$ 
by (rule f-join-all-conv[THEN iffD2])
corollary f-join-UNIV:  $xs \bowtie_f UNIV = xs$ 
by (simp add: f-join-all)

```

```

lemma f-join-union:
   $\llbracket \text{finite } A; \text{Max } A < i\text{Min } B \rrbracket \implies xs \bowtie_f (A \cup B) = xs \bowtie_f A @ (xs \bowtie_f B)$ 
apply (case-tac A = {}, simp add: f-join-empty)
apply (case-tac B = {}, simp add: f-join-empty)
apply (frule Max-less-iMin-imp-disjoint, assumption)
apply (simp add: list-eq-iff f-join-length cut-less-Un del: Max-less-iff)
apply (subgoal-tac A  $\downarrow < \text{length } xs \cap B \downarrow < \text{length } xs = \{\}$ )
  prefer 2
  apply (simp add: cut-less-Int[symmetric] cut-less-empty)
apply (frule card-Un-disjoint[OF nat-cut-less-finite nat-cut-less-finite])
apply (clarsimp simp del: Max-less-iff)
apply (subst f-join-nth)
  apply (simp add: f-join-length cut-less-Un)
apply (simp add: nth-append f-join-length del: Max-less-iff, intro conjI impI)
apply (simp add: f-join-nth f-join-length del: Max-less-iff)
thm inext-nth-card-append-eq1
  apply (rule ssubst[OF inext-nth-card-append-eq1], assumption)
  apply (rule order-less-le-trans[OF - cut-less-card], assumption+)
apply simp
apply (subst f-join-nth)
  apply (simp add: f-join-length)
apply (subgoal-tac iMin B < length xs)

```

```

prefer 2
apply (rule ccontr)
apply (simp add: linorder-not-less cut-less-Min-empty)
apply (frule order-less-trans, assumption)
apply (rule arg-cong[where  $f = \lambda x. xs ! x$ ])
apply (simp add: cut-less-Max-all inext-nth-card-append-eq2)
done

```

```

lemma f-join-singleton-if:
   $xs \bowtie_f \{n\} = (\text{if } n < \text{length } xs \text{ then } [xs ! n] \text{ else } [])$ 
apply (clar simp simp: list-eq-iff f-join-length cut-less-singleton)
apply (insert f-join-nth[of 0 xs {n}])
apply (simp add: f-join-length cut-less-singleton)
done

```

```

lemma f-join-insert:
   $n < \text{length } xs \implies$ 
   $xs \bowtie_f \text{insert } n I = xs \bowtie_f (I \downarrow < n) @ (xs ! n) \# (xs \bowtie_f (I \downarrow > n))$ 
apply (rule-tac  $t = \text{insert } n I$  and  $s = (I \downarrow < n) \cup \{n\} \cup (I \downarrow > n)$  in subst, fastsimp)
apply (insert nat-cut-less-finite[of I n])
apply (case-tac  $I \downarrow > n = \{\}$ )
apply (simp add: f-join-empty del: Un-insert-right)
apply (case-tac  $I \downarrow < n = \{\}$ )
apply (simp add: f-join-empty f-join-singleton-if)
apply (subgoal-tac  $\text{Max } (I \downarrow < n) < i\text{Min } \{n\}$ )
prefer 2
apply (simp add: cut-less-mem-iff)
apply (simp add: f-join-union f-join-singleton-if del: Un-insert-right)
apply (subgoal-tac  $\text{Max } \{n\} < i\text{Min } (I \downarrow > n)$ )
prefer 2
apply (simp add: iMin-gr-iff cut-greater-mem-iff)
apply (case-tac  $I \downarrow < n = \{\}$ )
apply (simp add: f-join-empty f-join-union f-join-singleton-if del: Un-insert-left)
apply (subgoal-tac  $\text{Max } (I \downarrow < n) < i\text{Min } \{n\}$ )
prefer 2
apply (simp add: cut-less-mem-iff)
apply (subgoal-tac  $\text{Max } (I \downarrow < n \cup \{n\}) < i\text{Min } (I \downarrow > n)$ )
prefer 2
apply (simp add: iMin-gr-iff i-cut-mem-iff)
apply (simp add: f-join-union f-join-singleton-if del: Un-insert-right)
done

```

```

lemma f-join-snoc:
   $(xs @ [x]) \bowtie_f I =$ 
   $xs \bowtie_f I @ (\text{if } \text{length } xs \in I \text{ then } [x] \text{ else } [])$ 
apply (simp add: list-eq-iff f-join-length)
apply (subgoal-tac
   $\text{card } (I \downarrow < \text{Suc } (\text{length } xs)) =$ 
   $\text{card } (I \downarrow < \text{length } xs) + (\text{if } \text{length } xs \in I \text{ then } \text{Suc } 0 \text{ else } 0)$ )

```

```

prefer 2
apply (simp add: nat-cut-le-less-conv[symmetric] cut-le-less-conv-if)
apply (simp add: card-insert-if[OF nat-cut-less-finite] cut-less-mem-iff)
apply simp
apply (case-tac length xs ∈ I)
apply (clarsimp simp: f-join-length)
apply (simp add: nth-append f-join-length, intro conjI impI)
apply (subst f-join-nth[of - xs @ [x]])
apply (simp add: f-join-length)
apply (simp add: nth-append less-card-cut-less-imp-inext-nth-less)
apply (simp add: f-join-nth f-join-length)
apply (simp add: linorder-not-less less-Suc-eq-le)
apply (subst f-join-nth)
apply (simp add: f-join-length)
apply (subgoal-tac I → i = length xs)
prefer 2
apply (rule-tac t=length xs and s=Max (I ↓< Suc (length xs)) in subst)
apply (rule Max-equality[OF - nat-cut-less-finite])
apply (simp add: cut-less-mem-iff)+
apply (subst inext-nth-cut-less-eq[of - - Suc (length xs), symmetric], simp)
apply (rule inext-nth-card-Max[OF nat-cut-less-finite])
apply (simp add: card-gr0-imp-not-empty)
apply simp+
apply (simp add: f-join-nth f-join-length)
apply (simp add: nth-append less-card-cut-less-imp-inext-nth-less)
done

```

lemma f-join-append:

```

(xs @ ys) ⋈f I = xs ⋈f I @ ys ⋈f (I ⊕ - (length xs))
apply (induct ys rule: rev-induct)
apply (simp add: f-join-Nil)
apply (simp add: append-assoc[symmetric] f-join-snoc del: append-assoc)
apply (simp add: iT-Plus-neg-mem-iff add-commute[of length xs])
done

```

lemma take-f-join-eq1:

```

n < card (I ↓< length xs) ⇒
(xs ⋈f I) ↓ n = xs ⋈f (I ↓< (I → n))
thm less-card-cut-less-imp-inext-nth-less
apply (frule less-card-cut-less-imp-inext-nth-less)
apply (simp add: list-eq-iff f-join-length cut-cut-less min-eqR)
apply (subgoal-tac n < card I ∨ infinite I)
prefer 2
apply (case-tac finite I)
apply (drule order-less-le-trans[OF - cut-less-card], simp+)
thm cut-less-inext-nth-card-eq1
apply (simp add: min-eqL cut-less-inext-nth-card-eq1)
apply clarify

```

```

thm f-join-nth
apply (subst f-join-nth)
apply (simp add: f-join-length)
apply (subst f-join-nth)
apply (simp add: f-join-length cut-cut-less min-eqL)
thm cut-less-inext-nth-card-eq1
apply (simp add: cut-less-inext-nth-card-eq1)
thm inext-nth-cut-less-eq
apply (simp add: cut-less-inext-nth-card-eq1 inext-nth-cut-less-eq)
done

```

lemma *take-f-join-eq2*:

$$\text{card } (I \downarrow < \text{length } xs) \leq n \implies (xs \bowtie_f I) \downarrow n = xs \bowtie_f I$$

by (*simp add: f-join-length*)

lemma *take-f-join-if*:

$$(xs \bowtie_f I) \downarrow n =$$

$$(if\ n < \text{card } (I \downarrow < \text{length } xs)\ \text{then } xs \bowtie_f (I \downarrow < (I \rightarrow n))\ \text{else } xs \bowtie_f I)$$

by (*simp add: take-f-join-eq1 take-f-join-eq2*)

lemma *drop-f-join-eq1*:

$$n < \text{card } (I \downarrow < \text{length } xs) \implies$$

$$(xs \bowtie_f I) \uparrow n = xs \bowtie_f (I \downarrow \geq (I \rightarrow n))$$

apply (*case-tac I = {}*)

apply (*simp add: cut-less-empty*)

apply (*case-tac I \downarrow < \text{length } xs = {}*)

apply (*simp add: cut-less-empty*)

apply (*rule same-append-eq[THEN iffD1, of xs \bowtie_f I \downarrow n]*)

First, a simplification step without *take-f-join-eq1* required for correct transformation, in order to eliminate $(xs \bowtie_f I) \downarrow n$ in the equation.

apply *simp*

Now, *take-f-join-eq1* can be applied

apply (*simp add: take-f-join-eq1*)

apply (*case-tac I \downarrow < (I \rightarrow n) = {}*)

apply (*simp add: f-join-empty*)

apply (*rule-tac t = I \rightarrow n and s = iMin I in subst*)

apply (*rule ccontr*)

apply (*drule neq-le-trans[of iMin I]*)

apply (*simp add: iMin-le[OF inext-nth-closed]*)

apply (*simp add: cut-less-Min-not-empty*)

apply (*simp add: cut-ge-Min-all*)

apply (*subst f-join-union[OF nat-cut-less-finite, symmetric]*)

apply (*subgoal-tac I \downarrow \geq (I \rightarrow n) \neq {}*)

prefer 2

apply (*simp add: cut-ge-not-empty-iff*)

apply (*blast intro: inext-nth-closed*)

apply (*simp add: nat-cut-less-finite i-cut-mem-iff iMin-gr-iff*)

apply (*simp add: cut-less-cut-ge-ident*)

done

lemma *drop-f-join-eq2*:

$\text{card } (I \downarrow < \text{length } xs) \leq n \implies (xs \bowtie_f I) \uparrow n = []$

by (*simp add: f-join-length*)

lemma *drop-f-join-if*:

$(xs \bowtie_f I) \uparrow n =$

$(\text{if } n < \text{card } (I \downarrow < \text{length } xs) \text{ then } xs \bowtie_f (I \downarrow \geq (I \rightarrow n)) \text{ else } [])$

by (*simp add: drop-f-join-eq1 drop-f-join-eq2*)

lemma *f-join-take*: $xs \downarrow n \bowtie_f I = xs \bowtie_f (I \downarrow < n)$

apply (*clarsimp simp: list-eq-iff f-join-length cut-cut-less min-max.inf commute*)

apply (*simp add: f-join-nth f-join-length cut-cut-less min-max.inf commute*)

apply (*case-tac n < length xs*)

apply (*simp add: min-eqL inext-nth-cut-less-eq*)

thm *less-card-cut-less-imp-inext-nth-less*

apply (*simp add: less-card-cut-less-imp-inext-nth-less*)

apply (*simp add: min-eqR linorder-not-less*)

thm *inext-nth-cut-less-eq*

apply (*subst inext-nth-cut-less-eq*)

thm *card-mono[OF nat-cut-less-finite cut-less-mono]*

apply (*rule order-less-le-trans, assumption*)

apply (*rule card-mono[OF nat-cut-less-finite cut-less-mono], assumption*)

apply *simp*

done

lemma *f-join-drop*: $xs \uparrow n \bowtie_f I = xs \bowtie_f (I \oplus n)$

apply (*case-tac length xs \leq n*)

apply (*simp add: f-join-Nil*)

apply (*rule sym*)

apply (*simp add: f-join-Nil-conv' iT-Plus-mem-iff*)

apply (*rule subst[OF append-take-drop-id, of $\lambda x. xs \uparrow n \bowtie_f I = x \bowtie_f (I \oplus n)$*)

apply (*simp only: f-join-append*)

apply (*simp add: f-join-take min-eqR*)

apply (*simp add: iT-Plus-Plus-neg-inverse*)

apply (*rule-tac $t=(I \oplus n) \downarrow < n$ and $s=\{\}$ in subst*)

apply (*rule sym*)

apply (*simp add: cut-less-empty-iff iT-Plus-mem-iff*)

apply (*simp add: f-join-empty*)

done

lemma *cut-less-eq-imp-f-join-eq*:

$A \downarrow < \text{length } xs = B \downarrow < \text{length } xs \implies xs \bowtie_f A = xs \bowtie_f B$

apply (*clarsimp simp: list-eq-iff f-join-length f-join-nth*)

apply (*rule subst[OF inext-nth-cut-less-eq, of - A length xs], simp*)

apply (*rule subst[OF inext-nth-cut-less-eq, of - B length xs], simp*)

apply *simp*

done

corollary *f-join-cut-less-eq*:

$length\ xs \leq t \implies xs \bowtie_f (I \downarrow < t) = xs \bowtie_f I$
apply (rule cut-less-eq-imp-f-join-eq)
apply (simp add: cut-cut-less min-eqR)
done

lemma take-Suc-Max-eq-imp-f-join-eq:

$\llbracket finite\ I; xs \downarrow Suc\ (Max\ I) = ys \downarrow Suc\ (Max\ I) \rrbracket \implies$
 $xs \bowtie_f I = ys \bowtie_f I$
apply (case-tac $I = \{\}$)
apply (simp add: f-join-empty)
apply (simp add: list-eq-iff f-join-length)
apply (case-tac $length\ xs < Suc\ (Max\ I)$)
apply (case-tac $length\ ys < Suc\ (Max\ I)$)
apply (clarsimp simp: min-eqL, rename-tac i)
thm f-join-nth2
apply (simp add: f-join-nth2)
apply (drule-tac $x=I \rightarrow i$ in spec)
apply (subgoal-tac $I \rightarrow i < length\ ys$)
prefer 2
apply (rule less-card-cut-less-imp-inext-nth-less, simp)
apply simp
apply (simp add: min-eq)
apply (case-tac $length\ ys < Suc\ (Max\ I)$)
apply (simp add: min-eq)
apply (simp add: linorder-not-less min-eqR Suc-le-eq del: Max-less-iff)
apply (subgoal-tac $I \downarrow < length\ xs = I \downarrow < length\ ys$)
prefer 2
apply (simp add: cut-less-Max-all)
apply (clarsimp simp: f-join-nth2 simp del: Max-less-iff, rename-tac i)
apply (drule-tac $x=I \rightarrow i$ in spec)
apply (subgoal-tac $I \rightarrow i < Suc\ (Max\ I)$)
prefer 2
apply (simp add: less-Suc-eq-le inext-nth-closed)
apply (simp del: Max-less-iff)
done
corollary f-join-take-Suc-Max-eq:
 $finite\ I \implies xs \downarrow Suc\ (Max\ I) \bowtie_f I = xs \bowtie_f I$
by (rule take-Suc-Max-eq-imp-f-join-eq, simp+)

Joining infinite streams and infinite intervals

lemma i-join-nth: $(f \bowtie_i I) n = f (I \rightarrow n)$
by (simp add: i-join-def)
lemma i-join-UNIV: $f \bowtie_i UNIV = f$
by (simp add: ilist-eq-iff i-join-nth inext-nth-UNIV)

thm f-join-union

lemma i-join-union:

$\llbracket finite\ A; Max\ A < iMin\ B; B \neq \{\} \rrbracket \implies$
 $f \bowtie_i (A \cup B) = (f \downarrow Suc\ (Max\ A) \bowtie_f A) \frown (f \bowtie_i B)$

```

apply (case-tac  $A = \{\}$ )
  apply (simp add: f-join-empty)
apply (simp (no-asm) add: ilist-eq-iff, clarify)
apply (simp add: i-join-nth i-append-nth f-join-length del: Max-less-iff)
apply (subgoal-tac  $A \downarrow < \text{Suc } (Max A) = A$ )
  prefer 2
  apply (simp add: nat-cut-le-less-conv[symmetric] cut-le-Max-all)
apply (simp del: Max-less-iff, intro conjI impI)
apply (simp add: inext-nth-card-append-eq1)
thm f-join-nth f-join-length i-take-length
apply (simp add: f-join-nth f-join-length)
thm less-card-cut-less-imp-inext-nth-less
apply (simp add: less-card-cut-less-imp-inext-nth-less)
apply (simp add: inext-nth-card-append-eq2)
done

lemma i-join-singleton:  $f \bowtie_i \{a\} = (\lambda n. f a)$ 
by (simp add: ilist-eq-iff i-join-nth inext-nth-singleton)

lemma i-join-insert:
   $f \bowtie_i (\text{insert } n I) =$ 
   $(f \downarrow n) \bowtie_f (I \downarrow < n) \frown [f n] \frown ($ 
     $\text{if } I \downarrow > n = \{\} \text{ then } (\lambda x. f x) \text{ else } f \bowtie_i (I \downarrow > n))$ 
thm ssubst[OF insert-eq-cut-less-cut-greater]
apply (rule ssubst[OF insert-eq-cut-less-cut-greater])
apply (case-tac  $I \downarrow < n = \{\}$ )
  apply (simp add: f-join-empty, intro conjI impI)
  apply (simp add: i-join-singleton ilist-eq-iff i-append-nth)
apply (subgoal-tac  $Max \{n\} < iMin (I \downarrow > n)$ )
  prefer 2
  apply (simp add: cut-greater-Min-greater)
apply simp
thm insert-is-Un
apply (subst insert-is-Un)
thm i-join-union[OF singleton-finite]
apply (subst i-join-union[OF singleton-finite])
apply (simp add: f-join-singleton-if)+
apply (intro conjI impI)
apply (subgoal-tac  $Max (I \downarrow < n) < iMin \{n\}$ )
  prefer 2
  apply (simp add: nat-cut-less-Max-less)
apply (rule-tac  $t = \text{insert } n (I \downarrow < n)$  and  $s = (I \downarrow < n) \cup \{n\}$  in subst, simp)
thm i-join-union[OF nat-cut-less-finite - singleton-not-empty]
apply (subst i-join-union[OF nat-cut-less-finite - singleton-not-empty], simp)
apply (simp add: i-join-singleton)
apply (rule-tac  $s = \lambda x. f x$  and  $t = [f n] \frown (\lambda x. f x)$  in subst)
  apply (simp add: ilist-eq-iff i-append-nth)
apply (subst i-append-assoc[symmetric])
apply (rule-tac  $t = [f n] \frown (\lambda x. f x)$  and  $s = (\lambda x. f x)$  in subst)

```

```

apply (simp add: ilist-eq-iff i-append-nth)
apply (rule arg-cong)
apply (simp add: take-Suc-Max-eq-imp-f-join-eq[OF nat-cut-less-finite] min-eqR)
apply (subgoal-tac Max (I ↓< n) < iMin {n} ∧ Max {n} < iMin (I ↓> n), elim
conjE)
prefer 2
apply (simp add: cut-greater-Min-greater nat-cut-less-Max-less)
apply (rule-tac t=insert n (I ↓< n ∪ I ↓> n) and s=(I ↓< n ∪ ({n} ∪ I ↓>
n)) in subst, simp)
apply (subgoal-tac ({n} ∪ I ↓> n) ≠ {} ∧ Max (I ↓< n) < iMin ({n} ∪ I ↓>
n), elim conjE)
prefer 2
apply (simp add: iMin-insert)
apply (simp add: i-join-union nat-cut-less-finite singleton-finite del: Un-insert-left
Un-insert-right Max-less-iff)
apply (simp add: f-join-singleton-if)
apply (rule arg-cong)
apply (simp add: take-Suc-Max-eq-imp-f-join-eq[OF nat-cut-less-finite] min-eqR)
done

```

lemma *i-join-i-append*:

```

infinite I ⇒ (xs ∩ f) ⋈i I = (xs ⋈f I) ∩ (f ⋈i (I ⊕- length xs))
apply (clarsimp simp: ilist-eq-iff)
apply (simp add: i-join-nth i-append-nth f-join-length)
apply (subgoal-tac I ↓≥ length xs ≠ {})
prefer 2
apply (fastsimp simp: cut-ge-not-empty-iff infinite-nat-iff-unbounded-le)
apply (simp add: inext-nth-less-less-card-conv)
apply (intro conjI impI)
apply (simp add: f-join-nth f-join-length)
apply (subgoal-tac I ⊕- length xs ≠ {})
prefer 2
apply (simp add: iT-Plus-neg-empty-iff infinite-imp-nonempty)
apply (simp add: iT-Plus-neg-inext-nth)
apply (case-tac I ↓< length xs = {})
thm cut-less-empty-iff[THEN iffD1, THEN cut-ge-all-iff[THEN iffD2]]
apply (frule cut-less-empty-iff[THEN iffD1, THEN cut-ge-all-iff[THEN iffD2]])
apply simp
apply (rule subst[OF inext-nth-card-append-eq2, OF nat-cut-less-finite], simp+)
apply (simp add: less-imp-Max-less-iMin[OF nat-cut-less-finite] i-cut-mem-iff)
apply simp
apply (simp add: cut-less-cut-ge-ident)
done

```

lemma *i-take-i-join*: infinite I ⇒ f ⋈_i I ↓ n = f ↓ (I → n) ⋈_f I

```

apply (clarsimp simp: list-eq-iff f-join-length cut-less-inext-nth-card-eq1, rename-tac
i)
apply (simp add: i-join-nth)
apply (frule inext-nth-mono2-infin[THEN iffD2], assumption)

```

```

apply (rule-tac t=f (I → i) and s=f ↓ (I → n) ! (I → i) in subst, simp)
apply (rule sym, rule f-join-nth)
apply (simp add: f-join-length)
apply (simp add: inext-nth-less-less-card-conv[OF nat-cut-ge-infinite-not-empty])
done

```

```

lemma i-drop-i-join: I ≠ {} ⇒ f ⋈i I ↑ n = f ⋈i (I ↓≥ (I → n))
apply (simp (no-asm) add: ilist-eq-iff)
apply (simp add: i-join-nth inext-nth-cut-ge-inext-nth)
done

```

```

lemma i-join-i-take: f ↓ n ⋈f I = f ⋈i I ↓ card (I ↓< n)
apply (clarisimp simp: list-eq-iff f-join-length)
apply (frule less-card-cut-less-imp-inext-nth-less)
apply (simp add: i-join-nth f-join-length f-join-nth)
done

```

```

lemma i-join-i-drop: I ≠ {} ⇒ f ↑ n ⋈i I = f ⋈i (I ⊕ n)
apply (simp (no-asm) add: ilist-eq-iff)
apply (simp add: i-join-nth iT-Plus-inext-nth add-commute[of - n])
done

```

```

lemma i-join-finite-nth-ge-card-eq-nth-Max:
  [ finite I; I ≠ {} ; card I ≤ Suc n ] ⇒ (f ⋈i I) n = f (Max I)
by (simp add: i-join-nth inext-nth-card-Max)
lemma i-join-finite-i-drop-card-eq-const-nth-Max:
  [ finite I; I ≠ {} ] ⇒ (f ⋈i I) ↑ (card I) = (λn. f (Max I))
by (simp add: ilist-eq-iff i-join-finite-nth-ge-card-eq-nth-Max)

```

```

lemma i-join-finite-i-append-nth-Max-conv:
  [ finite I; I ≠ {} ] ⇒ (f ⋈i I) = f ↓ Suc (Max I) ⋈f I ∩ (λn. f (Max I))
apply (simp (no-asm) add: ilist-eq-iff, clarify)
apply (subgoal-tac I ↓< (Suc (Max I)) = I)
prefer 2
apply (simp add: nat-cut-le-less-conv[symmetric] cut-le-Max-all)
apply (simp add: i-append-nth i-join-nth f-join-length)
apply (intro conjI impI)
thm f-join-nth f-join-length i-take-length
apply (simp add: f-join-nth f-join-length)
apply (rule sym, rule i-take-nth)
apply (simp add: less-card-cut-less-imp-inext-nth-less)
apply (simp add: inext-nth-card-Max)
done

```

Joining infinite streams and finite intervals

```

lemma i-f-join-length: finite I ⇒ length (f ⋈i-f I) = card I

```

apply (*simp add: i-f-join-def f-join-length*)
apply (*simp add: nat-cut-le-less-conv[symmetric] cut-le-Max-all*)
done

lemma *i-f-join-nth*: $n < \text{card } I \implies f \bowtie_{i-f} I ! n = f (I \rightarrow n)$
apply (*frule card-gr0-imp-finite[OF gr-implies-gr0]*)
apply (*frule card-gr0-imp-not-empty[OF gr-implies-gr0]*)
apply (*simp add: i-f-join-def*)
thm *i-take-nth*[*of* $I \rightarrow n \text{ Suc } (\text{Max } I) f$]
apply (*subst i-take-nth*[*of* $I \rightarrow n \text{ Suc } (\text{Max } I) f$, *symmetric*])
apply (*rule le-imp-less-Suc*)
apply (*simp add: Max-ge[OF - inext-nth-closed]*)
apply (*simp add: f-join-nth2 nat-cut-le-less-conv[symmetric] cut-le-Max-all*)
done

lemma *i-f-join-empty*: $f \bowtie_{i-f} \{\} = []$
by (*simp add: i-f-join-def f-join-empty*)

lemma *i-f-join-eq-i-join-i-take*:
 $\text{finite } I \implies f \bowtie_{i-f} I = f \bowtie_i I \Downarrow (\text{card } I)$
apply (*simp add: i-f-join-def*)
apply (*simp add: i-join-i-take nat-cut-le-less-conv[symmetric] cut-le-Max-all*)
done

lemma *i-f-join-union*:
 $\llbracket \text{finite } A; \text{finite } B; \text{Max } A < i\text{Min } B \rrbracket \implies$
 $f \bowtie_{i-f} (A \cup B) = f \bowtie_{i-f} A @ f \bowtie_{i-f} B$
apply (*case-tac A = \{\}*, *simp add: i-f-join-empty*)
apply (*case-tac B = \{\}*, *simp add: i-f-join-empty*)
apply (*simp add: i-f-join-def f-join-union del: Max-less-iff*)
apply (*subgoal-tac Max A < Max B*)
prefer 2
apply (*rule order-less-le-trans[OF - iMin-le-Max], assumption+*)
apply (*simp add: Max-Un max-eqR[OF less-imp-le]*)
apply (*rule take-Suc-Max-eq-imp-f-join-eq, assumption*)
apply (*simp add: min-eqR[OF less-imp-le]*)
done

lemma *i-f-join-singleton*: $f \bowtie_{i-f} \{n\} = [f n]$
by (*simp add: i-f-join-def f-join-singleton-if*)

thm *f-join-insert*

lemma *i-f-join-insert*:
 $\text{finite } I \implies$
 $f \bowtie_{i-f} \text{insert } n I = f \bowtie_{i-f} (I \Downarrow < n) @ f n \# f \bowtie_{i-f} (I \Downarrow > n)$
apply (*case-tac I = \{\}*)
apply (*simp add: i-f-join-singleton i-cut-empty i-f-join-empty*)
apply (*simp add: i-f-join-def*)

```

apply (simp add: f-join-insert)
apply (frule cut-greater-finite[of - n])
apply (case-tac I ↓> n = {})
  apply (simp add: f-join-empty)
  apply (case-tac I ↓< n = {})
    apply (simp add: f-join-empty)
apply (rule take-Suc-Max-eq-imp-f-join-eq[OF nat-cut-less-finite])
apply simp
apply (rule arg-cong[where f=λx. f ↓ x])
apply simp
apply (rule min-eqR, rule min-max.le-supI1, rule less-imp-le)
thm nat-cut-less-Max-less
apply (simp add: nat-cut-less-Max-less)
apply (simp add: cut-greater-Max-eq)
apply (subgoal-tac n < Max I)
  prefer 2
  apply (rule ccontr)
  apply (simp add: linorder-not-less cut-greater-Max-empty)
apply (simp add: max-eqR[OF less-imp-le])
apply (case-tac I ↓< n = {})
  apply (simp add: f-join-empty)
apply (rule take-Suc-Max-eq-imp-f-join-eq[OF nat-cut-less-finite])
apply simp
apply (rule arg-cong[where f=λx. f ↓ x])
apply simp
apply (rule min-eqR)
apply (blast intro: Max-subset)
done

```

lemma take-i-f-join-eq1:

```

  n < card I ⇒ f ⋈i-f I ↓ n = f ⋈i-f (I ↓< (I → n))
apply (frule card-ge-0-finite[OF gr-implies-gr0])
apply (case-tac I = {})
  apply (simp add: cut-less-empty i-f-join-empty)
apply (subgoal-tac n < card (I ↓< Suc (Max I)))
  prefer 2
  apply (simp add: cut-less-Max-all)
apply (simp add: i-f-join-def take-f-join-eq1)
apply (case-tac I ↓< (I → n) = {})
  apply (simp add: f-join-empty)
apply (rule take-Suc-Max-eq-imp-f-join-eq[OF nat-cut-less-finite])
apply simp
apply (rule arg-cong[where f=λx. f ↓ x])
apply simp
apply (rule min-eqR)
apply (rule order-trans[OF less-imp-le[OF cut-less-Max-less]])
apply (simp add: nat-cut-less-finite inext-nth-closed)+
done

```

lemma *take-i-f-join-eq2*:

$\llbracket \text{finite } I; \text{card } I \leq n \rrbracket \implies f \bowtie_{i-f} I \downarrow n = f \bowtie_{i-f} I$
apply (*case-tac* $I = \{\}$)
apply (*simp add: cut-less-empty i-f-join-empty*)
apply (*simp add: i-f-join-def take-f-join-eq2 cut-less-Max-all*)
done

lemma *take-i-f-join-if*:

finite $I \implies$
 $f \bowtie_{i-f} I \downarrow n = (\text{if } n < \text{card } I \text{ then } f \bowtie_{i-f} (I \downarrow < (I \rightarrow n)) \text{ else } f \bowtie_{i-f} I)$
by (*simp add: take-i-f-join-eq1 take-i-f-join-eq2*)

lemma *drop-i-f-join-eq1*:

$n < \text{card } I \implies f \bowtie_{i-f} I \uparrow n = f \bowtie_{i-f} (I \downarrow \geq (I \rightarrow n))$
apply (*frule card-ge-0-finite[OF gr-implies-gr0]*)
apply (*case-tac* $I = \{\}$)
apply (*simp add: cut-ge-empty i-f-join-empty*)
apply (*subgoal-tac* $n < \text{card } (I \downarrow < \text{Suc } (\text{Max } I))$)
prefer 2
apply (*simp add: cut-less-Max-all*)
apply (*simp add: i-f-join-def drop-f-join-eq1*)
apply (*subgoal-tac* $I \downarrow \geq (I \rightarrow n) \neq \{\}$)
prefer 2
apply (*rule in-imp-not-empty[of* $I \rightarrow n$ *]*)
apply (*simp add: cut-ge-mem-iff inext-nth-closed*)
apply (*rule take-Suc-Max-eq-imp-f-join-eq*)
apply (*rule cut-ge-finite, assumption*)
apply *simp*
apply (*rule arg-cong[where* $f = \lambda x. f \downarrow x$ *]*)
apply (*simp add: min-eqR cut-ge-Max-eq*)
done

lemma *drop-i-f-join-eq2*:

$\llbracket \text{finite } I; \text{card } I \leq n \rrbracket \implies f \bowtie_{i-f} I \uparrow n = []$
by (*simp add: i-f-join-length*)

lemma *drop-i-f-join-if*:

finite $I \implies$
 $f \bowtie_{i-f} I \uparrow n = (\text{if } n < \text{card } I \text{ then } f \bowtie_{i-f} (I \downarrow \geq (I \rightarrow n)) \text{ else } [])$
by (*simp add: drop-i-f-join-eq1 drop-i-f-join-eq2*)

lemma *i-f-join-i-drop*:

finite $I \implies f \uparrow n \bowtie_{i-f} I = f \bowtie_{i-f} (I \oplus n)$
apply (*case-tac* $I = \{\}$)
apply (*simp add: iT-Plus-empty i-f-join-empty*)
apply (*simp add: i-f-join-def iT-Plus-Max*)
apply (*simp add: i-take-i-drop f-join-drop*)
done

lemma *i-take-Suc-Max-eq-imp-i-f-join-eq*:
 $f \Downarrow \text{Suc } (\text{Max } I) = g \Downarrow \text{Suc } (\text{Max } I) \implies f \bowtie_{i-f} I = g \bowtie_{i-f} I$
by (*simp add: i-f-join-def*)

lemma *i-take-i-join-eq-i-f-join*:
 $\text{infinite } I \implies f \bowtie_i I \Downarrow n = f \bowtie_{i-f} (I \Downarrow < (I \rightarrow n))$
apply (*frule infinite-imp-nonempty*)
apply (*case-tac n = 0*)
apply (*simp add: cut-less-Min-empty i-f-join-empty*)
thm *inext-nth-gr-Min-conv-infinite*
apply (*frule inext-nth-gr-Min-conv-infinite[THEN iffD2], simp*)
apply (*simp add: i-take-i-join i-f-join-def*)
apply (*subgoal-tac Suc (Max (I \Downarrow < (I \rightarrow n))) \leq I \rightarrow n*)
prefer 2
apply (*rule Suc-leI*)
apply (*rule nat-cut-less-Max-less*)
apply (*simp add: cut-less-Min-not-empty*)
apply (*simp add: f-join-cut-less-eq*)
thm *i-join-i-take*
apply (*simp add: i-join-i-take*)
apply (*rule arg-cong[where f=\lambda x. f \bowtie_i I \Downarrow card x]*)
apply (*clarsimp simp: gr0-conv-Suc*)
thm *cut-le-less-inext-conv[OF inext-nth-closed]*
apply (*simp add: cut-le-less-inext-conv[OF inext-nth-closed, symmetric]*)
apply (*simp add: nat-cut-le-less-conv[symmetric]*)
apply (*rule arg-cong[where f=\lambda x. I \Downarrow \leq x]*)
apply (*rule sym, rule Max-equality[OF - nat-cut-le-finite]*)
apply (*simp add: cut-le-mem-iff inext-nth-closed*)
done

4.1.3 Results for intervals from *IL-Interval*

lemma *f-join-iFROM*: $xs \bowtie_f [n..] = xs \uparrow n$
apply (*clarsimp simp: list-eq-iff f-join-length iFROM-cut-less iIN-card Suc-diff-Suc*)
thm *f-join-nth2*
apply (*subst f-join-nth2*)
apply (*simp add: iFROM-cut-less iIN-card*)
apply (*simp add: iFROM-inext-nth*)
done

lemma *i-join-iFROM*: $f \bowtie_i [n..] = f \uparrow n$
by (*simp add: ilist-eq-iff i-join-nth iFROM-inext-nth*)

lemma *f-join-iIN*: $xs \bowtie_f [n..,d] = xs \uparrow n \Downarrow \text{Suc } d$
apply (*simp add: list-eq-iff f-join-length iIN-cut-less iIN-card Suc-diff-Suc min-eq*)
thm *f-join-nth2*
apply (*simp add: f-join-nth2 iIN-cut-less iIN-card iIN-inext-nth*)
done
lemma *i-f-join-iIN*: $f \bowtie_{i-f} [n..,d] = f \uparrow n \Downarrow \text{Suc } d$

by (simp add: i-f-join-def f-join-iIN iIN-Max i-take-drop)

lemma f-join-iTILL: $xs \bowtie_f [\dots n] = xs \downarrow (Suc\ n)$

by (simp add: iIN-0-iTILL-conv[symmetric] f-join-iIN)

lemma i-f-join-iTILL: $f \bowtie_{i-f} [\dots n] = f \downarrow Suc\ n$

by (simp add: iIN-0-iTILL-conv[symmetric] i-f-join-iIN)

lemma f-join-f-expand-iT-Mult:

$0 < k \implies xs \odot_f k \bowtie_f (I \otimes k) = xs \bowtie_f I$

apply (case-tac I = {})

 apply (simp add: iT-Mult-empty f-join-empty)

 apply (simp add: list-eq-iff f-join-length)

 apply (clarsimp simp: iT-Mult-cut-less2 iT-Mult-card)

 apply (simp add: f-join-nth2 iT-Mult-cut-less2 iT-Mult-card)

 apply (drule less-card-cut-less-imp-inext-nth-less)

 apply (simp add: iT-Mult-inext-nth f-expand-nth-mult)

done

lemma i-join-i-expand-iT-Mult:

$\llbracket 0 < k; I \neq \{\} \rrbracket \implies f \odot_i k \bowtie_i (I \otimes k) = f \bowtie_i I$

apply (simp (no-asm) add: ilist-eq-iff, clarify)

apply (simp add: i-join-nth iT-Mult-inext-nth i-expand-nth-mult)

done

lemma i-f-join-i-expand-iT-Mult:

$\llbracket 0 < k; \text{finite } I \rrbracket \implies f \odot_i k \bowtie_{i-f} (I \otimes k) = f \bowtie_{i-f} I$

apply (case-tac I = {})

 apply (simp add: iT-Mult-empty i-f-join-empty)

 apply (clarsimp simp: list-eq-iff i-f-join-length iT-Mult-finite-iff iT-Mult-not-empty iT-Mult-card)

 apply (simp add: i-f-join-nth iT-Mult-card iT-Mult-inext-nth i-expand-nth-mult)

done

lemma f-join-f-shrink-iT-Plus-iT-Div-mod:

$\llbracket 0 < k; \forall x \in I. x \bmod k = 0 \rrbracket \implies$

$(xs \mapsto_f k) \bowtie_f (I \oplus (k - 1)) = xs \div_f k \bowtie_f (I \odot k)$

apply (case-tac I = {})

 apply (simp add: iT-Plus-empty iT-Div-empty f-join-empty)

 apply (simp add: list-eq-iff f-join-length f-shrink-length)

 apply (subgoal-tac Suc (length xs) - k \leq length xs - length xs mod k)

 prefer 2

 apply (case-tac length xs < k, simp)

 apply (simp add: Suc-diff-le linorder-not-less)

 apply (rule Suc-leI)

 thm diff-less-mono2

 apply (rule diff-less-mono2, simp)

 apply (rule order-less-le-trans[OF mod-less-divisor], assumption+)

```

apply (rule context-conjI)
apply (simp add: iT-Plus-cut-less iT-Div-cut-less2 iT-Plus-card)
thm iT-Div-card-inj-on
apply (subst iT-Div-card-inj-on)
thm mod-eq-imp-div-right-inj-on
apply (rule mod-eq-imp-div-right-inj-on)
apply clarsimp+
apply (rule arg-cong[where f=card])
apply (simp (no-asm-simp) add: set-eq-iff cut-less-mem-iff, clarify)
apply (rule conj-cong, simp)
apply (rule iffI)
apply simp
apply (frule-tac x=x and m=k in less-mod-eq-imp-add-divisor-le)
apply (simp add: mod-diff-right-eq[symmetric])
apply simp
apply (clarsimp simp: f-join-nth f-join-length f-shrink-length)
apply (simp add: iT-Plus-inext-nth iT-Plus-not-empty)
thm iT-Div-mod-inext-nth
apply (simp add: iT-Div-mod-inext-nth)
thm f-shrink-nth-eq-f-last-message-hold-nth
apply (subst f-shrink-nth-eq-f-last-message-hold-nth)
apply (drule sym, simp, thin-tac card ?x = card ?y)
apply (simp add: iT-Plus-cut-less iT-Plus-card)
apply (rule less-mult-imp-div-less)
thm less-card-cut-less-imp-inext-nth-less
thm less-le-trans[OF less-card-cut-less-imp-inext-nth-less]
apply (rule less-le-trans[OF less-card-cut-less-imp-inext-nth-less], assumption)
apply (simp add: div-mult-cancel)
apply (simp add: div-mult-cancel inext-nth-closed)
done

```

lemma *i-join-i-shrink-iT-Plus-iT-Div-mod*:

```

[[ 0 < k; I ≠ {}; ∀ x∈I. x mod k = 0 ]] ⇒
(f ↦i k) ⋈i (I ⊕ (k - 1)) = f ÷i k ⋈i (I ⊙ k)
apply (simp (no-asm) add: ilit-eq-iff, clarify)
apply (simp add: i-join-nth)
apply (simp add: i-shrink-nth-eq-i-last-message-hold-nth)
apply (simp add: iT-Plus-inext-nth iT-Div-mod-inext-nth)
apply (drule-tac x=I → x in bspec)
apply (simp add: inext-nth-closed)
apply (simp add: mod-0-div-mult-cancel)
done

```

lemma *i-f-join-i-shrink-iT-Plus-iT-Div-mod*:

```

[[ 0 < k; finite I; ∀ x∈I. x mod k = 0 ]] ⇒
(f ↦i k) ⋈i-f (I ⊕ (k - 1)) = f ÷i k ⋈i-f (I ⊙ k)
apply (case-tac I = {})
apply (simp add: iT-Plus-empty iT-Div-empty i-f-join-empty)
apply (simp add: i-f-join-def iT-Plus-Max iT-Div-Max)

```

apply (*simp add: i-last-message-hold-i-take*[*symmetric*] *i-shrink-i-take-mult*[*symmetric*])
apply (*simp add: add-commute*[*of k*])
apply (*simp add: mod-0-div-mult-cancel*[*THEN iffD1*])
apply (*simp add: f-join-f-shrink-iT-Plus-iT-Div-mod*[*unfolded One-nat-def*])
done

corollary *f-join-f-shrink-iT-Plus-iT-Div-mod-subst*:

$\llbracket 0 < k; \forall x \in I. x \bmod k = 0; \\ A = I \oplus (k - 1); B = I \odot k \rrbracket \implies$

$(xs \mapsto_f k) \bowtie_f A = xs \div_f k \bowtie_f B$

by (*simp add: f-join-f-shrink-iT-Plus-iT-Div-mod*[*unfolded One-nat-def*])

corollary *i-join-i-shrink-iT-Plus-iT-Div-mod-subst*:

$\llbracket 0 < k; I \neq \{\}; \forall x \in I. x \bmod k = 0; \\ A = I \oplus (k - 1); B = I \odot k \rrbracket \implies$

$(f \mapsto_i k) \bowtie_i A = f \div_i k \bowtie_i B$

by (*simp add: i-join-i-shrink-iT-Plus-iT-Div-mod*[*unfolded One-nat-def*])

corollary *i-f-join-i-shrink-iT-Plus-iT-Div-mod-subst*:

$\llbracket 0 < k; \text{finite } I; \forall x \in I. x \bmod k = 0; \\ A = I \oplus (k - 1); B = I \odot k \rrbracket \implies$

$(f \mapsto_i k) \bowtie_{i-f} A = f \div_i k \bowtie_{i-f} B$

by (*simp add: i-f-join-i-shrink-iT-Plus-iT-Div-mod*[*unfolded One-nat-def*])

lemma *f-join-f-shrink-iT-Div-mod*:

$\llbracket 0 < k; \forall x \in I. x \bmod k = k - 1 \rrbracket \implies$

$(xs \mapsto_f k) \bowtie_f I = xs \div_f k \bowtie_f (I \odot k)$

apply (*case-tac* $I = \{\}$)

apply (*simp add: iT-Div-empty f-join-empty*)

apply (*frule Suc-leI, drule order-le-imp-less-or-eq, erule disjE*)

prefer 2

apply (*drule sym, simp add: iT-Div-1*)

apply (*rule-tac* $t=I$ **and** $s=I \oplus - (k - 1) \oplus (k - 1)$ **in** *subst*)

apply (*rule iT-Plus-neg-Plus-le-inverse*)

apply (*rule ccontr*)

apply (*drule-tac* $x=iMin I$ **in** *bspec, simp add: iMinI-ex2*)

apply (*simp add: iMinI-ex2*) $+$

apply (*subgoal-tac* $\bigwedge x. x + k - Suc\ 0 \in I \implies x \bmod k = 0$)

prefer 2

thm *mod-add-eq-imp-mod-0*[*THEN iffD1*]

apply (*rule mod-add-eq-imp-mod-0*[*THEN iffD1, of k - Suc 0*])

apply (*simp add: add-commute*[*of k*])

apply (*subst iT-Plus-Div-distrib-mod-less*)

apply (*clarsimp simp: iT-Plus-neg-mem-iff*)

apply (*simp add: iT-Plus-0*)

thm *f-join-f-shrink-iT-Plus-iT-Div-mod*[*unfolded One-nat-def*]

apply (*rule f-join-f-shrink-iT-Plus-iT-Div-mod*[*unfolded One-nat-def*], *simp*)

apply (*simp add: iT-Plus-neg-mem-iff*)

done

lemma *i-join-i-shrink-iT-Div-mod*:
 $\llbracket 0 < k; I \neq \{\}; \forall x \in I. x \bmod k = k - 1 \rrbracket \implies$
 $(f \mapsto_i k) \bowtie_i I = f \dot{\div}_i k \bowtie_i (I \odot k)$
apply (*simp* (*no-asm*) *add: ilist-eq-iff, clarify*)
apply (*simp* *add: i-join-nth*)
apply (*simp* *add: i-shrink-nth-eq-i-last-message-hold-nth*)
thm *iT-Div-mod-inext-nth*
apply (*simp* *add: iT-Div-mod-inext-nth*)
apply (*drule-tac* $x=I \rightarrow x$ **in** *bspec*)
apply (*rule* *inext-nth-closed, assumption*)
apply (*simp* *add: div-mult-cancel*)
apply (*subgoal-tac* $k - \text{Suc } 0 \leq I \rightarrow x$)
prefer 2
apply (*rule* *order-trans*[*OF - mod-le-dividend*[**where** $n=k$]])
apply *simp*
apply *simp*
done

lemma *i-f-join-i-shrink-iT-Div-mod*:
 $\llbracket 0 < k; \text{finite } I; \forall x \in I. x \bmod k = k - 1 \rrbracket \implies$
 $(f \mapsto_i k) \bowtie_{i-f} I = f \dot{\div}_i k \bowtie_{i-f} (I \odot k)$
apply (*case-tac* $I = \{\}$)
apply (*simp* *add: iT-Plus-empty iT-Div-empty i-f-join-empty*)
apply (*simp* *add: i-f-join-def*)
apply (*simp* *add: iT-Div-Max*)
apply (*simp* *add: i-last-message-hold-i-take*[*symmetric*] *i-shrink-i-take-mult*[*symmetric*] *add-commute*[*of k*])
apply (*simp* *add: div-mult-cancel*)
apply (*subgoal-tac* $k - \text{Suc } 0 \leq \text{Max } I$)
prefer 2
apply (*rule* *order-trans*[*OF - mod-le-dividend*[**where** $n=k$]])
apply *simp*
apply (*simp* *add: f-join-f-shrink-iT-Div-mod*)
done

lemma *f-join-f-expand-iMOD*:
 $0 < k \implies xs \odot_f k \bowtie_f [n * k, \text{mod } k] = xs \bowtie_f [n \dots]$
by (*subst* *iFROM-mult*[*symmetric*], *rule* *f-join-f-expand-iT-Mult*)
corollary *f-join-f-expand-iMOD-0*:
 $0 < k \implies xs \odot_f k \bowtie_f [0, \text{mod } k] = xs$
apply (*drule* *f-join-f-expand-iMOD*[*of k xs 0*])
apply (*simp* *add: iFROM-0 f-join-UNIV*)
done

lemma *f-join-f-expand-iMODb*:
 $0 < k \implies xs \odot_f k \bowtie_f [n * k, \text{mod } k, d] = xs \bowtie_f [n \dots, d]$
by (*subst* *iIN-mult*[*symmetric*], *rule* *f-join-f-expand-iT-Mult*)
corollary *f-join-f-expand-iMODb-0*:

$0 < k \implies xs \odot_f k \bowtie_f [0, \text{mod } k, n] = xs \bowtie_f [\dots n]$
apply (*drule f-join-f-expand-iMODb*[of k xs 0 n])
apply (*simp add: iIN-0-iTILL-conv*)
done

lemma *i-join-i-expand-iMOD*:

$0 < k \implies f \odot_i k \bowtie_i [n * k, \text{mod } k] = f \bowtie_i [n \dots]$
by (*subst iFROM-mult[symmetric]*, *rule i-join-i-expand-iT-Mult*[*OF - iFROM-not-empty*])
corollary *i-join-i-expand-iMOD-0*:
 $0 < k \implies f \odot_i k \bowtie_i [0, \text{mod } k] = f$
apply (*drule i-join-i-expand-iMOD*[of k f 0])
apply (*simp add: iFROM-0 i-join-UNIV*)
done

lemma *i-join-i-expand-iMODb*:

$0 < k \implies f \odot_i k \bowtie_i [n * k, \text{mod } k, d] = f \bowtie_i [n \dots, d]$
by (*subst iIN-mult[symmetric]*, *rule i-join-i-expand-iT-Mult*[*OF - iIN-not-empty*])
corollary *i-join-i-expand-iMODb-0*:
 $0 < k \implies f \odot_i k \bowtie_i [0, \text{mod } k, n] = f \bowtie_i [\dots n]$
apply (*drule i-join-i-expand-iMODb*[of k f 0 n])
apply (*simp add: iIN-0-iTILL-conv*)
done

lemma *i-f-join-i-expand-iMODb*:

$0 < k \implies f \odot_i k \bowtie_{i-f} [n * k, \text{mod } k, d] = f \bowtie_{i-f} [n \dots, d]$
by (*subst iIN-mult[symmetric]*, *rule i-f-join-i-expand-iT-Mult*[*OF - iIN-finite*])
corollary *i-f-join-i-expand-iMODb-0*:
 $0 < k \implies f \odot_i k \bowtie_{i-f} [0, \text{mod } k, n] = f \bowtie_{i-f} [\dots n]$
apply (*drule i-f-join-i-expand-iMODb*[of k f 0 n])
apply (*simp add: iIN-0-iTILL-conv*)
done

lemma *f-join-f-shrink-iMOD*:

$0 < k \implies (xs \mapsto_f k) \bowtie_f [n * k + (k - 1), \text{mod } k] = xs \div_f k \bowtie_f [n \dots]$
apply (*rule f-join-f-shrink-iT-Plus-iT-Div-mod-subst*[**where** $I=[n * k, \text{mod } k]$])
apply (*simp add: iMOD-iff iMOD-add iMOD-div-ge*)
done

corollary *f-join-f-shrink-iMOD-0*:

$0 < k \implies (xs \mapsto_f k) \bowtie_f [k - 1, \text{mod } k] = xs \div_f k$
apply (*frule f-join-f-shrink-iMOD*[of k xs 0])
apply (*simp add: iFROM-0 f-join-UNIV*)
done

lemma *f-join-f-shrink-iMODb*:

$0 < k \implies (xs \mapsto_f k) \bowtie_f [n * k + (k - 1), \text{mod } k, d] = xs \div_f k \bowtie_f [n \dots, d]$
apply (*rule f-join-f-shrink-iT-Plus-iT-Div-mod-subst*[**where** $I=[n * k, \text{mod } k, d]$])
apply (*simp add: iMODb-iff iMODb-add iMODb-div-ge*)
done

done

corollary *f-join-f-shrink-iMODb-0*:

$$0 < k \implies (xs \mapsto_f k) \bowtie_f [k - 1, \text{mod } k, n] = xs \div_f k \bowtie_f [\dots n]$$

apply (*frule f-join-f-shrink-iMODb[of k xs 0 n]*)

apply (*simp add: iIN-0-iTILL-conv*)

done

lemma *i-join-i-shrink-iMOD*:

$$0 < k \implies (f \mapsto_i k) \bowtie_i [n * k + (k - 1), \text{mod } k] = f \div_i k \bowtie_i [n \dots]$$

apply (*rule i-join-i-shrink-iT-Plus-iT-Div-mod-subst[where I=[n * k, mod k]]*)

apply (*simp add: iMOD-not-empty iMOD-iff iMOD-add iMOD-div-ge*)+

done

corollary *i-join-i-shrink-iMOD-0*:

$$0 < k \implies (f \mapsto_i k) \bowtie_i [k - 1, \text{mod } k] = f \div_i k$$

apply (*frule i-join-i-shrink-iMOD[of k f 0]*)

apply (*simp add: iFROM-0 i-join-UNIV*)

done

lemma *i-f-join-i-shrink-iMODb*:

$$0 < k \implies (f \mapsto_i k) \bowtie_{i-f} [n * k + (k - 1), \text{mod } k, d] = f \div_i k \bowtie_{i-f} [n \dots, d]$$

apply (*rule i-f-join-i-shrink-iT-Plus-iT-Div-mod-subst[where I=[n * k, mod k, d]]*)

apply (*simp add: iMODb-finite iMODb-iff iMODb-add iMODb-div-ge*)+

done

corollary *i-f-join-i-shrink-iMODb-0*:

$$0 < k \implies (f \mapsto_i k) \bowtie_{i-f} [k - 1, \text{mod } k, n] = f \div_i k \bowtie_{i-f} [\dots n]$$

apply (*frule i-f-join-i-shrink-iMODb[of k f 0 n]*)

apply (*simp add: iIN-0-iTILL-conv i-join-UNIV*)

done

4.2 Streams and temporal operators

thm *last-message-NoMsg-conv*

lemma *i-shrink-eq-NoMsg-iAll-conv*:

$$0 < k \implies ((s \div_i k) t = \varepsilon) = (\Box t1 [t * k \dots, k - \text{Suc } 0]. s t1 = \varepsilon)$$

apply (*simp add: i-shrink-nth last-message-NoMsg-conv iAll-def Ball-def iIN-iff*)

apply (*rule iffI*)

apply (*clarify, rename-tac i*)

apply (*drule-tac x=i - t * k in spec*)

apply *simp*

apply (*clarify, rename-tac i*)

apply (*drule-tac x=t * k + i in spec*)

apply *simp*

done

lemma *i-shrink-eq-NoMsg-iAll-conv2*:

$$0 < k \implies ((s \div_i k) t = \varepsilon) = (\Box t1 [\dots k - 1] \oplus (t * k). s t1 = \varepsilon)$$

by (*simp add: iT-add i-shrink-eq-NoMsg-iAll-conv*)

thm *last-message-conv*

lemma *i-shrink-eq-Msg-iEx-iAll-conv*:

$\llbracket 0 < k; m \neq \varepsilon \rrbracket \implies$
 $((s \dot{\div}_i k) t = m) =$
 $(\diamond t1 [t * k \dots, k - \text{Suc } 0]. s t1 = m \wedge$
 $(\square t2 [\text{Suc } t1 \dots]. t2 \leq t * k + k - \text{Suc } 0 \longrightarrow s t2 = \varepsilon))$
apply (*simp add: i-shrink-nth last-message-conv*)
apply (*simp add: iAll-def iEx-def Ball-def Bex-def iIN-iff iFROM-iff*)
apply (*rule iffI*)
apply (*clarsimp, rename-tac i*)
apply (*rule-tac x=t * k + i in exI*)
apply (*simp add: diff-add-assoc less-imp-le-pred del: add-diff-assoc*)
apply (*clarsimp, rename-tac j*)
apply (*drule-tac x=j - t * k in spec*)
apply *simp*
apply (*clarsimp, rename-tac i*)
apply (*rule-tac x=i - t * k in exI*)
apply *simp*
done
lemma *i-shrink-eq-Msg-iEx-iAll-conv2*:
 $\llbracket 0 < k; m \neq \varepsilon \rrbracket \implies$
 $((s \dot{\div}_i k) t = m) =$
 $(\diamond t1 [\dots k - 1] \oplus (t * k). s t1 = m \wedge$
 $(\square t2 [1 \dots] \oplus t1 . t2 \leq t * k + k - 1 \longrightarrow s t2 = \varepsilon))$
by (*simp add: iT-add i-shrink-eq-Msg-iEx-iAll-conv*)

lemma *i-shrink-eq-Msg-iEx-iAll-cut-greater-conv*:

$\llbracket 0 < k; m \neq \varepsilon \rrbracket \implies$
 $((s \dot{\div}_i k) t = m) =$
 $(\diamond t1 [t * k \dots, k - \text{Suc } 0]. s t1 = m \wedge$
 $(\square t2 [t * k \dots, k - \text{Suc } 0] \downarrow > t1. s t2 = \varepsilon))$
apply (*simp add: i-shrink-eq-Msg-iEx-iAll-conv*)
apply (*simp add: iIN-cut-greater iEx-def*)
apply (*rule bex-cong2[OF subset-refl]*)
apply (*force simp: iAll-def Ball-def iT-iff*)
done

lemma *i-shrink-eq-Msg-iEx-iAll-cut-greater-conv2*:

$\llbracket 0 < k; m \neq \varepsilon \rrbracket \implies$
 $((s \dot{\div}_i k) t = m) =$
 $(\diamond t1 [\dots k - 1] \oplus (t * k). s t1 = m \wedge$
 $(\square t2 ([\dots k - 1] \oplus (t * k)) \downarrow > t1. s t2 = \varepsilon))$
by (*simp add: iT-add i-shrink-eq-Msg-iEx-iAll-cut-greater-conv*)

lemma *i-shrink-eq-Msg-iSince-conv*:

$\llbracket 0 < k; m \neq \varepsilon \rrbracket \implies$
 $((s \dot{\div}_i k) t = m) =$
 $(s t2 = \varepsilon. t2 \mathcal{S} t1 [t * k \dots, k - \text{Suc } 0]. s t1 = m)$
by (*simp add: iSince-def iIN-cut-greater i-shrink-eq-Msg-iEx-iAll-cut-greater-conv*)

lemma *i-shrink-eq-Msg-iSince-conv2*:

$\llbracket 0 < k; m \neq \varepsilon \rrbracket \implies$
 $((s \dot{\div}_i k) t = m) =$

($s \ t2 = \varepsilon. t2 \ \mathcal{S} \ t1 \ [..k - 1] \oplus (t * k). s \ t1 = m$)
by (*simp add: iT-add i-shrink-eq-Msg-iSince-conv*)

lemma *iT-Mult-iAll-i-expand-nth-iff*:

$0 < k \implies (\Box t (I \otimes k). P ((f \odot_i k) t)) = (\Box t I. P (f t))$

apply (*rule iffI*)

apply *clarify*

apply (*drule-tac t=t * k in ispec*)

apply (*simp add: iT-Mult-mem-iff2*)

apply (*simp add: i-expand-nth-mult*)

apply (*fastsimp simp: iT-Mult-mem-iff mult-commute[of k] i-expand-nth-mod-eq-0*)

done

Streams and temporal operators cycle start/finish events

lemma *i-shrink-eq-NoMsg-iAll-start-event-conv*:

$\llbracket 0 < k; \bigwedge t. \text{event } t = (t \bmod k = 0); t0 = t * k \rrbracket \implies$

$((s \div_i k) t = \varepsilon) =$

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

apply (*case-tac k = Suc 0*)

apply (*simp add: iT-add iT-not-empty iNext-True*)

apply (*drule neq-le-trans[OF not-sym], simp*)

apply (*simp add: i-shrink-eq-NoMsg-iAll-conv iTL-defs Ball-def Bex-def iT-add iT-iff iFROM-cut-less iFROM-inert*)

apply (*rule iffI*)

apply *simp*

apply (*rule-tac x=t * k + k in exI*)

apply *fastsimp*

apply (*clarify, rename-tac x1 x2*)

apply (*case-tac x2 = Suc (t * k)*)

apply (*simp add: mod-Suc*)

apply (*clarsimp, rename-tac q*)

apply (*drule-tac y=x1 in order-le-imp-less-or-eq, erule disjE*)

prefer 2

apply *simp*

apply (*drule-tac x=x1 in spec*)

apply (*simp add: mult-commute[of k]*)

apply (*drule Suc-le-lessD*)

thm *less-mod-eq-imp-add-divisor-le*

apply (*drule-tac y=q * k and m=k in less-mod-eq-imp-add-divisor-le, simp*)

apply *simp*

done

thm *i-shrink-eq-Msg-iSince-conv*

lemma *i-shrink-eq-Msg-iUntil-start-event-conv*:

$\llbracket 0 < k; m \neq \varepsilon; \bigwedge t. \text{event } t = (t \bmod k = 0); t0 = t * k \rrbracket \implies$

$((s \div_i k) t = m) =$

$(s \ t0 = m \wedge (\bigcirc t' t0 \ [0..]). (s \ t1 = \varepsilon. t1 \ \mathcal{U} \ t2 \ ([0..] \oplus t'). \text{event } t2))) \vee$

```

( $\bigcirc t' t0 [0\dots]. (\neg \text{event } t1. t1 \mathcal{U} t2 ([0\dots] \oplus t'). ($ 
   $s t2 = m \wedge \neg \text{event } t2 \wedge (\bigcirc t'' t2 [0\dots].$ 
     $(s t3 = \varepsilon. t3 \mathcal{U} t4 ([0\dots] \oplus t''). \text{event } t4))))))$ 
apply (case-tac  $k = \text{Suc } 0$ )
apply (simp add: iT-add iT-not-empty iNext-iff)
apply (drule neq-le-trans[OF not-sym], simp)
thm i-shrink-eq-Msg-iSince-conv
apply (simp add: i-shrink-eq-Msg-iSince-conv iTL-defs iT-add iT-cut-greater iT-cut-less
Ball-def Bex-def iT-iff iFROM-inext)
apply (rule-tac  $t = \text{Suc } (t * k + k - 2)$  and  $s = t * k + k - \text{Suc } 0$  in subst, simp)
apply (rule iffI)
apply (elim exE conjE, rename-tac i)
apply (case-tac  $i = t * k$ )
apply (rule disjI1)
apply simp
apply (rule-tac  $x = t * k + k$  in exI)
apply force
apply (rule disjI2)
apply (rule-tac  $x = i$  in exI)
apply (case-tac  $i = \text{Suc } (t * k)$ )
apply simp
apply (case-tac  $\text{Suc } (t * k) < t * k + k - \text{Suc } 0$ )
apply (clarsimp simp: mod-Suc)
apply (case-tac  $k = \text{Suc } (\text{Suc } 0)$ , simp)
apply simp
apply (rule-tac  $x = t * k + k$  in exI)
apply force
apply clarsimp
apply (subgoal-tac  $k = \text{Suc } (\text{Suc } 0)$ )
prefer 2
apply simp
apply (simp add: mod-Suc)
apply (simp add: nat-mult-2-right[symmetric] numeral-2-eq-2 del: mult-Suc-right)
apply (rule-tac  $x = t * k + k$  in exI)
apply simp
apply simp
apply (subgoal-tac  $\text{Suc } (t * k) \leq i$ )
prefer 2
apply (rule ccontr, simp)
apply simp
apply (case-tac  $i < t * k + k - \text{Suc } 0$ )
apply clarsimp
apply (subgoal-tac  $0 < i \bmod k$ )
prefer 2
thm between-imp-mod-gr0[OF Suc-le-lessD]
apply (simp add: mult-commute[of t])
apply (rule between-imp-mod-gr0[OF Suc-le-lessD], simp+)
apply (rule conjI)
apply (rule-tac  $x = t * k + k$  in exI)

```

```

apply force
apply clarify
apply (simp add: mult-commute[of t])
apply (rule between-imp-mod-gr0[OF Suc-le-lessD], assumption)
apply simp
apply clarsimp
apply (subgoal-tac Suc (Suc 0) < k)
prefer 2
apply simp
thm mod-0-imp-mod-pred
apply (simp add: mod-0-imp-mod-pred)
apply (rule conjI, blast)
apply clarify
apply (simp add: mult-commute[of t])
apply (rule between-imp-mod-gr0[OF Suc-le-lessD], assumption)
apply simp
apply (simp add: mod-Suc)
apply (erule disjE)
apply (clarsimp simp: mult-commute[of k], rename-tac i)
apply (subgoal-tac t < i)
prefer 2
apply (rule ccontr)
apply (simp add: linorder-not-less)
apply (drule-tac i=i and k=k in mult-le-mono1)
apply simp
apply (rule-tac x=t * k in exI)
apply simp
apply (subgoal-tac t * k < t * k + k - Suc 0)
prefer 2
apply simp
apply (clarsimp, rename-tac j)
apply (drule-tac x=j in spec)
apply (simp add: numeral-2-eq-2 Suc-diff-Suc)
apply (drule mp)
apply (rule order-trans, assumption)
apply (drule-tac m=t and n=i in Suc-leI)
apply (drule mult-le-mono1[of Suc t- k])
apply simp
apply simp
apply (clarsimp, rename-tac i)
apply (case-tac i = Suc (t * k))
apply (clarsimp, rename-tac i1)
apply (rule-tac x=Suc (t * k) in exI)
apply simp
apply (case-tac k = Suc (Suc 0), simp)
apply (clarsimp simp: mult-commute[of k], rename-tac q)
apply (subgoal-tac Suc (t * k) < t * k + k - Suc 0)
prefer 2
apply simp

```

```

apply (clarsimp, rename-tac j)
apply (drule-tac x=j in spec)
apply (simp add: numeral-3-eq-3 Suc-diff-Suc)
apply (subgoal-tac t * k + k ≤ q * k)
prefer 2
apply (rule less-mod-eq-imp-add-divisor-le)
apply (rule Suc-le-lessD, simp)
apply simp
apply simp
apply (clarsimp, rename-tac i1)
apply (rule-tac x=i in exI)
apply (simp add: numeral-2-eq-2 Suc-diff-Suc)
apply (case-tac i1 = Suc i)
apply simp
apply (case-tac Suc (i mod k) = k)
apply simp
apply (subgoal-tac i ≤ t * k + k - Suc 0)
prefer 2
apply (rule ccontr)
apply (drule-tac x=t * k + k in spec)
apply (simp add: linorder-not-le)
apply (drule pred-less-imp-le)+
apply clarsimp
apply simp
thm le-imp-less-or-eq
apply (drule-tac x=i in le-imp-less-or-eq, erule disjE)
apply simp
thm between-imp-mod-between[of k - Suc (Suc 0) k t Suc 0 i]
apply (cut-tac b=k - Suc (Suc 0) and m=k and k=t and a=Suc 0 and n=i)
in between-imp-mod-between)
apply (simp add: mult-commute[of k])+
apply clarsimp+
apply (rename-tac q)
apply (simp add: mult-commute[of k])
apply (subgoal-tac Suc t ≤ q)
prefer 2
apply (rule Suc-leI)
thm mult-less-cancel2[THEN iffD1, THEN conjunct2]
apply (rule mult-less-cancel2[where k=k, THEN iffD1, THEN conjunct2])
apply (rule Suc-le-lessD)
apply simp
apply (frule mult-le-mono1[of Suc t - k])
apply (simp add: add-commute[of k])
apply (intro conjI impI allI)
apply force
apply (simp add: linorder-not-less)
apply (case-tac i > t * k + k)
apply (drule-tac x=t * k + k in spec)
apply simp

```

```

apply (case-tac  $i = t * k + k$ , simp)
apply simp
done

lemma i-shrink-eq-NoMsg-iAll-finish-event-conv:
   $\llbracket 1 < k; \bigwedge t. \text{event } t = (t \bmod k = k - 1); t0 = t * k \rrbracket \implies$ 
   $((s \div_i k) t = \varepsilon) =$ 
   $(s t0 = \varepsilon \wedge (\bigcirc t' t0 [0\dots]. (s t1 = \varepsilon. t1 \mathcal{U} t2 ([0\dots] \oplus t'). (\text{event } t2 \wedge s t2 =$ 
   $\varepsilon))))$ 
apply (simp add: i-shrink-eq-NoMsg-iAll-conv iT-add)
apply (unfold iTL-defs Ball-def Bex-def)
apply (simp add: iT-iff div-mult-cancel iFROM-cut-less iFROM-inext)
apply (subgoal-tac  $t * k < t * k + k - \text{Suc } 0$ )
  prefer 2
  apply simp
apply (rule iffI)
  apply simp
  apply (rule-tac  $x=t * k + k - \text{Suc } 0$  in exI)
  apply (simp add: mod-pred)
  apply (clarify, rename-tac t1)
  apply (drule Suc-leI[of  $t * k$ ])
  apply (drule order-le-less[THEN iffD1], erule disjE)
  prefer 2
  apply simp
  apply (clarsimp simp: iIN-iff)
apply (clarify, rename-tac t1 t2)
apply (case-tac  $t2 \leq \text{Suc } (t * k)$ )
  apply (clarsimp simp: mod-Suc)
  apply (drule-tac  $s=\text{Suc } 0$  in sym, drule-tac  $x=k - \text{Suc } 0$  and  $f=\text{Suc}$  in arg-cong)
  apply (drule-tac  $y=t1$  in order-le-imp-less-or-eq, erule disjE)
  apply (drule-tac  $n=t1$  in Suc-leI)
  apply simp
  apply simp
apply clarsimp
apply (drule-tac  $x=t1$  in spec)
apply (simp add: iIN-iff linorder-not-le)
apply (drule-tac  $y=t1$  in order-le-imp-less-or-eq, erule disjE)
  prefer 2
  apply simp
apply (subgoal-tac  $t * k + k - \text{Suc } 0 \leq t2$ )
  prefer 2
  apply (rule le-diff-conv[THEN iffD2])
  apply (rule less-mod-eq-imp-add-divisor-le, simp)
  apply (simp add: mod-Suc)
apply simp
apply (drule-tac  $x=t * k + k - \text{Suc } 0$  and  $y=t2$  in order-le-imp-less-or-eq, erule
disjE)
  prefer 2
  apply (drule-tac  $t=t2$  in sym, simp)

```

apply (*drule-tac* $x=t1$ **in** *order-le-imp-less-or-eq*, *erule disjE*)
apply *simp+*
done

lemma *i-shrink-eq-Msg-iUntil-finish-event-conv*:

$\llbracket 1 < k; m \neq \varepsilon; \bigwedge t. \text{event } t = (t \bmod k = k - 1); t0 = t * k \rrbracket \implies$
 $((s \div_i k) t = m) = ($
 $(\neg \text{event } t1. t1 \mathcal{U} t2 ([0..] \oplus t0). \text{event } t2 \wedge s t2 = m) \vee$
 $(\neg \text{event } t1. t1 \mathcal{U} t2 ([0..] \oplus t0). (\neg \text{event } t2 \wedge s t2 = m \wedge ($
 $\bigcirc t' t2 [0..]. (s t3 = \varepsilon. t3 \mathcal{U} t4 ([0..] \oplus t'). \text{event } t4 \wedge s t4 = \varepsilon))))$
apply (*simp add: i-shrink-eq-Msg-iSince-conv split del: split-if*)
apply (*simp only: iTL-defs iT-add iT-cut-greater iT-cut-less Ball-def Bex-def iT-iff*
iFROM-inext)
apply (*subgoal-tac* $t * k < t * k + k - \text{Suc } 0$)
prefer 2
apply *simp*
apply (*rule iffI*)
apply (*subgoal-tac* $\bigwedge x. t * k \leq x \implies x < t * k + k - \text{Suc } 0 \implies x \bmod k \neq k$
 $- \text{Suc } 0$)
prefer 2
apply (*rule less-imp-neg*)
apply (*rule le-pred-imp-less, simp*)
apply (*simp only: mult-commute[of t k]*)
apply (*rule between-imp-mod-le[of k - Suc 0 - Suc 0 k t]*)
apply (*simp split del: split-if*)+
apply (*elim exE conjE, rename-tac t1*)
apply (*drule-tac* $x=t1$ **in** *order-le-imp-less-or-eq*, *erule disjE*)
prefer 2
apply (*rule disjI1*)
apply (*rule-tac* $x=t1$ **in** *exI*)
apply (*clarsimp simp add: mod-pred iIN-iff*)
apply (*rule disjI2*)
apply (*rule-tac* $x=t1$ **in** *exI*)
apply (*simp split del: split-if*)
apply (*rule conjI*)
apply (*rule-tac* $x=t * k + k - \text{Suc } 0$ **in** *exI*)
apply (*clarsimp simp: mod-pred iIN-iff*)
apply (*clarsimp simp: iIN-iff*)
apply (*erule disjE*)
apply (*clarsimp, rename-tac t1*)
apply (*drule-tac* $y=t1$ **in** *order-le-imp-less-or-eq*, *erule disjE*)
prefer 2
apply (*drule-tac* $t=t1$ **in** *sym, simp*)
apply (*simp add: iIN-iff*)
apply (*subgoal-tac* $t1 \leq t * k + k - \text{Suc } 0$)
prefer 2
apply (*rule ccontr*)
apply (*drule-tac* $x=t * k + k - \text{Suc } 0$ **in** *spec*)
apply (*simp add: mod-pred*)

```

thm le-mod-add-eq-imp-add-mod-le[OF less-imp-le, rule-format, of  $t * k - k -$ 
  Suc 0  $k$ ]
apply (frule-tac  $a=t * k$  and  $b=t1$  and  $k=k - Suc 0$  and  $m=k$ 
  in le-mod-add-eq-imp-add-mod-le[OF less-imp-le, rule-format])
apply (simp add: add-commute[of  $t * k$ ] mod-pred)
apply (rule-tac  $x=t1$  in exI)
apply simp
apply (clarsimp, rename-tac  $t1 t2$ )
apply (rule-tac  $x=t1$  in exI)
apply (drule-tac  $y=t1$  in order-le-imp-less-or-eq, erule disjE)
prefer 2
apply (drule-tac  $t=t1$  in sym)
apply (clarsimp simp: iIN-iff, rename-tac  $t3$ )
apply (split split-if-asm)
apply (subgoal-tac  $t2 = Suc (t * k)$ )
prefer 2
apply simp
apply (subgoal-tac  $k = Suc (Suc 0)$ )
prefer 2
apply (simp add: mod-Suc)
apply (simp add: mod-Suc)
apply (simp add: iIN-iff)
apply (subgoal-tac  $t * k + k - Suc 0 \leq t2$ )
prefer 2
apply (rule ccontr)
apply (simp add: linorder-not-le)
apply (drule-tac  $m=t2$  in less-imp-le-pred)
thm between-imp-mod-le[of  $k - Suc (Suc 0) k t -$ , OF diff-Suc-less, OF gr-implies-gr0]
apply (simp only: mult-commute[of  $t$ ])
apply (frule-tac  $n=t2$  in between-imp-mod-le[of  $k - Suc (Suc 0) k t -$ , OF
  diff-Suc-less, OF gr-implies-gr0])
apply simp+
apply (drule-tac  $x=t3$  in spec)
apply simp
apply (drule-tac  $x=t3$  in order-le-imp-less-or-eq)
apply (drule-tac  $x=t * k + k - Suc 0$  and  $y=t2$  in order-le-imp-less-or-eq)
apply (fastsimp simp: numeral-2-eq-2 Suc-diff-Suc)
apply simp
apply (case-tac Suc  $t1 = t2$ )
apply (drule-tac  $t=t2$  in sym)
apply (simp add: iIN-iff numeral-2-eq-2 Suc-diff-Suc)
apply (subgoal-tac  $t1 \leq t * k + k - Suc 0$ )
prefer 2
apply (rule ccontr)
apply (drule-tac  $x=t * k + k - Suc 0$  in spec)
apply (simp add: mod-pred)
apply (intro conjI impI)
apply (subgoal-tac Suc  $t1 = t * k + k - Suc 0$ , clarsimp)
apply (subgoal-tac  $t * k + (k - Suc 0) \leq Suc t1$ )

```

```

prefer 2
apply (rule ccontr)
apply (subgoal-tac k - Suc 0 - Suc 0 < k)
prefer 2
apply simp
apply (simp only: mult-commute[of t])
thm between-imp-mod-le[of k - Suc 0 - Suc 0 k t Suc t1]
apply (drule-tac n=Suc t1 in between-imp-mod-le[of k - Suc 0 - Suc 0 k t])
apply simp+
apply (simp add: iN-iff)
apply (subgoal-tac t1 ≤ t * k + k - Suc 0)
prefer 2
apply (rule ccontr)
apply (drule-tac x=t * k + k - Suc 0 in spec)
apply (simp add: mod-pred)
apply (clarsimp, rename-tac t3)
apply (thin-tac All (λx. ?A x → ?B (x mod k)))
apply (drule-tac x=t3 in spec)
apply (subgoal-tac t3 ≤ t2 ⇒ s t3 = ε)
prefer 2
apply (drule-tac x=t3 and y=t2 in order-le-imp-less-or-eq, erule disjE)
apply simp
apply simp
apply (drule-tac P=t3 ≤ t2 in meta-mp)
apply (subgoal-tac t * k < t2)
prefer 2
apply (rule-tac y=t1 in less-trans, assumption+)
apply (case-tac t * k + (k - Suc 0) < t2)
apply simp
apply simp
apply (subgoal-tac t * k + (k - Suc 0) ≤ t2)
prefer 2
apply (simp only: mult-commute[of t])
thm mult-divisor-le-mod-ge-imp-ge
apply (rule mult-divisor-le-mod-ge-imp-ge)
apply simp+
done

```

end

5 IL-AF-Stream-Exec: 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

thm

i-Exec-Stream-Pre-Post1
i-Exec-Stream-Pre-Post2
i-Exec-Stream-Init-Pre-Post1
i-Exec-Stream-Init-Pre-Post2

thm *i-Exec-Stream-Pre-Post1*

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

$\llbracket \text{result} = i\text{-Exec-Comp-Stream trans-fun input } c;$
 $\forall x\text{-}n \text{ } c\text{-}n. P1 \ x\text{-}n \wedge P2 \ c\text{-}n \longrightarrow Q \ (\text{trans-fun } x\text{-}n \ c\text{-}n) \rrbracket \Longrightarrow$
 $\square t \ I. (P1 \ (\text{input } t) \wedge P2 \ (\text{result}^{\leftarrow c} \ t) \longrightarrow Q \ (\text{result } t))$

by (*simp add: i-Exec-Stream-Pre-Post1*)

Direct relation between input and result after transition

thm *i-Exec-Stream-Pre-Post2*

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

$\llbracket \text{result} = i\text{-Exec-Comp-Stream trans-fun input } c;$
 $\forall x\text{-}n \ c\text{-}n. P \ c\text{-}n \longrightarrow Q \ x\text{-}n \ (\text{trans-fun } x\text{-}n \ c\text{-}n) \rrbracket \Longrightarrow$
 $\square t \ I. P \ (\text{result}^{\leftarrow c} \ t) \longrightarrow Q \ (\text{input } t) \ (\text{result } t)$

by (*simp add: i-Exec-Stream-Pre-Post2*)

lemma *i-Exec-Stream-Pre-Post3-iAll-iNext*:

$\llbracket \text{result} = i\text{-Exec-Comp-Stream trans-fun input } c;$
 $\forall x\text{-}n \ c\text{-}n. P \ c\text{-}n \longrightarrow Q \ x\text{-}n \ (\text{trans-fun } x\text{-}n \ c\text{-}n);$
 $\forall t \in I. \text{inext } t \ I' = \text{Suc } t \rrbracket \Longrightarrow$
 $\square t \ I. P \ (\text{result } t) \longrightarrow (\bigcirc t1 \ t \ I'. Q \ (\text{input } t1) \ (\text{result } t1))$

by (*rule iallI, simp add: iNext-def i-Exec-Stream-Pre-Post2-Suc*)

thm *i-Exec-Stream-Init-Pre-Post1*

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

$\llbracket \text{result} = i\text{-Exec-Comp-Stream-Init trans-fun input } c;$
 $\forall x\text{-}n \ c\text{-}n. P1 \ x\text{-}n \wedge P2 \ c\text{-}n \longrightarrow Q \ (\text{trans-fun } x\text{-}n \ c\text{-}n);$
 $\forall t \in I. \text{inext } t \ I' = \text{Suc } t \rrbracket \Longrightarrow$
 $\square t \ I. (P1 \ (\text{input } t) \wedge P2 \ (\text{result } t) \longrightarrow (\bigcirc t1 \ t \ I'. Q \ (\text{result } t1)))$

by (*rule iallI, simp add: iNext-def i-Exec-Stream-Init-Pre-Post1*)

Direct relation between input and state before transition

thm *i-Exec-Stream-Init-Pre-Post2*

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

$\llbracket \text{result} = i\text{-Exec-Comp-Stream-Init trans-fun input } c;$
 $\forall x\text{-}n \ c\text{-}n. P \ x\text{-}n \ c\text{-}n \longrightarrow Q \ (\text{trans-fun } x\text{-}n \ c\text{-}n);$
 $\forall t \in I. \text{inext } t \ I' = \text{Suc } t \rrbracket \Longrightarrow$
 $\square t \ I. (P \ (\text{input } t) \ (\text{result } t) \longrightarrow (\bigcirc t1 \ t \ I'. Q \ (\text{result } t1)))$

by (*rule iallI, simp add: iNext-def i-Exec-Stream-Init-Pre-Post2*)

Relation between input and output

thm *i-Exec-Stream-Pre-Post3-iAll-iNext*

lemma *i-Exec-Stream-Init-Pre-Post3-iAll-iNext*:

$\llbracket \text{result} = i\text{-Exec-Comp-Stream-Init trans-fun input } c;$
 $\forall x\text{-}n \text{ } c\text{-}n. P \text{ } c\text{-}n \longrightarrow Q \text{ } x\text{-}n \text{ } (trans\text{-fun } x\text{-}n \text{ } c\text{-}n);$
 $\forall t \in I. \text{inext } t \text{ } I' = \text{Suc } t \rrbracket \Longrightarrow$
 $\square t \text{ } I. (P \text{ } (result \text{ } t) \longrightarrow (\bigcirc t1 \text{ } t \text{ } I'. Q \text{ } (input \leftarrow^\varepsilon t1) \text{ } (result \text{ } t1)))$
apply (rule *iallI*, unfold *iNext-def*)
apply (simp add: *ilist-Previous-Suc i-Exec-Stream-Init-nth-Suc-eq-i-Exec-Stream-nth*
i-Exec-Stream-Previous-i-Exec-Stream-Init)
thm *i-Exec-Stream-Pre-Post2-iAll[OF refl]*
apply (blast dest: *i-Exec-Stream-Pre-Post2-iAll[OF refl]*)
done

5.2 *i-Exec-Comp-Stream-Acc-Output* and temporal operators with bounded intervals.

Temporal relation between uncompressed and compressed output of accelerated components.

thm *i-shrink-eq-NoMsg-iAll-conv*

lemma *i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-iAll-conv:*

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

by (simp add: *i-Exec-Comp-Stream-Acc-Output-def i-shrink-eq-NoMsg-iAll-conv*)

lemma *i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-iAll-conv2:*

$0 < k \Longrightarrow$
 $((i\text{-Exec-Comp-Stream-Acc-Output } k \text{ } output\text{-fun } trans\text{-fun } input \text{ } c) \text{ } t = \varepsilon) =$
 $(\square t1 \text{ } [\dots k - \text{Suc } 0] \oplus (t * k). (output\text{-fun} \circ i\text{-Exec-Comp-Stream } trans\text{-fun}$
 $(input \odot_i k) \text{ } c) \text{ } t1 = \varepsilon)$

by (simp add: *iT-add i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-iAll-conv*)

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-NoMsg-iAll-conv:*

$0 < k \Longrightarrow$
 $((i\text{-Exec-Comp-Stream-Acc-Output } k \text{ } output\text{-fun } trans\text{-fun } input \text{ } c) \text{ } t = \varepsilon) =$
 $(\square t1 \text{ } [\text{Suc } (t * k) \dots, k - \text{Suc } 0]. (output\text{-fun} \circ i\text{-Exec-Comp-Stream-Init } trans\text{-fun}$
 $(input \odot_i k) \text{ } c) \text{ } t1 = \varepsilon)$

apply (unfold *i-Exec-Comp-Stream-Acc-Output-def*)

thm *i-shrink-eq-NoMsg-iAll-conv*

apply (simp add: *i-shrink-eq-NoMsg-iAll-conv i-Exec-Stream-Init-eq-i-Exec-Stream-Cons*)

apply (rule-tac $t = [\text{Suc } (t * k) \dots, k - \text{Suc } 0]$ and $s = [t * k \dots, k - \text{Suc } 0] \oplus 1$ in *subst*)

apply (simp add: *iIN-add*)

apply (simp add: *iT-Plus-iAll-conv*)

done

thm *i-shrink-eq-Msg-iEx-iAll-cut-greater-conv*

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iEx-iAll-cut-greater-conv:*

$\llbracket 0 < k; m \neq \varepsilon; s = (output\text{-fun} \circ i\text{-Exec-Comp-Stream } trans\text{-fun} \text{ } (input \odot_i k)$
 $c) \rrbracket \Longrightarrow$

$((i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) \ t = m) =$
 $(\diamond t1 \ [t * k \dots, k - \text{Suc } 0]. \ (s \ t1 = m \wedge$
 $\ (\square t2 \ [t * k \dots, k - \text{Suc } 0] \ \downarrow > t1 \ . \ s \ t2 = \varepsilon)))$
by (*simp add: i-Exec-Comp-Stream-Acc-Output-def i-shrink-eq-Msg-iEx-iAll-cut-greater-conv*)
thm *i-shrink-eq-Msg-iEx-iAll-cut-greater-conv2*
lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iEx-iAll-cut-greater-conv2*:
 $\llbracket 0 < k; m \neq \varepsilon; s = (\text{output-fun} \circ i\text{-Exec-Comp-Stream trans-fun } (\text{input} \odot_i k)$
 $c) \rrbracket \implies$
 $((i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) \ t = m) =$
 $(\diamond t1 \ [\dots k - \text{Suc } 0] \oplus (t * k). \ (s \ t1 = m \wedge$
 $\ (\square t2 \ [\dots k - \text{Suc } 0] \oplus (t * k)) \ \downarrow > t1 \ . \ s \ t2 = \varepsilon)))$
by (*simp add: i-Exec-Comp-Stream-Acc-Output-def i-shrink-eq-Msg-iEx-iAll-cut-greater-conv2*)

thm *i-shrink-eq-Msg-iSince-conv*

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iSince-conv*:

$\llbracket 0 < k; m \neq \varepsilon; s = (\text{output-fun} \circ i\text{-Exec-Comp-Stream trans-fun } (\text{input} \odot_i k)$
 $c) \rrbracket \implies$

$((i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) \ t = m) =$
 $(s \ t2 = \varepsilon. \ t2 \ \mathcal{S} \ t1 \ [t * k \dots, k - \text{Suc } 0]. \ s \ t1 = m)$

by (*simp add: i-Exec-Comp-Stream-Acc-Output-def i-shrink-eq-Msg-iSince-conv*)

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iSince-conv2*:

$\llbracket 0 < k; m \neq \varepsilon; s = (\text{output-fun} \circ i\text{-Exec-Comp-Stream trans-fun } (\text{input} \odot_i k)$
 $c) \rrbracket \implies$

$((i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) \ t = m) =$
 $(s \ t2 = \varepsilon. \ t2 \ \mathcal{S} \ t1 \ [\dots k - \text{Suc } 0] \oplus (t * k). \ s \ t1 = m)$

by (*simp add: i-Exec-Comp-Stream-Acc-Output--eq-Msg-iSince-conv iT-add*)

thm *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iSince-conv[OF - - refl]*

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-Msg-iSince-conv*:

$\llbracket 0 < k; m \neq \varepsilon; s = (\text{output-fun} \circ i\text{-Exec-Comp-Stream-Init trans-fun } (\text{input} \odot_i$
 $k) \ c) \rrbracket \implies$

$((i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) \ t = m) =$
 $(s \ t2 = \varepsilon. \ t2 \ \mathcal{S} \ t1 \ [\text{Suc } (t * k) \dots, k - \text{Suc } 0]. \ s \ t1 = m)$

apply (*unfold i-Exec-Comp-Stream-Acc-Output-def*)

apply (*simp add: i-shrink-eq-Msg-iSince-conv i-Exec-Stream-Init-eq-i-Exec-Stream-Cons*)

apply (*rule-tac t=[Suc (t * k) \dots, k - Suc 0] and s=[t * k \dots, k - Suc 0] \oplus 1 in*
 subst)

apply (*simp add: iIN-add*)

apply (*simp add: iT-Plus-iSince-conv*)

done

thm *i-Exec-Comp-Stream-Acc-Output--Init--eq-Msg-iSince-conv[OF - - refl]*

lemma *i-Exec-Comp-Stream-Acc-Output--eq-iAll-iSince-conv*:

$\llbracket 0 < k; s = (\text{output-fun} \circ i\text{-Exec-Comp-Stream trans-fun } (\text{input} \odot_i k) \ c) \rrbracket \implies$
 $((i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) \ t = m) =$

$((m = \varepsilon \longrightarrow (\square t1 \ [t * k \dots, k - \text{Suc } 0]. \ s \ t1 = \varepsilon)) \wedge$

$((m \neq \varepsilon \longrightarrow (s \ t2 = \varepsilon. \ t2 \ \mathcal{S} \ t1 \ [t * k \dots, k - \text{Suc } 0]. \ s \ t1 = m))))$

apply (*case-tac m = \varepsilon*)

apply (*simp add: i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-iAll-conv*)
apply (*simp add: i-Exec-Comp-Stream-Acc-Output--eq-Msg-iSince-conv*)
done

lemma *i-Exec-Comp-Stream-Acc-Output--eq-iAll-iSince-conv2*:

$\llbracket 0 < k; s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream trans-fun} (\text{input} \odot_i k) c) \rrbracket \implies$
 $((\text{i-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) t = m) =$
 $((m = \varepsilon \longrightarrow (\Box t1 [\dots k - \text{Suc } 0] \oplus (t * k). s t1 = \varepsilon)) \wedge$
 $((m \neq \varepsilon \longrightarrow (s t2 = \varepsilon. t2 \mathcal{S} t1 [\dots k - \text{Suc } 0] \oplus (t * k). s t1 = m))))$
by (*simp add: i-Exec-Comp-Stream-Acc-Output--eq-iAll-iSince-conv iT-add*)

5.3 *i-Exec-Comp-Stream-Acc-Output* and temporal operators with unbounded intervals and start/finish events.

thm *i-shrink-eq-NoMsg-iAll-start-event-conv*

lemma *i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-iAll-start-event-conv*:

$\llbracket 0 < k; \bigwedge t. \text{event } t = (t \bmod k = 0); t0 = t * k;$
 $s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream trans-fun} (\text{input} \odot_i k) c) \rrbracket \implies$
 $((\text{i-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) t = \varepsilon) =$
 $(s t0 = \varepsilon \wedge (\bigcirc t' t0 [0\dots]. (s t1 = \varepsilon. t1 \mathcal{U} t2 [0\dots] \oplus t'. \text{event } t2)))$
by (*simp add: i-Exec-Comp-Stream-Acc-Output-def i-shrink-eq-NoMsg-iAll-start-event-conv*)

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-NoMsg-iAll-start-event-conv*:

$\llbracket 0 < k; \bigwedge t. \text{event } t = ((t + k - \text{Suc } 0) \bmod k = 0); t0 = \text{Suc } (t * k);$
 $s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream-Init trans-fun} (\text{input} \odot_i k) c) \rrbracket \implies$
 $((\text{i-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) t = \varepsilon) =$
 $(s t0 = \varepsilon \wedge (\bigcirc t' t0 [0\dots]. (s t1 = \varepsilon. t1 \mathcal{U} t2 [0\dots] \oplus t'. \text{event } t2)))$

apply (*simp add: i-Exec-Comp-Stream-Acc-Output-def i-shrink-eq-NoMsg-iAll-start-event-conv*)

apply (*simp add: iT-add iNext-def iFROM-inext iT-iff*)

apply (*simp add: i-Exec-Stream-Init-eq-i-Exec-Stream-Cons*)

apply (*rule-tac t=[Suc (Suc (t*k))\dots] and s=[Suc (t*k)\dots] \oplus Suc 0 in subst*)

apply (*simp add: iFROM-add*)

thm *iT-Plus-iUntil-conv*

apply (*simp add: iT-Plus-iUntil-conv*)

done

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-NoMsg-iAll-start-event2-conv*:

$\llbracket \text{Suc } 0 < k; \bigwedge t. \text{event } t = (t \bmod k = \text{Suc } 0); t0 = \text{Suc } (t * k);$
 $s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream-Init trans-fun} (\text{input} \odot_i k) c) \rrbracket \implies$
 $((\text{i-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) t = \varepsilon) =$
 $(s t0 = \varepsilon \wedge (\bigcirc t' t0 [0\dots]. (s t1 = \varepsilon. t1 \mathcal{U} t2 [0\dots] \oplus t'. \text{event } t2)))$

thm *i-Exec-Comp-Stream-Acc-Output--Init--eq-NoMsg-iAll-start-event-conv*

thm *mod-eq-Suc-0-conv*

by (*simp add: i-Exec-Comp-Stream-Acc-Output--Init--eq-NoMsg-iAll-start-event-conv mod-eq-Suc-0-conv*)

thm *i-shrink-eq-Msg-iUntil-start-event-conv*

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iUntil-start-event-conv*:

$\llbracket 0 < k; m \neq \varepsilon; \bigwedge t. \text{event } t = (t \bmod k = 0); t0 = t * k;$
 $s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream trans-fun} (\text{input} \odot_i k) c) \rrbracket \implies$
 $((\text{i-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) t = m) = ($
 $(s t0 = m \wedge (\bigcirc t' t0 [0\dots]. (s t1 = \varepsilon. t1 \mathcal{U} t2 ([0\dots] \oplus t'). \text{event } t2))) \vee$

$(\bigcirc t' t0 [0\dots]. (\neg \text{event } t1. t1 \mathcal{U} t2 ([0\dots] \oplus t'). ($
 $s t2 = m \wedge \neg \text{event } t2 \wedge (\bigcirc t'' t2 [0\dots].$
 $(s t3 = \varepsilon. t3 \mathcal{U} t4 ([0\dots] \oplus t''). \text{event } t4))))))$
by (*simp add: i-Exec-Comp-Stream-Acc-Output-def i-shrink-eq-Msg-iUntil-start-event-conv*)
lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-Msg-iUntil-start-event-conv*:
 $\llbracket 0 < k; m \neq \varepsilon; \bigwedge t. \text{event } t = ((t + k - \text{Suc } 0) \bmod k = 0); t0 = \text{Suc } (t * k);$
 $s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream-Init trans-fun } (\text{input} \odot_i k) c) \rrbracket \implies$
 $((\text{i-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) t = m) = ($
 $(s t0 = m \wedge (\bigcirc t' t0 [0\dots]. (s t1 = \varepsilon. t1 \mathcal{U} t2 ([0\dots] \oplus t'). \text{event } t2))) \vee$
 $(\bigcirc t' t0 [0\dots]. (\neg \text{event } t1. t1 \mathcal{U} t2 ([0\dots] \oplus t'). ($
 $s t2 = m \wedge \neg \text{event } t2 \wedge (\bigcirc t'' t2 [0\dots].$
 $(s t3 = \varepsilon. t3 \mathcal{U} t4 ([0\dots] \oplus t''). \text{event } t4))))))$
apply (*simp add: i-Exec-Comp-Stream-Acc-Output-def i-shrink-eq-Msg-iUntil-start-event-conv*)
apply (*simp add: iNext-def iFROM-inext iFROM-iff iT-add*)
apply (*simp add: i-Exec-Stream-Init-eq-i-Exec-Stream-Cons*)
apply (*simp only: Suc-eq-plus1 iFROM-add[symmetric]*)
apply (*simp add: iT-Plus-iUntil-conv*)
apply (*simp only: Suc-eq-plus1 iFROM-add[symmetric]*)
apply (*simp add: iT-Plus-iUntil-conv*)
done
lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-Msg-iUntil-start-event2-conv*:
 $\llbracket \text{Suc } 0 < k; m \neq \varepsilon; \bigwedge t. \text{event } t = (t \bmod k = \text{Suc } 0); t0 = \text{Suc } (t * k);$
 $s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream-Init trans-fun } (\text{input} \odot_i k) c) \rrbracket \implies$
 $((\text{i-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) t = m) = ($
 $(s t0 = m \wedge (\bigcirc t' t0 [0\dots]. (s t1 = \varepsilon. t1 \mathcal{U} t2 ([0\dots] \oplus t'). \text{event } t2))) \vee$
 $(\bigcirc t' t0 [0\dots]. (\neg \text{event } t1. t1 \mathcal{U} t2 ([0\dots] \oplus t'). ($
 $s t2 = m \wedge \neg \text{event } t2 \wedge (\bigcirc t'' t2 [0\dots].$
 $(s t3 = \varepsilon. t3 \mathcal{U} t4 ([0\dots] \oplus t''). \text{event } t4))))))$
by (*simp add: i-Exec-Comp-Stream-Acc-Output--Init--eq-Msg-iUntil-start-event-conv*
mod-eq-Suc-0-conv)

thm *i-shrink-eq-NoMsg-iAll-finish-event-conv*

thm *i-shrink-eq-Msg-iUntil-finish-event-conv*

lemma *i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-iAll-finish-event-conv*:

$\llbracket \text{Suc } 0 < k; \bigwedge t. \text{event } t = (t \bmod k = k - \text{Suc } 0); t0 = t * k;$
 $s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream trans-fun } (\text{input} \odot_i k) c) \rrbracket \implies$
 $((\text{i-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) t = \varepsilon) =$
 $(s t0 = \varepsilon \wedge (\bigcirc t' t0 [0\dots]. (s t1 = \varepsilon. t1 \mathcal{U} t2 [0\dots] \oplus t'. \text{event } t2 \wedge s t2 = \varepsilon)))$
by (*simp add: i-Exec-Comp-Stream-Acc-Output-def i-shrink-eq-NoMsg-iAll-finish-event-conv*)

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-NoMsg-iAll-finish-event-conv*:

$\llbracket \text{Suc } 0 < k; \bigwedge t. \text{event } t = (t \bmod k = 0); t0 = \text{Suc } (t * k);$
 $s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream-Init trans-fun } (\text{input} \odot_i k) c) \rrbracket \implies$
 $((\text{i-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) t = \varepsilon) =$
 $(s t0 = \varepsilon \wedge (\bigcirc t' t0 [0\dots]. (s t1 = \varepsilon. t1 \mathcal{U} t2 [0\dots] \oplus t'. \text{event } t2 \wedge s t2 = \varepsilon)))$
apply (*simp add: i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-iAll-finish-event-conv*)
apply (*simp add: iNext-def iFROM-inext iFROM-iff iT-add*)
apply (*simp add: i-Exec-Stream-Init-eq-i-Exec-Stream-Cons*)
apply (*rule-tac t=[Suc (Suc (t * k))...] and s=[Suc (t * k)...] \oplus 1 in subst*)

```

apply (simp add: iFROM-add)
thm iT-Plus-iUntil-conv
apply (simp add: iT-Plus-iUntil-conv)
apply (simp add: mod-eq-divisor-minus-Suc-0-conv)
done

```

```

thm i-shrink-eq-Msg-iUntil-finish-event-conv

```

```

lemma i-Exec-Comp-Stream-Acc-Output--eq-Msg-iUntil-finish-event-conv:

```

$$\begin{aligned} & \llbracket 0 < k; m \neq \varepsilon; \bigwedge t. \text{event } t = (t \bmod k = k - \text{Suc } 0); t0 = t * k; \\ & \quad s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream trans-fun } (\text{input} \odot_i k) c) \rrbracket \Longrightarrow \\ & ((\text{i-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) t = m) = \\ & ((\neg \text{event } t1. t1 \mathcal{U} t2 ([0..] \oplus t0). \text{event } t2 \wedge s t2 = m) \vee \\ & (\neg \text{event } t1. t1 \mathcal{U} t2 ([0..] \oplus t0). (\neg \text{event } t2 \wedge s t2 = m \wedge (\\ & \quad \bigcirc t' t2 [0..]. (s t3 = \varepsilon. t3 \mathcal{U} t4 ([0..] \oplus t'). \text{event } t4 \wedge s t4 = \varepsilon)))))) \end{aligned}$$

```

apply (case-tac k = Suc 0)

```

```

apply (simp add: iT-add iT-not-empty iFROM-Min)

```

```

apply (drule neq-le-trans[OF not-sym], simp)

```

```

thm i-shrink-eq-Msg-iUntil-finish-event-conv[unfolded One-nat-def]

```

```

apply (simp add: i-Exec-Comp-Stream-Acc-Output-def i-shrink-eq-Msg-iUntil-finish-event-conv)
done

```

```

lemma i-Exec-Comp-Stream-Acc-Output--Init--eq-Msg-iUntil-finish-event-conv:

```

$$\begin{aligned} & \llbracket \text{Suc } 0 < k; m \neq \varepsilon; \bigwedge t. \text{event } t = (t \bmod k = 0); t0 = \text{Suc } (t * k); \\ & \quad s = (\text{output-fun} \circ \text{i-Exec-Comp-Stream-Init trans-fun } (\text{input} \odot_i k) c) \rrbracket \Longrightarrow \\ & ((\text{i-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c) t = m) = \\ & ((\neg \text{event } t1. t1 \mathcal{U} t2 ([0..] \oplus t0). \text{event } t2 \wedge s t2 = m) \vee \\ & (\neg \text{event } t1. t1 \mathcal{U} t2 ([0..] \oplus t0). (\neg \text{event } t2 \wedge s t2 = m \wedge (\\ & \quad \bigcirc t' t2 [0..]. (s t3 = \varepsilon. t3 \mathcal{U} t4 ([0..] \oplus t'). \text{event } t4 \wedge s t4 = \varepsilon)))))) \end{aligned}$$

```

apply (simp add: i-Exec-Comp-Stream-Acc-Output--eq-Msg-iUntil-finish-event-conv)

```

```

apply (simp add: iNext-def iFROM-inext iT-iff)

```

```

apply (simp add: i-Exec-Stream-Init-eq-i-Exec-Stream-Cons)

```

```

apply (simp add: iT-Plus-iUntil-conv)

```

```

apply (simp add: mod-eq-divisor-minus-Suc-0-conv add-Suc[symmetric] del: add-Suc)
done

```

5.4 *i-Exec-Comp-Stream-Acc-Output* and temporal operators with idle states.

```

thm

```

```

  f-expand-nth-interval-eq-nth-append-replicate-NoMsg

```

```

  f-expand-nth-interval-eq-replicate-NoMsg

```

```

  i-expand-nth-interval-eq-nth-append-replicate-NoMsg

```

```

  i-expand-nth-interval-eq-replicate-NoMsg

```

```

thm i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-iAll-conv

```

```

thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-iSince-conv

```

```

thm i-Exec-Stream-Acc-Output--State-Idle-nth

```

lemma *i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-State-Idle-conv*:

```

[[ 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 = 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))))
apply (case-tac k = Suc 0)
apply (simp add: iUntil-def)
apply (rule iffI)
apply (rule-tac t=t in iexI)
apply (simp add: iT-add iT-cut-less)
apply (simp add: iT-add iT-iff)
apply (clarify, rename-tac t1)
apply (simp add: iT-add iT-iff iT-cut-less)
apply (drule order-le-less[THEN iffD1])
apply (erule disjE)
apply (drule-tac t=t in ispec)
apply (simp add: iT-iff)+
apply (drule order-neq-le-trans[OF not-sym Suc-leI], assumption)
thm i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-iAll-conv
apply (simp add: i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-iAll-conv)
apply (simp add: iT-add i-Exec-Stream-nth i-Exec-Stream-Acc-LocalState-nth)
apply (simp add: i-take-Suc-conv-app-nth[of t])
apply (simp add: i-expand-i-take-mult[symmetric] f-Exec-append)
apply (subgoal-tac ∀ t1 ∈ [t * k.., k - Suc 0]. input ⊙i k ↓ Suc t1 ↑ (t * k) =
input t # εt1 - t * k)
prefer 2
apply (simp add: i-expand-nth-interval-eq-nth-append-replicate-NoMsg iIN-iff)
apply (case-tac output-fun (f-Exec-Comp trans-fun (input ⊙i k ↓ Suc (t * k)) c)
≠ ε)
apply (subgoal-tac
  ¬ (□ t1 [t * k.., k - Suc 0]. output-fun (f-Exec-Comp trans-fun (input ⊙i k ↓
Suc t1) c) = ε))
prefer 2
apply simp
apply (rule-tac t=t * k in iexI, assumption)
apply (simp add: iIN-iff)
apply (simp add: not-iUntil del: not-iAll)
apply (clarsimp simp: iT-iff, rename-tac t1 t2)
apply (case-tac t1 = t * k, simp)
apply (drule order-le-neq-trans[OF not-sym], assumption)
apply (rule-tac t=t * k in iexI, simp)
apply (simp add: iFROM-cut-less1 iIN-iff)
apply (case-tac

```

```

State-Idle localState output-fun trans-fun
  (localState ((trans-fun (input t) (f-Exec-Comp trans-fun (input  $\odot_i k \Downarrow (t * k)$ )
c))))))
apply (subgoal-tac
  ( $\square$  t1 [t * k...k - Suc 0]. output-fun (f-Exec-Comp trans-fun (input  $\odot_i k \Downarrow$ 
Suc t1) c) = NoMsg))
prefer 2
apply (clarsimp simp: iIN-iff, rename-tac t1)
thm subst[OF i-take-drop-append, rule-format]
apply (rule-tac m=t * k and n=Suc t1 in subst[OF i-take-drop-append, rule-format],
simp)
apply (drule-tac x=t1 in bspec, simp add: iT-iff)
apply (simp add: f-Exec-append del: i-take-drop-append)
apply (simp add: i-take-Suc-conv-app-nth f-Exec-append i-expand-nth-mult)
thm f-Exec-State-Idle-replicate-NoMsg-output
apply (rule f-Exec-State-Idle-replicate-NoMsg-output, assumption+)
apply (simp add: iUntil-def)
apply (rule-tac t=t * k in iexI)
apply (simp add: i-take-Suc-conv-app-nth f-Exec-append i-expand-nth-mult iFROM-cut-less)
apply (simp add: iFROM-iff)
apply (subgoal-tac  $\forall i < k$ . input  $\odot_i k \Uparrow$  Suc (t * k)  $\Downarrow$  i = NoMsgi)
prefer 2
apply (simp add: list-eq-iff i-expand-nth-if)
apply (rule iffI)
thm State-Idle-imp-exists-state-change2
apply (frule State-Idle-imp-exists-state-change2, assumption)
apply (elim exE conjE, rename-tac i)
apply (frule Suc-less-pred-conv[THEN iffD2])
apply (simp only: iUntil-def)
apply (rule-tac t=Suc (t * k + i) in iexI)
apply (rule conjI)
  apply (drule-tac t=Suc (t * k + i) in ispec)
    apply (simp add: iIN-iff)
    apply (rule conjI, simp)
  apply (rule-tac t=Suc (Suc (t * k + i)) and s=Suc (t * k) + Suc i in subst,
simp)
apply (subst i-take-add)
apply (drule-tac x=Suc i in spec)+
apply (simp add: i-take-Suc-conv-app-nth f-Exec-append i-expand-nth-mult)
apply (rule iallI, rename-tac t1)
apply (drule-tac t=t1 in ispec)
apply (drule-tac m=Suc i in less-imp-le-pred)
apply (clarsimp simp: iIN-iff iFROM-cut-less1)
apply (rule order-trans, assumption)
apply simp
apply assumption
apply (simp add: iFROM-iff)
apply (rule iallI)
apply (unfold iUntil-def, elim iexE conjE, rename-tac t2)

```

```

apply (case-tac  $t1 < t2$ )
  apply (drule-tac  $t=t1$  in ispec)
    apply (simp add: cut-less-mem-iff iT-iff)
    apply simp
  apply (simp add: linorder-not-less)
apply (case-tac  $t1 = t2$ , simp)
apply (drule le-neq-trans[OF - not-sym], assumption)
apply (drule-tac  $i=t2$  in less-imp-add-positive, elim exE conjE, rename-tac i)
apply (drule-tac  $t=t1$  in sym)
apply (simp del: add-Suc add: add-Suc[symmetric] i-take-add f-Exec-append iFROM-iff)
apply (rule-tac  $t=input \odot_i k \uparrow \text{Suc } t2 \downarrow i$  and  $s=\varepsilon^i$  in subst)
  apply (rule list-eq-iff[THEN iffD2])
  apply simp
  apply (intro allI impI, rename-tac i1)
  apply (simp add: i-expand-nth-if)
  apply (subst imp-conv-disj, rule disjI1)
thm if-not-P'
apply simp
apply (subgoal-tac  $t * k < \text{Suc } (t2 + i1) \wedge \text{Suc } (t2 + i1) < t * k + k$ , elim conjE)
  prefer 2
  apply (simp add: iIN-iff)
  apply (simp only: mult-commute[of - k])
  apply (rule between-imp-mod-gr0, assumption+)
thm f-Exec-State-Idle-replicate-NoMsg-gr0-output
apply (rule f-Exec-State-Idle-replicate-NoMsg-gr0-output, assumption+)
done
thm i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-State-Idle-conv
thm i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-State-Idle-conv[OF - - refl refl]

lemma i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-imp:
  [|  $0 < k$ ;
     $s = i\text{-Exec-Comp-Stream trans-fun } (input \odot_i k) c$ ;
     $t0 = t * k$ ;
     $t1 \in [0.., k - \text{Suc } 0] \oplus t0$ ;
     $\text{State-Idle localState output-fun trans-fun } (localState (s t1))$ ;
     $\text{output-fun } (s t1) \neq \varepsilon$  |]  $\implies$ 
   $i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun } input c t = \text{output-fun } (s t1)$ 
apply (case-tac  $k = \text{Suc } 0$ )
  apply (simp add: iIN-0 iT-Plus-singleton)
apply (drule order-neq-le-trans[OF not-sym], rule Suc-leI, assumption)
apply (simp add: iT-add iT-iff, erule conjE)
apply (simp only: i-Exec-Stream-Acc-Output-nth i-Exec-Stream-nth)
apply (rule-tac  $t=\text{Suc } t1$  and  $s=t * k + (\text{Suc } t1 - t * k)$  in subst, simp)
apply (simp only: i-take-add f-Exec-append i-expand-i-take-mult)
apply (subgoal-tac  $input \odot_i k \uparrow (t * k) \downarrow (\text{Suc } t1 - t * k) = input t \# \varepsilon^{t1 - t * k}$ )
  prefer 2
  apply (simp add: i-take-i-drop)

```

```

apply (subst i-expand-nth-interval-eq-nth-append-replicate-NoMsg)
apply (simp del: f-Exec-Comp-Stream.simps)+
apply (subgoal-tac  $\exists i. k - \text{Suc } 0 = t1 - t * k + i$ )
prefer 2
thm le-iff-add[THEN iffD1]
apply (rule le-iff-add[THEN iffD1])
apply (simp add: le-diff-conv)
apply (erule exE)
apply (simp only: replicate-add)
apply (subst append-Cons[symmetric])
thm State-Idle-append-replicate-NoMsg-output-last-message
apply (subst State-Idle-append-replicate-NoMsg-output-last-message)
apply (simp only: f-Exec-append[symmetric])
apply (rule-tac  $t = \text{input} \Downarrow t \odot_f k @ \text{input } t \# \text{NoMsg}^{t1 - t * k}$  and  $s = \text{input} \odot_i$ 
 $k \Downarrow \text{Suc } t1$  in subst)
apply (subst i-expand-i-take-mult[symmetric])
apply (drule-tac  $t = \text{input } t \# \text{NoMsg}^{t1 - t * k}$  in sym)
apply (simp add: i-take-add[symmetric])
apply assumption
apply (subgoal-tac
  f-Exec-Comp-Stream trans-fun (input t # NoMsgt1 - t * k)
  (f-Exec-Comp trans-fun (input  $\Downarrow t \odot_f k$ ) c)  $\neq []$ )
prefer 2
apply (simp add: f-Exec-Stream-not-empty-conv)
thm last-message-Msg-eq-last
apply (rule ssubst[OF last-message-Msg-eq-last])
apply simp
apply (subst map-last, simp)
thm f-Exec-eq-f-Exec-Stream-last2
apply (subst f-Exec-eq-f-Exec-Stream-last2[symmetric], simp)
apply (subst f-Exec-append[symmetric])
apply (rule-tac  $t = \text{input} \Downarrow t \odot_f k @ \text{input } t \# \text{NoMsg}^{t1 - t * k}$  and  $s = \text{input} \odot_i$ 
 $k \Downarrow \text{Suc } t1$  in subst)
apply (subst i-expand-i-take-mult[symmetric])
apply (rule-tac  $t = \text{Suc } t1$  and  $s = t * k + (\text{Suc } t1 - t * k)$  in subst, simp)
apply (subst i-take-add, simp)
apply assumption
apply (subst map-last, simp)
apply (subst f-Exec-eq-f-Exec-Stream-last2[symmetric], simp+)
done

```

thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-iSince-conv

lemma i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv2:

$[[0 < k;$

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;
    t1  $\in$  [0..., k - Suc 0]  $\oplus$  t0;
    State-Idle localState output-fun trans-fun (localState (s t1));
    output-fun (s t1)  $\neq$   $\varepsilon$  ]  $\implies$ 
    (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  $\wedge$  State-Idle localState output-fun trans-fun (localState
      (s t1))))))
  apply (case-tac k = Suc 0)
  apply (simp add: iIN-0 iT-Plus-singleton)
  apply (drule order-neq-le-trans[OF not-sym], rule Suc-leI, assumption)
  apply simp
  thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-imp
  apply (simp add: i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-imp)
  apply (rule iffI)
  apply blast
  apply (clarify, rename-tac t1')
  apply (subgoal-tac t1' = t1)
  prefer 2
  apply (rule ccontr)
  apply (simp add: i-Exec-Stream-nth)
  apply (subgoal-tac
     $\wedge$  n1 n2.
    [ n1 < n2; n1  $\in$  [0..., k - Suc 0]  $\oplus$  t * k; n2  $\in$  [0..., k - Suc 0]  $\oplus$  t * k;
    State-Idle localState output-fun trans-fun (localState (f-Exec-Comp trans-fun
    (input  $\odot_i$  k  $\Downarrow$  Suc n1) c));
    output-fun (f-Exec-Comp trans-fun (input  $\odot_i$  k  $\Downarrow$  Suc n2) c)  $\neq$  NoMsg ]  $\implies$ 
    False)
  prefer 2
  apply (drule-tac i=n1 in less-imp-add-positive, elim exE conjE, rename-tac i)
  apply (drule-tac t=n2 in sym, simp)
  apply (simp only: add-Suc[symmetric] i-take-add f-Exec-append)
  thm i-expand-nth-interval-eq-replicate-NoMsg
  apply (subgoal-tac input  $\odot_i$  k  $\Uparrow$  Suc n1  $\Downarrow$  i =  $\varepsilon^i$ )
  prefer 2
  apply (subst i-take-i-drop)
  apply (rule-tac t= $\varepsilon^i$  and s= $\varepsilon^i$  + Suc n1 - Suc n1 in subst, simp)
  thm i-expand-nth-interval-eq-replicate-NoMsg
  apply (rule-tac t=t in i-expand-nth-interval-eq-replicate-NoMsg)
  apply (simp add: iT-add iT-iff)+
  thm f-Exec-State-Idle-replicate-NoMsg-gr0-output
  apply (frule-tac c=f-Exec-Comp trans-fun (input  $\odot_i$  k  $\Downarrow$  Suc n1) c and n=i
    in f-Exec-State-Idle-replicate-NoMsg-gr0-output)
  apply (fastsimp dest: linorder-neq-iff[THEN iffD1])+
done

```

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-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;
  State-Idle localState output-fun trans-fun (localState (s t1));
  output-fun (s t1) ≠ ε ]] ⇒
(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))))))
thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv2
apply (subst i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv2, as-
sumption+)
apply (unfold iUntil-def)
apply (rule iffI)
apply (elim iexE conjE, rename-tac t2)
apply (rule-tac t=t2 in iexI)
prefer 2
apply (simp add: iT-add iT-iff)
apply simp
apply (rule iallI, rename-tac t2')
apply (rule ccontr)
apply (simp add: cut-less-mem-iff iT-iff iT-add, elim conjE)
apply (frule-tac n=t2' in le-imp-less-Suc)
apply (frule-tac i=t2' in less-imp-add-positive, elim exE conjE, rename-tac i)
apply (drule-tac t=t2 in sym)
apply (simp only: i-Exec-Stream-nth add-Suc[symmetric] i-take-add f-Exec-append)
apply (simp only: i-take-i-drop)
apply (subgoal-tac input ⊙i k ↓ (i + Suc t2') ↑ Suc t2' = εi)
prefer 2
thm i-expand-nth-interval-eq-replicate-NoMsg
apply (rule-tac t=εi and s=εi + Suc t2' - Suc t2' in subst, simp)
apply (rule-tac t=t in i-expand-nth-interval-eq-replicate-NoMsg)
apply simp+
thm f-Exec-State-Idle-replicate-NoMsg-gr0-output
apply (drule-tac c=(f-Exec-Comp trans-fun (input ⊙i k ↓ Suc t2') c) and n=i
in f-Exec-State-Idle-replicate-NoMsg-gr0-output, assumption)
apply simp
apply (fastsimp simp: iT-add iT-iff i-Exec-Stream-Acc-LocalState-nth i-Exec-Stream-nth)
done

```

lemma i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-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
apply (clarsimp simp: iUntil-def iNext-def iT-inext iT-iff, rename-tac t2)
apply (simp only: i-Exec-Stream-Acc-Output-nth i-Exec-Stream-Acc-LocalState-nth
i-Exec-Stream-nth)
apply (rule last-message-conv[THEN iffD2], assumption)
apply (clarsimp simp: iT-add iT-iff simp del: f-Exec-Comp-Stream.simps)
apply (subgoal-tac t1 - t * k < k)
prefer 2
apply simp
apply (rule-tac x=t1 - t * k in exI)
apply (rule conjI, simp)
apply (rule conjI)
apply (simp add: f-Exec-Stream-nth min-eqL del: f-Exec-Comp.simps f-Exec-Comp-Stream.simps)
apply (simp only: f-Exec-append[symmetric])
thm i-expand-i-take-mult-Suc
apply (subst i-expand-i-take-mult-Suc[symmetric], assumption)
apply simp
apply (intro allI impI)
apply (simp only: f-Exec-Stream-length length-Cons length-replicate Suc-pred
nth-map f-Exec-Stream-nth take-Suc-Cons take-replicate min-eqL[OF less-imp-le-pred])
apply (subst f-Exec-append[symmetric])
apply (subst i-expand-i-take-mult-Suc[symmetric], assumption)
apply (case-tac t2 ≤ t * k + j)
prefer 2
apply fastsimp
apply (drule-tac x=t2 in order-le-less[THEN iffD1, rule-format])
apply (erule disjE)
prefer 2
apply simp
apply (subgoal-tac
  State-Idle localState output-fun trans-fun
  (localState (f-Exec-Comp trans-fun (input ⊙i k ↓ (t * k + Suc j)) c)))
prefer 2
apply (rule-tac t=t * k + Suc j and s=Suc t2 + (t * k + j - t2) in subst,
simp)
apply (simp only: i-take-add f-Exec-append)
apply (simp only: i-take-i-drop)
apply simp
thm i-expand-nth-interval-eq-replicate-NoMsg
apply (rule-tac t=t in ssubst[OF i-expand-nth-interval-eq-replicate-NoMsg, rule-format],

```

```

simp+)
thm f-Exec-State-Idle-replicate-NoMsg-state
apply (simp add: f-Exec-State-Idle-replicate-NoMsg-state)
apply (subgoal-tac t1 div k = t  $\wedge$  t2 div k = t, elim conjE)
prefer 2
thm le-less-imp-div
apply (simp add: le-less-imp-div)
apply (simp only: i-expand-i-take-Suc i-expand-i-take-mult-Suc f-Exec-append)
apply (simp add: f-Exec-append)
thm f-Exec-State-Idle-replicate-NoMsg-gr-output
apply (rule-tac m=t2 mod k in f-Exec-State-Idle-replicate-NoMsg-gr-output, as-
sumption)
apply (simp add: mod-div-equality')
done

lemma i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-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  $\neq$   $\varepsilon$ ;
    t0 = t * k;
    s = i-Exec-Comp-Stream trans-fun (input  $\odot_i$  k) c;
     $\square$  t1 [0.., k - Suc 0]  $\oplus$  t0.  $\neg$  (
      State-Idle localState output-fun trans-fun (localState (s t1))  $\wedge$ 
      output-fun (s t1)  $\neq$   $\varepsilon$ ) ]  $\implies$ 
    (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) =  $\varepsilon$ . t3  $\mathcal{U}$  t4 ([0..]  $\oplus$  t2).
          (output-fun (s t4) =  $\varepsilon$   $\wedge$  State-Idle localState output-fun trans-fun (localState
            (s t4))))))))))
apply (rule iffI)
apply (simp only: i-Exec-Stream-Acc-Output-nth i-Exec-Stream-nth)
apply (simp only: iNext-def iFROM-iff iFROM-inext)
thm last-message-conv
thm last-message-conv[THEN iffD1]
apply (frule last-message-conv[THEN iffD1], assumption)
apply (elim exE conjE, rename-tac i)
apply (simp add: f-Exec-Stream-nth min-eqL del: f-Exec-Comp.simps f-Exec-Comp-Stream.simps
de-Morgan-conj)
apply (subgoal-tac
   $\square$  t' ([0..]  $\oplus$  (Suc (t * k + i)))  $\downarrow$  < (t * k + k).
  output-fun (f-Exec-Comp trans-fun (input  $\odot_i$  k  $\downarrow$  Suc t') c) =  $\varepsilon$ )
prefer 2
apply (rule iallI, rename-tac t')
apply (simp only: iT-add iT-iff cut-less-mem-iff, erule conjE)
apply (drule-tac x=t' - t * k in spec)
apply (subgoal-tac t' - t * k < k)

```

```

prefer 2
apply simp
apply (simp add: f-Exec-Stream-nth min-eqL del: f-Exec-Comp-Stream.simps
de-Morgan-conj)
apply (subgoal-tac  $t * k \leq t'$ )
prefer 2
apply simp
apply (rule-tac  $t = \text{Suc } t'$  and  $s = t * k + (\text{Suc } t' - t * k)$  in subst, simp)
apply (simp only: i-take-add f-Exec-append i-expand-i-take-mult)
apply (simp add: i-take-i-drop)
thm i-expand-nth-interval-eq-nth-append-replicate-NoMsg
apply (rule ssubst[OF i-expand-nth-interval-eq-nth-append-replicate-NoMsg])
apply (simp del: f-Exec-Comp-Stream.simps de-Morgan-conj)+
apply (rule-tac  $t = t * k + i$  in iexI)
prefer 2
apply (simp add: iT-add iT-iff)
apply (rule conjI)
thm i-expand-i-take-mult-Suc
apply (simp add: add-Suc-right[symmetric] i-expand-i-take-mult-Suc f-Exec-append
del: add-Suc-right)
apply (simp only: i-Exec-Stream-Acc-LocalState-nth i-expand-i-take-mult[symmetric]
mult-Suc add-commute[of k])
apply (subgoal-tac
 $\neg \text{State-Idle localState output-fun trans-fun}$ 
 $(\text{localState } (f\text{-Exec-Comp trans-fun } (\text{input } \odot_i k \Downarrow (t * k + \text{Suc } i)) c)))$ )
prefer 2
apply (drule-tac  $t = t * k + i$  in ispec)
apply (simp add: iT-add iT-iff)
apply (simp add: add-Suc-right[symmetric] i-expand-i-take-mult-Suc f-Exec-append
i-expand-i-take-mult del: add-Suc-right)
apply (thin-tac last-message ?x = m)
apply (drule-tac
 $a = t * k + k$  and  $b = t * k + \text{Suc } (k - \text{Suc } 0)$  and
 $P = \lambda x. \text{State-Idle localState output-fun trans-fun}$ 
 $(\text{localState } (f\text{-Exec-Comp trans-fun } (\text{input } \odot_i k \Downarrow x) c))$  in back-subst,
simp)
apply (simp only: i-expand-i-take-mult-Suc f-Exec-append)
thm State-Idle-imp-exists-state-change
apply (frule-tac  $n = k - \text{Suc } 0 - i$  in State-Idle-imp-exists-state-change)
apply (simp add: f-Exec-append[symmetric] replicate-add[symmetric])
apply (elim exE conjE, rename-tac i1)
thm less-diff-conv[THEN iffD1, rule-format]
apply (frule-tac  $i = i1$  in less-diff-conv[THEN iffD1, rule-format])
thm subst[OF add-commute, rule-format]
apply (drule-tac  $a = i1$  and  $P = \lambda x. (x < k - \text{Suc } 0)$  in subst[OF add-commute,
rule-format])
apply (frule Suc-less-pred-conv[THEN iffD2])
apply (simp only: iUntil-def)
apply (rule-tac  $t = t * k + \text{Suc } (i + i1)$  in iexI)

```

```

prefer 2
apply (simp add: iT-add iT-iff)
apply (rule conjI)
apply (drule-tac t=t * k + Suc (i + i1) in ispec)
  apply (simp add: iT-add iT-iff cut-less-mem-iff)
apply (subgoal-tac Suc (t * k + Suc (i + i1)) = t * k + Suc (Suc (i + i1)))
  prefer 2
  apply simp
thm i-expand-i-take-mult-Suc
apply (simp only: i-expand-i-take-mult-Suc f-Exec-append)
apply (simp add: add-Suc-right[symmetric] replicate-add f-Exec-append del: add-Suc-right
replicate.simps)
apply (clarsimp simp: cut-less-mem-iff iT-add iT-iff simp del: f-Exec-Comp-Stream.simps,
rename-tac t^')
apply (subgoal-tac  $\exists i' > i. t' = t * k + i'$ )
  prefer 2
  apply (rule-tac  $x=t' - t * k$  in exI)
  apply simp
apply (thin-tac iAll ?I ?P)+
apply (elim exE conjE)
apply (subgoal-tac  $i' < k$ )
  prefer 2
  apply simp
apply (simp add: add-Suc-right[symmetric] i-expand-i-take-mult-Suc f-Exec-append
f-Exec-Stream-nth min-eqL i-expand-i-take-mult del: add-Suc-right f-Exec-Comp-Stream.simps)
apply (elim iexE conjE, rename-tac t1)
thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-imp2
apply (rule i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-imp2, as-
sumption+)
done

```

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:*

```

[[ 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 t1))). t1 U t2 [0..]
⊕ t0. (
  (output-fun (s t2) = m) ∧
  (○ t3 t2 [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
apply (case-tac

```

```

◇ t1 [0.., k - Suc 0] ⊕ t0. (
  State-Idle localState output-fun trans-fun (localState (s t1)) ∧
  output-fun (s t1) ≠ ε)
apply (clarsimp, rename-tac t1)
thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-imp[OF Suc-lessD
refl refl]
apply (frule i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-imp[OF Suc-lessD
refl refl], assumption+)
apply (simp only: iNext-def iT-inext iT-iff iUntil-def)
apply (elim iexE conjE, rename-tac t2 t3)
apply (subgoal-tac t2 ≤ t1)
prefer 2
apply (rule ccontr)
apply (drule-tac t=t1 in ispec)
apply (simp add: cut-less-mem-iff iT-add iT-iff)
apply simp
apply (thin-tac iAll ?I ?P)
apply (subgoal-tac t1 ≤ t2)
prefer 2
apply (rule ccontr)
apply (subgoal-tac t3 < t1 → output-fun (i-Exec-Comp-Stream trans-fun (input
⊙i k) c t1) = ε)
prefer 2
apply (rule impI)
apply (subgoal-tac t * k ≤ t3)
prefer 2
apply (simp add: iT-add iT-iff)
apply (subgoal-tac t1 div k = t ∧ t3 div k = t, elim conjE)
prefer 2
apply (simp add: iT-add iT-iff le-less-imp-div)
apply (simp (no-asm-simp) add: i-Exec-Stream-nth i-expand-i-take-Suc f-Exec-append)
thm f-Exec-State-Idle-replicate-NoMsg-gr-output
apply (rule-tac m=t3 mod k in f-Exec-State-Idle-replicate-NoMsg-gr-output[of
localState output-fun trans-fun])
apply (simp add: i-Exec-Stream-nth i-expand-i-take-Suc f-Exec-append)
apply (simp add: mod-div-equality^)
apply (case-tac t1 < t3)
apply (drule-tac t=t1 in ispec)
apply (simp add: cut-less-mem-iff iT-add iT-iff)
apply simp+
thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-conv2
apply (rule ssubst[OF i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-conv2],
simp+)
apply (simp only: iNext-def iT-inext iT-iff iUntil-def)
apply (elim iexE conjE, rename-tac t1 t2)
apply (subgoal-tac t1 ≤ t * k + (k - Suc 0))
prefer 2
apply (rule ccontr)
apply (simp add: i-Exec-Stream-Acc-LocalState-nth i-expand-i-take-mult[symmetric]

```

```

add-commute[of k]
  apply (thin-tac iAll ?I ?P)
  apply (drule-tac t=t * k + (k - Suc 0) in ispec)
  apply (simp add: cut-less-mem-iff iT-add iT-iff)
  apply (simp add: i-Exec-Stream-nth)
apply (rule-tac t=t1 in iexI)
  prefer 2
  apply (simp add: iT-add iT-iff)
apply simp
apply (rule-tac t=t2 in iexI)
apply simp+
done
lemma i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-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;
   □ t1 [0.., k - Suc 0] ⊕ t0. ¬ (
     State-Idle localState output-fun trans-fun (localState (s t1)) ∧
     output-fun (s t1) ≠ ε) |] ⇒
  (i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c t = m) =
  ((¬ State-Idle localState output-fun trans-fun (localState (s t1))). t1  $\mathcal{U}$  t2 [0..]
  ⊕ t0. (
    (output-fun (s t2) = m) ∧
    (○ t3 t2 [0..].
      ((output-fun (s t4) = ε. t4  $\mathcal{U}$  t5 ([0..] ⊕ t3).
        (output-fun (s t5) = ε ∧ State-Idle localState output-fun trans-fun (localState
        (s t5))))))))))
  apply (rule iffI)
  thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-conv2
  thm subst[OF i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-conv2,
  where P=λx. x]
  apply (frule subst[OF i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-conv2,
  where P=λx. x], assumption+)
  apply (simp only: iNext-def iT-inext iT-iff iUntil-def)
  apply (elim iexE conjE, rename-tac t1 t2)
  apply (rule-tac t=t1 in iexI)
  prefer 2
  apply (simp add: iT-add iT-iff)
  apply (intro conjI)
  apply simp
  apply (rule-tac t=t2 in iexI)
  prefer 2
  apply (simp add: iT-add iT-iff)
  apply simp
  apply (rule iallI, rename-tac t')
  apply (rule ccontr)

```

```

apply (clarsimp simp: cut-less-mem-iff)
apply (drule-tac i=t' in less-imp-add-positive)
apply (elim exE conjE, rename-tac i)
apply (drule-tac t=t1 in sym)
apply (simp only: i-Exec-Stream-nth)
apply (simp only: add-Suc[symmetric] i-take-add f-Exec-append)
apply (simp only: i-take-i-drop)
thm i-expand-nth-interval-eq-replicate-NoMsg
apply (subgoal-tac i + Suc t' ≤ t * k + k)
  prefer 2
  apply (simp add: iT-add iT-iff)
  apply (simp only: iT-add iT-iff)
thm i-expand-nth-interval-eq-replicate-NoMsg[of k t Suc t', OF - le-imp-less-Suc
le-add2]
apply (simp only: i-expand-nth-interval-eq-replicate-NoMsg[of k t, OF - le-imp-less-Suc
le-add2])
thm f-Exec-State-Idle-replicate-NoMsg-gr0-output
apply (drule-tac c=f-Exec-Comp trans-fun (input ⊙i k ↓ Suc t') c and n=i in
f-Exec-State-Idle-replicate-NoMsg-gr0-output)
apply simp+
thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-imp
apply (rule i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-imp, simp+)
done

thm
  i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv2
  i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-conv2
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 ∪ t4 ([0..] ⊕ t2).
        (output-fun (s t4) = ε ∧ State-Idle localState output-fun trans-fun (localState
(s t4))))))))))
apply (subst conj-disj-distribL)
thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv2
apply (case-tac
  ◇ t1 [0.., k - Suc 0] ⊕ t0.
  (State-Idle localState output-fun trans-fun (localState (s t1)) ∧ output-fun (s
t1) ≠ ε))
apply (elim iexE conjE, rename-tac t1)

```

```

apply (rule iffI)
thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv2 [THEN iffD1,
OF Suc-lessD]
apply (frule i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv2 [THEN
iffD1, OF Suc-lessD], assumption+)
apply fastsimp
apply (elim iexE conjI, rename-tac t2)
apply (erule disjE)
thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv2
apply (rule i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv2 [THEN
iffD2], simp+)
apply (rule-tac t=t2 in iexI, simp+)
thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-imp2
apply (rule-tac ?t1.0=t2 in i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-imp2,
simp+)
thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-conv2 [OF - - - refl
refl]
apply (rule ssubst [OF i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-conv2 [OF
- - - refl refl]], simp+)
apply fastsimp
done
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))))))))))
apply (subst i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv2, assumption+)
apply blast
done

```

thm *i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-State-Idle-conv*

thm *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv2*

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);

```

$t0 = t * k;$
 $s = i\text{-Exec-Comp-Stream trans-fun (input } \odot_i k) c \llbracket \implies$
 $(i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c \text{ } t = m) = ($
 $(m = \varepsilon \longrightarrow$
 $(\text{output-fun (s } t1) = \varepsilon. t1 \mathcal{U} t2 ([0..] \oplus t0). ($
 $\text{output-fun (s } t2) = \varepsilon \wedge \text{State-Idle localState output-fun trans-fun (localState$
 $(s \ t2)))))) \wedge$
 $(m \neq \varepsilon \longrightarrow$
 $(\diamond t1 [0.., k - \text{Suc } 0] \oplus t0. ($
 $\text{output-fun (s } t1) = m \wedge$
 $(\text{State-Idle localState output-fun trans-fun (localState (s } t1)) \vee$
 $(\bigcirc t2 \ t1 [0..].$
 $((\text{output-fun (s } t3) = \varepsilon. t3 \mathcal{U} t4 ([0..] \oplus t2).$
 $(\text{output-fun (s } t4) = \varepsilon \wedge \text{State-Idle localState output-fun trans-fun (localState$
 $(s \ t4))))))))))$
apply (case-tac $m = \varepsilon$)
apply (simp add: $i\text{-Exec-Comp-Stream-Acc-Output--eq-NoMsg-State-Idle-conv}$)
apply (simp add: $i\text{-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv2}$)
done

thm

$i\text{-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv}$

$i\text{-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-conv}$

lemma $i\text{-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv}'$:

$\llbracket \text{Suc } 0 < k;$
 $\text{State-Idle localState output-fun trans-fun (}$
 $\text{ } i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input } c \text{ } t);$
 $m \neq \varepsilon;$
 $t0 = t * k;$
 $s = i\text{-Exec-Comp-Stream trans-fun (input } \odot_i k) c \llbracket \implies$
 $(i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c \text{ } t = m) =$
 $((\neg \text{State-Idle localState output-fun trans-fun (localState (s } t2))). t2 \mathcal{U} t1 [0..]$
 $\oplus t0.$
 $(\text{output-fun (s } t1) = m \wedge \text{State-Idle localState output-fun trans-fun (localState$
 $(s \ t1)))) \vee$
 $((\neg \text{State-Idle localState output-fun trans-fun (localState (s } t2))). t2 \mathcal{U} t1 [0..]$
 $\oplus t0.$
 $(\text{output-fun (s } t1) = m \wedge$
 $(\bigcirc t3 \ t1 [0..].$
 $((\text{output-fun (s } t4) = \varepsilon. t4 \mathcal{U} t5 ([0..] \oplus t3).$
 $(\text{output-fun (s } t5) = \varepsilon \wedge \text{State-Idle localState output-fun trans-fun (localState$
 $(s \ t5))))))))))$
thm $i\text{-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv}$
apply (case-tac
 $\diamond t1 [0.., k - \text{Suc } 0] \oplus t0.$
 $(\text{State-Idle localState output-fun trans-fun (localState (s } t1)) \wedge \text{output-fun (s}$
 $t1) \neq \varepsilon)$
apply (elim $iexE \text{ conjE, rename-tac } t1$)
apply (rule $iffI$)

```

thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv [THEN iffD1,
OF Suc-lessD]
apply (frule i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv [THEN
iffD1, OF Suc-lessD], simp+)
apply (erule disjE)
thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv
apply (rule i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv [THEN
iffD2], simp+)
thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-imp
apply (rule-tac i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-imp, simp+)
thm i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-conv [OF - - - refl
refl]
apply (subst i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-conv [OF -
- - refl refl], simp+)
apply (rule iffI)
apply simp
apply (unfold iUntil-def, erule disjE)
apply (elim iexE conjE, rename-tac t1)
apply (case-tac t1 ≤ t * k + (k - Suc 0))
prefer 2
apply (simp add: i-Exec-Stream-Acc-LocalState-nth i-Exec-Stream-nth i-expand-i-take-mult [symmetric])
apply (thin-tac iAll ?I ?P)
apply (drule-tac t=t * k + (k - Suc 0) in ispec)
apply (simp add: cut-less-mem-iff iT-add iT-iff)
apply (simp add: add-commute[of k])
apply (fastsimp simp: iT-add iT-iff)+
done
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))))))))))
apply (subst i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv', assumption+)
apply (subst iUntil-disj-distrib [symmetric])
apply (rule iUntil-cong2)
apply blast
done

```

thm *i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-State-Idle-conv*

thm *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv*

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 \odot_i *k*) *c*]] \implies
 (*i-Exec-Comp-Stream-Acc-Output* *k* output-fun trans-fun input *c* *t* = *m*) = (
 (*m* = ε \longrightarrow
 (*output-fun* (*s* *t1*) = ε . *t1* \mathcal{U} *t2* ([0..] \oplus *t0*). (
 output-fun (*s* *t2*) = ε \wedge State-Idle localState output-fun trans-fun (localState
 (*s* *t2*)))))) \wedge
 (*m* \neq ε \longrightarrow
 (((\neg State-Idle localState output-fun trans-fun (localState (*s* *t2*))). *t2* \mathcal{U} *t1* [0..]
 \oplus *t0*.
 (*output-fun* (*s* *t1*) = *m* \wedge
 (State-Idle localState output-fun trans-fun (localState (*s* *t1*)) \vee
 (\bigcirc *t3* *t1* [0..].
 ((*output-fun* (*s* *t4*) = ε . *t4* \mathcal{U} *t5* ([0..] \oplus *t3*).
 (*output-fun* (*s* *t5*) = ε \wedge State-Idle localState output-fun trans-fun
 (localState (*s* *t5*)))))))))))))
 apply (case-tac *m* = ε)
 apply (simp add: *i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-State-Idle-conv*)
 apply (simp add: *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv*)
 done

Sufficient conditions for output messages.

thm *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv2'*

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 \neq ε ;
t0 = *t* * *k*;
s = *i-Exec-Comp-Stream* trans-fun (input \odot_i *k*) *c*;
 (\diamond *t1* [0.., *k* - *Suc* 0] \oplus *t0*. (
 output-fun (*s* *t1*) = *m* \wedge State-Idle localState output-fun trans-fun (localState
 (*s* *t1*))))]] \implies
i-Exec-Comp-Stream-Acc-Output *k* output-fun trans-fun input *c* *t* = *m*

by (blast intro: *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv2'* [THEN
 iffD2])

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 \neq ε ;
t0 = *t* * *k*;

$s = i\text{-Exec-Comp-Stream trans-fun (input } \odot_i k) c;$
 $\diamond t1 [0.., k - \text{Suc } 0] \oplus t0. ($
 $\quad ((\text{output-fun (s t1) = m}) \wedge$
 $\quad (\bigcirc t2 t1 [0..].$
 $\quad \quad ((\text{output-fun (s t3) = } \varepsilon. t3 \mathcal{U} t4 ([0..] \oplus t2).$
 $\quad \quad \quad (\text{output-fun (s t4) = } \varepsilon \wedge \text{State-Idle localState output-fun trans-fun}$
 $\quad \quad \quad \quad (\text{localState (s t4))))))))) \mathbb{I} \implies$
 $i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c \text{ t} = m$
by (blast intro: $i\text{-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv2}$ [THEN iffD2])

thm $i\text{-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv}'$

lemma $i\text{-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-iUntil-imp1}:$

$\mathbb{I} \text{Suc } 0 < k;$
 $\text{State-Idle localState output-fun trans-fun (}$
 $\quad i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input } c \text{ t);}$
 $m \neq \varepsilon;$
 $t0 = t * k;$
 $s = i\text{-Exec-Comp-Stream trans-fun (input } \odot_i k) c;$
 $(\neg \text{State-Idle localState output-fun trans-fun (localState (s t2))). t2 \mathcal{U} t1 [0..]$
 $\oplus t0.$
 $(\text{output-fun (s t1) = m} \wedge \text{State-Idle localState output-fun trans-fun (localState}$
 $\quad (\text{s t1}))) \mathbb{I} \implies$
 $i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c \text{ t} = m$
by (blast intro: $i\text{-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv}$ [THEN iffD2])

lemma $i\text{-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-iUntil-imp2}:$

$\mathbb{I} \text{Suc } 0 < k;$
 $\text{State-Idle localState output-fun trans-fun (}$
 $\quad i\text{-Exec-Comp-Stream-Acc-LocalState } k \text{ localState trans-fun input } c \text{ t);}$
 $m \neq \varepsilon;$
 $t0 = t * k;$
 $s = i\text{-Exec-Comp-Stream trans-fun (input } \odot_i k) c;$
 $(\neg \text{State-Idle localState output-fun trans-fun (localState (s t2))). t2 \mathcal{U} t1 [0..]$
 $\oplus t0.$
 $(\text{output-fun (s t1) = m} \wedge$
 $\quad (\bigcirc t3 t1 [0..].$
 $\quad \quad ((\text{output-fun (s t4) = } \varepsilon. t4 \mathcal{U} t5 ([0..] \oplus t3).$
 $\quad \quad \quad (\text{output-fun (s t5) = } \varepsilon \wedge \text{State-Idle localState output-fun trans-fun (localState}$
 $\quad \quad \quad \quad (\text{s t5))))))))) \mathbb{I} \implies$
 $i\text{-Exec-Comp-Stream-Acc-Output } k \text{ output-fun trans-fun input } c \text{ t} = m$
by (blast intro: $i\text{-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv}$ [THEN iffD2])

List of selected lemmas about output of accelerated components.

thm $i\text{-Exec-Comp-Stream-Acc-Output--eq-NoMsg-iAll-conv}$

thm $i\text{-Exec-Comp-Stream-Acc-Output--eq-Msg-iEx-iAll-cut-greater-conv}$

thm $i\text{-Exec-Comp-Stream-Acc-Output--eq-Msg-iSince-conv}$

thm $i\text{-Exec-Comp-Stream-Acc-Output--eq-iAll-iSince-conv}$

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