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

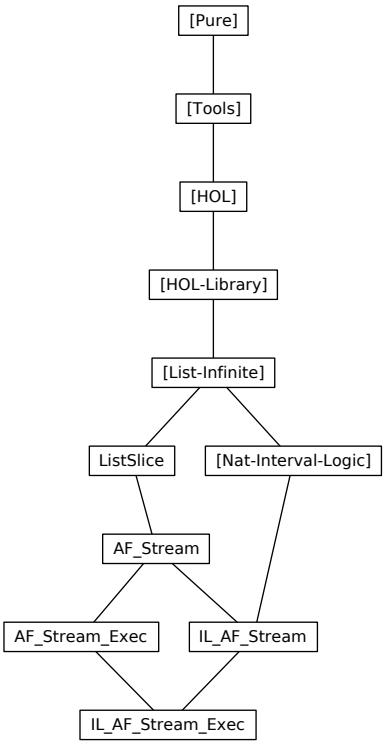
David Trachtenherz

May 26, 2024

Abstract

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

Contents


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

1 Additional definitions and results for lists

theory *ListSlice*
imports *List−Infinite.ListInf*
begin

1.1 Slicing lists into lists of lists

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

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

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

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

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

definition *ilist-unslice* :: *'a list ilist ⇒ 'a ilist*
  where *ilist-unslice f ≡ λn. f (n div length (f 0)) ! (n mod length (f 0))*

lemma *list-slice-aux-length*: *⋀xs. length (list-slice-aux xs k n) = n*
by (*induct n*, *simp+*)

lemma *list-slice-aux-nth*:
*⋀m xs. m < n ⟹ (list-slice-aux xs k n) ! m = (xs ↑ (m * k)) ↓ 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-slice xs 0 = []*
by (*simp add*: *list-slice-def*)

lemma *list-slice-1*: *list-slice xs (Suc 0) = map (λx. [x]) xs*
by (*fastforce simp*: *list-eq-iff list-slice-def list-slice-aux-nth list-slice-aux-length*)

lemma *list-slice-less*: *length xs* $<$ *k* $\Longrightarrow$ *list-slice xs k* $=$ []
by (*simp add*: *list-slice-def*)

lemma *list-slice-Nil*: *list-slice* [] *k* $=$ []
by (*simp add*: *list-slice-def*)

lemma *list-slice-nth*:
m $<$ *length xs div k* $\Longrightarrow$ *list-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* $\Longrightarrow$ *length* ((*list-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

lemma *list-slice-nth-eq-sublist-list*:
m $<$ *length xs div k* $\Longrightarrow$ *list-slice xs k* ! *m* $=$ *sublist-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* ⟧ $\Longrightarrow$
(*list-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* $\Longrightarrow$
(*list-slice xs k*) ! (*n div k*) ! (*n mod k*) $=$ *xs* ! *n*
apply (*case-tac k* $=$ *0*, *simp*)
apply (*simp add*: *list-slice-nth-nth div-less-conv*)
done

lemma *list-slice-eq-list-slice-take*:
list-slice (*xs* $\downarrow$ (*length xs div k* $*$ *k*)) *k* $=$ *list-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* $\leq$ *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*)
by (*simp add*: *less-diff-conv*)

lemma *list-slice-append-mult*:
⋀*xs*. *length xs = m * k ⟹*
list-slice (*xs* @ *ys*) *k* = *list-slice xs k* @ *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*:
length xs mod k = 0 ⟹
list-slice (*xs* @ *ys*) *k* = *list-slice xs k* @ *list-slice ys k*
by (*auto intro*: *list-slice-append-mult elim*!: *dvdE*)

lemma *list-slice-div-eq-1*[*rule-format*]:
length xs div k = Suc 0 ⟹ list-slice xs k = [*take k xs*]
by (*simp add*: *list-slice-def*)

lemma *list-slice-div-eq-Suc*[*rule-format*]:
length xs div k = Suc n ⟹
list-slice xs k = *list-slice* (*xs* ↓ (*n * k*)) *k* @ [*xs* ↑ (*n * k*) ↓ *k*]
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*=λ*x*. *x − Suc 0*], *drule sym*)
apply (*simp add*: *diff-mult-distrib div-mult-cancel*)
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

lemma *list-slice2-mod-0*:
length xs mod k = 0 ⟹ list-slice2 xs k = list-slice xs k
by (*simp add*: *list-slice2-def*)

lemma *list-slice2-mod-gr0*:
*0 < length xs mod k ⟹ list-slice2 xs k = list-slice xs k @ [xs ↑ (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 (λx. [x]) xs*
by (*simp add*: *list-slice2-def list-slice-1*)

lemma *list-slice2-le*:
length xs ≤ k ⟹ 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 ⟹ 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* ⟧ ⟹
*list-slice2 xs k ! m = xs ↑ (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* ⟧ ⟹
*list-slice2 xs k ! m = xs ↑ (m * k) ↓ 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 ⟹ 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*:
⟦ *length xs mod k > 0*; *m = length xs div k* ⟧ ⟹

length (*list-slice2 xs k* ! *m*) = *length xs mod k*
by (*simp add*: *list-slice2-def list-slice-length nth-append minus-div-mult-eq-mod* [*symmetric*])

lemma *list-slice2-nth-nth-eq1*:
⟦ *m* < *length xs div k*; *n* < *k* ⟧ ⟹
(*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*:
⟦ *m* = *length xs div k*; *n* < *length xs mod k* ⟧ ⟹
(*list-slice2 xs k*) ! *m* ! *n* = *xs* ! (*m* ∗ *k* + *n*)
by (*simp add*: *mult.commute*[*of* - *k*] *minus-mod-eq-mult-div* [*symmetric*] *list-slice2-last*)

lemma *list-slice2-nth-nth-rev*:
n < *length xs* ⟹ (*list-slice2 xs k*) ! (*n div k*) ! (*n 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*)
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*:
length xs = *m* ∗ *k* ⟹
list-slice2 (*xs* @ *ys*) *k* = *list-slice2 xs k* @ *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*:
length xs mod k = *0* ⟹
list-slice2 (*xs* @ *ys*) *k* = *list-slice2 xs k* @ *list-slice2 ys k*
by (*auto intro*: *list-slice2-append-mult elim*!: *dvdE*)

lemma *ilist-slice-nth*:
(*ilist-slice f k*) *m* = *map f* [*m* ∗ *k*..<*Suc m* ∗ *k*]
by (*simp add*: *ilist-slice-def*)

lemma *ilist-slice-nth-length*: *length* ((*ilist-slice f k*) *m*) = *k*

by (*simp add*: *ilist-slice-def*)

lemma *ilist-slice-nth-nth*:
n < *k* ⟹ (*ilist-slice f k*) *m* ! *n* = *f* (*m* ∗ *k* + *n*)
by (*simp add*: *ilist-slice-def*)

lemma *ilist-slice-nth-nth-rev*:
0 < *k* ⟹ (*ilist-slice f k*) (*n div k*) ! (*n mod k*) = *f n*
by (*simp add*: *ilist-slice-nth-nth*)

lemma *list-slice-concat*:
concat (*list-slice xs k*) = *xs* ↓ (*length xs div k* ∗ *k*)
(**is** *?P xs k*)
apply (*case-tac k* = *0*)
apply (*simp add*: *list-slice-0*)
apply *simp*
apply (*subgoal-tac* ⋀*m*. ∀ *xs*. *length xs div k* = *m* ⟶ *?P xs k*, *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* ≤ *length xs*)
prefer *2*
apply (*simp add*: *le-less-div-conv*[*symmetric*])
apply (*simp add*: *list-slice-div-eq-Suc*)
apply (*drule-tac x*=*xs* ↓ (*n* ∗ *k*) **in** *spec*)
apply (*simp add*: *min-eqR*)
apply (*simp add*: *take-add*)
done

lemma *list-slice-unslice-mult*:
length xs = *m* ∗ *k* ⟹ *concat* (*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* ⟹ *ilist-unslice* (*ilist-slice f k*) = *f*
by (*simp add*: *ilist-unslice-def ilist-slice-nth-length ilist-slice-nth-nth*)

lemma *i-take-ilist-slice-eq-list-slice*:
0 < *k* ⟹ *ilist-slice f k* ⇓ *n* = *list-slice* (*f* ⇓ (*n* ∗ *k*)) *k*
apply (*simp add*: *list-eq-iff list-slice-length ilist-slice-nth list-slice-nth*)
apply (*clarify*, *rename-tac i*)
apply (*subgoal-tac k* ≤ *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-ilist-slice*:
list-slice $(f \Downarrow n)\ k = ilist\text{-}slice\ f\ k \Downarrow (n\ div\ k)$
apply (*case-tac* $k = 0$)
apply (*simp add*: *list-slice-0*)
apply (*simp add*: *i-take-ilist-slice-eq-list-slice*)
apply (*subst list-slice-eq-list-slice-take*[*of* $f \Downarrow n$, *symmetric*])
apply (*simp add*: *div-mult-le min-eqR*)
done

lemma *ilist-slice-i-append-mod*:
$length\ xs\ mod\ k = 0 \Longrightarrow$
$ilist\text{-}slice\ (xs \frown f)\ k = list\text{-}slice\ xs\ k \frown ilist\text{-}slice\ f\ k$
apply (*simp add*: *ilist-eq-iff ilist-slice-nth i-append-nth list-slice-length*)
apply (*clarsimp simp*: *mult.commute*[*of k*] *elim*!: *dvdE*, *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 (*fastforce simp*: *list-eq-iff i-append-nth min-eqR*)
apply (*simp add*: *ilist-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 *ilist-slice-append-mult*:
$length\ xs = m * k \Longrightarrow$
$ilist\text{-}slice\ (xs \frown f)\ k = list\text{-}slice\ xs\ k \frown ilist\text{-}slice\ f\ k$
by (*simp add*: *ilist-slice-i-append-mod*)

end

2 AutoFocus message streams

theory *AF-Stream*
imports *ListSlice*
begin

2.1 Basic definitions

2.1.1 Time-synchronous streams

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

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

Abbreviation for finite streams

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

Abbreviation for infinite streams

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

lemma *not-NoMsg-eq*: $(m \neq \varepsilon) = (\exists x.\ m = \mathsf{Msg}\ x)$
by (*case-tac m*, *simp-all*)

lemma *not-Msg-eq*: $(\forall x.\ m \neq \mathsf{Msg}\ x) = (m = \varepsilon)$
by (*case-tac m*, *simp-all*)

primrec *the-af* :: *'a message-af* ⇒ *'a*
where *the-af* (***Msg*** *x*) = *x*

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

consts *is-NoMsg* :: *'a* ⇒ *bool*

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

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

end

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

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

end

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

begin

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

end

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

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

end

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

lemma *is-NoMsg-message-af-conv*: *is-NoMsg m* = (*case m of ε* ⇒ *True* | *Msg x* ⇒ *False*)
by (*case-tac m*, *simp*+)

lemma *is-NoMsg-message-af-conv2*: *is-NoMsg m* = (*m* = *ε*)
by (*case-tac m*, *simp*+)

lemma *is-Msg-message-af-conv*: *is-Msg m* = (*case m of ε* ⇒ *False* | *Msg x* ⇒ *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*.

named-theorems *is-NoMsg-defs*

declare
is-NoMsg-tuple-def[*is-NoMsg-defs*]
is-NoMsg-set-def [*is-NoMsg-defs*]
is-NoMsg-SomeRecordExample-def[*is-NoMsg-defs*]
is-Msg-def[*is-NoMsg-defs*]

lemma *not-is-NoMsg*: (¬ *is-NoMsg m*) = *is-Msg m*
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* (ε::(*nat message-af*))
by *simp*

lemma *is-NoMsg* (ε::(*nat message-af*), ε::(*nat message-af*))
by (*simp add*: *is-NoMsg-defs*)

lemma *is-NoMsg* (ε::(*nat message-af*), ε::(*nat message-af*), ε::(*nat message-af*))
by (*simp add*: *is-NoMsg-defs*)

lemma *is-Msg* (ε::(*nat message-af*), *Msg* (*1*::*nat*), ε::(*nat message-af*))
by (*simp add*: *is-NoMsg-defs*)

lemma *is-NoMsg* {ε::(*nat message-af*), ε}
by (*simp add*: *is-NoMsg-defs*)

lemma *is-Msg* {ε::(*nat message-af*), *Msg 1*}
by (*simp add*: *is-NoMsg-defs*)

lemma *is-NoMsg* (| *Field1* $= \varepsilon$, *Field2* $= \varepsilon$, *Field3* $= \varepsilon$ |)
by (*simp add*: *is-NoMsg-defs*)

lemma *is-Msg* (| *Field1* $= \varepsilon$, *Field2* $=$ *Msg 1*, *Field3* $= \varepsilon$ |)
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* $x = \varepsilon$
     then (*untime xs*)
     else (*the-af x*) # (*untime xs*))

lemma *untime-eq-filter*[*rule-format*]:
  map (λx. *Msg x*) (*untime s*) = *filter* (λx. $x \neq \varepsilon$) 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. $x \neq \varepsilon$) s)
by (*induct s*, *simp-all*)

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* (λ*x*. *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*: (*untime s* = []) = (∀ *n*<*length s*. *s* ! *n* = *ε*)
apply (*induct s*)
 apply *simp*
apply (*force simp add*: *nth.simps split*: *nat.split*)
done

lemma *untime-not-empty-conv*: (*untime s* ≠ []) = (∃ *n*<*length s*. *s* ! *n* ≠ *ε*)
by (*simp add*: *untime-empty-conv*)

corollary *untime-empty-imp-NoMsg*[*rule-format*]:
  ⟦ *untime s* = []; *n* < *length s* ⟧ ⟹ *s* ! *n* = *ε*
by (*rule untime-empty-conv*[*THEN iffD1*, *rule-format*])

lemma *untime-nth-eq-filter*:
  n < *untime-length s* ⟹
  Msg (*untime s* ! *n*) = (*filter* (λ*x*. *x* ≠ *ε*) *s*) ! *n*
by (*simp add*: *untime-eq-filter*[*symmetric*] *untime-length-def*)

corollary *untime-nth-eq-filter2*:
  n < *untime-length s* ⟹
  untime s ! *n* = *the-af* ((*filter* (λ*x*. *x* ≠ *ε*) *s*) ! *n*)
by (*simp add*: *untime-length-def untime-nth-eq-filter*[*symmetric*])

lemma *untime-hd-eq-filter-hd*:
untime $s \neq [] \Longrightarrow$
Msg (*hd* (*untime* s)) = *hd* (*filter* ($\lambda x.\ x \neq \varepsilon$) s)
by (*simp add*: *untime-eq-filter*[*symmetric*] *hd-eq-first*[*symmetric*])

corollary *untime-hd-eq-filter-hd2*:
untime $s \neq [] \Longrightarrow$
hd (*untime* s) = *the-af* (*hd* (*filter* ($\lambda x.\ x \neq \varepsilon$) s))
by (*simp add*: *untime-hd-eq-filter-hd*[*symmetric*])

lemma *untime-last-eq-filter-last*:
untime $s \neq [] \Longrightarrow$
Msg (*last* (*untime* s)) = *last* (*filter* ($\lambda x.\ x \neq \varepsilon$) s)
by (*simp add*: *untime-eq-filter*[*symmetric*] *last-nth*)

corollary *untime-last-eq-filter-last2*:
untime $s \neq [] \Longrightarrow$
last (*untime* s) = *the-af* (*last* (*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* ⇒ *nat* ⇒ *'a fstream-af* (**infixl** $\odot_f$ *100*)
where
f-expand-Nil: [] $\odot_f$ k = []
| *f-expand-Cons*: (x # xs) $\odot_f$ k =
(*if* 0 < k *then* x # $\varepsilon^{k - Suc\ 0}$ @ (xs $\odot_f$ k) *else* [])

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

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

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

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

lemma *length-f-expand-Suc*[*simp*]: *length* (*f-expand-Suc xs k*) = *length xs* ∗ *Suc k*
by (*induct xs*, *simp*+)

lemma *i-expand-if*:
f ⊙$_i$ *k* = (*if k* = *0 then* (λ*n*. ε) *else*
(λ*n*. *if n mod k* = *0 then f* (*n div k*) *else* ε))
by (*simp add*: *i-expand-def ilist-eq-iff*)

lemma *f-expand-one*: *0* < *k* ⟹ [*a*] ⊙$_f$ *k* = *a* # ε$^{k - Suc\ 0}$
by *simp*

lemma *f-expand-0*[*simp*]: *xs* ⊙$_f$ *0* = []
by (*induct xs*, *simp*+)
corollary *f-expand-0-is-zero-element*: *xs* ⊙$_f$ *0* = *ys* ⊙$_f$ *0*
by *simp*
lemma *i-expand-0*[*simp*]: *f* ⊙$_i$ *0* = (λ*n*. ε)
by (*simp add*: *i-expand-def*)
corollary *i-expand-0-is-zero-element*: *f* ⊙$_i$ *0* = *g* ⊙$_i$ *0*
by *simp*

lemma *f-expand-gr0-f-expand-Suc*: *0* < *k* ⟹ *xs* ⊙$_f$ *k* = *f-expand-Suc xs* (*k* − *Suc 0*)
by (*induct xs*, *simp*+)
lemma *i-expand-gr0-i-expand-Suc*: *0* < *k* ⟹ *f* ⊙$_i$ *k* = *i-expand-Suc f* (*k* − *Suc 0*)
by (*simp add*: *i-expand-def i-expand-Suc-def ilist-eq-iff*)
lemma *i-expand-gr0*:
0 < *k* ⟹ *f* ⊙$_i$ *k* = (λ*n*. *if n mod k* = *0 then f* (*n div k*) *else* ε)
by (*simp add*: *i-expand-if*)
lemma *f-expand-1*[*simp*]: *xs* ⊙$_f$ *Suc 0* = *xs*
by (*induct xs*, *simp*+)
lemma *i-expand-1*[*simp*]: *f* ⊙$_i$ *Suc 0* = *f*
by (*simp add*: *i-expand-gr0*)

lemma *f-expand-length*[*simp*]: *length* (*xs* ⊙$_f$ *k*) = *length xs* ∗ *k*
apply (*case-tac k*, *simp*)
apply (*simp add*: *f-expand-gr0-f-expand-Suc*)
done

lemma *f-expand-empty-conv*: (*xs* ⊙$_f$ *k* = []) = (*xs* = [] ∨ *k* = *0*)
by (*simp add*: *length-0-conv*[*symmetric*] *del*: *length-0-conv*)
lemma *f-expand-not-empty-conv*: (*xs* ⊙$_f$ *k* ≠ []) = (*xs* ≠ [] ∧ *0* < *k*)
by (*simp add*: *f-expand-empty-conv*)

lemma *f-expand-Cons*:
0 < *k* ⟹ (*x* # *xs*) ⊙$_f$ *k* = *x* # ε$^{k - Suc\ 0}$ @ (*xs* ⊙$_f$ *k*)
by *simp*
lemma *f-expand-append*[*simp*]: ⋀*ys*. (*xs* @ *ys*) ⊙$_f$ *k* = (*xs* ⊙$_f$ *k*) @ (*ys* ⊙$_f$ *k*)
apply (*case-tac k* = *0*, *simp*)

apply (*induct xs*, *simp*+)
done

lemma *f-expand-snoc*:
  $0 < k \Longrightarrow (xs \mathbin{@} [x]) \odot_f k = xs \odot_f k \mathbin{@} x \# replicate\ (k - Suc\ 0)\ \varepsilon$
by *simp*

lemma *f-expand-nth-mult*: $\bigwedge n.$
  $[\![\ n < length\ xs;\ 0 < k\ ]\!] \Longrightarrow (xs \odot_f k)\ !\ (n * k) = xs\ !\ n$
apply (*induct xs*)
 apply *simp*
apply (*case-tac n*, *simp*)
apply (*simp add*: *nth-append append-Cons*[*symmetric*] *del*: *append-Cons*)
done

lemma *i-expand-nth-mult*: $0 < k \Longrightarrow (f \odot_i k)\ (n * k) = f\ n$
by (*simp add*: *i-expand-gr0*)

lemma *f-expand-nth-if*: $\bigwedge n.$
  $n < length\ xs * k \Longrightarrow$
  $(xs \odot_f k)\ !\ n = (if\ n\ mod\ k = 0\ then\ xs\ !\ (n\ div\ k)\ else\ \varepsilon)$
apply (*case-tac* $k = 0$, *simp*)
apply (*simp*, *intro conjI impI*)
 apply (*clarsimp simp*: *f-expand-nth-mult mult.commute*[*of k*] *elim*!: *dvdE*)
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 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*:
  $[\![\ n < length\ xs * k;\ n\ mod\ k = 0\ ]\!] \Longrightarrow (xs \odot_f k)\ !\ n = xs\ !\ (n\ div\ k)$
by (*simp add*: *f-expand-nth-if*)

corollary *f-expand-nth-mod-neq-0*:
  $[\![\ n < length\ xs * k;\ 0 < n\ mod\ k\ ]\!] \Longrightarrow (xs \odot_f k)\ !\ n = \varepsilon$
by (*simp add*: *f-expand-nth-if*)

lemma *f-expand-nth-0-upto-k-minus-1-if*:
  $[\![\ t < length\ xs;\ n = t * k + i;\ i < k\ ]\!] \Longrightarrow$
  $(xs \odot_f k)\ !\ n = (if\ i = 0\ then\ xs\ !\ t\ else\ \varepsilon)$
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* $i < length\ xs * k$)
prefer *2*
apply (*rule-tac* $y{=}n * k$ **in** *order-le-less-trans*, *simp+*)
apply (*clarsimp simp*: *f-expand-nth-if elim*!: *dvdE*)
done

lemma *f-expand-take-mod*:
$n\ mod\ k = 0 \Longrightarrow xs \odot_f k \downarrow n = xs \downarrow (n\ div\ k) \odot_f k$
by (*clarsimp simp*: *mult.commute*[*of k*] *f-expand-take-mult elim*!: *dvdE*)

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*:
$n\ mod\ k = 0 \Longrightarrow xs \odot_f k \uparrow n = xs \uparrow (n\ div\ k) \odot_f k$
by (*clarsimp simp*: *mult.commute*[*of k*] *f-expand-drop-mult elim*!: *dvdE*)

lemma *f-expand-take-mult-Suc*:
$[\![\ n < length\ xs;\ i < k\ ]\!] \Longrightarrow$
$xs \odot_f k \downarrow (n * k + Suc\ i) = (xs \downarrow n) \odot_f k\ @\ (xs\ !\ n\ \#\ \varepsilon^{i})$
apply (*subgoal-tac* $n * k + Suc\ i \leq length\ xs * k$)
prefer *2*
apply (*drule Suc-leI*[*of n*])
apply (*drule mult-le-mono1*[*of Suc n* - *k*])
apply *simp*
apply (*clarsimp simp*: *list-eq-iff min-eqR nth-append f-expand-nth-if min-def nth-Cons'*
elim!: *dvdE*)
apply (*simp add*: *mult.commute*[*of k*] *linorder-not-less*)
apply (*drule-tac* $n{=}ka$ **in** *le-neq-implies-less*, *simp+*)
apply (*drule-tac* $n{=}ka$ **in** *Suc-leI*)
apply (*drule-tac* $j{=}ka$ **in** *mult-le-mono1*[*of* - - *k*])
apply *simp*
done

lemma *f-expand-take-Suc*:
$n < length\ xs * k \Longrightarrow$
$xs \odot_f k \downarrow Suc\ n = (xs \downarrow (n\ div\ k)) \odot_f k\ @\ (xs\ !\ (n\ div\ k)\ \#\ \varepsilon^{n\ mod\ k})$
apply (*case-tac* $k = 0$, *simp*)
apply (*insert f-expand-take-mult-Suc*[*of n div k xs n mod k k*])

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

lemma *i-expand-nth-if*:
  $0 < k \Longrightarrow (f \odot_i k)\ n = (\mathit{if}\ n\ \mathit{mod}\ k = 0\ \mathit{then}\ f\ (n\ \mathit{div}\ k)\ \mathit{else}\ \varepsilon)$
by (*simp add*: *i-expand-gr0*)
corollary *i-expand-nth-mod-eq-0*:
  $[\![\ 0 < k;\ n\ \mathit{mod}\ k = 0\ ]\!] \Longrightarrow (f \odot_i k)\ n = f\ (n\ \mathit{div}\ k)$
by (*simp add*: *i-expand-gr0*)
corollary *i-expand-nth-mod-neq-0*:
  $0 < n\ \mathit{mod}\ k \Longrightarrow (f \odot_i k)\ n = \varepsilon$
apply (*case-tac* $k = 0$, *simp*)
apply (*simp add*: *i-expand-gr0*)
done

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

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 elim*!: *dvdE*)
done

lemma *i-expand-i-take-mod*:
  $n\ \mathit{mod}\ k = 0 \Longrightarrow f \odot_i k \Downarrow n = f \Downarrow (n\ \mathit{div}\ k) \odot_f k$
  by (*clarsimp simp*: *mult.commute*[*of k*] *i-expand-i-take-mult elim*!: *dvdE*)

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*:
  $n\ \mathit{mod}\ k = 0 \Longrightarrow f \odot_i k \Uparrow n = f \Uparrow (n\ \mathit{div}\ k) \odot_i k$
  by (*clarsimp simp*: *mult.commute*[*of k*] *i-expand-i-drop-mult elim*!: *dvdE*)

lemma *i-expand-i-take-mult-Suc*:
  $i < k \Longrightarrow f \odot_i k \Downarrow (n * k + \mathit{Suc}\ i) = (f \Downarrow n) \odot_f k\ @\ (f\ n\ \#\ \varepsilon^i)$
apply (*clarsimp simp*: *list-eq-iff*, *rename-tac i'*)
apply (*clarsimp simp*: *i-expand-nth-if f-expand-nth-if nth-append nth-Cons' elim*!: *dvdE*)
apply (*simp add*: *linorder-not-less mult.commute*[*of k*])
apply (*drule-tac n=ka* **in** *le-neq-implies-less*, *simp+*)
apply (*drule-tac n=ka* **in** *Suc-leI*)
apply (*drule-tac j=ka* **in** *mult-le-mono1*[*of - - k*])
apply *simp*
done

lemma *i-expand-i-take-Suc*:
$0 < k \Longrightarrow f \odot_i k \Downarrow Suc\ n = (f \Downarrow (n\ div\ k)) \odot_f k\ @\ (f\ (n\ div\ k)\ \#\ \varepsilon^{n\ mod\ k})$
apply (*insert i-expand-i-take-mult-Suc*[*of n mod k k n div k f*])
apply *simp*
done

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

lemma *f-expand-nth-interval-eq-replicate-NoMsg*:
$[\![\ 0 < k;\ t * k < t1;\ t1 \leq t2;\ t2 \leq t * k + k;\ t2 \leq length\ xs * k]\!] \Longrightarrow$
$xs \odot_f k \downarrow t2 \uparrow t1 = \varepsilon^{t2\ -\ t1}$
apply (*clarsimp simp*: *list-eq-iff min-eqR f-expand-nth-if elim*!: *dvdE*, *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*)
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*]:
$[\![\ 0 < k;\ t * k \leq t1;\ t1 \leq t * k + k - Suc\ 0\ ]\!] \Longrightarrow$
$f \odot_i k \Downarrow Suc\ t1 \uparrow (t * k) = f\ t\ \#\ \varepsilon^{t1\ -\ t\ *\ k}$
by (*simp add*: *list-eq-iff Suc-diff-le i-expand-nth-if nth-Cons'*)

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

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 (*fastforce 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*])

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*:
$0 < k \Longrightarrow (xs \odot_f k = ys \odot_f k) = (xs = ys)$
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*:
$0 < k \Longrightarrow (f \odot_i k = g \odot_i k) = (f = g)$
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'*:

$(xs' \odot_f\ k = xs) =$
$(length\ xs' * k = length\ xs \wedge$
$(\forall i{<}length\ xs.\ xs\ !\ i = (if\ i\ mod\ k = 0\ then\ xs'\ !\ (i\ div\ k)\ else\ \varepsilon)))$
by (*fastforce simp*: *list-eq-iff f-expand-nth-if*)

lemma *i-expand-eq-conv'*:
$0 < k \Longrightarrow (f' \odot_i\ k = f) =$
$(\forall i.\ f\ i = (if\ i\ mod\ k = 0\ then\ f'\ (i\ div\ k)\ else\ \varepsilon))$
by (*fastforce simp*: *ilist-eq-iff i-expand-nth-if*)

2.2.2 Aggregating lists

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

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

lemma *f-aggregate-0*[*simp*]: *f-aggregate xs 0 ag* = []
by (*simp add*: *f-aggregate-def list-slice-0*)

lemma *f-aggregate-1*:
(⋀*x*. *ag* [*x*] = *x*) ⟹
f-aggregate xs (*Suc 0*) *ag* = *xs*
by (*simp add*: *list-eq-iff f-aggregate-def list-slice-1*)

lemma *f-aggregate-Nil*[*simp*]: *f-aggregate* [] *k ag* = []
by (*simp add*: *f-aggregate-def list-slice-Nil*)

lemma *f-aggregate-length*[*simp*]: *length* (*f-aggregate xs k ag*) = *length xs div k*
by (*simp add*: *f-aggregate-def list-slice-length*)

lemma *f-aggregate-empty-conv*:
$0 < k \Longrightarrow$ (*f-aggregate xs k ag* = []) = (*length xs* < *k*)
by (*simp add*: *length-0-conv*[*symmetric*] *div-eq-0-conv' del*: *length-0-conv*)

lemma *f-aggregate-one*:
⟦ $0 < k$; *length xs* = *k* ⟧ ⟹ *f-aggregate xs k ag* = [*ag xs*]
by (*simp add*: *f-aggregate-def list-slice-def*)

lemma *f-aggregate-Cons*:
⟦ $0 < k$; *length xs* = *k* ⟧ ⟹
f-aggregate (*xs* @ *ys*) *k ag* = *ag xs* # (*f-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*:
  $n < length\ xs\ div\ k \Longrightarrow$
  $(f\text{-}aggregate\ xs\ k\ ag)\ !\ n = ag\ (xs \uparrow (n * k) \downarrow k)$
by (*simp add*: *f-aggregate-def list-slice-length list-slice-nth*)

lemma *f-aggregate-nth-eq-sublist-list*:
  $n < length\ xs\ div\ k \Longrightarrow$
  $(f\text{-}aggregate\ xs\ k\ ag)\ !\ n = ag\ (sublist\text{-}list\ xs\ [n * k..<n * k + k])$
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*:
  $\bigwedge xs\ m.\ [\![\ n < length\ xs\ div\ k;\ n < m\ div\ k\ ]\!] \Longrightarrow$
  $f\text{-}aggregate\ (xs \downarrow m)\ k\ ag\ !\ n = f\text{-}aggregate\ xs\ k\ ag\ !\ n$
apply (*simp add*: *f-aggregate-nth drop-take*)
apply (*drule-tac n=m* **in** *less-div-imp-mult-add-divisor-le*)
apply (*simp add*: *min-eqL*)
done

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

lemma *f-aggregate-append-mod*:
  $length\ xs\ mod\ k = 0 \Longrightarrow$
  $f\text{-}aggregate\ (xs\ @\ ys)\ k\ ag =$
  $f\text{-}aggregate\ xs\ k\ ag\ @\ f\text{-}aggregate\ ys\ k\ ag$
by (*simp add*: *f-aggregate-def list-slice-append-mod*)
lemma *f-aggregate-append-mult*:
  $length\ xs = m * k \Longrightarrow$
  $f\text{-}aggregate\ (xs\ @\ ys)\ k\ ag =$
  $f\text{-}aggregate\ xs\ k\ ag\ @\ f\text{-}aggregate\ ys\ k\ ag$
by (*simp add*: *f-aggregate-append-mod*)

lemma *f-aggregate-snoc*:
  $[\![\ 0 < k;\ length\ ys = k;\ length\ xs\ mod\ k = 0\ ]\!] \Longrightarrow$
  $f\text{-}aggregate\ (xs\ @\ ys)\ k\ ag = f\text{-}aggregate\ xs\ k\ ag\ @\ [ag\ ys]$
by (*simp add*: *f-aggregate-append-mod f-aggregate-one*)

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

lemma *f-aggregate-drop-mult*:
f-aggregate (*xs* ↑ (*n* ∗ *k*)) *k ag* = *f-aggregate xs k ag* ↑ *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 mod k = *0* ⟹ *f-aggregate* (*xs* ↑ *n*) *k ag* = *f-aggregate xs k ag* ↑ (*n div k*)
by (*clarsimp simp*: *mult.commute*[*of k*] *f-aggregate-drop-mult elim*!: *dvdE*)

lemma *f-aggregate-assoc*:
(⋀*xs*. *length xs mod a* = *0* ⟹ *ag* (*f-aggregate xs a ag*) = *ag xs*) ⟹
f-aggregate (*f-aggregate xs a ag*) *b ag* = *f-aggregate xs* (*a* ∗ *b*) *ag*
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*:
⟦ ⋀*xs*. *length xs mod a* = *0* ⟹ *ag* (*f-aggregate xs a ag*) = *ag xs*;
⋀*xs*. *length xs mod b* = *0* ⟹ *ag* (*f-aggregate xs b ag*) = *ag xs* ⟧ ⟹
f-aggregate (*f-aggregate xs a ag*) *b ag* = *f-aggregate* (*f-aggregate xs b ag*) *a ag*
by (*simp add*: *f-aggregate-assoc mult.commute*[*of* - *b*])

lemma *i-aggregate-0*[*simp*]: *i-aggregate f 0 ag* = (λ*x*. *ag* [])
by (*simp add*: *i-aggregate-def*)
lemma *i-aggregate-1*: (⋀*x*. *ag* [*x*] = *x*) ⟹ *i-aggregate f* (*Suc 0*) *ag* = *f*
by (*simp add*: *i-aggregate-def i-take-first*)
lemma *i-aggregate-nth*: *i-aggregate f k ag n* = *ag* (*f* ⇑ (*n* ∗ *k*) ⇓ *k*)
by (*simp add*: *i-aggregate-def*)
lemma *i-aggregate-hd*: *i-aggregate f k ag 0* = *ag* (*f* ⇓ *k*)
by (*simp add*: *i-aggregate-nth*)

lemma *i-aggregate-nth-eq-map*: *i-aggregate f k ag n* = *ag* (*map f* [*n* ∗ *k*..<*n* ∗ *k* + *k*])
by (*simp add*: *i-aggregate-nth i-take-drop-eq-map*)

lemma *i-aggregate-i-append-mod*:
length xs mod k = *0* ⟹
i-aggregate (*xs* ⌢ *f*) *k ag* = *f-aggregate xs k ag* ⌢ *i-aggregate f k ag*
apply (*clarsimp simp*: *ilist-eq-iff i-aggregate-nth i-append-nth f-aggregate-nth mult.commute*[*of k*] *diff-mult-distrib elim*!: *dvdE*, *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*:
length xs = *m* ∗ *k* ⟹
i-aggregate (*xs* ⌢ *f*) *k ag* = *f-aggregate xs k ag* ⌢ *i-aggregate f k ag*
by (*rule i-aggregate-i-append-mod*, *simp*)

lemma *i-aggregate-Cons*:
⟦ *0* < *k*; *length xs* = *k* ⟧ ⟹
i-aggregate (*xs* ⌢ *f*) *k ag* = [*ag xs*] ⌢ (*i-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 div k* ⟹ *f-aggregate* (*f* ⇓ *m*) *k ag* ! *n* = *i-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* ⇓ *n*) *k ag* = *i-aggregate f k ag* ⇓ (*n div k*)
by (*simp add*: *list-eq-iff i-aggregate-take-nth*)

lemma *i-aggregate-i-take-mult*:
0 < *k* ⟹ *f-aggregate* (*f* ⇓ (*n* ∗ *k*)) *k ag* = *i-aggregate f k ag* ⇓ *n*
by (*simp add*: *i-aggregate-i-take*)

lemma *i-aggregate-i-drop-mult*:
i-aggregate (*f* ⇑ (*n* ∗ *k*)) *k ag* = *i-aggregate f k ag* ⇑ *n*
by (*simp add*: *ilist-eq-iff i-aggregate-nth add-mult-distrib*)

lemma *i-aggregate-i-drop-mod*:
n mod k = *0* ⟹
i-aggregate (*f* ⇑ *n*) *k ag* = *i-aggregate f k ag* ⇑ (*n div k*)
by (*clarsimp simp*: *mult.commute*[*of k*] *i-aggregate-i-drop-mult ilist-eq-iff elim*!: *dvdE*)

lemma *i-aggregate-assoc*:
⟦ *0* < *a*; *0* < *b*;
⋀*xs*. *length xs mod a* = *0* ⟹ *ag* (*f-aggregate xs a ag*) = *ag xs* ⟧ ⟹
i-aggregate (*i-aggregate f a ag*) *b ag* = *i-aggregate f* (*a* ∗ *b*) *ag*
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*:
⟦ *0* < *a*; *0* < *b*;
⋀*xs*. *length xs mod a* = *0* ⟹ *ag* (*f-aggregate xs a ag*) = *ag xs*;
⋀*xs*. *length xs mod b* = *0* ⟹ *ag* (*f-aggregate xs b ag*) = *ag xs* ⟧ ⟹
i-aggregate (*i-aggregate xs a ag*) *b ag* = *i-aggregate* (*i-aggregate xs b ag*) *a ag*
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* ⇒ *'a message-af*
where
last-message [] = *ε*
| *last-message* (*x* # *xs*) = (*if last-message xs* = *ε then x else last-message xs*)

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

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

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

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

lemma *last-message-Nil*: *last-message* [] = ε
by *simp*

lemma *last-message-one*: *last-message* [*m*] = *m*
by *simp*

lemma *last-message-replicate*: *0* < *n* $\Longrightarrow$ *last-message* (m^n) = *m*
apply (*induct n*, *simp*)
apply (*case-tac n*, *simp*+)
done

lemma *last-message-replicate-NoMsg*: *last-message* (ε^n) = ε
apply (*case-tac n* = *0*, *simp*)
apply (*simp add*: *last-message-replicate*)
done

lemma *last-message-Cons-NoMsg*: *last-message* (ε # *xs*) = *last-message xs*
by *simp*

lemma *last-message-append-one*:
last-message (*xs* @ [*m*]) = (*if m* = ε *then last-message xs else m*)
apply (*induct xs*, *simp*)
apply (*case-tac m* = ε, *simp*+)
done

lemma *last-message-append*: $\bigwedge$*xs*.
last-message (*xs* @ *ys*) = (
if last-message ys = ε *then last-message xs 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 *last-message-append-replicate-NoMsg*:
last-message (*xs* @ ε^n) = *last-message xs*
by (*simp add*: *last-message-append last-message-replicate-NoMsg*)

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

lemma *last-message-NoMsg-conv*:
(*last-message xs* = ε) = ($\forall$ *i*<*length xs*. *xs* ! *i* = ε)
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-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*:
$[\![$ *i* < *length xs*; *xs* ! *i* $\neq \varepsilon$ $]\!] \Longrightarrow$ *last-message xs* $\neq \varepsilon$
by (*rule last-message-not-NoMsg-conv*[*THEN iffD2*, *OF exI*, *OF conjI*])

lemma *last-message-exists-nth*:
last-message xs $\neq \varepsilon \Longrightarrow$
$\exists i$<*length xs*. *last-message xs* = *xs* ! *i* $\wedge$ ($\forall j$<*length xs*. *i* < *j* $\longrightarrow$ *xs* ! *j* = ε)
apply (*induct xs*, *simp*)
apply (*rename-tac a xs*)
apply (*case-tac last-message xs* = ε)
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-message xs $\neq \varepsilon \Longrightarrow \exists i$<*length xs*. *last-message xs* = *xs* ! *i*
by (*blast dest*: *last-message-exists-nth*)

lemma *last-messageI2-aux*: $\bigwedge i$.
$[\![$ *i* < *length xs*; *xs* ! *i* $\neq \varepsilon$;
$\forall j$. *i* < *j* $\wedge$ *j* < *length xs* $\longrightarrow$ *xs* ! *j* = ε $]\!] \Longrightarrow$
last-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* < *length xs* $\longrightarrow$ *xs* ! *j* = ε)
prefer *2*
apply (*clarify*, *drule-tac x=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* < *length xs* $\longrightarrow$ *xs* ! *j* = ε)

prefer *2*
apply (*clarify*, *drule-tac x=Suc j* **in** *spec*)
apply *simp*
apply *simp*
done

lemma *last-messageI2*:
⟦ *i* < *length xs*; *xs* ! *i* ≠ *ε*;
⋀*j*. ⟦ *i* < *j*; *j* < *length xs* ⟧ ⟹ *xs* ! *j* = *ε* ⟧ ⟹
last-message xs = *xs* ! *i*
by (*blast intro*: *last-messageI2-aux*)

lemma *last-messageI*:
⟦ *m* ≠ *ε*; *i* < *length xs*; *xs* ! *i* = *m*;
⋀*j*. ⟦ *i* < *j*; *j* < *length xs* ⟧ ⟹ *xs* ! *j* = *ε* ⟧ ⟹
last-message xs = *m*
by (*blast intro*: *last-messageI2*)

lemma *last-message-Msg-eq-last*:
⟦ *xs* ≠ []; *last xs* ≠ *ε* ⟧ ⟹ *last-message xs* = *last xs*
by (*simp add*: *last-nth last-messageI2*)

lemma *last-message-conv*:
m ≠ *ε* ⟹
(*last-message xs* = *m*) =
(∃ *i*<*length xs*. *xs* ! *i* = *m* ∧ (∀ *j*<*length xs*. *i* < *j* ⟶ *xs* ! *j* = *ε*))
apply (*rule iffI*)
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* = *ε then* ∀ *i*<*length xs*. *xs* ! *i* = *ε*
else ∃ *i*<*length xs*. *xs* ! *i* = *m* ∧ (∀ *j*<*length xs*. *i* < *j* ⟶ *xs* ! *j* = *ε*))
by (*simp add*: *last-message-NoMsg-conv last-message-conv*)

lemma *last-message-not-NoMsg-eq-conv*:
⟦ *last-message xs* ≠ *ε*; *last-message ys* ≠ *ε* ⟧ ⟹
(*last-message xs* = *last-message ys*) =
(∃ *i j*. *i* < *length xs* ∧ *j* < *length ys* ∧ *xs* ! *i* ≠ *ε* ∧
xs ! *i* = *ys* ! *j* ∧
(∀ *n*<*length xs*. *i* < *n* ⟶ *xs* ! *n* = *ε*) ∧
(∀ *n*<*length ys*. *j* < *n* ⟶ *ys* ! *n* = *ε*))
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*)
apply (*rule last-messageI2*)
apply *simp*+
done

lemma *f-shrink-0*[*simp*]: *xs* ÷$_f$ *0* = []
by (*simp add*: *f-shrink-defs list-slice-0*)

lemma *f-shrink-1*[*simp*]: *xs* ÷$_f$ *Suc 0* = *xs*
by (*simp add*: *f-shrink-def f-aggregate-1*)

lemma *f-shrink-Nil*[*simp*]: [] ÷$_f$ *k* = []
by (*simp add*: *f-shrink-def list-slice-Nil*)

lemma *f-shrink-length*: *length* (*xs* ÷$_f$ *k*) = *length xs div k*
by (*simp add*: *f-shrink-def*)

lemma *f-shrink-empty-conv*: *0* < *k* ⟹ (*xs* ÷$_f$ *k* = []) = (*length xs* < *k*)
by (*simp add*: *f-shrink-def f-aggregate-empty-conv*)

lemma *f-shrink-Cons*:
 ⟦ *0* < *k*; *length xs* = *k* ⟧ ⟹ (*xs* @ *ys*) ÷$_f$ *k* = *last-message xs* # (*ys* ÷$_f$ *k*)
by (*simp add*: *f-shrink-def f-aggregate-Cons*)

lemma *f-shrink-one*:
 ⟦ *0* < *k*; *length xs* = *k* ⟧ ⟹ *xs* ÷$_f$ *k* = [*last-message xs*]
by (*simp add*: *f-shrink-def f-aggregate-one*)

lemma *f-shrink-eq-f-shrink-take*:
 xs ↓ (*length xs div k* ∗ *k*) ÷$_f$ *k* = *xs* ÷$_f$ *k*
by (*simp add*: *f-shrink-defs list-slice-eq-list-slice-take*)

lemma *f-shrink-nth*:
 n < *length xs div k* ⟹
 (*xs* ÷$_f$ *k*) ! *n* = *last-message* (*xs* ↑ (*n* ∗ *k*) ↓ *k*)
by (*simp add*: *f-shrink-def f-aggregate-nth*)

lemma *f-shrink-nth-eq-sublist-list*:
 n < *length xs div k* ⟹
 (*xs* ÷$_f$ *k*) ! *n* = *last-message* (*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*:
⟦ $n < length\ xs\ div\ k$; $n < m\ div\ k$ ⟧ ⟹ $(xs \downarrow m) \div_f k\ !\ n = xs \div_f k\ !\ n$
by (*simp add*: *f-shrink-def f-aggregate-take-nth*)

lemma *f-shrink-hd*:
⟦ $0 < k$; $k \leq length\ xs$ ⟧ ⟹ $hd\ (xs \div_f k) = last\text{-}message\ (xs \downarrow k)$
by (*simp add*: *f-shrink-def f-aggregate-hd*)

lemma *f-shrink-append-mod*:
$length\ xs\ mod\ k = 0$ ⟹ $(xs\ @\ ys) \div_f k = xs \div_f k\ @\ (ys \div_f k)$
by (*simp add*: *f-shrink-defs list-slice-append-mod*)

lemma *f-shrink-append-mult*:
$length\ xs = m * k$ ⟹ $(xs\ @\ ys) \div_f k = xs \div_f k\ @\ (ys \div_f k)$
by (*simp add*: *f-shrink-append-mod*)

lemma *f-shrink-snoc*:
⟦ $0 < k$; $length\ ys = k$; $length\ xs\ mod\ k = 0$ ⟧ ⟹
$(xs\ @\ ys) \div_f k = xs \div_f k\ @\ [last\text{-}message\ ys]$
by (*simp add*: *f-shrink-append-mod f-shrink-one*)

lemma *f-shrink-last-message*[*rule-format*]:
$length\ xs\ mod\ k = 0 \longrightarrow last\text{-}message\ (xs \div_f k) = last\text{-}message\ xs$
apply (*case-tac* $k = 0$, *simp*)
apply (*rule append-constant-length-induct*[*of k*])
apply *simp*
apply (*simp add*: *f-shrink-Cons last-message-append*)
done

lemma *f-shrink-replicate*: $m^n \div_f k = m^{n\ div\ k}$
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*)
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*: $0 < k$ ⟹ $xs \odot_f k \div_f k = xs$
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*]:
⟦ $\forall i{<}length\ xs.\ 0 < i\ mod\ k \longrightarrow xs\ !\ i = \varepsilon$ ⟧ ⟹

xs $\div_f$ *k* $\odot_f$ *k* = *xs* $\downarrow$ (*length xs div k* * *k*)
apply (*case-tac k* = *0*, *simp*)
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

corollary *f-expand-f-shrink-id-mod-0*:
⟦ *length xs mod k* = *0*;
⋀*i*. ⟦ *i* < *length xs*; *0* < *i mod k* ⟧ ⟹ *xs* ! *i* = ε ⟧ ⟹
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 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 mod k = *0* ⟹ *xs* $\uparrow$ *n* $\div_f$ *k* = *xs* $\div_f$ *k* $\uparrow$ (*n div k*)
by (*simp add*: *f-shrink-def f-aggregate-drop-mod*)

lemma *f-shrink-eq-conv*:
(*xs* $\div_f$ *k1* = *ys* $\div_f$ *k2*) =
(*length xs div k1* = *length ys div k2* ∧
(∀ *i*<*length xs div k1*.
last-message (*xs* $\uparrow$ (*i* * *k1*) $\downarrow$ *k1*) = *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 (*fastforce 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

lemma *f-shrink-eq-conv'*:
(*xs'* ÷$_f$ *k* = *xs*) =
(*length xs' div k* = *length xs* ∧
(∀ *i*<*length xs*.
if xs ! *i* = ε *then* (∀ *j*<*k*. *xs'* ! (*i* ∗ *k* + *j*) = ε)
else (∃ *n*<*k*. *xs'* ! (*i* ∗ *k* + *n*) = *xs* ! *i* ∧
(∀ *j*<*k*. *n* < *j* ⟶ *xs'* ! (*i* ∗ *k* + *j*) = ε))))
apply (*case-tac k* = *0*, *fastforce*)
apply (*simp add*: *list-eq-iff f-shrink-length split del*: *if-split*)
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* ÷$_f$ *a* ÷$_f$ *b* = *xs* ÷$_f$ (*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* ÷$_f$ *a* ÷$_f$ *b* = *xs* ÷$_f$ *b* ÷$_f$ *a*
by (*simp add*: *f-shrink-assoc mult.commute*[*of a*])

lemma *i-shrink-0*[*simp*]: *f* ÷$_i$ *0* = (λ*n*. ε)
by (*simp add*: *i-shrink-defs*)
lemma *i-shrink-1*[*simp*]: *f* ÷$_i$ *Suc 0* = *f*
by (*simp add*: *i-shrink-def i-aggregate-1*)
lemma *i-shrink-nth*: (*f* ÷$_i$ *k*) *n* = *last-message* (*f* ⇑ (*n* ∗ *k*) ⇓ *k*)
by (*simp add*: *i-shrink-defs*)
lemma *i-shrink-nth-eq-map*: (*f* ÷$_i$ *k*) *n* = *last-message* (*map f* [*n* ∗ *k*..<*n* ∗ *k* +
k])
by (*simp add*: *i-shrink-def i-aggregate-nth-eq-map*)
lemma *i-shrink-hd*: (*f* ÷$_i$ *k*) *0* = *last-message* (*f* ⇓ *k*)
by (*simp add*: *i-shrink-nth*)

lemma *i-shrink-i-append-mod*:
length xs mod k = *0* ⟹ (*xs* ⌢ *f*) ÷$_i$ *k* = *xs* ÷$_f$ *k* ⌢ (*f* ÷$_i$ *k*)
by (*simp add*: *f-shrink-def i-shrink-def i-aggregate-i-append-mod*)

lemma *i-shrink-i-append-mult*:
length xs = *m* ∗ *k* ⟹ (*xs* ⌢ *f*) ÷$_i$ *k* = *xs* ÷$_f$ *k* ⌢ (*f* ÷$_i$ *k*)
by (*simp add*: *i-shrink-i-append-mod*)

lemma *i-shrink-Cons*:

$[\![\ 0 < k;\ length\ xs = k\ ]\!] \Longrightarrow (xs \frown f) \div_i k = [last\text{-}message\ xs] \frown (f \div_i k)$
by (*simp add*: *f-shrink-def i-shrink-def i-aggregate-Cons*)

lemma *i-shrink-take-nth*:
$n < m\ div\ k \Longrightarrow (f \Downarrow m) \div_f k\ !\ n = (f \div_i k)\ n$
by (*simp add*: *f-shrink-def i-shrink-def i-aggregate-take-nth*)

lemma *i-shrink-const*[*simp*]: $0 < k \Longrightarrow (\lambda x.\ m) \div_i k = (\lambda x.\ m)$
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*: $0 < k \Longrightarrow f \odot_i k \div_i k = f$
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*: $0 < k \Longrightarrow f \Downarrow (n * k) \div_f k = f \div_i k \Downarrow n$
by (*simp add*: *f-shrink-def i-shrink-def i-aggregate-i-take-mult*)

lemma *i-shrink-i-take*:
$f \Downarrow n \div_f k = f \div_i k \Downarrow (n\ div\ k)$
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*:
$n\ mod\ k = 0 \Longrightarrow f \Uparrow n \div_i k = f \div_i k \Uparrow (n\ div\ k)$
by (*simp add*: *f-shrink-def i-shrink-def i-aggregate-i-drop-mod*)

lemma *i-shrink-eq-conv*:
$(f \div_i k1 = g \div_i k2) =$
$(\forall i.\ last\text{-}message\ (f \Uparrow (i * k1) \Downarrow k1) =$
$\quad last\text{-}message\ (g \Uparrow (i * k2) \Downarrow k2))$
by (*simp add*: *ilist-eq-iff i-shrink-nth*)

lemma *i-shrink-eq-conv'*:
$(f' \div_i k = f) =$
$(\forall i.\ if\ f\ i = \varepsilon\ then\ \forall j{<}k.\ f'\ (i * k + j) = \varepsilon$
$\quad else\ \exists n{<}k.\ f'\ (i * k + n) = f\ i \wedge$
$\qquad (\forall j{<}k.\ n < j \longrightarrow f'\ (i * k + j) = \varepsilon))$
apply (*simp add*: *ilist-eq-iff*)
apply (*case-tac* $k = 0$, *fastforce*)
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

lemma *i-shrink-assoc*: $f \div_i a \div_i b = f \div_i (a * b)$
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*: $f \div_i a \div_i b = f \div_i b \div_i a$
by (*simp add*: *i-shrink-assoc mult.commute*[*of a*])

2.2.4 Holding last messages in everly cycle of a stream

primrec *last-message-hold-init* :: *'a fstream-af* ⇒ *'a message-af* ⇒ *'a fstream-af*
where
  last-message-hold-init [] *m* = []
| *last-message-hold-init* (*x* # *xs*) *m* =
    (*if x* = ε *then m else x*) #
    (*last-message-hold-init xs* (*if x* = ε *then m else x*))

definition *last-message-hold* :: *'a fstream-af* ⇒ *'a fstream-af*
  where *last-message-hold xs* ≡ *last-message-hold-init xs* ε

lemma *last-message-hold-init-length*[*simp*]:
  ⋀*m*. *length* (*last-message-hold-init xs m*) = *length xs*
by (*induct xs*, *simp+*)

lemma *last-message-hold-init-nth*:
  ⋀*i m*. *i* < *length xs* ⟹
  (*last-message-hold-init xs m*) ! *i* = *last-message* (*m* # *xs* ↓ *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*]:
  ⋀*xs m*. *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* ⟹ *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 ≠ [] ⟹ *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* ⇒ *nat* ⇒ *'a fstream-af* (**infixl** ⟼$_f$ *100*)
where *f-last-message-hold xs k* ≡ *concat* (*map last-message-hold* (*list-slice2 xs k*))

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

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

lemma *f-last-message-hold-0*[*simp*]: *xs* ⟼$_f$ *0* = *last-message-hold xs*
by (*simp add*: *f-last-message-hold-def list-slice2-0*)

lemma *f-last-message-hold-1*[*simp*]: *xs* ⟼$_f$ (*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*]: [] ⟼$_f$ *k* = []
by (*simp add*: *f-last-message-hold-def list-slice2-Nil*)

lemma *f-last-message-hold-length*[*simp*]: *length* (*xs* ⟼$_f$ *k*) = *length xs*
apply (*case-tac k* = *0*, *simp*)
apply (*simp add*: *f-last-message-hold-def*)
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*: *length xs* ≤ *k* ⟹ *xs* ⟼$_f$ *k* = *last-message-hold xs*
by (*simp add*: *f-last-message-hold-def list-slice2-le*)

lemma *f-last-message-hold-append-mult*:
length xs = *m* ∗ *k* ⟹ (*xs* @ *ys*) ⟼$_f$ *k* = *xs* ⟼$_f$ *k* @ (*ys* ⟼$_f$ *k*)
by (*simp add*: *f-last-message-hold-def list-slice2-append-mod*)

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

length xs mod k = 0 ⟹ (xs @ ys) ⟼$_f$ k = xs ⟼$_f$ k @ (ys ⟼$_f$ k)
by (*simp add*: *f-last-message-hold-def list-slice2-append-mod*)

lemma *f-last-message-hold-nth*[*rule-format*]:
*∀ n. n < length xs ⟶ xs ⟼$_f$ k ! n = last-message (xs ↑ (n div k * k) ↓ Suc (n mod 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 mod k ≤ 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 ↓ n ⟼$_f$ k = xs ⟼$_f$ k ↓ 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 ↑ (n * k) ⟼$_f$ k = xs ⟼$_f$ k ↑ (n * k)*
apply (*rule subst*[*OF append-take-drop-id*, *of - n * k xs*])
apply (*case-tac length xs ≤ 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 mod k = 0 ⟹ xs ↑ n ⟼$_f$ k = xs ⟼$_f$ k ↑ n
by (*clarsimp simp*: *mult.commute*[*of k*] *f-last-message-hold-drop-mult elim*!: *dvdE*)

lemma *f-last-message-hold-idem*: *xs ⟼$_f$ k ⟼$_f$ k = xs ⟼$_f$ 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.coboundedI2 Suc-mod-le-divisor*)
done

lemma *f-shrink-nth-eq-f-last-message-hold-last*:
*n < length xs div k ⟹ xs ÷$_f$ k ! n = last (xs ⟼$_f$ k ↑ (n * k) ↓ 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*:
  $n < length\ xs\ div\ k \Longrightarrow xs \div_f k\ !\ n = xs \longmapsto_f k\ !\ (n * k + k - Suc\ 0)$
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*:
  $last\text{-}message\ (xs \longmapsto_f k) = last\text{-}message\ xs$
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*]: $f \longmapsto_i 0 = (\lambda n.\ last\text{-}message\ (f \Downarrow Suc\ n))$
by (*simp add*: *i-last-message-hold-def*)
lemma *i-last-message-hold-1*[*simp*]: $f \longmapsto_i Suc\ 0 = f$
by (*simp add*: *i-last-message-hold-def i-drop-i-take-1*)

lemma *i-last-message-hold-nth*:
  $(f \longmapsto_i k)\ n = last\text{-}message\ (f \Uparrow (n - n\ mod\ k) \Downarrow Suc\ (n\ mod\ k))$
by (*simp add*: *i-last-message-hold-def*)

lemma *i-last-message-hold-i-append-mult*:
  $length\ xs = m * k \Longrightarrow (xs \frown f) \longmapsto_i k = (xs \longmapsto_f k) \frown (f \longmapsto_i k)$
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* $length\ xs \leq x - x\ mod\ k$)
 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* $j{=}x - x\ mod\ k$ **and** $k{=}x\ mod\ k$ **in** *add-le-mono1*)
apply (*simp add*: *add.commute*[*of* $m * k$])
done

lemma *i-last-message-hold-i-append-mod*:
  $length\ xs\ mod\ k = 0 \Longrightarrow (xs \frown f) \longmapsto_i k = (xs \longmapsto_f k) \frown (f \longmapsto_i k)$
 by (*clarsimp simp*: *mult.commute*[*of k*] *elim*!: *dvdE*, *rule i-last-message-hold-i-append-mult*)

lemma *i-last-message-hold-i-take*: $f \Downarrow n \longmapsto_f k = (f \longmapsto_i k) \Downarrow n$
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*:
$f \Uparrow (n * k) \longmapsto_i k = f \longmapsto_i k \Uparrow (n * k)$
by (*simp add*: *ilist-eq-iff i-last-message-hold-nth*)

lemma *i-last-message-hold-i-drop-mod*:
$n \bmod k = 0 \Longrightarrow f \Uparrow n \longmapsto_i k = f \longmapsto_i k \Uparrow n$
by (*clarsimp simp*: *mult.commute*[*of k*] *elim*!: *dvdE*, *rule i-last-message-hold-i-drop-mult*)

lemma *i-last-message-hold-idem*: $f \longmapsto_i k \longmapsto_i k = f \longmapsto_i k$
by (*simp add*: *ilist-eq-iff i-last-message-hold-nth minus-mod-eq-mult-div i-last-message-hold-i-drop-mod*[*symmet*
i-last-message-hold-i-take[*symmetric*] *last-message-f-last-message-hold*)

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

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

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

notation
f-shrink-last (**infixl** $\div_l$ *100*) **and**
i-shrink-last (**infixl** $\div_l$ *100*)

lemma *f-shrink-last-0*[*simp*]: *xs* $\div_{fl}$ *0* = []
by (*simp add*: *f-shrink-last-def f-aggregate-def list-slice-0*)

lemma *f-shrink-last-1*[*simp*]: *xs* $\div_{fl}$ *Suc 0* = *xs*
by (*simp add*: *f-shrink-last-def f-aggregate-1*)

lemma *f-shrink-last-Nil*[*simp*]: [] $\div_{fl}$ *k* = []
by (*simp add*: *f-shrink-last-def f-aggregate-def list-slice-Nil*)

lemma *f-shrink-last-length*: *length* (*xs* ÷$_{fl}$ *k*) = *length xs div k*
by (*simp add*: *f-shrink-last-def*)

lemma *f-shrink-last-empty-conv*:
0 < *k* ⟹ (*xs* ÷$_{fl}$ *k* = []) = (*length xs* < *k*)
by (*simp add*: *f-shrink-last-def f-aggregate-empty-conv*)

lemma *f-shrink-last-Cons*:
⟦ *0* < *k*;
length xs = *k* ⟧ ⟹ (*xs* @ *ys*) ÷$_{fl}$ *k* = *last xs* # (*ys* ÷$_{fl}$ *k*)
by (*simp add*: *f-shrink-last-def f-aggregate-Cons*)

lemma *f-shrink-last-one*:
⟦ *0* < *k*; *length xs* = *k* ⟧ ⟹ *xs* ÷$_{fl}$ *k* = [*last xs*]
by (*simp add*: *f-shrink-last-def f-aggregate-one*)

lemma *f-shrink-last-eq-f-shrink-last-take*:
xs ↓ (*length xs div k* ∗ *k*) ÷$_{fl}$ *k* = *xs* ÷$_{fl}$ *k*
by (*simp add*: *f-shrink-last-def f-aggregate-eq-f-aggregate-take*)

lemma *f-shrink-last-nth*:
n < *length xs div k* ⟹ (*xs* ÷$_{fl}$ *k*) ! *n* = *xs* ! (*n* ∗ *k* + *k* − *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 < *length xs div k* ⟹ (*xs* ÷$_{fl}$ *k*) ! *n* = *xs* ! (*Suc n* ∗ *k* − *Suc 0*)
by (*simp add*: *f-shrink-last-nth add.commute*[*of k*])

lemma *f-shrink-last-hd*:
⟦ *0* < *k*; *k* ≤ *length xs* ⟧ ⟹ *hd* (*xs* ÷$_{fl}$ *k*) = *xs* ! (*k* − *Suc 0*)
by (*simp add*: *f-shrink-last-def f-aggregate-hd last-take2*)

lemma *f-shrink-last-map*: (*map f xs*) ÷$_{fl}$ *k* = *map f* (*xs* ÷$_{fl}$ *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*:
length xs mod k = *0* ⟹ (*xs* @ *ys*) ÷$_{fl}$ *k* = *xs* ÷$_{fl}$ *k* @ (*ys* ÷$_{fl}$ *k*)
by (*simp add*: *f-shrink-last-def f-aggregate-append-mod*)

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

*length xs = m * k ⟹ (xs @ ys) ÷ʃl k = xs ÷ʃl k @ (ys ÷ʃl k)*
by (*unfold f-shrink-last-def*, *rule f-aggregate-append-mult*)

lemma *f-shrink-last-snoc*:
⟦ 0 < k; length ys = k; length xs mod k = 0 ⟧ ⟹
(xs @ ys) ÷ʃl k = xs ÷ʃl k @ [last ys]
by (*simp add*: *f-shrink-last-append-mod f-shrink-last-one*)

lemma *f-shrink-last-last*:
length xs mod k = 0 ⟹ last (xs ÷ʃl k) = last xs
apply (*case-tac k = 0*, *simp*)
apply (*case-tac xs = []*, *simp*)
apply (*subgoal-tac k ≤ length xs*)
prefer *2*
apply (*rule ccontr*, *simp*)
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 ÷ʃl k = m^(n 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-take*:
xs ↓ n ÷ʃl k = xs ÷ʃl k ↓ (n div k)
by (*unfold f-shrink-last-def*, *rule f-aggregate-take*)

lemma *f-shrink-last-take-mult*: *xs ↓ (n * k) ÷ʃl k = xs ÷ʃl k ↓ n*
by (*unfold f-shrink-last-def*, *rule f-aggregate-take-mult*)

lemma *f-shrink-last-drop-mult*: *xs ↑ (n * k) ÷ʃl k = xs ÷ʃl k ↑ n*
by (*unfold f-shrink-last-def*, *rule f-aggregate-drop-mult*)

lemma *f-shrink-last-drop-mod*:
n mod k = 0 ⟹ xs ↑ n ÷ʃl k = xs ÷ʃl k ↑ (n div k)
by (*unfold f-shrink-last-def*, *rule f-aggregate-drop-mod*)

lemma *f-shrink-last-assoc*: *xs ÷ʃl a ÷ʃl b = xs ÷ʃl (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 ÷ʃl a ÷ʃl b = xs ÷ʃl b ÷ʃl a*
by (*simp add*: *f-shrink-last-assoc mult.commute*[*of a*])

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

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

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

lemma *i-shrink-last-hd*: $(f \div_{il} k)\ 0 = last\ (f \Downarrow k)$
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*: $0 < k \Longrightarrow (f \circ g) \div_{il} k = f \circ (g \div_{il} k)$
by (*simp add*: *ilist-eq-iff i-shrink-last-nth*)

lemma *i-shrink-last-i-append-mod*:
$length\ xs\ mod\ k = 0 \Longrightarrow (xs \frown f) \div_{il} k = xs \div_{fl} k \frown (f \div_{il} k)$
by (*simp add*: *f-shrink-last-def i-shrink-last-def i-aggregate-i-append-mod*)

lemma *i-shrink-last-i-append-mult*:
$length\ xs = m * k \Longrightarrow (xs \frown f) \div_{il} k = xs \div_{fl} k \frown (f \div_{il} k)$
by (*simp add*: *i-shrink-last-i-append-mod*)

lemma *i-shrink-last-Cons*:
$[\![\ 0 < k;\ length\ xs = k\ ]\!] \Longrightarrow (xs \frown f) \div_{il} k = [last\ xs] \frown (f \div_{il} k)$
by (*simp add*: *f-shrink-last-def i-shrink-last-def i-aggregate-Cons*)

lemma *i-shrink-last-const*: $0 < k \Longrightarrow (\lambda x.\ m) \div_{il} k = (\lambda x.\ m)$
by (*simp add*: *ilist-eq-iff i-shrink-last-nth*)

lemma *i-shrink-last-i-take-mult*:
$0 < k \Longrightarrow f \Downarrow (n * k) \div_{fl} k = f \div_{il} k \Downarrow n$
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*:
$n\ mod\ k = 0 \Longrightarrow f \Uparrow n \div_{il} k = f \div_{il} k \Uparrow (n\ div\ k)$
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*:
$xs \div_f k = xs \longmapsto_f k \div_{fl} k$
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*:
$0 < k \Longrightarrow f \div_i k = f \longmapsto_i k \div_{il} k$
by (*simp add*: *ilist-eq-iff i-shrink-last-nth i-shrink-nth-eq-i-last-message-hold-nth*)

end

3 Processing of message streams

theory *AF-Stream-Exec*
imports *AF-Stream List−Infinite.ListInf-Prefix List−Infinite.SetIntervalStep*
begin

3.1 Executing components with state transition functions

3.1.1 Basic definitions

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

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

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

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

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

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

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

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

— Execute a component for all inputs in the input stream *'input list*
primrec *f-Exec-Comp* :: (*'comp*, *'input*) *Comp-Trans-Fun* ⇒ *'input list* ⇒ *'comp* ⇒ *'comp*
where
f-Exec-Nil: *f-Exec-Comp trans-fun* [] *c* = *c*
| *f-Exec-Cons*: *f-Exec-Comp trans-fun* (*x*#*xs*) *c* = *f-Exec-Comp trans-fun xs* (*trans-fun x c*)

— Execute the component for at most n steps
definition *f-Exec-Comp-N* :: (*'comp*, *'input*) *Comp-Trans-Fun* ⇒ *nat* ⇒ *'input list* ⇒ *'comp* ⇒ *'comp*
where *f-Exec-Comp-N trans-fun n xs c* ≡ *f-Exec-Comp trans-fun* (*xs* ↓ *n*) *c*

— Produce the component stream for all inputs in the input stream
primrec *f-Exec-Comp-Stream* :: (*'comp*, *'input*) *Comp-Trans-Fun* ⇒ *'input list* ⇒ *'comp* ⇒ *'comp list*
where
f-Exec-Stream-Nil: *f-Exec-Comp-Stream trans-fun* [] *c* = []
| *f-Exec-Stream-Cons*: *f-Exec-Comp-Stream trans-fun* (*x* # *xs*) *c* =
(*trans-fun x c*) # (*f-Exec-Comp-Stream trans-fun xs* (*trans-fun x c*))

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

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

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

3.1.2 Basic results

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

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

lemma *f-Exec-Stream-empty-conv*:
(*f-Exec-Comp-Stream trans-fun xs c* = []) = (*xs* = [])
by (*simp add*: *length-0-conv*[*symmetric*] *del*: *length-0-conv*)

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

lemma *f-Exec-eq-f-Exec-Stream-last*[*rule-format*]:
∀ *c*. *f-Exec-Comp trans-fun xs c* = *last* (*c* # (*f-Exec-Comp-Stream trans-fun xs c*))
by (*induct xs*, *simp-all*)

corollary *f-Exec-eq-f-Exec-Stream-last2*[*rule-format*]:
xs ≠ [] ⟹
f-Exec-Comp trans-fun xs c = *last* (*f-Exec-Comp-Stream trans-fun xs c*)
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-Exec-Comp trans-fun xs c = (*if xs* = [] *then c else last* (*f-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* ≠ []; *0* < *n* ⟧ ⟹
f-Exec-Comp trans-fun (*xs* ↓ *n*) *c* =
last (*f-Exec-Comp-Stream trans-fun* (*xs* ↓ *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* ≠ []; *0* < *n* ⟧ ⟹
f-Exec-Comp-N trans-fun n xs c =
last (*f-Exec-Comp-Stream trans-fun* (*xs* ↓ *n*) *c*)
by (*simp add*: *f-Exec-Comp-N-def f-Exec-take-eq-last-f-Exec-Stream-take*)

lemma *f-Exec-Stream-nth*:
⋀*n c*. *n* < *length xs* ⟹
f-Exec-Comp-Stream trans-fun xs c ! *n* = *f-Exec-Comp trans-fun* (*xs* ↓ *Suc n*) *c*
apply (*induct xs*, *simp*)
apply (*simp add*: *nth-Cons'*)
done

lemma *f-Exec-Stream-nth2*:
n ≤ *length xs* ⟹
(*c* # *f-Exec-Comp-Stream trans-fun xs c*) ! *n* = *f-Exec-Comp trans-fun* (*xs* ↓ *n*) *c*
by (*simp add*: *nth-Cons' f-Exec-Stream-nth*)

lemma *f-Exec-N-all*:
length xs $\leq$ *n* $\Longrightarrow$
f-Exec-Comp-N trans-fun n xs c = *f-Exec-Comp trans-fun xs c*
by (*simp add*: *f-Exec-Comp-N-def*)

lemma *f-Exec-Stream-append*[*rule-format*]:$\forall$ *c*.
f-Exec-Comp-Stream trans-fun (*xs* @ *ys*) *c* =
(*f-Exec-Comp-Stream trans-fun xs c*) @
(*f-Exec-Comp-Stream trans-fun ys* (*f-Exec-Comp trans-fun xs c*))
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 $\neq$ [] $\Longrightarrow$
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*)

corollary *f-Exec-Stream-Cons-rev*:
xs $\neq$ [] $\Longrightarrow$
(*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
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*)
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-Exec-Comp-Stream trans-fun input c (Suc n) =
(i-Exec-Comp-Stream trans-fun (input ⇑ Suc 0) (trans-fun (input 0) c) n)
by (*simp add*: *i-Exec-Stream-nth i-take-Suc*)

lemma *f-Exec-Stream-nth-eq-i-Exec-Stream-nth*:
n < n' ⟹
f-Exec-Comp-Stream trans-fun (input ⇓ n') c ! n =
i-Exec-Comp-Stream trans-fun input c n
by (*simp add*: *f-Exec-Stream-nth i-Exec-Stream-nth min-eqR*)

lemma *i-Exec-Stream-append*:
i-Exec-Comp-Stream trans-fun (xs ⌢ input) c =
f-Exec-Comp-Stream trans-fun xs c ⌢
i-Exec-Comp-Stream trans-fun input (f-Exec-Comp trans-fun xs c)
by (*simp add*: *ilist-eq-iff i-Exec-Stream-nth f-Exec-Stream-nth f-Exec-append i-append-nth Suc-diff-le*)

lemma *i-Exec-Stream-append-last-Cons*:
i-Exec-Comp-Stream trans-fun (xs ⌢ input) c =
f-Exec-Comp-Stream trans-fun xs c ⌢
i-Exec-Comp-Stream trans-fun input (
last (c # f-Exec-Comp-Stream trans-fun xs c))
by (*simp add*: *f-Exec-eq-f-Exec-Stream-last i-Exec-Stream-append*)

lemma *i-Exec-Stream-append-last*:
xs ≠ [] ⟹
i-Exec-Comp-Stream trans-fun (xs ⌢ input) c =
f-Exec-Comp-Stream trans-fun xs c ⌢
i-Exec-Comp-Stream trans-fun input (
last (f-Exec-Comp-Stream trans-fun xs c))
by (*simp add*: *f-Exec-Stream-empty-conv i-Exec-Stream-append-last-Cons*)

lemma *i-Exec-Stream-append-if*:
i-Exec-Comp-Stream trans-fun (xs ⌢ input) c =
f-Exec-Comp-Stream trans-fun xs c ⌢
i-Exec-Comp-Stream trans-fun input (
if xs = [] then c
else last (f-Exec-Comp-Stream trans-fun xs c))
by (*simp add*: *i-Exec-Stream-append-last*)

corollary *i-Exec-Stream-Cons*:
i-Exec-Comp-Stream trans-fun ([x] ⌢ input) c =
[trans-fun x c] ⌢ i-Exec-Comp-Stream trans-fun input (trans-fun x c)

by (*simp add*: *i-Exec-Stream-append*)

corollary *i-Exec-Stream-Cons-rev*:
[trans-fun (input 0) c] ⌢
i-Exec-Comp-Stream trans-fun (input ⇑ Suc 0) (trans-fun (input 0) c) =
i-Exec-Comp-Stream trans-fun input c
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 ⟹
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 ⟹
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*)

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* ⊙$_f$ *k*) *c*) *k ag* ↑ *n* =
f-aggregate (*f-Exec-Comp-Stream trans-fun* ((*xs* ↑ *n*) ⊙$_f$ *k*) (
f-Exec-Comp trans-fun ((*xs* ↓ *n*) ⊙$_f$ *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* ⟹
i-aggregate (*f* ∘ (*i-Exec-Comp-Stream trans-fun* (*input* ⊙$_i$ *k*) *c*)) *k ag* ⇑ *n* =
i-aggregate (*f* ∘ (*i-Exec-Comp-Stream trans-fun* ((*input* ⇑ *n*) ⊙$_i$ *k*) (
f-Exec-Comp trans-fun ((*input* ⇓ *n*) ⊙$_f$ *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* ⟹
i-aggregate (*i-Exec-Comp-Stream trans-fun* (*input* ⊙$_i$ *k*) *c*) *k ag* ⇑ *n* =
i-aggregate (*i-Exec-Comp-Stream trans-fun* ((*input* ⇑ *n*) ⊙$_i$ *k*) (
f-Exec-Comp trans-fun ((*input* ⇓ *n*) ⊙$_f$ *k*) *c*)) *k ag*
by (*drule i-Exec-Stream-expand-aggregate-map-drop*[*of k n id trans-fun input c ag*], *simp*)

lemma *f-Exec-Stream-expand-aggregate-map-nth-eq-i-nth*:
⟦ *0* < *k*; *n* < *n'* ⟧ ⟹
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* (*input* ⇓ *n'* ⊙$_f$ *k*) *c*)) *k ag* !
n =
i-aggregate (*f* ∘ (*i-Exec-Comp-Stream trans-fun* (*input* ⊙$_i$ *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* ⟹
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* (*input* ⇓ *Suc n* ⊙$_f$ *k*) *c*)) *k ag*
! *n* =
i-aggregate (*f* ∘ (*i-Exec-Comp-Stream trans-fun* (*input* ⊙$_i$ *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*:
⟦ *0* < *k*; *n* < *n'* ⟧ ⟹

f-aggregate (*f-Exec-Comp-Stream trans-fun* (*input* ⇓ n' ⊙$_f$ *k*) *c*) *k ag* ! *n* =
i-aggregate (*i-Exec-Comp-Stream trans-fun* (*input* ⊙$_i$ *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* ⟹
f-aggregate (*f-Exec-Comp-Stream trans-fun* (*input* ⇓ *Suc n* ⊙$_f$ *k*) *c*) *k ag* ! *n* =
i-aggregate (*i-Exec-Comp-Stream trans-fun* (*input* ⊙$_i$ *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*:
⟦ *0* < *k*; *n* < *length xs* ⟧ ⟹
map f (*f-Exec-Comp-Stream trans-fun* (*xs* ⊙$_f$ *k*) *c*) ÷$_{fl}$ *k* ! *n* =
f (*f-Exec-Comp trans-fun* ((*xs* ↓ *Suc n*) ⊙$_f$ *k*) *c*)
apply (*simp add*: *f-shrink-last-map f-shrink-last-length f-shrink-last-nth*)
apply (*subgoal-tac n* ∗ *k* + *k* − *Suc 0* < *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*:
⟦ *0* < *k*; *n* < *length xs* ⟧ ⟹
f-Exec-Comp-Stream trans-fun (*xs* ⊙$_f$ *k*) *c* ÷$_{fl}$ *k* ! *n* =
f-Exec-Comp trans-fun ((*xs* ↓ *Suc n*) ⊙$_f$ *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*:
⟦ *0* < *k*; *n* < *length xs* ⟧ ⟹
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* (*xs* ⊙$_f$ *k*) *c*)) *k ag* ! *n* =
ag (*map f* (*f-Exec-Comp-Stream trans-fun* (*xs* ! *n* # $\varepsilon^{k - Suc\ 0}$)
(*f-Exec-Comp trans-fun* (*xs* ↓ *n* ⊙$_f$ *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*:
⟦ *0* < *k*; *n* < *length xs* ⟧ ⟹
f-aggregate (*f-Exec-Comp-Stream trans-fun* (*xs* ⊙$_f$ *k*) *c*) *k ag* ! *n* =
ag (*f-Exec-Comp-Stream trans-fun* (*xs* ! *n* # $\varepsilon^{k - Suc\ 0}$)
(*f-Exec-Comp trans-fun* (*xs* ↓ *n* ⊙$_f$ *k*) *c*))
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*:

$[\![\ 0 < k;\ n < \mathit{length}\ \mathit{xs}\ ]\!] \Longrightarrow$
$(\mathit{map}\ f\ (\mathit{f\text{-}Exec\text{-}Comp\text{-}Stream}\ \mathit{trans\text{-}fun}\ (\mathit{xs} \odot_f k)\ c)) \div_f k\ !\ n =$
$\mathit{last\text{-}message}\ (\mathit{map}\ f\ (\mathit{f\text{-}Exec\text{-}Comp\text{-}Stream}\ \mathit{trans\text{-}fun}\ (\mathit{xs}\ !\ n\ \#\ \varepsilon^{k - \mathit{Suc}\ 0})$
$(\mathit{f\text{-}Exec\text{-}Comp}\ \mathit{trans\text{-}fun}\ (\mathit{xs} \downarrow n \odot_f k)\ c)))$
by (*simp add*: *f-shrink-def f-Exec-Stream-expand-aggregate-map-nth*)

lemma *i-Exec-Stream-expand-aggregate-map-nth*:
$0 < k \Longrightarrow$
$\mathit{i\text{-}aggregate}\ (f \circ (\mathit{i\text{-}Exec\text{-}Comp\text{-}Stream}\ \mathit{trans\text{-}fun}\ (\mathit{input} \odot_i k)\ c))\ k\ \mathit{ag}\ n =$
$\mathit{ag}\ (\mathit{map}\ f\ (\mathit{f\text{-}Exec\text{-}Comp\text{-}Stream}\ \mathit{trans\text{-}fun}\ (\mathit{input}\ n\ \#\ \varepsilon^{k - \mathit{Suc}\ 0})$
$(\mathit{f\text{-}Exec\text{-}Comp}\ \mathit{trans\text{-}fun}\ ((\mathit{input} \Downarrow n) \odot_f k)\ c)))$
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*:
$0 < k \Longrightarrow$
$\mathit{i\text{-}aggregate}\ (\mathit{i\text{-}Exec\text{-}Comp\text{-}Stream}\ \mathit{trans\text{-}fun}\ (\mathit{input} \odot_i k)\ c)\ k\ \mathit{ag}\ n =$
$\mathit{ag}\ (\mathit{f\text{-}Exec\text{-}Comp\text{-}Stream}\ \mathit{trans\text{-}fun}\ (\mathit{input}\ n\ \#\ \varepsilon^{k - \mathit{Suc}\ 0})$
$(\mathit{f\text{-}Exec\text{-}Comp}\ \mathit{trans\text{-}fun}\ ((\mathit{input} \Downarrow n) \odot_f k)\ c))$
by (*drule i-Exec-Stream-expand-aggregate-map-nth*[**where** *f*=*id*], *simp add*: *map-id*)

corollary *i-Exec-Stream-expand-shrink-map-nth*:
$0 < k \Longrightarrow$
$((f \circ (\mathit{i\text{-}Exec\text{-}Comp\text{-}Stream}\ \mathit{trans\text{-}fun}\ (\mathit{input} \odot_i k)\ c)) \div_i k)\ n =$
$\mathit{last\text{-}message}\ (\mathit{map}\ f\ (\mathit{f\text{-}Exec\text{-}Comp\text{-}Stream}\ \mathit{trans\text{-}fun}\ (\mathit{input}\ n\ \#\ \varepsilon^{k - \mathit{Suc}\ 0})$
$(\mathit{f\text{-}Exec\text{-}Comp}\ \mathit{trans\text{-}fun}\ (\mathit{input} \Downarrow n \odot_f k)\ c)))$
by (*simp add*: *i-shrink-def i-Exec-Stream-expand-aggregate-map-nth*)

lemma *f-Exec-Stream-expand-snoc*:
$[\![\ 0 < k;\ n < \mathit{length}\ \mathit{xs}\ ]\!] \Longrightarrow$
$\mathit{f\text{-}Exec\text{-}Comp\text{-}Stream}\ \mathit{trans\text{-}fun}\ (\mathit{xs} \odot_f k)\ c \uparrow (n * k) \downarrow k =$
$\mathit{f\text{-}Exec\text{-}Comp\text{-}Stream}\ \mathit{trans\text{-}fun}\ (\mathit{xs}\ !\ n\ \#\ \varepsilon^{k - \mathit{Suc}\ 0})$
$(\mathit{f\text{-}Exec\text{-}Comp}\ \mathit{trans\text{-}fun}\ (\mathit{xs} \downarrow n \odot_f k)\ c)$
by (*simp add*: *f-Exec-Stream-drop f-Exec-Stream-take f-expand-take-mod f-expand-drop-mod drop-take-1*)

lemma *f-Exec-Stream-expand-map-aggregate-append*:
$\mathit{f\text{-}aggregate}\ (\mathit{map}\ f\ (\mathit{f\text{-}Exec\text{-}Comp\text{-}Stream}\ \mathit{trans\text{-}fun}\ ((\mathit{xs} @ \mathit{ys}) \odot_f k)\ c))\ k\ \mathit{ag} =$
$\mathit{f\text{-}aggregate}\ (\mathit{map}\ f\ (\mathit{f\text{-}Exec\text{-}Comp\text{-}Stream}\ \mathit{trans\text{-}fun}\ (\mathit{xs} \odot_f k)\ c))\ k\ \mathit{ag} @$
$\mathit{f\text{-}aggregate}\ (\mathit{map}\ f\ (\mathit{f\text{-}Exec\text{-}Comp\text{-}Stream}\ \mathit{trans\text{-}fun}\ (\mathit{ys} \odot_f k)\ ($
$\mathit{f\text{-}Exec\text{-}Comp}\ \mathit{trans\text{-}fun}\ (\mathit{xs} \odot_f k)\ c)))\ k\ \mathit{ag}$
by (*simp add*: *f-Exec-Stream-append f-aggregate-append-mod*)

lemma *i-Exec-Stream-expand-map-aggregate-append*:
$\mathit{i\text{-}aggregate}\ (f \circ (\mathit{i\text{-}Exec\text{-}Comp\text{-}Stream}\ \mathit{trans\text{-}fun}\ ((\mathit{xs} \frown \mathit{input}) \odot_i k)\ c))\ k\ \mathit{ag} =$
$\mathit{f\text{-}aggregate}\ (\mathit{map}\ f\ (\mathit{f\text{-}Exec\text{-}Comp\text{-}Stream}\ \mathit{trans\text{-}fun}\ (\mathit{xs} \odot_f k)\ c))\ k\ \mathit{ag} \frown$
$\mathit{i\text{-}aggregate}\ (f \circ (\mathit{i\text{-}Exec\text{-}Comp\text{-}Stream}\ \mathit{trans\text{-}fun}\ (\mathit{input} \odot_i k)\ ($
$\mathit{f\text{-}Exec\text{-}Comp}\ \mathit{trans\text{-}fun}\ (\mathit{xs} \odot_f k)\ c)))\ k\ \mathit{ag}$
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 $\Longrightarrow$
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* ((*x # xs*) $\odot_f$ *k*) *c*)) *k ag* =
ag (*map f* (*f-Exec-Comp-Stream trans-fun* (*x # ε*$^{k - Suc\ 0}$) *c*)) #
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* (*xs* $\odot_f$ *k*) (
f-Exec-Comp trans-fun (*x # ε*$^{k - Suc\ 0}$) *c*))) *k 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 $\Longrightarrow$
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* ((*xs* @ [*x*]) $\odot_f$ *k*) *c*)) *k ag* =
f-aggregate (*map f* (*f-Exec-Comp-Stream trans-fun* (*xs* $\odot_f$ *k*) *c*)) *k ag* @
[*ag* (*map f* (*f-Exec-Comp-Stream trans-fun* (*x # ε*$^{k - Suc\ 0}$) (
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 $\Longrightarrow$
i-aggregate (*f* ∘ (*i-Exec-Comp-Stream trans-fun* (([*x*] ⌢ *input*) $\odot_i$ *k*) *c*)) *k ag* =
[*ag* (*map f* (*f-Exec-Comp-Stream trans-fun* (*x # ε*$^{k - Suc\ 0}$) *c*))] ⌢
i-aggregate (*f* ∘ (*i-Exec-Comp-Stream trans-fun* (*input* $\odot_i$ *k*) (
f-Exec-Comp trans-fun (*x # ε*$^{k - Suc\ 0}$) *c*))) *k ag*
apply (*subst i-Exec-Stream-expand-map-aggregate-append*)
apply (*simp add*: *f-aggregate-one*)
done

lemma *f-Exec-N-eq-f-Exec-Stream-nth*:
n ≤ *length xs* $\Longrightarrow$
f-Exec-Comp-N trans-fun n xs c = (*c # f-Exec-Comp-Stream trans-fun xs c*) ! *n*
by (*simp add*: *f-Exec-Comp-N-def f-Exec-Stream-nth2*)

theorem *f-Exec-Stream-causal*:
xs ↓ *n* = *ys* ↓ *n* $\Longrightarrow$
(*f-Exec-Comp-Stream trans-fun xs c*) ↓ *n* = (*f-Exec-Comp-Stream trans-fun ys c*) ↓ *n*
by (*simp add*: *f-Exec-Stream-take*)
theorem *i-Exec-Stream-causal*:
input1 ⇓ *n* = *input2* ⇓ *n* $\Longrightarrow$
(*i-Exec-Comp-Stream trans-fun input1 c*) ⇓ *n* = (*i-Exec-Comp-Stream trans-fun input2 c*) ⇓ *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*]:
∀ *c*. *length* (*f-Exec-Comp-Stream-Init trans-fun xs c*) = *Suc* (*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*)

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*)
apply (*simp add*: *take-Suc-conv-app-nth f-Exec-append Cons-nth-drop-Suc*[*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*)

lemma *i-Exec-Stream-Init-eq-i-Exec-Stream-Cons*:
i-Exec-Comp-Stream-Init trans-fun input c = [c] ⌢ i-Exec-Comp-Stream trans-fun input c
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 ⟹
output-fun ∘ i-Exec-Comp-Stream-Init trans-fun input c =
[ε] ⌢ (output-fun ∘ 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 ⌢ input2) c =
(f-Exec-Comp-Stream-Init trans-fun input1 c) ⌢
((i-Exec-Comp-Stream-Init trans-fun input2 (f-Exec-Comp trans-fun input1 c)) ⇑ Suc 0)
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 ⌢ input2) c =
(f-Exec-Comp-Stream-Init trans-fun input1 c) ⌢
((i-Exec-Comp-Stream-Init trans-fun input2 (last (f-Exec-Comp-Stream-Init trans-fun input1 c))) ⇑ Suc 0)
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 ⌢ input2) c =
(f-Exec-Comp-Stream-Init trans-fun input1 c) ⌢
(i-Exec-Comp-Stream trans-fun input2 (f-Exec-Comp trans-fun input1 c))
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) ⇓ Suc n =
f-Exec-Comp-Stream-Init trans-fun (input ⇓ 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*) ⇑ *n* =
i-Exec-Comp-Stream-Init trans-fun (*input* ⇑ *n*)
(*f-Exec-Comp trans-fun* (*input* ⇓ *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*)
apply (*simp add*: *upt-conv-Cons*)
done

theorem *f-Exec-Stream-Init-strictly-causal*:
xs ↓ *n* = *ys* ↓ *n* ⟹
(*f-Exec-Comp-Stream-Init trans-fun xs c*) ↓ *Suc n* = (*f-Exec-Comp-Stream-Init trans-fun ys c*) ↓ *Suc n*
by (*simp add*: *f-Exec-Stream-Init-take*)

theorem *i-Exec-Stream-Init-strictly-causal*:
input1 ⇓ *n* = *input2* ⇓ *n* ⟹
(*i-Exec-Comp-Stream-Init trans-fun input1 c*) ⇓ *Suc n* = (*i-Exec-Comp-Stream-Init trans-fun input2 c*) ⇓ *Suc n*
by (*simp add*: *i-Exec-Stream-Init-take*)

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

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

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

abbreviation *ilist-Previous'* :: *'value ilist* ⇒ *'value* ⇒ *nat* ⇒ *'value*

$(\ -^{\leftarrow\ '}\ -\ [1000,\ 10,\ 100]\ 100)$
where $f^{\leftarrow\ init}\ n \equiv ilist\text{-}Previous\ f\ init\ n$

lemma *list-Previous-nth*: $xs^{\leftarrow'\ init}\ n = (case\ n\ of\ 0 \Rightarrow init \mid Suc\ n' \Rightarrow xs\ !\ n')$
by (*simp add*: *list-Previous-def*)

lemma *ilist-Previous-nth*: $f^{\leftarrow\ init}\ n = (case\ n\ of\ 0 \Rightarrow init \mid Suc\ n' \Rightarrow f\ n')$
by (*simp add*: *ilist-Previous-def*)

lemma *list-Previous-nth-if*: $xs^{\leftarrow'\ init}\ n = (if\ n = 0\ then\ init\ else\ xs\ !\ (n - Suc\ 0))$
by (*case-tac n*, *simp-all add*: *list-Previous-nth*)

lemma *ilist-Previous-nth-if*: $f^{\leftarrow\ init}\ n = (if\ n = 0\ then\ init\ else\ f\ (n - Suc\ 0))$
by (*case-tac n*, *simp-all add*: *ilist-Previous-nth*)

lemma *list-Previous-Cons*: $xs^{\leftarrow'\ init}\ n = (init\ \#\ xs)\ !\ n$
by (*case-tac n*, *simp-all add*: *list-Previous-nth*)

lemma *ilist-Previous-Cons*: $f^{\leftarrow\ init}\ n = ([init] \frown f)\ n$
by (*case-tac n*, *simp-all add*: *ilist-Previous-nth*)

lemma *list-Previous-0*: $xs^{\leftarrow'\ init}\ 0 = init$
by (*simp add*: *list-Previous-def*)

lemma *ilist-Previous-0*: $f^{\leftarrow\ init}\ 0 = init$
by (*simp add*: *ilist-Previous-def*)

lemma *list-Previous-gr0*: $0 < n \Longrightarrow xs^{\leftarrow'\ init}\ n = xs\ !\ (n - Suc\ 0)$
by (*case-tac n*, *simp-all add*: *list-Previous-nth*)

lemma *ilist-Previous-gr0*: $0 < n \Longrightarrow f^{\leftarrow\ init}\ n = f\ (n - Suc\ 0)$
by (*case-tac n*, *simp-all add*: *ilist-Previous-nth*)

lemma *list-Previous-Suc*: $xs^{\leftarrow'\ init}\ (Suc\ n) = xs\ !\ n$
by (*simp add*: *list-Previous-def*)

lemma *ilist-Previous-Suc*: $f^{\leftarrow\ init}\ (Suc\ n) = f\ n$
by (*simp add*: *ilist-Previous-def*)

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

by (*simp add*: *i-Exec-Stream-Init-eq-i-Exec-Stream-Cons ilist-Previous-Cons*)

lemma *f-Exec-Stream-hd*:
0 < length xs ⟹ hd (*f-Exec-Comp-Stream trans-fun xs c*) = *trans-fun* (*hd xs*) *c*
by (*case-tac xs*, *simp+*)

lemma *f-Exec-Stream-nth-0*:
0 < length xs ⟹ (*f-Exec-Comp-Stream trans-fun xs c*) ! *0*= *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*:
⟦ *n < length xs*; *0 < n* ⟧ ⟹
f-Exec-Comp-Stream trans-fun xs c ! *n* =
trans-fun (*xs* ! *n*) (*f-Exec-Comp-Stream trans-fun xs c* ! (*n* − *1*))
by (*simp add*: *f-Exec-Stream-nth take-Suc-conv-app-nth f-Exec-append*)

lemma *f-Exec-Stream-nth-calc-Previous*:
n < length xs ⟹
f-Exec-Comp-Stream trans-fun xs c ! *n* =
trans-fun (*xs* ! *n*) ((*f-Exec-Comp-Stream trans-fun xs c*)$^{\leftarrow' \ c}$ *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-Exec-Comp-Stream trans-fun input c*) *0* = *trans-fun* (*input 0*) *c*
by (*simp add*: *i-Exec-Stream-nth i-take-first*)

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

lemma *i-Exec-Stream-nth-calc-Previous*:
i-Exec-Comp-Stream trans-fun input c n =
trans-fun (*input n*) ((*i-Exec-Comp-Stream trans-fun input c*)$^{\leftarrow \ c}$ *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 < length xs ⟹
f-Exec-Comp-Stream-Init trans-fun xs c ! Suc n =
trans-fun (xs ! n) (f-Exec-Comp-Stream-Init trans-fun xs c ! n)
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 < length xs ⟹
f-Exec-Comp-Stream-Init trans-fun xs c ! (n + 1)=
trans-fun (xs ! n) (f-Exec-Comp-Stream-Init trans-fun xs c ! n)
by (*simp add*: *f-Exec-Stream-Init-nth-Suc-calc*)

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

The component state (and thus its output) at time point $n + (1::'a)$ for the execution stream with initial state is computed from the previous state (the state at time n) and the previous input (input at time n), contrary to the *i-Exec-Comp-Stream*, where each state at time n represents the resulting state after processing the input at time n.

lemma *i-Exec-Stream-Init-nth-Suc-calc*:
i-Exec-Comp-Stream-Init trans-fun input c (Suc n) =
trans-fun (input n) (i-Exec-Comp-Stream-Init trans-fun input c n)
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-Exec-Comp-Stream-Init trans-fun input c (n + 1) =
trans-fun (input n) (i-Exec-Comp-Stream-Init trans-fun input c n)
by (*simp add*: *i-Exec-Stream-Init-nth-Suc-calc*)

lemma *i-Exec-Stream-Init-nth-gr0-calc*:
0 < n ⟹
i-Exec-Comp-Stream-Init trans-fun input c n =
trans-fun (input (n − 1)) (i-Exec-Comp-Stream-Init trans-fun input c (n − 1))
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*:
⟦ *n < length xs*;
c-n = (*f-Exec-Comp-Stream trans-fun xs c*)$^{\leftarrow' c}$ *n*; *x-n = xs ! n* ⟧ *⟹*
(*P1 x-n ∧ P2 c-n ⟶ Q* (*f-Exec-Comp-Stream trans-fun xs c ! n*)) =

$(P1\ x\text{-}n \wedge P2\ c\text{-}n \longrightarrow Q\ (trans\text{-}fun\ x\text{-}n\ c\text{-}n))$
by (*simp add*: *f-Exec-Stream-nth-calc-Previous*)

Direct relation between input and result after transition

lemma *f-Exec-Stream-Pre-Post2*:
$[\![\ n < length\ xs;$
$c\text{-}n = (f\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ xs\ c)^{\leftarrow'\ c}\ n;\ x\text{-}n = xs\ !\ n\ ]\!] \Longrightarrow$
$(P\ c\text{-}n \longrightarrow Q\ (xs\ !\ n)\ (f\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ xs\ c\ !\ n)) =$
$(P\ c\text{-}n \longrightarrow Q\ x\text{-}n\ (trans\text{-}fun\ x\text{-}n\ c\text{-}n))$
by (*simp add*: *f-Exec-Stream-nth-calc-Previous*)

lemma *f-Exec-Stream-Pre-Post2-Suc*:
$[\![\ Suc\ n < length\ xs;$
$c\text{-}n = f\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ xs\ c\ !\ n;\ x\text{-}n1 = xs\ !\ Suc\ n\ ]\!] \Longrightarrow$
$(P\ c\text{-}n \longrightarrow Q\ (xs\ !\ Suc\ n)\ (f\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ xs\ c\ !\ Suc\ n)) =$
$(P\ c\text{-}n \longrightarrow Q\ x\text{-}n1\ (trans\text{-}fun\ x\text{-}n1\ c\text{-}n))$
by (*simp add*: *f-Exec-Stream-nth-gr0-calc*)

lemma *f-Exec-Stream-Init-Pre-Post1*:
$[\![\ n < length\ xs;$
$c\text{-}n = f\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ xs\ c\ !\ n;\ x\text{-}n = xs\ !\ n\ ]\!] \Longrightarrow$
$(P1\ x\text{-}n \wedge P2\ c\text{-}n \longrightarrow Q\ (f\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ xs\ c\ !\ Suc\ n)) =$
$(P1\ x\text{-}n \wedge P2\ c\text{-}n \longrightarrow Q\ (trans\text{-}fun\ x\text{-}n\ c\text{-}n))$
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*:
$[\![\ n < length\ xs;$
$c\text{-}n = f\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ xs\ c\ !\ n;\ x\text{-}n = xs\ !\ n\ ]\!] \Longrightarrow$
$(P\ (xs\ !\ n)\ (f\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ xs\ c\ !\ n) \longrightarrow$
$Q\ (f\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ xs\ c\ !\ Suc\ n)) =$
$(P\ x\text{-}n\ c\text{-}n \longrightarrow Q\ (trans\text{-}fun\ x\text{-}n\ c\text{-}n))$
by (*simp add*: *f-Exec-Stream-Init-nth-Suc-calc*)

lemma *i-Exec-Stream-Pre-Post1*:
$[\![\ c\text{-}n = (i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ input\ c)^{\leftarrow\ c}\ n;\ x\text{-}n = input\ n\ ]\!] \Longrightarrow$
$(P1\ x\text{-}n \wedge P2\ c\text{-}n \longrightarrow Q\ (i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ input\ c\ n)) =$
$(P1\ x\text{-}n \wedge P2\ c\text{-}n \longrightarrow Q\ (trans\text{-}fun\ x\text{-}n\ c\text{-}n))$
by (*simp add*: *i-Exec-Stream-nth-calc-Previous*)

Direct relation between input and result after transition

lemma *i-Exec-Stream-Pre-Post2*:
$[\![\ c\text{-}n = (i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ input\ c)^{\leftarrow\ c}\ n;\ x\text{-}n = input\ n\ ]\!] \Longrightarrow$
$(P\ c\text{-}n \longrightarrow Q\ (input\ n)\ (i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ input\ c\ n)) =$
$(P\ c\text{-}n \longrightarrow Q\ x\text{-}n\ (trans\text{-}fun\ x\text{-}n\ c\text{-}n))$
by (*simp add*: *i-Exec-Stream-nth-calc-Previous*)

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

⟦ *c-n* = *i-Exec-Comp-Stream trans-fun input c n*; *x-n1* = *input* (*Suc n*) ⟧ ⟹
(*P c-n* ⟶ *Q* (*input* (*Suc n*)) (*i-Exec-Comp-Stream trans-fun input c* (*Suc n*)))
=
(*P c-n* ⟶ *Q x-n1* (*trans-fun x-n1 c-n*))
by (*simp add*: *i-Exec-Stream-nth-gr0-calc*)

lemma *i-Exec-Stream-Init-Pre-Post1*:
⟦ *c-n* = *i-Exec-Comp-Stream-Init trans-fun input c n*; *x-n* = *input n* ⟧ ⟹
(*P1 x-n* ∧ *P2 c-n* ⟶ *Q* (*i-Exec-Comp-Stream-Init trans-fun input c* (*Suc n*))) =
(*P1 x-n* ∧ *P2 c-n* ⟶ *Q* (*trans-fun x-n c-n*))
by (*simp add*: *i-Exec-Stream-Init-nth-Suc-calc*)

Direct relation between input and state before transition

lemma *i-Exec-Stream-Init-Pre-Post2*:
⟦ *c-n* = *i-Exec-Comp-Stream-Init trans-fun input c n*; *x-n* = *input n* ⟧ ⟹
(*P* (*input n*) (*i-Exec-Comp-Stream-Init trans-fun input c n*) ⟶
Q (*i-Exec-Comp-Stream-Init trans-fun input c* (*Suc n*))) =
(*P x-n c-n* ⟶ *Q* (*trans-fun x-n c-n*))
by (*simp add*: *i-Exec-Stream-Init-nth-Suc-calc*)

Basic results for stream prefices

lemma *f-Exec-Stream-prefix*:
prefix xs ys ⟹
prefix (*f-Exec-Comp-Stream trans-fun xs c*)
(*f-Exec-Comp-Stream trans-fun ys c*)
by (*clarsimp simp*: *prefix-def f-Exec-Stream-append*)

lemma *i-Exec-Stream-prefix*:
xs ⊑ *input* ⟹
f-Exec-Comp-Stream trans-fun xs c ⊑
i-Exec-Comp-Stream trans-fun input c
by (*simp add*: *iprefix-eq-iprefix-take i-Exec-Stream-take*)

lemma *f-Exec-N-prefix*:
⟦ *n* ≤ *length xs*; *prefix xs ys* ⟧ ⟹
f-Exec-Comp-N trans-fun n xs c =
f-Exec-Comp-N trans-fun n ys c
by (*simp add*: *f-Exec-Comp-N-def prefix-imp-take-eq*)

theorem *f-Exec-Stream-prefix-causal*[*rule-format*]:
n ≤ *length* (*xs* ⊓ *ys*) ⟹
f-Exec-Comp-Stream trans-fun xs c ↓ *n* =
f-Exec-Comp-Stream trans-fun ys c ↓ *n*
by (*rule f-Exec-Stream-causal*, *rule inf-prefix-take-correct*)

lemma *f-Exec-Stream-Init-prefix*:
prefix xs ys ⟹
prefix (*f-Exec-Comp-Stream-Init trans-fun xs c*)
(*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* $\Longrightarrow$
f-Exec-Comp-Stream-Init trans-fun xs c $\sqsubseteq$
i-Exec-Comp-Stream-Init trans-fun input c
by (*simp add*: *iprefix-eq-iprefix-take i-Exec-Stream-Init-take*)

theorem *f-Exec-Stream-Init-prefix-strictly-causal*[*rule-format*]:
n $\leq$ *length* (*xs* $\sqcap$ *ys*) $\Longrightarrow$
f-Exec-Comp-Stream-Init trans-fun xs c $\downarrow$ *Suc n* =
f-Exec-Comp-Stream-Init trans-fun ys c $\downarrow$ *Suc n*
by (*rule f-Exec-Stream-Init-strictly-causal*, *rule inf-prefix-take-correct*)

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

definition *Deterministic-Trans-Fun* ::
(*'comp*, *'input*) *Comp-Trans-Fun* $\Rightarrow$ (*'comp*, *'state*) *Comp-Local-State* $\Rightarrow$ *bool*
where *Deterministic-Trans-Fun trans-fun localState* $\equiv$
$\forall$ *c1 c2 x*. *localState c1* = *localState c2* $\longrightarrow$ *trans-fun x c1* = *trans-fun x c2*

lemma *Deterministic-f-Exec*:
$[\![$ *Deterministic-Trans-Fun trans-fun localState*; *localState c1* = *localState c2*; *xs* $\neq$ [] $]\!]$ $\Longrightarrow$
f-Exec-Comp trans-fun xs c1 = *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*:
$[\![$ *Deterministic-Trans-Fun trans-fun localState*; *localState c1* = *localState c2* $]\!]$ $\Longrightarrow$
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 (*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* $]\!]$ $\Longrightarrow$
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* ⇒ *'a fstream-af* ⇒ *bool*
  where *f-Streams-Connected outS inS* ≡ *inS* = *ε* # *outS*

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

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

lemma *f-Streams-Connected-imp-not-empty*: *f-Streams-Connected outS inS* ⟹ *inS* ≠ []
by (*simp add*: *f-Streams-Connected-def*)

lemma *f-Streams-Connected-nth-conv*:
  f-Streams-Connected outS inS =
  (*length inS* = *Suc* (*length outS*) ∧
  (∀ *i*<*length inS*. *inS* ! *i* = (*case i of 0* ⇒ *ε* | *Suc k* ⇒ *outS* ! *k*)))
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*) ∧
  (∀ *i*<*length inS*. *inS* ! *i* = (*if i* = *0 then ε else outS* ! (*i* − *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 =
  (∀ *i*. *inS i* = (*case i of 0* ⇒ *ε* | *Suc k* ⇒ *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 =
  (∀ *i*. *inS i* = (*if i* = *0 then ε else outS* (*i* − *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*:
⟦ *output-fun c* = *ε*;
f-Streams-Connected
(*map output-fun* (*f-Exec-Comp-Stream trans-fun xs c*))
channel ⟧ ⟹
map output-fun (*f-Exec-Comp-Stream-Init trans-fun xs c*) = *channel*
by (*simp add*: *f-Streams-Connected-def f-Exec-Stream-Init-eq-f-Exec-Stream-Cons*)

lemma *i-Exec-Stream-Init-eq-output-channel*:
⟦ *output-fun c* = *ε*;
i-Streams-Connected
(*output-fun* ∘ (*i-Exec-Comp-Stream trans-fun input c*))
channel ⟧ ⟹
output-fun ∘ (*i-Exec-Comp-Stream-Init trans-fun input c*) = *channel*
by (*simp add*: *i-Streams-Connected-def i-Exec-Stream-Init-eq-i-Exec-Stream-Cons*)

lemma *f-Exec-Stream-output-causal*:
⟦ *xs* ↓ *n* = *ys* ↓ *n*;
output1 = *map output-fun* (*f-Exec-Comp-Stream trans-fun xs c*);
output2 = *map output-fun* (*f-Exec-Comp-Stream trans-fun ys c*) ⟧ ⟹
output1 ↓ *n* = *output2* ↓ *n*
by (*simp add*: *take-map f-Exec-Stream-causal*[*of n xs*])

lemma *f-Exec-Stream-Init-output-strictly-causal*:
⟦ *xs* ↓ *n* = *ys* ↓ *n*;
output1 = *map output-fun* (*f-Exec-Comp-Stream-Init trans-fun xs c*);
output2 = *map output-fun* (*f-Exec-Comp-Stream-Init trans-fun ys c*) ⟧ ⟹
output1 ↓ *Suc n* = *output2* ↓ *Suc n*
by (*simp add*: *take-map f-Exec-Stream-Init-strictly-causal*[*of n xs*])

lemma *i-Exec-Stream-output-causal*:
⟦ *input1* ⇓ *n* = *input2* ⇓ *n*;
output1 = *output-fun* ∘ *i-Exec-Comp-Stream trans-fun input1 c*;
output2 = *output-fun* ∘ *i-Exec-Comp-Stream trans-fun input2 c* ⟧ ⟹
output1 ⇓ *n* = *output2* ⇓ *n*
by (*simp add*: *i-Exec-Stream-causal*[*of n input1*])

lemma *i-Exec-Stream-Init-output-strictly-causal*:
⟦ *input1* ⇓ *n* = *input2* ⇓ *n*;
output1 = *output-fun* ∘ *i-Exec-Comp-Stream-Init trans-fun input1 c*;
output2 = *output-fun* ∘ *i-Exec-Comp-Stream-Init trans-fun input2 c* ⟧ ⟹
output1 ⇓ *Suc n* = *output2* ⇓ *Suc n*
by (*simp add*: *i-Exec-Stream-Init-strictly-causal*[*of n input1*])

lemma *f-Exec-Stream-Connected-strictly-causal*:
⟦ *xs* ↓ *n* = *ys* ↓ *n*;

f-Streams-Connected
(*map output-fun* (*f-Exec-Comp-Stream trans-fun xs c*))
channel1;
f-Streams-Connected
(*map output-fun* (*f-Exec-Comp-Stream trans-fun ys c*))
channel2 ⟧ ⟹
channel1 ↓ *Suc n* = *channel2* ↓ *Suc n*
by (*simp add*: *f-Streams-Connected-def take-map f-Exec-Stream-take*)

lemma *i-Exec-Stream-Connected-strictly-causal*:
⟦ *input1* ⇓ *n* = *input2* ⇓ *n*;
i-Streams-Connected
(*portOutput* ∘ (*i-Exec-Comp-Stream trans-fun input1 c*))
channel1;
i-Streams-Connected
(*portOutput* ∘ (*i-Exec-Comp-Stream trans-fun input2 c*))
channel2 ⟧ ⟹
channel1 ⇓ *Suc n* = *channel2* ⇓ *Suc n*
by (*simp add*: *i-Streams-Connected-def i-take-Suc-Cons i-Exec-Stream-take*)

A predicate for the semantics with initial state in result stream indicating for two message streams that the ports, they correspond to, are connected.

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

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

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

lemma *f-Streams-Connected-Init-nth-conv*:
f-Streams-Connected-Init outS inS =
(*length inS* = *length outS* ∧ (∀ *i*<*length inS*. *inS* ! *i* = *outS* ! *i*))
by (*simp add*: *f-Streams-Connected-Init-def list-eq-iff*)

lemma *i-Streams-Connected-Init-nth-conv*:
i-Streams-Connected-Init outS inS =
(∀ *i*. *inS i* = *outS i*)
by (*simp add*: *i-Streams-Connected-Init-def ilist-eq-iff*)

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

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*:
⟦ *output-fun c* = *ε*;
i-Streams-Connected-Init
(*output-fun* ∘ (*i-Exec-Comp-Stream-Init trans-fun input c*))
channel ⟧ ⟹
output-fun ∘ (*i-Exec-Comp-Stream-Init trans-fun input c*) = *channel*
by (*simp add*: *i-Streams-Connected-Init-def i-Exec-Stream-Init-eq-i-Exec-Stream-Cons*)

lemma *f-Exec-Stream-Connected-Init-strictly-causal*:
⟦ *xs* ↓ *n* = *ys* ↓ *n*;
f-Streams-Connected-Init
(*map output-fun* (*f-Exec-Comp-Stream-Init trans-fun xs c*))
channel1;
f-Streams-Connected-Init
(*map output-fun* (*f-Exec-Comp-Stream-Init trans-fun ys c*))
channel2 ⟧ ⟹
channel1 ↓ *Suc n* = *channel2* ↓ *Suc n*
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*:
⟦ *input1* ⇓ *n* = *input2* ⇓ *n*;
i-Streams-Connected-Init
(*portOutput* ∘ (*i-Exec-Comp-Stream-Init trans-fun input1 c*))
channel1;
i-Streams-Connected-Init
(*portOutput* ∘ (*i-Exec-Comp-Stream-Init trans-fun input2 c*))
channel2 ⟧ ⟹
channel1 ⇓ *Suc n* = *channel2* ⇓ *Suc n*
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*:
⟦ *n1* ≤ *n2*; *n1* ∈ *I*; *n2* ∈ *I*;
¬ *P* (*f-Exec-Comp trans-fun* (*input* ↓ *n1*) *c*);
P (*f-Exec-Comp trans-fun* (*input* ↓ *n2*) *c*) ⟧ ⟹
∃ *n*∈*I*. *n1* ≤ *n* ∧ *n* < *n2* ∧
¬ *P* (*f-Exec-Comp trans-fun* (*input* ↓ *n*) *c*) ∧
P (*f-Exec-Comp trans-fun* (*input* ↓ (*inext n I*)) *c*)
by (*rule inext-predicate-change-exists*)

lemma *f-State-Change-exists*:
⟦ *n1* ≤ *n2*;

$\neg\ P\ (f\text{-}Exec\text{-}Comp\ trans\text{-}fun\ (input \downarrow n1)\ c);$
$P\ (f\text{-}Exec\text{-}Comp\ trans\text{-}fun\ (input \downarrow n2)\ c)\ \rrbracket \Longrightarrow$
$\exists\, n{\geq}n1.\ n < n2\ \wedge$
$\neg\ P\ (f\text{-}Exec\text{-}Comp\ trans\text{-}fun\ (input \downarrow n)\ c)\ \wedge$
$P\ (f\text{-}Exec\text{-}Comp\ trans\text{-}fun\ (input \downarrow (Suc\ n))\ c)$
by (*rule nat-Suc-predicate-change-exists*)

lemma *i-State-Change-exists-set*:
$\llbracket\ n1 \leq n2;\ n1 \in I;\ n2 \in I;$
$\neg\ P\ (i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ input\ c\ n1);$
$P\ (i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ input\ c\ n2)\ \rrbracket \Longrightarrow$
$\exists\, n{\in}I.\ n1 \leq n \wedge n < n2\ \wedge$
$\neg\ P\ (i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ input\ c\ n)\ \wedge$
$P\ (i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ input\ c\ (inext\ n\ I))$
by (*rule inext-predicate-change-exists*)

lemma *i-State-Change-exists*:
$\llbracket\ n1 \leq n2;$
$\neg\ P\ (i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ input\ c\ n1);$
$P\ (i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ input\ c\ n2)\ \rrbracket \Longrightarrow$
$\exists\, n{\geq}n1.\ n < n2\ \wedge$
$\neg\ P\ (i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ input\ c\ n)\ \wedge$
$P\ (i\text{-}Exec\text{-}Comp\text{-}Stream\ trans\text{-}fun\ input\ c\ (Suc\ n))$
by (*rule nat-Suc-predicate-change-exists*)

lemma *i-State-Change-Init-exists-set*:
$\llbracket\ n1 \leq n2;\ n1 \in I;\ n2 \in I;$
$\neg\ P\ (i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ input\ c\ n1);$
$P\ (i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ input\ c\ n2)\ \rrbracket \Longrightarrow$
$\exists\, n{\in}I.\ n1 \leq n \wedge n < n2\ \wedge$
$\neg\ P\ (i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ input\ c\ n)\ \wedge$
$P\ (i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ input\ c\ (inext\ n\ I))$
by (*rule inext-predicate-change-exists*)

lemma *i-State-Change-Init-exists*:
$\llbracket\ n1 \leq n2;$
$\neg\ P\ (i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ input\ c\ n1);$
$P\ (i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ input\ c\ n2)\ \rrbracket \Longrightarrow$
$\exists\, n{\geq}n1.\ n < n2\ \wedge$
$\neg\ P\ (i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ input\ c\ n)\ \wedge$
$P\ (i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ input\ c\ (Suc\ n))$
by (*rule nat-Suc-predicate-change-exists*)

3.2 Components with accelerated execution

This section deals with variable execution speed components. A component accelerated by a (clocking) factor k processes streams expanded by factor k and its output streams are compressed by factor k.

3.2.1 Equivalence relation for executions

A predicate indicating for two components together with transition functions and a given equivalence predicate for their local states, that the components exhibit equivalent observable behaviour after expanding input streams and shrinking output streams by a constant factor, given that their local states are equivalent with respect to the specified equivalence relations.

definition
Equiv-Exec ::
'input ⇒
(*'state1* ⇒ *'state2* ⇒ *bool*) ⇒ — Equivalence predicate for local states
(*'comp1*, *'state1*) *Comp-Local-State* ⇒
(*'comp2*, *'state2*) *Comp-Local-State* ⇒
(*'input*, *'input1*) *Port-Input-Value* ⇒ — Input adaptor for first component
(*'input*, *'input2*) *Port-Input-Value* ⇒ — Input adaptor for second component
(*'comp1*, *'output*) *Port-Output-Value* ⇒
(*'comp2*, *'output*) *Port-Output-Value* ⇒
(*'comp1*, *'input1 message-af*) *Comp-Trans-Fun* ⇒
(*'comp2*, *'input2 message-af*) *Comp-Trans-Fun* ⇒
nat ⇒ *nat* ⇒ *'comp1* ⇒ *'comp2* ⇒ *bool*
where
Equiv-Exec
m equiv-states
localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2 ≡
equiv-states (*localState1 c1*) (*localState2 c2*) ⟶ (
last-message (*map output-fun1* (
f-Exec-Comp-Stream trans-fun1 (*input-fun1 m* # $\varepsilon^{k1 - Suc\ 0}$) *c1*)) =
last-message (*map output-fun2* (
f-Exec-Comp-Stream trans-fun2 (*input-fun2 m* # $\varepsilon^{k2 - Suc\ 0}$) *c2*)) ∧
equiv-states
(*localState1* (*f-Exec-Comp trans-fun1* (*input-fun1 m* # $\varepsilon^{k1 - Suc\ 0}$) *c1*))
(*localState2* (*f-Exec-Comp trans-fun2* (*input-fun2 m* # $\varepsilon^{k2 - Suc\ 0}$) *c2*)))

Predicate indicating for two components together with transition functions and a given equivalence predicate for their local states, that the equivalence predicate is stable with respect to component execution, i.e., it determines the equivalence of components' local states both for the initial states and after the components have processed an arbitrary input. The restricting version *Equiv-Exec-stable-set* guarantees stability only for inputs from a given restriction set, the not-restricting version guarantees stability for all inputs.

definition
Equiv-Exec-stable-set ::
'input set ⇒
(*'state1* ⇒ *'state2* ⇒ *bool*) ⇒ — Equivalence predicate for local states
(*'comp1*, *'state1*) *Comp-Local-State* ⇒
(*'comp2*, *'state2*) *Comp-Local-State* ⇒

$('input, 'input1)\ Port\text{-}Input\text{-}Value \Rightarrow$ — Input adaptor for first component
$('input, 'input2)\ Port\text{-}Input\text{-}Value \Rightarrow$ — Input adaptor for second component
$('comp1, 'output)\ Port\text{-}Output\text{-}Value \Rightarrow$
$('comp2, 'output)\ Port\text{-}Output\text{-}Value \Rightarrow$
$('comp1, 'input1\ message\text{-}af)\ Comp\text{-}Trans\text{-}Fun \Rightarrow$
$('comp2, 'input2\ message\text{-}af)\ Comp\text{-}Trans\text{-}Fun \Rightarrow$
$nat \Rightarrow nat \Rightarrow 'comp1 \Rightarrow 'comp2 \Rightarrow bool$
where
$Equiv\text{-}Exec\text{-}stable\text{-}set\ A$
$equiv\text{-}states\ localState1\ localState2\ input\text{-}fun1\ input\text{-}fun2\ output\text{-}fun1\ output\text{-}fun2$
$trans\text{-}fun1\ trans\text{-}fun2\ k1\ k2\ c1\ c2 \equiv$
$\forall\ input\ m.\ set\ input \subseteq A \wedge m \in A \longrightarrow$
$Equiv\text{-}Exec\ m$
$equiv\text{-}states\ localState1\ localState2\ input\text{-}fun1\ input\text{-}fun2\ output\text{-}fun1\ output\text{-}fun2$
$trans\text{-}fun1\ trans\text{-}fun2\ k1\ k2$
$(f\text{-}Exec\text{-}Comp\ trans\text{-}fun1\ (map\ input\text{-}fun1\ input \odot_f k1)\ c1)$
$(f\text{-}Exec\text{-}Comp\ trans\text{-}fun2\ (map\ input\text{-}fun2\ input \odot_f k2)\ c2)$

definition
$Equiv\text{-}Exec\text{-}stable ::$
$('state1 \Rightarrow 'state2 \Rightarrow bool) \Rightarrow$ — Equivalence predicate for local states
$('comp1, 'state1)\ Comp\text{-}Local\text{-}State \Rightarrow$
$('comp2, 'state2)\ Comp\text{-}Local\text{-}State \Rightarrow$
$('input, 'input1)\ Port\text{-}Input\text{-}Value \Rightarrow$ — Input adaptor for first component
$('input, 'input2)\ Port\text{-}Input\text{-}Value \Rightarrow$ — Input adaptor for second component
$('comp1, 'output)\ Port\text{-}Output\text{-}Value \Rightarrow$
$('comp2, 'output)\ Port\text{-}Output\text{-}Value \Rightarrow$
$('comp1, 'input1\ message\text{-}af)\ Comp\text{-}Trans\text{-}Fun \Rightarrow$
$('comp2, 'input2\ message\text{-}af)\ Comp\text{-}Trans\text{-}Fun \Rightarrow$
$nat \Rightarrow nat \Rightarrow 'comp1 \Rightarrow 'comp2 \Rightarrow bool$
where
$Equiv\text{-}Exec\text{-}stable$
$equiv\text{-}states\ localState1\ localState2\ input\text{-}fun1\ input\text{-}fun2\ output\text{-}fun1\ output\text{-}fun2$
$trans\text{-}fun1\ trans\text{-}fun2\ k1\ k2\ c1\ c2 \equiv$
$\forall\ input\ m.$
$Equiv\text{-}Exec\ m$
$equiv\text{-}states\ localState1\ localState2\ input\text{-}fun1\ input\text{-}fun2\ output\text{-}fun1\ output\text{-}fun2$
$trans\text{-}fun1\ trans\text{-}fun2\ k1\ k2$
$(f\text{-}Exec\text{-}Comp\ trans\text{-}fun1\ (map\ input\text{-}fun1\ input \odot_f k1)\ c1)$
$(f\text{-}Exec\text{-}Comp\ trans\text{-}fun2\ (map\ input\text{-}fun2\ input \odot_f k2)\ c2)$

lemma $Equiv\text{-}Exec\text{-}equiv\text{-}statesI$:
$[\![\ equiv\text{-}states\ (localState1\ c1)\ (localState2\ c2);$
$Equiv\text{-}Exec$
$m\ equiv\text{-}states$
$localState1\ localState2\ input\text{-}fun1\ input\text{-}fun2\ output\text{-}fun1\ output\text{-}fun2$
$trans\text{-}fun1\ trans\text{-}fun2\ k1\ k2\ c1\ c2\ ]\!] \Longrightarrow$

equiv-states
(*localState1* (*f-Exec-Comp trans-fun1* (*input-fun1 m* # $\varepsilon^{k1 - Suc\ 0}$) *c1*))
(*localState2* (*f-Exec-Comp trans-fun2* (*input-fun2 m* # $\varepsilon^{k2 - Suc\ 0}$) *c2*))
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* # $\varepsilon^{k1 - Suc\ 0}$) *c1*)) =
last-message (*map output-fun2* (
f-Exec-Comp-Stream trans-fun2 (*input-fun2 m* # $\varepsilon^{k2 - Suc\ 0}$) *c2*))
by (*simp add*: *Equiv-Exec-def*)

lemma *Equiv-Exec-equiv-statesI'*:
⟦ *equiv-states* (*localState1 c1*) (*localState2 c2*);
Equiv-Exec
m equiv-states
localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2 ⟧ ⟹
equiv-states
(*localState1* (*f-Exec-Comp trans-fun1 NoMsg*$^{k1 - Suc\ 0}$ (*trans-fun1* (*input-fun1 m*) *c1*)))
(*localState2* (*f-Exec-Comp trans-fun2 NoMsg*$^{k2 - Suc\ 0}$ (*trans-fun2* (*input-fun2 m*) *c2*)))
by (*simp add*: *Equiv-Exec-def*)

lemma *Equiv-Exec-le1*:
⟦ *k1* ≤ *Suc 0*; *k2* ≤ *Suc 0*;
equiv-states (*localState1 c1*) (*localState2 c2*);
Equiv-Exec m
equiv-states localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2 ⟧ ⟹
output-fun1 (*trans-fun1* (*input-fun1 m*) *c1*) =
output-fun2 (*trans-fun2* (*input-fun2 m*) *c2*) ∧
equiv-states
(*localState1* (*trans-fun1* (*input-fun1 m*) *c1*))
(*localState2* (*trans-fun2* (*input-fun2 m*) *c2*))
by (*simp add*: *Equiv-Exec-def*)

lemma *Equiv-Exec-stable-set-UNIV*:
Equiv-Exec-stable-set
UNIV equiv-states
localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2

trans-fun1 trans-fun2 k1 k2 c1 c2 =
Equiv-Exec-stable
equiv-states
localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2
by (*simp add*: *Equiv-Exec-stable-set-def Equiv-Exec-stable-def*)

lemma *Equiv-Exec-stable-setI*:
⟦ *Equiv-Exec-stable-set A*
equiv-states localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2;
set input ⊆ *A*; *m* ∈ *A* ⟧ ⟹
Equiv-Exec
m equiv-states
localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2
(*f-Exec-Comp trans-fun1* (*map input-fun1 input* ⊙$_f$ *k1*) *c1*)
(*f-Exec-Comp trans-fun2* (*map input-fun2 input* ⊙$_f$ *k2*) *c2*)
by (*simp add*: *Equiv-Exec-stable-set-def*)

lemma *Equiv-Exec-stableI*:
Equiv-Exec-stable
equiv-states localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2 ⟹
Equiv-Exec m
equiv-states localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2
(*f-Exec-Comp trans-fun1* (*map input-fun1 input* ⊙$_f$ *k1*) *c1*)
(*f-Exec-Comp trans-fun2* (*map input-fun2 input* ⊙$_f$ *k2*) *c2*)
by (*simp add*: *Equiv-Exec-stable-def*)

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

lemma *Equiv-Exec-refl*:
⟦ ⋀*c*. *equiv-states* (*localState c*) (*localState c*) ⟧ ⟹
Equiv-Exec
m equiv-states
localState localState input-fun input-fun output-fun output-fun
trans-fun trans-fun k k c c
by (*simp add*: *Equiv-Exec-def*)

lemma *Equiv-Exec-sym*[*rule-format*]:
⟦ ∀ *c1 c2*.
equiv-states (*localState1 c1*) (*localState2 c2*) =
equiv-states (*localState2 c2*) (*localState1 c1*) ⟧ ⟹
Equiv-Exec
m equiv-states
localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2 =
Equiv-Exec

m equiv-states
localState2 localState1 input-fun2 input-fun1 output-fun2 output-fun1
trans-fun2 trans-fun1 k2 k1 c2 c1
by (*fastforce simp*: *Equiv-Exec-def*)

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

lemma *Equiv-Exec-sym2-ex*:
∃ *equiv-states-sym*.
Equiv-Exec
m equiv-states
localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2 =
Equiv-Exec
m equiv-states-sym
localState2 localState1 input-fun2 input-fun1 output-fun2 output-fun1
trans-fun2 trans-fun1 k2 k1 c2 c1
by (*rule exI*, *rule Equiv-Exec-sym2*, *simp*)

lemma *Equiv-Exec-trans*:
⟦ *Equiv-Exec*
m equiv-states12
localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2;
Equiv-Exec
m equiv-states23
localState2 localState3 input-fun2 input-fun3 output-fun2 output-fun3
trans-fun2 trans-fun3 k2 k3 c2 c3;
equiv-states13 = (λ*s1 s3*. (
if s1 = *localState1 c1* ∧ *s3* = *localState3 c3 then*
equiv-states12 s1 (*localState2 c2*) ∧
equiv-states23 (*localState2 c2*) *s3*
else
equiv-states12 s1 (
localState2 (*f-Exec-Comp trans-fun2* (*input-fun2 m* # $\varepsilon^{k2 - Suc\ 0}$) *c2*)))
∧
equiv-states23 (
localState2 (*f-Exec-Comp trans-fun2* (*input-fun2 m* # $\varepsilon^{k2 - Suc\ 0}$) *c2*))
s3) ⟧ ⟹

Equiv-Exec
m equiv-states13
localState1 localState3 input-fun1 input-fun3 output-fun1 output-fun3
trans-fun1 trans-fun3 k1 k3 c1 c3
by (*fastforce simp*: *Equiv-Exec-def*)

lemma *Equiv-Exec-trans-ex*:
⟦ *Equiv-Exec*
m equiv-states12
localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2;
Equiv-Exec
m equiv-states23
localState2 localState3 input-fun2 input-fun3 output-fun2 output-fun3
trans-fun2 trans-fun3 k2 k3 c2 c3 ⟧ ⟹
∃ *equiv-states13*. *Equiv-Exec*
m equiv-states13
localState1 localState3 input-fun1 input-fun3 output-fun1 output-fun3
trans-fun1 trans-fun3 k1 k3 c1 c3
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 componenents, when the transition function is applied with the same input.

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

lemma *Exec-Equal-StateD*:
⟦ *Exec-Equal-State localState trans-fun*;
localState c1 = *localState c2* ⟧ ⟹
trans-fun m c1 = *trans-fun m c2*
by (*unfold Exec-Equal-State-def*, *blast*)

lemma *Exec-Equal-StateD'*:
Exec-Equal-State localState trans-fun ⟹
∀ *c1 c2 m*. *localState c1* = *localState c2* ⟶ *trans-fun m c1* = *trans-fun m c2*
by (*unfold Exec-Equal-State-def*, *blast*)

lemma *Exec-Equal-StateI*:
(⋀*c1 c2 m*. *localState c1* = *localState c2* ⟹ *trans-fun m c1* = *trans-fun m c2*)
⟹ *Exec-Equal-State localState trans-fun*
by (*unfold Exec-Equal-State-def*, *blast*)

lemma *f-Exec-Equal-State*: ⋀*c1 c2*.
⟦ *Exec-Equal-State localState trans-fun*;

localState c1 = localState c2; *xs* ≠ [] ⟧ ⟹
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*)
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*:
⟦ *Exec-Equal-State localState trans-fun*;
localState c1 = localState c2 ⟧ ⟹
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*:
⟦ *Exec-Equal-State localState trans-fun*;
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 f-Exec-Equal-State*)
apply *simp*+
done

3.2.2 Idle states

definition *State-Idle* ::
(*'comp*, *'state*) *Comp-Local-State* ⇒ (*'comp* ⇒ *'output message-af*) ⇒
(*'comp*, *'input message-af*) *Comp-Trans-Fun* ⇒ *'state* ⇒ *bool*
where *State-Idle localState output-fun trans-fun state* ≡
∀ *c*. *localState c* = *state* ⟶
localState (*trans-fun* ε *c*) = *state* ∧
output-fun (*trans-fun* ε *c*) = ε

lemma *State-IdleD*:
⟦ *State-Idle localState output-fun trans-fun state*;
localState c = *state* ⟧ ⟹
localState (*trans-fun* ε *c*) = *state* ∧
output-fun (*trans-fun* ε *c*) = ε

by (*unfold State-Idle-def*, *blast*)

lemma *State-IdleD'*:
State-Idle localState output-fun trans-fun state $\Longrightarrow$
$\forall$ *c*. *localState c* = *state* $\longrightarrow$
localState (*trans-fun* ε *c*) = *state* $\wedge$
output-fun (*trans-fun* ε *c*) = ε
by (*unfold State-Idle-def*, *blast*)

lemma *State-IdleI*:
$[\![$ $\bigwedge$*c*. *localState c* = *state* $\Longrightarrow$
localState (*trans-fun* ε *c*) = *state* $\wedge$
output-fun (*trans-fun* ε *c*) = ε $]\!]$ $\Longrightarrow$
State-Idle localState output-fun trans-fun state
by (*unfold State-Idle-def*, *blast*)

lemma *State-Idle-step*[*rule-format*]:
$[\![$ *State-Idle localState output-fun trans-fun* (*localState c*) $]\!]$ $\Longrightarrow$
State-Idle localState output-fun trans-fun (*localState* (*trans-fun* ε *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*. *State-Idle localState output-fun trans-fun* (*localState c*) $\Longrightarrow$
localState (*f-Exec-Comp trans-fun* ε^n *c*) = *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*.
$[\![$ *State-Idle localState output-fun trans-fun* (*localState c*); *0* < *n* $]\!]$ $\Longrightarrow$
output-fun (*f-Exec-Comp trans-fun* ε^n *c*) = ε
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*]:
$[\![$ *State-Idle localState output-fun trans-fun* (*localState c*);
output-fun c = ε $]\!]$ $\Longrightarrow$

output-fun (*f-Exec-Comp trans-fun* ε^n *c*) = ε
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*]:
⟦ *State-Idle localState output-fun trans-fun* (*localState c*) ⟧ ⟹
map output-fun (*f-Exec-Comp-Stream trans-fun* ε^n *c*) = ε^n
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*:
⟦ *State-Idle localState output-fun trans-fun* (
localState (*f-Exec-Comp trans-fun xs c*)) ⟧ ⟹
localState (*f-Exec-Comp trans-fun* (*xs* @ ε^n) *c*) =
localState (*f-Exec-Comp trans-fun xs c*)
by (*simp add*: *f-Exec-append f-Exec-State-Idle-replicate-NoMsg-state*)

corollary *f-Exec-State-Idle-append-replicate-NoMsg-ge-state*:
⟦ *State-Idle localState output-fun trans-fun* (
localState (*f-Exec-Comp trans-fun* (*xs* @ ε^m) *c*));
$m \leq n$ ⟧ ⟹
localState (*f-Exec-Comp trans-fun* (*xs* @ ε^n) *c*) =
localState (*f-Exec-Comp trans-fun* (*xs* @ ε^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*:
⟦ *State-Idle localState output-fun trans-fun* (
localState (*f-Exec-Comp trans-fun* ε^m *c*));
$m \leq n$ ⟧ ⟹
localState (*f-Exec-Comp trans-fun* ε^n *c*) =
localState (*f-Exec-Comp trans-fun* ε^m *c*)
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*:
⟦ *State-Idle localState output-fun trans-fun* (
localState (*f-Exec-Comp trans-fun xs c*));
$0 < n$ ⟧ ⟹
output-fun (*f-Exec-Comp trans-fun* (*xs* @ ε^n) *c*) = ε
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*:
⟦ *State-Idle localState output-fun trans-fun* (
localState (*f-Exec-Comp trans-fun xs c*)) ⟧ ⟹
map output-fun (*f-Exec-Comp-Stream trans-fun* (*xs* @ ε^n) *c*) =
map output-fun (*f-Exec-Comp-Stream trans-fun xs c*) @ ε^n

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*:
⟦ *State-Idle localState output-fun trans-fun* (
localState (*f-Exec-Comp trans-fun* (*xs* @ ε^m) *c*));
$m < n$ ⟧ ⟹
output-fun (*f-Exec-Comp trans-fun* (*xs* @ ε^n) *c*) = ε
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*:
⟦ *State-Idle localState output-fun trans-fun* (
localState (*f-Exec-Comp trans-fun* (*xs* @ ε^m) *c*));
output-fun (*f-Exec-Comp trans-fun* (*xs* @ ε^m) *c*) = ε; $m \leq n$ ⟧ ⟹
output-fun (*f-Exec-Comp trans-fun* (*xs* @ ε^n) *c*) = ε
by (*fastforce simp*: *order-le-less f-Exec-State-Idle-append-replicate-NoMsg-gr-output*)

corollary *f-Exec-State-Idle-replicate-NoMsg-gr-output*:
⟦ *State-Idle localState output-fun trans-fun* (
localState (*f-Exec-Comp trans-fun* ε^m *c*));
$m < n$ ⟧ ⟹
output-fun (*f-Exec-Comp trans-fun* ε^n *c*) = ε
by (*cut-tac xs*=[] **in** *f-Exec-State-Idle-append-replicate-NoMsg-gr-output*, *simp*+)

corollary *f-Exec-State-Idle-replicate-NoMsg-ge-output*:
⟦ *State-Idle localState output-fun trans-fun* (
localState (*f-Exec-Comp trans-fun* ε^m *c*));
output-fun (*f-Exec-Comp trans-fun* ε^m *c*) = ε; $m \leq n$ ⟧ ⟹
output-fun (*f-Exec-Comp trans-fun* ε^n *c*) = ε
by (*fastforce simp*: *order-le-less f-Exec-State-Idle-replicate-NoMsg-gr-output*)

lemma *State-Idle-append-replicate-NoMsg-output-last-message*:
⟦ *State-Idle localState output-fun trans-fun* (
localState (*f-Exec-Comp trans-fun xs c*)) ⟧ ⟹
last-message (*map output-fun* (*f-Exec-Comp-Stream trans-fun* (*xs* @ ε^n) *c*)) =
last-message (*map output-fun* (*f-Exec-Comp-Stream trans-fun xs c*))
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*:
⟦ *State-Idle localState output-fun trans-fun* (
localState (*f-Exec-Comp trans-fun xs c*));
output-fun (*f-Exec-Comp trans-fun xs c*) ≠ ε;
xs ≠ [] ⟧ ⟹
last-message (*map output-fun* (*f-Exec-Comp-Stream trans-fun* (*xs* @ ε^n) *c*)) =
output-fun (*f-Exec-Comp trans-fun xs c*)
apply (*simp add*: *State-Idle-append-replicate-NoMsg-output-last-message f-Exec-eq-f-Exec-Stream-last2*

)
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*:
⟦ *State-Idle localState output-fun trans-fun* (
localState (*f-Exec-Comp trans-fun xs c*));
output-fun (*f-Exec-Comp trans-fun xs c*) ≠ *ε*;
xs ≠ [] ⟧ ⟹
last-message (*map output-fun* (*f-Exec-Comp-Stream trans-fun xs c*)) =
output-fun (*f-Exec-Comp trans-fun xs c*)
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*:
⟦ ¬ *State-Idle localState output-fun trans-fun* (*localState c*);
State-Idle localState output-fun trans-fun (*localState* (*f-Exec-Comp trans-fun* ε^n *c*)) ⟧ ⟹
∃ *i*<*n*. (
¬ *State-Idle localState output-fun trans-fun* (*localState* (*f-Exec-Comp trans-fun* ε^i *c*)) ∧ (
∀ *j*≤*n*. *i* < *j* ⟶ *State-Idle localState output-fun trans-fun* (*localState* (*f-Exec-Comp trans-fun* ε^j *c*))))
apply (*cut-tac*
a=0 **and** *b=n* **and**
P=λ*x*. *State-Idle localState output-fun trans-fun* (*localState* (*f-Exec-Comp trans-fun* $NoMsg^x$ *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* ε^n *c*)) ⟧ ⟹
∃ *i*<*n*. (
(∀ *j*≤*i*. ¬ *State-Idle localState output-fun trans-fun* (*localState* (*f-Exec-Comp trans-fun* ε^i *c*))) ∧
(∀ *j*≤*n*. *i* < *j* ⟶ *State-Idle localState output-fun trans-fun* (*localState* (*f-Exec-Comp trans-fun* ε^j *c*))))
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*)) ÷$_{fl}$ *k*

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

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

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

(*output-fun c*) # *f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c*

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

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

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

lemma *f-Exec-Stream-Acc-Output-length*[*simp*]:
0 < *k* ⟹
length (*f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c*) = *length xs*
by (*simp add*: *f-Exec-Comp-Stream-Acc-Output-def f-shrink-length*)

lemma *f-Exec-Stream-Acc-LocalState-length*[*simp*]:
0 < *k* ⟹
length (*f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c*) = *length xs*
by (*simp add*: *f-Exec-Comp-Stream-Acc-LocalState-def f-shrink-last-length*)

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

3.2.4 Basic results for accelerated execution

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

by (*simp add*: *f-Exec-Comp-Stream-Acc-Output-def*)

lemma *f-Exec-Stream-Acc-LocalState-Nil*[*simp*]:
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun [] *c* = []
by (*simp add*: *f-Exec-Comp-Stream-Acc-LocalState-def*)

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

lemma *f-Exec-Stream-Acc-Output-0*[*simp*]:
f-Exec-Comp-Stream-Acc-Output 0 output-fun trans-fun xs c = []
by (*simp add*: *f-Exec-Comp-Stream-Acc-Output-def*)

lemma *f-Exec-Stream-Acc-LocalState-0*[*simp*]:
f-Exec-Comp-Stream-Acc-LocalState 0 localState trans-fun xs c = []
by (*simp add*: *f-Exec-Comp-Stream-Acc-LocalState-def*)

lemmas *f-Exec-Stream-Acc-0* =
f-Exec-Stream-Acc-LocalState-0
f-Exec-Stream-Acc-Output-0

lemma *f-Exec-Stream-Acc-Output-1*[*simp*]:
f-Exec-Comp-Stream-Acc-Output (*Suc 0*) *output-fun trans-fun xs c* =
map output-fun (*f-Exec-Comp-Stream trans-fun xs c*)
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 ∘ (*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 ∘ (*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*)) $\longmapsto_f$ *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* ⟹
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c =

(*output-fun* ∘ (*i-Exec-Comp-Stream trans-fun* (*input* $\odot_i$ *k*) *c*)) $\longmapsto_i$ *k* $\div_{il}$ *k*
by (*simp add*: *i-Exec-Comp-Stream-Acc-Output-def i-shrink-eq-i-last-message-hold-shrink-last*)

lemma *f-Exec-Stream-Acc-Output-take*:
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c ↓ *n* =
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun (*xs* ↓ *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 ↑ *n* =
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun (*xs* ↑ *n*) (
f-Exec-Comp trans-fun (*xs* ↓ *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* ⟹
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c ⇓ *n* =
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun (*input* ⇓ *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* ⟹
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c ⇑ *n* =
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun (*input* ⇑ *n*) (
f-Exec-Comp trans-fun (*input* ⇓ *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* ⟹
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input c ⇓ *n* =
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun (*input* ⇓ *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* ⟹
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input c ⇑ *n* =
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun (*input* ⇑ *n*) (
f-Exec-Comp trans-fun (*input* ⇓ *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-d*

lemma *f-Exec-Stream-Acc-Output-append*:
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun (*xs* @ *ys*) *c* =
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c @
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun ys (
f-Exec-Comp trans-fun (*xs* $\odot_f$ *k*) *c*)
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 ⟹
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun (x # xs) c =
last-message (map output-fun (f-Exec-Comp-Stream trans-fun (x # ε^k − Suc 0) c)) #
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs (f-Exec-Comp trans-fun (x # ε^k − Suc 0) c)
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 ⟹
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun [x] c =
[last-message (map output-fun (f-Exec-Comp-Stream trans-fun (x # ε^k − Suc 0) c))]
by (*simp add*: *f-Exec-Stream-Acc-Output-Cons*)

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

lemma *i-Exec-Stream-Acc-Output-append*:
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun (xs ⌢ input) c =
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c ⌢
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input (f-Exec-Comp trans-fun (xs ⊙_f k) c)
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*:
0 < k ⟹
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun ([x] ⌢ input) c =
[last-message (map output-fun (f-Exec-Comp-Stream trans-fun (x # ε^k − Suc 0) c))] ⌢
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input (f-Exec-Comp trans-fun (x # ε^k − Suc 0) c)
by (*simp add*: *i-Exec-Stream-Acc-Output-append f-Exec-Stream-Acc-Output-one*)

lemma *f-Exec-Stream-Acc-LocalState-append*:
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun (xs @ ys) c =
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c @
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun ys (f-Exec-Comp trans-fun (xs ⊙_f k) c)
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*:
0 < k $\Longrightarrow$
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun (x # xs) c =
localState (f-Exec-Comp trans-fun (x # ε^k − Suc 0) c) #
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs (
f-Exec-Comp trans-fun (x # ε^k − Suc 0) c)
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*:
0 < k $\Longrightarrow$
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun [x] c =
[localState (f-Exec-Comp trans-fun (x # ε^k − Suc 0) c)]
by (*simp add*: *f-Exec-Stream-Acc-LocalState-Cons*)

lemma *f-Exec-Stream-Acc-LocalState-snoc*:
0 < k $\Longrightarrow$
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun (xs @ [x]) c =
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c @
[localState (f-Exec-Comp trans-fun ((xs @ [x]) ⊙_f k) c)]
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*:
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun (xs ⌢ input) c =
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c ⌢
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input (
f-Exec-Comp trans-fun (xs ⊙_f k) 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-map-aggregate-append*)

lemma *i-Exec-Stream-Acc-LocalState-Cons*:
0 < k $\Longrightarrow$
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun ([x] ⌢ input) c =
[localState (f-Exec-Comp trans-fun (x # ε^k − Suc 0) c)] ⌢
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input (
f-Exec-Comp trans-fun (x # ε^k − Suc 0) c)
by (*simp add*: *i-Exec-Stream-Acc-LocalState-append f-Exec-Stream-Acc-LocalState-one f-expand-one*)

lemma *f-Exec-Stream-Acc-Output-nth*:
⟦ *0 < k*; *n < length xs* ⟧ $\Longrightarrow$
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c ! n =
last-message (map output-fun (
f-Exec-Comp-Stream trans-fun (xs ! n # ε^k − Suc 0) (

f-Exec-Comp trans-fun (*xs* ↓ *n* ⊙$_f$ *k*) *c*)))
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*:
⟦ *0* < *k*; *n* < *n'* ⟧ ⟹
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun (*input* ⇓ *n'*) *c* ! *n* =
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c n
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*:
0 < *k* ⟹
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c n =
last-message (*map output-fun* (
f-Exec-Comp-Stream trans-fun (*input n* # $\varepsilon^{k - Suc\ 0}$) (
f-Exec-Comp trans-fun (*input* ⇓ *n* ⊙$_f$ *k*) *c*)))
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*:
0 < *k* ⟹
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c n =
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun (*input* ⇓ *Suc n*) *c* ! *n*
by (*simp add*: *f-Exec-Stream-Acc-Output-nth-eq-i-nth*)

corollary *i-Exec-Stream-Acc-Output-nth-f-last*:
0 < *k* ⟹
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c n =
last (*f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun* (*input* ⇓ *Suc n*) *c*)
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*:
⟦ *0* < *k*; *n* < *length xs* ⟧ ⟹
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c ! *n* =
localState (*f-Exec-Comp trans-fun* (*xs* ↓ *Suc n* ⊙$_f$ *k*) *c*)
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*:
⟦ *0* < *k*; *n* < *n'* ⟧ ⟹
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun (*input* ⇓ *n'*) *c* ! *n* =
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input c n
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 ⟹
i-Exec-Comp-Stream-Acc-LocalState k output-fun trans-fun input c n =
f-Exec-Comp-Stream-Acc-LocalState k output-fun trans-fun (input ⇓ Suc n) c ! n
by (*simp add*: *f-Exec-Stream-Acc-LocalState-nth-eq-i-nth*)

corollary *i-Exec-Stream-Acc-LocalState-nth-f-last*:
0 < k ⟹
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input c n =
last (*f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun* (*input ⇓ Suc n*) *c*)
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 ⟹
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input c n =
localState (*f-Exec-Comp trans-fun* (*input ⇓ Suc n ⊙$_f$ k*) *c*)
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 ↓ n = ys ↓ n ⟹
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c ↓ n =
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun ys c ↓ n
by (*simp add*: *f-Exec-Stream-Acc-Output-take*)

lemma *i-Exec-Stream-Acc-Output-causal*:
input1 ⇓ n = input2 ⇓ n ⟹
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input1 c ⇓ n =
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input2 c ⇓ n
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*:
⟦ *xs ↓ n = ys ↓ n*;
f-Streams-Connected
(*f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c*)
channel1;
f-Streams-Connected
(*f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun ys c*)
channel2 ⟧ *⟹*
channel1 ↓ Suc n = channel2 ↓ Suc n
by (*simp add*: *f-Streams-Connected-def f-Exec-Stream-Acc-Output-take*)

lemma *i-Exec-Stream-Acc-Output-Connected-strictly-causal*:
⟦ *input1 ⇓ n = input2 ⇓ n*;
i-Streams-Connected
(*i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input1 c*)

channel1;
i-Streams-Connected
(*i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input2 c*)
channel2 ⟧ ⟹
channel1 ⇓ *Suc n* = *channel2* ⇓ *Suc n*
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 accelrated execution

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

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

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

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

lemma *Deterministic-trans-fun-imp-acc-trans-fun*:
Deterministic-Trans-Fun trans-fun localState ⟹

Deterministic-Trans-Fun (*Acc-Trans-Fun-Step k trans-fun pointwise-shrink*) *localState*
apply (*simp* (*no-asm*) *only*: *Deterministic-Trans-Fun-def Acc-Trans-Fun-Step-def*)
apply *clarify*
apply (*subst Deterministic-f-Exec-Stream*, *simp*+)
done

lemma *is-Pointwise-Output-Shrink-list-imp-is-Pointwise-Output-Shrink*:
⟦ *is-Pointwise-Output-Shrink-list pointwise-shrink fs*; *output-fun* ∈ *set fs* ⟧ ⟹
is-Pointwise-Output-Shrink pointwise-shrink output-fun
apply (*induct fs*, *simp*)
apply *fastforce*
done

lemma *is-Pointwise-Output-Shrink-list-eq-is-Pointwise-Output-Shrink-all*:
(*is-Pointwise-Output-Shrink-list pointwise-shrink fs*) =
(∀ *output-fun* ∈ *set fs*. *is-Pointwise-Output-Shrink pointwise-shrink output-fun*)
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*:
⟦ *is-Pointwise-Output-Shrink-list pointwise-shrink fs*; *set fs′* ⊆ *set fs* ⟧ ⟹
is-Pointwise-Output-Shrink-list pointwise-shrink fs′
by (*fastforce simp*: *is-Pointwise-Output-Shrink-list-eq-is-Pointwise-Output-Shrink-all*)

lemma *f-Exec-Stream-Acc-LocalState-eq-Acc-Trans-Fun-Step-LocalState*: ⋀*c*.
⟦ *0* < *k*;
Deterministic-Trans-Fun trans-fun localState;
is-correct-localState-Pointwise-Output-Shrink pointwise-shrink localState ⟧ ⟹
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c =
map localState (*f-Exec-Comp-Stream* (*Acc-Trans-Fun-Step k trans-fun pointwise-shrink*) *xs c*)
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*])
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*: $\bigwedge c.$
⟦ *0 < k*;
Deterministic-Trans-Fun trans-fun localState;
is-correct-localState-Pointwise-Output-Shrink pointwise-shrink localState;
is-Pointwise-Output-Shrink pointwise-shrink output-fun ⟧ ⟹
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c =
map output-fun (*f-Exec-Comp-Stream* (*Acc-Trans-Fun-Step k trans-fun pointwise-shrink*) *xs c*)
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* $\varepsilon^{k - Suc\ 0}$ (*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=output-fun*])
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 *i-Exec-Stream-Acc-LocalState-eq-Acc-Trans-Fun-Step-LocalState*: $\bigwedge c.$
⟦ *0 < k*;

Deterministic-Trans-Fun trans-fun localState;
is-correct-localState-Pointwise-Output-Shrink pointwise-shrink localState ⟧ ⟹
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input c =
localState ∘ (*i-Exec-Comp-Stream* (*Acc-Trans-Fun-Step k trans-fun pointwise-shrink*)
input c)
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-*
done

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

lemma *f-Exec-Stream-Acc-LocalState-Init-length*:
0 < *k* ⟹
length (*f-Exec-Comp-Stream-Acc-LocalState-Init k localState trans-fun xs c*) = *Suc*
(*length xs*)
by (*simp add*: *f-Exec-Comp-Stream-Acc-LocalState-Init-def*)

lemma *f-Exec-Stream-Acc-Output-Init-Nil*:
f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun [] *c* = [*output-fun c*]
by (*simp add*: *f-Exec-Comp-Stream-Acc-Output-Init-def*)

lemma *f-Exec-Stream-Acc-LocalState-Init-Nil*:
f-Exec-Comp-Stream-Acc-LocalState-Init k 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-Exec-Comp-Stream-Acc-Output-Init (*Suc 0*) *output-fun trans-fun xs c* =
map output-fun (*f-Exec-Comp-Stream-Init trans-fun xs c*)
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-Exec-Comp-Stream-Acc-LocalState-Init (*Suc 0*) *localState trans-fun xs c* =
map localState (*f-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-Exec-Comp-Stream-Acc-Output-Init (*Suc 0*) *output-fun trans-fun input c* =
output-fun ∘ (*i-Exec-Comp-Stream-Init trans-fun input c*)
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-Exec-Comp-Stream-Acc-LocalState-Init (*Suc 0*) *localState trans-fun input c* =
localState ∘ (*i-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-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun xs c ↓ (*Suc n*) =
f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun (*xs* ↓ *n*) *c*
by (*simp add*: *f-Exec-Comp-Stream-Acc-Output-Init-def f-Exec-Stream-Acc-Output-take*)

lemma *f-Exec-Stream-Acc-Output-Init-drop′*:
⟦ *0* < *k*; *n* < *length xs* ⟧ ⟹
f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun xs c ↑ *Suc n* =
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c ↑ *n*
by (*simp add*: *f-Exec-Comp-Stream-Acc-Output-Init-def*)

lemma *i-Exec-Stream-Acc-Output-Init-take*:
0 < *k* ⟹
i-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun input c ⇓ (*Suc n*) =
f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun (*input* ⇓ *n*) *c*
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* ⟹
i-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun xs c ⇑ *Suc n* =
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c ⇑ *n*
by (*simp add*: *i-Exec-Comp-Stream-Acc-Output-Init-def*)

lemma *f-Exec-Stream-Acc-Output-Init-strictly-causal*:
xs ↓ *n* = *ys* ↓ *n* ⟹
f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun xs c ↓ *Suc n* =
f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun ys c ↓ *Suc n*
by (*simp add*: *f-Exec-Comp-Stream-Acc-Output-Init-def*, *rule f-Exec-Stream-Acc-Output-causal*)

lemma *i-Exec-Stream-Acc-Output-Init-strictly-causal*:
input1 ⇓ *n* = *input2* ⇓ *n* ⟹

i-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun input1 c ⇓ Suc n =
i-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun input2 c ⇓ Suc n
by (*simp add*: *i-Exec-Comp-Stream-Acc-Output-Init-def*, *rule i-Exec-Stream-Acc-Output-causal*)

lemma *f-Exec-Stream-Acc-Output-Init-eq-f-Exec-Stream-Acc-Output-Cons*:
f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun xs c =
output-fun c # f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c
by (*simp add*: *f-Exec-Comp-Stream-Acc-Output-def f-Exec-Comp-Stream-Acc-Output-Init-def*)

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

output-fun c = ε ⟹
f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun xs c =
ε # f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c
by (*simp add*: *f-Exec-Stream-Acc-Output-Init-eq-f-Exec-Stream-Acc-Output-Cons*)

lemma *f-Exec-Stream--Acc-OutputInit-tl-eq-f-Exec-Stream-Acc-Output*:
tl (*f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun xs c*) =
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c
by (*simp add*: *f-Exec-Stream-Acc-Output-Init-eq-f-Exec-Stream-Acc-Output-Cons*)

lemma *f-Exec-Stream-Previous-f-Exec-Stream-Acc-Output-Init*:
f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun xs c ! n =
(*f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c*)$^{\leftarrow'\ output\text{-}fun\ c\ \ n}$
by (*simp add*: *f-Exec-Stream-Acc-Output-Init-eq-f-Exec-Stream-Acc-Output-Cons list-Previous-nth-if nth-Cons'*)

lemma *f-Exec-Stream-Acc-Output-Init-eq-output-channel*:
⟦ *output-fun c = ε*;
f-Streams-Connected
(*f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c*)
channel ⟧ ⟹
f-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun xs c = channel
by (*simp add*: *f-Streams-Connected-def f-Exec-Stream-Acc-Output-Init-eq-f-Exec-Stream-Acc-Output-Cons-outp*

lemma *i-Exec-Stream-Acc-Output-Init-eq-i-Exec-Stream-Acc-Output-Cons*:
i-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun input c =
[*output-fun c*] ⌢ *i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c*
by (*simp add*: *i-Exec-Comp-Stream-Acc-Output-def i-Exec-Comp-Stream-Acc-Output-Init-def*)

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

output-fun c = ε ⟹
i-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun input c =
[*ε*] ⌢ *i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c*
by (*simp add*: *i-Exec-Stream-Acc-Output-Init-eq-i-Exec-Stream-Acc-Output-Cons*)

lemma *i-Exec-Stream-Previous-i-Exec-Stream-Acc-Output-Init*:

i-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun input c n =
(i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c)$^{\leftarrow\ output\text{-}fun\ c\ n}$
by (*simp add*: *i-Exec-Stream-Acc-Output-Init-eq-i-Exec-Stream-Acc-Output-Cons ilist-Previous-nth-if*)

lemma *i-Exec-Stream-Acc-Output-Init-eq-output-channel*:
⟦ *output-fun c = ε*;
i-Streams-Connected
(*i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c*)
channel ⟧ ⟹
i-Exec-Comp-Stream-Acc-Output-Init k output-fun trans-fun input c = channel
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*]:
⋀*c1 c2 i*. ⟦
0 < k1; *0 < k2*;
equiv-states (*localState1 c1*) (*localState2 c2*);
∀ *input0*. *set input0* ⊆ *A* ⟶ (∀ *m*∈*A*.
Equiv-Exec m equiv-states
localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2
(*f-Exec-Comp trans-fun1* (*map input-fun1 input0* ⊙$_f$ *k1*) *c1*)
(*f-Exec-Comp trans-fun2* (*map input-fun2 input0* ⊙$_f$ *k2*) *c2*));
— *equiv-states* relation implies equivalent executions
— not only at the beginning but also after processing an input
set input ⊆ *A*; *i < length input* ⟧ ⟹
equiv-states
(*localState1* ((*f-Exec-Comp-Stream trans-fun1* (*map input-fun1 input* ⊙$_f$ *k1*) *c1*) ÷$_{fl}$ *k1* ! *i*))
(*localState2* ((*f-Exec-Comp-Stream trans-fun2* (*map input-fun2 input* ⊙$_f$ *k2*) *c2*) ÷$_{fl}$ *k2* ! *i*))
apply (*induct input*, *simp*)
apply (*clarsimp simp*: *append-Cons*[*symmetric*] *f-Exec-Stream-append-if f-shrink-last-Cons nth-Cons simp del*: *last.simps f-Exec-Stream-Cons append-Cons*)
apply (*case-tac i*)
apply (*drule-tac x*=[] **in** *spec*)
apply (*drule mp*, *simp*)
apply (*drule-tac x*=*a* **in** *bspec*, *assumption*)
apply (*simp del*: *last.simps f-Exec-Stream-Cons*)
apply (*subst f-Exec-eq-f-Exec-Stream-last2*[*symmetric*], *simp*)+
apply (*rule Equiv-Exec-equiv-statesI*[*of equiv-states localState1 - localState2 - - input-fun1*], *assumption*+)

apply (*rename-tac i'*)
apply (*subst f-Exec-eq-f-Exec-Stream-last2*[*symmetric*], *simp*)+
apply (*drule-tac x=f-Exec-Comp trans-fun1* (*input-fun1 a* # $\varepsilon^{k1 - Suc\ 0}$) *c1* **in** *meta-spec*)
apply (*drule-tac x=f-Exec-Comp trans-fun2* (*input-fun2 a* # $\varepsilon^{k2 - Suc\ 0}$) *c2* **in** *meta-spec*)
apply (*drule-tac x=i'* **in** *meta-spec*)
apply (*drule meta-mp*, *simp*)+
apply (*drule-tac x=*[] **in** *spec*, *simp*)
apply (*drule-tac x=a* **in** *bspec*, *assumption*)
apply (*rule Equiv-Exec-equiv-statesI'*[*of equiv-states localState1 - localState2 - - input-fun1*], *simp*+)
apply *clarsimp*
apply (*drule meta-mp*)
apply *clarify*
apply (*drule-tac x=a* # *input0* **in** *spec*)
apply (*simp add*: *f-Exec-append*)
apply *simp*
done

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

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

(*localState1* (*f-Exec-Comp trans-fun1* (*map input-fun1 input* $\odot_f$ *k1*) *c1*))
(*localState2* (*f-Exec-Comp trans-fun2* (*map input-fun2 input* $\odot_f$ *k2*) *c2*))
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*)
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*)
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*:
⟦ *0* < *k1*; *0* < *k2*; *equiv-states* (*localState1 c1*) (*localState2 c2*);
⋀*input0 m*.
Equiv-Exec
m equiv-states localState1 localState2
input-fun1 input-fun2 output-fun1 output-fun2 trans-fun1 trans-fun2 k1 k2
(*f-Exec-Comp trans-fun1* (*map input-fun1 input0* $\odot_f$ *k1*) *c1*)
(*f-Exec-Comp trans-fun2* (*map input-fun2 input0* $\odot_f$ *k2*) *c2*) ⟧ ⟹
equiv-states
(*localState1* (*f-Exec-Comp trans-fun1* (*map input-fun1 input* $\odot_f$ *k1*) *c1*))
(*localState2* (*f-Exec-Comp trans-fun2* (*map input-fun2 input* $\odot_f$ *k2*) *c2*))
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*]:
⟦ *0* < *k1*; *0* < *k2*; *equiv-states* (*localState1 c1*) (*localState2 c2*);
⋀*input0 m*. ⟦*set input0* ⊆ *A*; *m* ∈ *A*⟧ ⟹
Equiv-Exec
m equiv-states localState1 localState2
input-fun1 input-fun2 output-fun1 output-fun2 trans-fun1 trans-fun2 k1 k2
(*f-Exec-Comp trans-fun1* (*map input-fun1 input0* $\odot_f$ *k1*) *c1*)
(*f-Exec-Comp trans-fun2* (*map input-fun2 input0* $\odot_f$ *k2*) *c2*);
range input ⊆ *A* ⟧ ⟹
equiv-states
(*localState1* ((*i-Exec-Comp-Stream trans-fun1* ((*input-fun1* ∘ *input*) $\odot_i$ *k1*) *c1* $\div_{il}$ *k1*) *i*))
(*localState2* ((*i-Exec-Comp-Stream trans-fun2* ((*input-fun2* ∘ *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*:
⟦ *0* < *k1*; *0* < *k2*; *equiv-states* (*localState1 c1*) (*localState2 c2*);
⋀*input0 m*.
Equiv-Exec
m equiv-states localState1 localState2
input-fun1 input-fun2 output-fun1 output-fun2 trans-fun1 trans-fun2 k1 k2
(*f-Exec-Comp trans-fun1* (*map input-fun1 input0* $\odot_f$ *k1*) *c1*)
(*f-Exec-Comp trans-fun2* (*map input-fun2 input0* $\odot_f$ *k2*) *c2*) ⟧ ⟹
equiv-states
(*localState1* ((*i-Exec-Comp-Stream trans-fun1* ((*input-fun1* ∘ *input*) $\odot_i$ *k1*) *c1* $\div_{il}$ *k1*) *i*))
(*localState2* ((*i-Exec-Comp-Stream trans-fun2* ((*input-fun2* ∘ *input*) $\odot_i$ *k2*) *c2* $\div_{il}$ *k2*) *i*))
by (*rule i-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*+)

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* $\odot_f$ *k1*) *c1*)
(*f-Exec-Comp trans-fun2* (*map input-fun2 input0* $\odot_f$ *k2*) *c2*);
set input ⊆ *A* ⟧ ⟹
(*map output-fun1* (
f-Exec-Comp-Stream trans-fun1 (*map input-fun1 input* $\odot_f$ *k1*) *c1*)) $\div_f$ *k1* =
(*map output-fun2* (
f-Exec-Comp-Stream trans-fun2 (*map input-fun2 input* $\odot_f$ *k2*) *c2*)) $\div_f$ *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*)
apply (*simp add*: *take-map*[*symmetric*] *f-Exec-Stream-expand-shrink-last-nth-eq-f-Exec-Comp*[*symmetric*])
apply (*frule Suc-lessD*)
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

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

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

input-fun1 input-fun2 output-fun1 output-fun2 trans-fun1 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*:
⟦ *0 < k1*; *0 < k2*;
equiv-states (*localState1 c1*) (*localState2 c2*);
⋀*input0 m*.
Equiv-Exec
m equiv-states localState1 localState2
input-fun1 input-fun2 output-fun1 output-fun2 trans-fun1 trans-fun2 k1 k2
(*f-Exec-Comp trans-fun1* (*map input-fun1 input0* $\odot_f$ *k1*) *c1*)
(*f-Exec-Comp trans-fun2* (*map input-fun2 input0* $\odot_f$ *k2*) *c2*) ⟧ ⟹
(*output-fun1* ∘
i-Exec-Comp-Stream trans-fun1 ((*input-fun1* ∘ *input*) $\odot_i$ *k1*) *c1*) $\div_i$ *k1* =
(*output-fun2* ∘
i-Exec-Comp-Stream trans-fun2 ((*input-fun2* ∘ *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 ⊆ *A*;
i < length input ⟧ ⟹
equiv-states
(*f-Exec-Comp-Stream-Acc-LocalState k1 localState1 trans-fun1* (*map input-fun1 input*) *c1* ! *i*)
(*f-Exec-Comp-Stream-Acc-LocalState k2 localState2 trans-fun2* (*map input-fun2 input*) *c2* ! *i*)
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*)
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

lemma *f-Equiv-Exec-Stream-Acc-Output-set-eq*:
⟦ *0 < k1*; *0 < k2*;
equiv-states (*localState1 c1*) (*localState2 c2*);
Equiv-Exec-stable-set A
equiv-states localState1 localState2
input-fun1 input-fun2 output-fun1 output-fun2 trans-fun1 trans-fun2 k1 k2 c1 c2;
set input ⊆ *A* ⟧ ⟹
f-Exec-Comp-Stream-Acc-Output k1 output-fun1 trans-fun1 (*map input-fun1 input*) *c1* =
f-Exec-Comp-Stream-Acc-Output k2 output-fun2 trans-fun2 (*map input-fun2 input*) *c2*
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 input*) *c1* =
f-Exec-Comp-Stream-Acc-Output k2 output-fun2 trans-fun2 (*map input-fun2 input*) *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

lemma *i-Equiv-Exec-Stream-Acc-LocalState-set*:
⟦ *0 < k1*; *0 < k2*;
equiv-states (*localState1 c1*) (*localState2 c2*);
Equiv-Exec-stable-set A
equiv-states localState1 localState2 input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2;
range input ⊆ *A* ⟧ ⟹
equiv-states
(*i-Exec-Comp-Stream-Acc-LocalState k1 localState1 trans-fun1* (*input-fun1* ∘ *input*) *c1 i*)
(*i-Exec-Comp-Stream-Acc-LocalState k2 localState2 trans-fun2* (*input-fun2* ∘ *input*) *c2 i*)
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*:
⟦ *0 < k1*; *0 < k2*;
equiv-states (*localState1 c1*) (*localState2 c2*);
Equiv-Exec-stable
equiv-states localState1 localState2
input-fun1 input-fun2 output-fun1 output-fun2
trans-fun1 trans-fun2 k1 k2 c1 c2 ⟧ ⟹
equiv-states
(*i-Exec-Comp-Stream-Acc-LocalState k1 localState1 trans-fun1* (*input-fun1* ∘ *input*) *c1 i*)
(*i-Exec-Comp-Stream-Acc-LocalState k2 localState2 trans-fun2* (*input-fun2* ∘ *input*) *c2 i*)
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*:
⟦ *0 < k1*; *0 < k2*;
equiv-states (*localState1 c1*) (*localState2 c2*);
Equiv-Exec-stable-set A
equiv-states localState1 localState2
input-fun1 input-fun2 output-fun1 output-fun2 trans-fun1 trans-fun2 k1 k2 c1 c2;
range input ⊆ *A* ⟧ ⟹
i-Exec-Comp-Stream-Acc-Output k1 output-fun1 trans-fun1 (*input-fun1* ∘ *input*) *c1* =

i-Exec-Comp-Stream-Acc-Output k2 output-fun2 trans-fun2 (input-fun2 ∘ input) c2
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 ∘ input) c1 =
i-Exec-Comp-Stream-Acc-Output k2 output-fun2 trans-fun2 (input-fun2 ∘ 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*]:
⋀*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=*$\varepsilon^{l\ -\ Suc\ 0}$ *@* ε^{k-l} **and** *t=*$\varepsilon^{k\ -\ Suc\ 0}$ **in** *subst*)
apply (*simp add*: *replicate-le-diff2*)
apply (*subst append-Cons*[*symmetric*])
apply (*induct xs*, *simp*)
apply (*case-tac i*)
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 # *$\varepsilon^{l\ -\ Suc\ 0}$) *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*)

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* $\odot_f$ *k* **in** *f-Exec-Equal-State*)
apply (*simp add*: *f-expand-not-empty-conv*)+
done

corollary *f-Exec-Stream-Acc-LocalState--State-Idle-eq*[*rule-format*]:
⟦ *0* < *l*; *l* ≤ *k*; *Exec-Equal-State localState trans-fun*;
∀ *n*<*length xs*. *State-Idle localState output-fun trans-fun* (
f-Exec-Comp-Stream-Acc-LocalState l localState trans-fun xs c ! *n*) ⟧ ⟹
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c =
f-Exec-Comp-Stream-Acc-LocalState l localState trans-fun xs c
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*]:
⟦ *0* < *l*; *l* ≤ *k*; *Exec-Equal-State localState trans-fun*;
∀ *n*≤*i*. *State-Idle localState output-fun trans-fun* (
i-Exec-Comp-Stream-Acc-LocalState l localState trans-fun input c n) ⟧ ⟹
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input c i =
i-Exec-Comp-Stream-Acc-LocalState l localState trans-fun input c i
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'*)
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*)
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*
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*)
apply (*frule-tac x=Suc n* **in** *spec*)
apply (*simp only*: *f-Exec-Stream-Acc-LocalState-nth*)
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*)
apply (*rule f-Exec-State-Idle-replicate-NoMsg-gr0-output*, *assumption*)
apply *simp*
apply (*rule arg-cong*[**where** *f*=λ*x*. *last-message* (*map output-fun x*)])
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* ≤ *k*; *Exec-Equal-State localState trans-fun*;
∀ *n*. *State-Idle localState output-fun trans-fun* (

```
    i-Exec-Comp-Stream-Acc-LocalState l localState trans-fun input c n) ⟧ ⟹
  i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c =
  i-Exec-Comp-Stream-Acc-Output l output-fun trans-fun input c
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]:
  ⋀c xs. ⟦ 0 < l; l ≤ k;
    State-Idle localState output-fun trans-fun (localState c);
    ∀ c m. State-Idle localState output-fun trans-fun (localState c) ⟶
      State-Idle localState output-fun trans-fun (
        localState (f-Exec-Comp trans-fun (m # ε^l - Suc 0) c));
    i < length xs ⟧ ⟹
  State-Idle localState output-fun trans-fun (
    f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c ! i)
apply (frule length-greater-imp-not-empty)
apply (subgoal-tac
  State-Idle localState output-fun trans-fun (
    localState (f-Exec-Comp trans-fun (hd xs # NoMsg^k - Suc 0) 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+)
 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 - Suc 0) c in meta-spec)
apply (drule-tac x=tl xs in meta-spec)
apply (subgoal-tac i < length (tl xs) ∧ tl xs ≠ [], 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 - 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)
apply (simp add: f-Exec-Stream-Acc-LocalState-nth)
done
```

lemma *i-Exec-Stream-Acc-LocalState--State-Idle-all*[*rule-format*]:
⟦ *0* < *l*; *l* ≤ *k*;
State-Idle localState output-fun trans-fun (*localState c*);
∀ *c m*. *State-Idle localState output-fun trans-fun* (*localState c*) ⟶
State-Idle localState output-fun trans-fun (
localState (*f-Exec-Comp trans-fun* (*m* # $\varepsilon^{l - Suc\ 0}$) *c*)) ⟧ ⟹
State-Idle localState output-fun trans-fun (
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c i)
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* # $\varepsilon^{l - Suc\ 0}$) *c*)) ⟧ ⟹
f-Exec-Comp-Stream-Acc-Output k output-fun trans-fun xs c =
f-Exec-Comp-Stream-Acc-Output l output-fun trans-fun xs c
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* # $\varepsilon^{l - Suc\ 0}$) *c*)) ⟧ ⟹
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c =
i-Exec-Comp-Stream-Acc-Output l output-fun trans-fun input c
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*);
∀ *c m*. *State-Idle localState output-fun trans-fun* (*localState c*) ⟶
State-Idle localState output-fun trans-fun (
localState (*f-Exec-Comp trans-fun* (*m* # $\varepsilon^{l - Suc\ 0}$) *c*)) ⟧ ⟹
f-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c =
f-Exec-Comp-Stream-Acc-LocalState l localState trans-fun xs c
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* ≤ *k*; *Exec-Equal-State localState trans-fun*;
State-Idle localState output-fun trans-fun (*localState c*);
∀ *c m*. *State-Idle localState output-fun trans-fun* (*localState c*) ⟶
State-Idle localState output-fun trans-fun (
localState (*f-Exec-Comp trans-fun* (*m* # $\varepsilon^{l - Suc\ 0}$) *c*)) ⟧ ⟹
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun xs c =
i-Exec-Comp-Stream-Acc-LocalState l localState trans-fun xs c
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*: ⋀*c*.
f-Exec-Comp trans-fun (*map f xs*) *c* = *f-Exec-Comp* (*trans-fun* ∘ *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* ∘ *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* ∘ *input*) *c* =
i-Exec-Comp-Stream (*trans-fun* ∘ *f*) *input c*
by (*simp add*: *ilist-eq-iff i-Exec-Stream-nth f-Exec-input-map*)

end

4 AutoFocus message streams and temporal logic on intervals

theory *IL-AF-Stream*
imports *Main Nat−Interval−Logic.IL-TemporalOperators AF-Stream*
begin

4.1 Stream views – joining streams and intervals

4.1.1 Basic definitions

primrec *f-join-aux* :: *'a list* ⇒ *nat* ⇒ *iT* ⇒ *'a list*
where
f-join-aux [] *n I* = []
| *f-join-aux* (*x* # *xs*) *n I* =
(*if n* ∈ *I then* [*x*] *else* []) @ *f-join-aux xs* (*Suc n*) *I*

The functions *f-join* and *i-join* deliver views of finite and infinite streams

through intervals (more exactly: arbitrary natural sets). A stream view contains only the elements of the original stream at positions, which are contained in the interval. For instance, *f-join* [*0*,*10*,*20*,*30*,*40*] {*1*,*4*} = [*10*,*40*]

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

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

notation
f-join (**infixl** ⋈ *100*) **and**
i-join (**infixl** ⋈ *100*)

The function *i-f-join* can be used for the case, when an infinite stream is joined with a finite interval. The function *i-join* would then deliver an infinite stream, whose elements after position *card I* are equal to initial stream's element at position *Max I*. The function *i-f-join* in contrast cuts the resulting stream at this position and returns a finite stream.

definition *i-f-join* :: *'a ilist* ⇒ *iT* ⇒ *'a list* (**infixl** ⋈$_{i-f}$ *100*)
where *f* ⋈$_{i-f}$ *I* ≡ *f* ⇓ *Suc* (*Max I*) ⋈$_f$ *I*
notation
i-f-join (**infixl** ⋈ *100*)

The function *i-f-join* should be used only for finite sets in order to deliver well-defined results. The function *i-join* should be used for infinite sets, because joining an infinite stream *s* and a finite set *I* using *i-join* would deliver an infinite stream, ending with an infinite sequence of elements equal to *s* (*Max I*).

4.1.2 Basic results

lemma *f-join-aux-length*:
⋀*n*. *length* (*f-join-aux xs n I*) = *card* (*I* ∩ {*n*..<*n* + *length xs*})
apply (*induct xs*, *simp*)
apply (*simp add*: *atLeastLessThan-def*)
apply (*rule-tac t*={*n*..} **and** *s*=*insert n* {*Suc n*..} **in** *subst*, *fastforce*)
apply *simp*
done

lemma *f-join-aux-nth*[*rule-format*]:
∀ *n i*. *i* < *card* (*I* ∩ {*n*..<*n* + *length xs*}) ⟶
(*f-join-aux xs n I*) ! *i* = *xs* ! (((*I* ∩ {*n*..<*n* + *length xs*}) → *i*) − *n*)
apply (*induct xs*, *simp*)
apply (*clarsimp split del*: *if-split*)
apply (*subgoal-tac* {*n*..<*Suc* (*n* + *length xs*)} = *insert n* {*Suc n*..<*Suc* (*n* + *length xs*)})

prefer *2*
apply *fastforce*
apply (*frule card-gr0-imp-not-empty*[*OF gr-implies-gr0*])
apply (*case-tac* $n \in I$)
prefer *2*
apply (*simp add*: *nth-Cons'*)
apply (*subgoal-tac Suc* $n \leq (I \cap \{Suc\ n..<Suc\ (n + length\ xs)\}) \rightarrow i$, *simp*)
apply (*rule order-trans*[*OF - iMin-le*[*OF inext-nth-closed*]])
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* $I \cap \{Suc\ n..<Suc\ (n + length\ xs)\} = \{\}$, *simp*)
apply (*case-tac i*)
apply (*simp add*: *iMin-insert*)
apply (*subgoal-tac Suc* $n \leq iMin\ \{Suc\ n..<Suc\ (n + length\ xs)\}$)
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*)
apply (*subst inext-nth-insert-Suc*)
apply *simp*
apply (*rule Suc-le-lessD*)
apply (*rule order-trans*[*OF - iMin-Int-ge2*])
apply *assumption*+
apply (*simp add*: *nth-Cons'*)
apply (*subgoal-tac Suc* $n \leq (I \cap \{Suc\ n..<Suc\ (n + length\ xs)\}) \rightarrow i1$, *simp*)
apply (*rule order-trans*[*OF - iMin-le*[*OF inext-nth-closed*]])
apply (*rule order-trans*[*OF - iMin-Int-ge2*])
apply *assumption*+
done

Joining finite streams and intervals

lemma *f-join-length*: $length\ (xs \bowtie_f I) = card\ (I \downarrow< length\ xs)$
by (*simp add*: *f-join-def f-join-aux-length atLeast0LessThan cut-less-Int-conv*)

lemma *f-join-nth*: $n < length\ (xs \bowtie_f I) \Longrightarrow (xs \bowtie_f I)\ !\ n = xs\ !\ (I \rightarrow n)$
apply (*simp add*: *f-join-length*)
apply (*unfold f-join-def*)
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*: $n < card\ (I \downarrow< length\ xs) \Longrightarrow (xs \bowtie_f I)\ !\ n = xs\ !\ (I \rightarrow n)$
by (*simp add*: *f-join-nth f-join-length*)

lemma *f-join-empty*: $xs \bowtie_f \{\} = []$
by (*simp add*: *length-0-conv*[*symmetric*] *f-join-length cut-less-empty del*: *length-0-conv*)

lemma *f-join-Nil*: $[] \bowtie_f I = []$
by (*simp add*: *length-0-conv*[*symmetric*] *f-join-length cut-less-0-empty del*: *length-0-conv*)

lemma *f-join-Nil-conv*: $(xs \bowtie_f I = []) = (I \downarrow< length\ xs = \{\})$
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'*: $(xs \bowtie_f I = []) = (\forall i{<}length\ xs.\ i \notin I)$
by (*fastforce simp*: *f-join-Nil-conv*)

lemma *f-join-all-conv*: $(xs \bowtie_f I = xs) = (\{..{<}length\ xs\} \subseteq I)$
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* $\downarrow<$ *length xs* $\subset \{..{<}length\ xs\}$)
prefer *2*
apply *blast*
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*=(!) *xs*])
apply (*subgoal-tac I* $\downarrow<$ *length xs* $= \{..{<}length\ xs\}$)
prefer *2*
apply *fastforce*
apply (*subst inext-nth-cut-less-eq*[**where** *t*=*length xs*, *symmetric*], *simp*)
apply (*simp add*: *inext-nth-lessThan*)
done

lemma *f-join-all*: $\{..{<}length\ xs\} \subseteq I \Longrightarrow xs \bowtie_f I = xs$
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*:
$[\![\ finite\ A;\ Max\ A < iMin\ B\ ]\!] \Longrightarrow xs \bowtie_f (A \cup B) = xs \bowtie_f A\ @\ (xs \bowtie_f B)$

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*)
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*:
$xs \bowtie_f \{n\} = (\textit{if } n < \text{length } xs \textit{ then } [xs\ !\ n] \textit{ else } [])$
apply (*clarsimp 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*:
$n < \text{length } xs \Longrightarrow$
$xs \bowtie_f \textit{insert } n\ I = xs \bowtie_f (I \downarrow< n) \mathbin{@} (xs\ !\ n) \mathbin{\#} (xs \bowtie_f (I \downarrow> n))$
apply (*rule-tac t*=*insert n I* **and** s=$(I \downarrow< n) \cup \{n\} \cup (I \downarrow> n)$ **in** *subst*, *fastforce*)
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 Max* $(I \downarrow< n) <$ *iMin* $\{n\}$)
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 Max* $\{n\} <$ *iMin* $(I \downarrow> n)$)
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 Max* $(I \downarrow< n) < iMin\ \{n\}$)
prefer *2*
apply (*simp add*: *cut-less-mem-iff*)
apply (*subgoal-tac Max* $(I \downarrow< n \cup \{n\}) < iMin\ (I \downarrow> n)$)
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*:
$(xs\ @\ [x]) \bowtie_f I =$
$xs \bowtie_f I\ @$ (*if length xs* $\in I$ *then* $[x]$ *else* $[]$)
apply (*simp add*: *list-eq-iff f-join-length*)
apply (*subgoal-tac*
card $(I \downarrow< Suc\ (length\ xs)) =$
card $(I \downarrow< length\ xs) +$ (*if length xs* $\in I$ *then Suc 0 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* $\in I$)
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* $I \rightarrow i =$ *length xs*)
prefer *2*
apply (*rule-tac t*=*length xs* **and** *s*=*Max* $(I \downarrow< Suc\ (length\ xs))$ **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*:
$(xs\ @\ ys) \bowtie_f I = xs \bowtie_f I\ @\ ys \bowtie_f (I \oplus- (length\ xs))$

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*:
$n < card\ (I \downarrow< length\ xs) \Longrightarrow$
$(xs \bowtie_f I) \downarrow n = xs \bowtie_f (I \downarrow< (I \rightarrow n))$
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* $n < card\ I \vee infinite\ I$)
prefer *2*
apply (*case-tac finite I*)
apply (*drule order-less-le-trans*[*OF - cut-less-card*], *simp*+)
apply (*simp add*: *min-eqL cut-less-inext-nth-card-eq1*)
apply *clarify*
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*)
apply (*simp add*: *cut-less-inext-nth-card-eq1*)
apply (*simp add*: *cut-less-inext-nth-card-eq1 inext-nth-cut-less-eq*)
done

lemma *take-f-join-eq2*:
$card\ (I \downarrow< length\ xs) \leq n \Longrightarrow (xs \bowtie_f I) \downarrow n = xs \bowtie_f I$
by (*simp add*: *f-join-length*)
lemma *take-f-join-if*:
$(xs \bowtie_f I) \downarrow n =$
$(if\ n < card\ (I \downarrow< length\ xs)\ then\ xs \bowtie_f (I \downarrow< (I \rightarrow n))\ else\ xs \bowtie_f I)$
by (*simp add*: *take-f-join-eq1 take-f-join-eq2*)

lemma *drop-f-join-eq1*:
$n < card\ (I \downarrow< length\ xs) \Longrightarrow$
$(xs \bowtie_f I) \uparrow n = xs \bowtie_f (I \downarrow\geq (I \rightarrow n))$
apply (*case-tac* $I = \{\}$)
apply (*simp add*: *cut-less-empty*)
apply (*case-tac* $I \downarrow< length\ xs = \{\}$)
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*= $I \to n$ **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* $I \downarrow\geq (I \to n) \neq \{\}$)
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*:
$card\ (I \downarrow< length\ xs) \leq n \Longrightarrow (xs \bowtie_f I) \uparrow n = []$
by (*simp add*: *f-join-length*)

lemma *drop-f-join-if*:
$(xs \bowtie_f I) \uparrow n =$
$(if\ n < card\ (I \downarrow< length\ xs)\ then\ xs \bowtie_f (I \downarrow\geq (I \to n))\ else\ [])$
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.commute*)
apply (*simp add*: *f-join-nth f-join-length cut-cut-less min.commute*)
apply (*case-tac* $n < length\ xs$)
apply (*simp add*: *min-eqL inext-nth-cut-less-eq*)
apply (*simp add*: *less-card-cut-less-imp-inext-nth-less*)
apply (*simp add*: *min-eqR linorder-not-less*)
apply (*subst inext-nth-cut-less-eq*)
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* $\lambda x.\ xs \uparrow n \bowtie_f I = x \bowtie_f (I \oplus n)\ n$])
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 ↓< *length xs* = *B* ↓< *length xs* ⟹ *xs* ⋈$_f$ *A* = *xs* ⋈$_f$ *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 ≤ *t* ⟹ *xs* ⋈$_f$ (*I* ↓< *t*) = *xs* ⋈$_f$ *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*:
⟦ *finite I*; *xs* ↓ *Suc* (*Max I*) = *ys* ↓ *Suc* (*Max I*) ⟧ ⟹
xs ⋈$_f$ *I* = *ys* ⋈$_f$ *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*)
apply (*simp add*: *f-join-nth2*)
apply (*drule-tac x*=*I* → *i* **in** *spec*)
apply (*subgoal-tac I* → *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* ↓< *length xs* = *I* ↓< *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* → *i* **in** *spec*)
apply (*subgoal-tac I* → *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 $\Longrightarrow$ *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*)

lemma *i-join-union*:
$[\![$ *finite A*; *Max A* < *iMin B*; *B* $\neq$ {} $]\!] \Longrightarrow$
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* $\downarrow<$ *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*)
apply (*simp add*: *f-join-nth f-join-length*)
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*} = (λ*n*. *f a*)
by (*simp add*: *ilist-eq-iff i-join-nth inext-nth-singleton*)

lemma *i-join-insert*:
f $\bowtie_i$ (*insert n I*) =
(*f* $\Downarrow$ *n*) $\bowtie_f$ (*I* $\downarrow<$ *n*) $\frown$ [*f n*] $\frown$ (
if I $\downarrow>$ *n* = {} *then* (λ*x*. *f n*) *else f* $\bowtie_i$ (*I* $\downarrow>$ *n*))
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*
apply (*subst insert-is-Un*)
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=insert n* (*I* ↓< *n*) **and** *s*=(*I* ↓< *n*) ∪ {*n*} **in** *subst*, *simp*)
apply (*subst i-join-union*[*OF nat-cut-less-finite - singleton-not-empty*], *simp*)
apply (*simp add*: *i-join-singleton*)
apply (*rule-tac s*=λ*x*. *f n* **and** *t*=[*f n*] ⌢ (λ*x*. *f n*) **in** *subst*)
apply (*simp add*: *ilist-eq-iff i-append-nth*)
apply (*subst i-append-assoc*[*symmetric*])
apply (*rule-tac t*=[*f n*] ⌢ (λ*x*. *f n*) **and** *s*=(λ*x*. *f n*) **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 (*fastforce 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* = {})
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* $\Longrightarrow$ *f* $\bowtie_i$ *I* $\Downarrow$ *n* = *f* $\Downarrow$ (*I* $\rightarrow$ *n*) $\bowtie_f$ *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* $\rightarrow$ *i*) **and** *s*=*f* $\Downarrow$ (*I* $\rightarrow$ *n*) ! (*I* $\rightarrow$ *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* $\neq$ {} $\Longrightarrow$ *f* $\bowtie_i$ *I* $\Uparrow$ *n* = *f* $\bowtie_i$ (*I* $\downarrow\geq$ (*I* $\rightarrow$ *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* $\Downarrow$ *n* $\bowtie_f$ *I* = *f* $\bowtie_i$ *I* $\Downarrow$ *card* (*I* $\downarrow<$ *n*)
apply (*clarsimp 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* $\neq$ {} $\Longrightarrow$ *f* $\Uparrow$ *n* $\bowtie_i$ *I* = *f* $\bowtie_i$ (*I* $\oplus$ *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* $\neq$ {}; *card I* $\leq$ *Suc n* ⟧ $\Longrightarrow$ (*f* $\bowtie_i$ *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* $\neq$ {} ⟧ $\Longrightarrow$ (*f* $\bowtie_i$ *I*) $\Uparrow$ (*card I*) = (λ*n*. *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* $\neq$ {} ⟧ $\Longrightarrow$ (*f* $\bowtie_i$ *I*) = *f* $\Downarrow$ *Suc* (*Max I*) $\bowtie_f$ *I* $\frown$ (λ*n*. *f* (*Max I*))
apply (*simp* (*no-asm*) *add*: *ilist-eq-iff*, *clarify*)
apply (*subgoal-tac I* $\downarrow<$ (*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*)
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* $\Longrightarrow$ *length* (*f* $\bowtie_{i-f}$ *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* < *card I* $\Longrightarrow$ *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*)
apply (*subst i-take-nth*[*of I* $\rightarrow$ *n Suc* (*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*:
finite I $\Longrightarrow$ *f* $\bowtie_{i-f}$ *I* = *f* $\bowtie_i$ *I* $\Downarrow$ (*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*:
⟦ *finite A*; *finite B*; *Max A* < *iMin B* ⟧ $\Longrightarrow$
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*)

lemma *i-f-join-insert*:
finite I $\Longrightarrow$
f $\bowtie_{i-f}$ *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* $I \downarrow> n = \{\}$)
 apply (*simp add*: *f-join-empty*)
 apply (*case-tac* $I \downarrow< n = \{\}$)
  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** $f=\lambda x.\ f \Downarrow x$])
 apply *simp*
 apply (*rule min-eqR*, *rule max.coboundedI1*, *rule less-imp-le*)
 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* $I \downarrow< n = \{\}$)
 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** $f=\lambda x.\ f \Downarrow x$])
apply *simp*
apply (*rule min-eqR*)
apply (*blast intro*: *Max-subset*)
done

lemma *take-i-f-join-eq1*:
  $n < card\ I \Longrightarrow f \bowtie_{i-f} I \downarrow n = f \bowtie_{i-f} (I \downarrow< (I \to n))$
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* $n < card\ (I \downarrow< Suc\ (Max\ I))$)
 prefer *2*
 apply (*simp add*: *cut-less-Max-all*)
apply (*simp add*: *i-f-join-def take-f-join-eq1*)
apply (*case-tac* $I \downarrow< (I \to n) = \{\}$)
 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** $f=\lambda x.\ f \Downarrow x$])
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*:
⟦ *finite I*; *card I* ≤ *n* ⟧ ⟹ *f* ⋈$_{i-f}$ *I* ↓ *n* = *f* ⋈$_{i-f}$ *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 ⟹
f ⋈$_{i-f}$ *I* ↓ *n* = (*if n* < *card I then f* ⋈$_{i-f}$ (*I* ↓< (*I* → *n*)) *else f* ⋈$_{i-f}$ *I*)
by (*simp add*: *take-i-f-join-eq1 take-i-f-join-eq2*)

lemma *drop-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-ge-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 drop-f-join-eq1*)
apply (*subgoal-tac I* ↓≥ (*I* → *n*) ≠ {})
prefer *2*
apply (*rule in-imp-not-empty*[*of I* → *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*=λ*x*. *f* ⇓ *x*])
apply (*simp add*: *min-eqR cut-ge-Max-eq*)
done

lemma *drop-i-f-join-eq2*:
⟦ *finite I*; *card I* ≤ *n* ⟧ ⟹ *f* ⋈$_{i-f}$ *I* ↑ *n* = []
by (*simp add*: *i-f-join-length*)

lemma *drop-i-f-join-if*:
finite I ⟹
f ⋈$_{i-f}$ *I* ↑ *n* = (*if n* < *card I then f* ⋈$_{i-f}$ (*I* ↓≥ (*I* → *n*)) *else* [])
by (*simp add*: *drop-i-f-join-eq1 drop-i-f-join-eq2*)

lemma *i-f-join-i-drop*:
finite I ⟹ *f* ⇑ *n* ⋈$_{i-f}$ *I* = *f* ⋈$_{i-f}$ (*I* ⊕ *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 ⇓ *Suc* (*Max I*) = *g* ⇓ *Suc* (*Max I*) ⟹ *f* ⋈$_{i-f}$ *I* = *g* ⋈$_{i-f}$ *I*
by (*simp add*: *i-f-join-def*)

lemma *i-take-i-join-eq-i-f-join*:
infinite I ⟹ *f* ⋈$_i$ *I* ⇓ *n* = *f* ⋈$_{i-f}$ (*I* ↓< (*I* → *n*))
apply (*frule infinite-imp-nonempty*)
apply (*case-tac n* = *0*)
apply (*simp add*: *cut-less-Min-empty i-f-join-empty*)
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* ↓< (*I* → *n*))) ≤ *I* → *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*)
apply (*simp add*: *i-join-i-take*)
apply (*rule arg-cong*[**where** *f*=λ*x*. *f* ⋈$_i$ *I* ⇓ *card x*])
apply (*clarsimp simp*: *gr0-conv-Suc*)
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*=λ*x*. *I* ↓≤ *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* ⋈$_f$ [*n*...] = *xs* ↑ *n*
apply (*clarsimp simp*: *list-eq-iff f-join-length iFROM-cut-less iIN-card Suc-diff-Suc*)
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* ⋈$_i$ [*n*...] = *f* ⇑ *n*
by (*simp add*: *ilist-eq-iff i-join-nth iFROM-inext-nth*)

lemma *f-join-iIN*: *xs* ⋈$_f$ [*n*...,*d*] = *xs* ↑ *n* ↓ *Suc d*
apply (*simp add*: *list-eq-iff f-join-length iIN-cut-less iIN-card Suc-diff-Suc min-eq*)
apply (*simp add*: *f-join-nth2 iIN-cut-less iIN-card iIN-inext-nth*)
done

lemma *i-f-join-iIN*: *f* ⋈$_{i-f}$ [*n*...,*d*] = *f* ⇑ *n* ⇓ *Suc d*
by (*simp add*: *i-f-join-def f-join-iIN iIN-Max i-take-drop*)

lemma *f-join-iTILL*: *xs* ⋈$_f$ [...*n*] = *xs* ↓ (*Suc n*)

by (*simp add*: *iIN-0-iTILL-conv*[*symmetric*] *f-join-iIN*)

lemma *i-f-join-iTILL*: $f \bowtie_{i-f} [...n] = f \Downarrow Suc\ n$
by (*simp add*: *iIN-0-iTILL-conv*[*symmetric*] *i-f-join-iIN*)

lemma *f-join-f-expand-iT-Mult*:
$0 < k \Longrightarrow xs \odot_f k \bowtie_f (I \otimes k) = xs \bowtie_f I$
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*:
$[\![\ 0 < k;\ I \neq \{\}\ ]\!] \Longrightarrow f \odot_i k \bowtie_i (I \otimes k) = f \bowtie_i I$
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*:
$[\![\ 0 < k;\ finite\ I\ ]\!] \Longrightarrow f \odot_i k \bowtie_{i-f} (I \otimes k) = f \bowtie_{i-f} I$
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*:
$[\![\ 0 < k;\ \forall x{\in}I.\ x\ mod\ k = 0\ ]\!] \Longrightarrow$
$(xs \longmapsto_f k) \bowtie_f (I \oplus (k - 1)) = xs \div_f k \bowtie_f (I \oslash k)$
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*)
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*)
apply (*subst iT-Div-card-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*)
apply (*simp add*: *iT-Div-mod-inext-nth*)
apply (*subst f-shrink-nth-eq-f-last-message-hold-nth*)
apply (*drule sym*, *simp*, *thin-tac card x = card y* **for** *x y*)
apply (*simp add*: *iT-Plus-cut-less iT-Plus-card*)
apply (*rule less-mult-imp-div-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*: *ilist-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*:
⟦ *0 < k*; ∀ *x*∈*I*. *x mod k = 0*;
A = I ⊕ (*k − 1*); *B = I* ⊘ *k* ⟧ ⟹
(*xs* ⟼$_f$ *k*) ⋈$_f$ *A* = *xs* ÷$_f$ *k* ⋈$_f$ *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*:
⟦ $0 < k$; $I \neq \{\}$; $\forall x{\in}I.\ x\ mod\ k = 0$;
$A = I \oplus (k - 1)$; $B = I \oslash k$ ⟧ ⟹
$(f \longmapsto_i k) \bowtie_i A = f \div_i k \bowtie_i B$
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*:
⟦ $0 < k$; *finite* I; $\forall x{\in}I.\ x\ mod\ k = 0$;
$A = I \oplus (k - 1)$; $B = I \oslash k$ ⟧ ⟹
$(f \longmapsto_i k) \bowtie_{i-f} A{=} f \div_i k \bowtie_{i-f} B$
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*:
⟦ $0 < k$; $\forall x{\in}I.\ x\ mod\ k = k - 1$ ⟧ ⟹
$(xs \longmapsto_f k) \bowtie_f I = xs \div_f k \bowtie_f (I \oslash k)$
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* $\bigwedge x.\ x + k - Suc\ 0 \in I \Longrightarrow x\ mod\ k = 0$)
prefer *2*
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*)
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*:
⟦ $0 < k$; $I \neq \{\}$; $\forall x{\in}I.\ x\ mod\ k = k - 1$ ⟧ ⟹
$(f \longmapsto_i k) \bowtie_i I{=} f \div_i k \bowtie_i (I \oslash k)$
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*)
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* $k - Suc\ 0 \leq I \rightarrow x$)

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*:
⟦ *0* < *k*; *finite I*; ∀ *x*∈*I*. *x mod k* = *k* − *1* ⟧ ⟹
(*f* ⟼$_i$ *k*) ⋈$_{i-f}$ *I* = *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*)
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* − *Suc 0* ≤ *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* ⟹ *xs* ⊙$_f$ *k* ⋈$_f$ [*n* ∗ *k*, *mod k*] = *xs* ⋈$_f$ [*n*...]
by (*subst iFROM-mult*[*symmetric*], *rule f-join-f-expand-iT-Mult*)
corollary *f-join-f-expand-iMOD-0*:
0 < *k* ⟹ *xs* ⊙$_f$ *k* ⋈$_f$ [*0*, *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* ⟹ *xs* ⊙$_f$ *k* ⋈$_f$ [*n* ∗ *k*, *mod k*, *d*] = *xs* ⋈$_f$ [*n*...,*d*]
by (*subst iIN-mult*[*symmetric*], *rule f-join-f-expand-iT-Mult*)

corollary *f-join-f-expand-iMODb-0*:
0 < *k* ⟹ *xs* ⊙$_f$ *k* ⋈$_f$ [*0*, *mod k*, *n*] = *xs* ⋈$_f$ [...*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* ⟹ *f* ⊙$_i$ *k* ⋈$_i$ [*n* ∗ *k*, *mod k*] = *f* ⋈$_i$ [*n*...]
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* ⟹ *f* ⊙$_i$ *k* ⋈$_i$ [*0*, *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* ⟹ *f* ⊙$_i$ *k* ⋈$_i$ [*n* ∗ *k*, *mod k*, *d*] = *f* ⋈$_i$ [*n*. . .,*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* ⟹ *f* ⊙$_i$ *k* ⋈$_i$ [*0*, *mod k*, *n*] = *f* ⋈$_i$ [. . .*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* ⟹ *f* ⊙$_i$ *k* ⋈$_{i-f}$ [*n* ∗ *k*, *mod k*, *d*] = *f* ⋈$_{i-f}$ [*n*. . .,*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* ⟹ *f* ⊙$_i$ *k* ⋈$_{i-f}$ [*0*, *mod k*, *n*] = *f* ⋈$_{i-f}$ [. . .*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* ⟹ (*xs* ⟼$_f$ *k*) ⋈$_f$ [*n* ∗ *k* + (*k* − *1*), *mod k*] = *xs* ÷$_f$ *k* ⋈$_f$ [*n*. . .]
apply (*rule f-join-f-shrink-iT-Plus-iT-Div-mod-subst*[**where** *I*=[*n* ∗ *k*, *mod k*]])
apply (*simp add*: *iMOD-iff iMOD-add iMOD-div-ge*)+
done

corollary *f-join-f-shrink-iMOD-0*:
0 < *k* ⟹ (*xs* ⟼$_f$ *k*) ⋈$_f$ [*k* − *1*, *mod k*] = *xs* ÷$_f$ *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* ⟹ (*xs* ⟼$_f$ *k*) ⋈$_f$ [*n* ∗ *k* + (*k* − *1*), *mod k*, *d*] = *xs* ÷$_f$ *k* ⋈$_f$ [*n*. . .,*d*]
apply (*rule f-join-f-shrink-iT-Plus-iT-Div-mod-subst*[**where** *I*=[*n* ∗ *k*, *mod k*, *d*]])
apply (*simp add*: *iMODb-iff iMODb-add iMODb-div-ge*)+
done

corollary *f-join-f-shrink-iMODb-0*:
0 < *k* ⟹ (*xs* ⟼$_f$ *k*) ⋈$_f$ [*k* − *1*, *mod k*, *n*] = *xs* ÷$_f$ *k* ⋈$_f$ [. . .*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*:

$0 < k \Longrightarrow (f \longmapsto_i k) \bowtie_i [n * k + (k - 1),\ mod\ k] = f \div_i k \bowtie_i [n\ldots]$
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*:
$0 < k \Longrightarrow (f \longmapsto_i k) \bowtie_i [k - 1,\ mod\ k] = f \div_i k$
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*:
$0 < k \Longrightarrow (f \longmapsto_i k) \bowtie_{i-f} [n * k + (k - 1),\ mod\ k,\ d] = f \div_i k \bowtie_{i-f} [n\ldots,d]$
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*:
$0 < k \Longrightarrow (f \longmapsto_i k) \bowtie_{i-f} [k - 1,\ mod\ k,\ n] = f \div_i k \bowtie_{i-f} [\ldots n]$
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

lemma *i-shrink-eq-NoMsg-iAll-conv*:
$0 < k \Longrightarrow ((s \div_i k)\ t = \varepsilon) = (\Box\ t1\ [t * k\ldots,k - Suc\ 0].\ s\ t1 = \varepsilon)$
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*:
$0 < k \Longrightarrow ((s \div_i k)\ t = \varepsilon) = (\Box\ t1\ [\ldots k - 1] \oplus (t * k).\ s\ t1 = \varepsilon)$
by (*simp add*: *iT-add i-shrink-eq-NoMsg-iAll-conv*)

lemma *i-shrink-eq-Msg-iEx-iAll-conv*:
$[\![\ 0 < k;\ m \neq \varepsilon\ ]\!] \Longrightarrow$
$((s \div_i k)\ t = m) =$
$(\Diamond\ t1\ [t * k\ldots,k - Suc\ 0].\ s\ t1 = m \wedge$
$(\Box\ t2\ [Suc\ t1\ldots].\ t2 \leq t * k + k - Suc\ 0 \longrightarrow s\ t2 = \varepsilon))$
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*:
  ⟦ *0 < k*; *m ≠ ε* ⟧ ⟹
  ((*s ÷ᵢ k*) *t* = *m*) =
  (◇ *t1* [*...k − 1*] ⊕ (*t ∗ k*). *s t1* = *m* ∧
    (□ *t2* [*1...*] ⊕ *t1* . *t2* ≤ *t ∗ k* + *k* − *1* ⟶ *s t2* = *ε*))
by (*simp add*: *iT-add i-shrink-eq-Msg-iEx-iAll-conv*)

lemma *i-shrink-eq-Msg-iEx-iAll-cut-greater-conv*:
  ⟦ *0 < k*; *m ≠ ε* ⟧ ⟹
  ((*s ÷ᵢ k*) *t* = *m*) =
  (◇ *t1* [*t ∗ k...*,*k − Suc 0*]. *s t1* = *m* ∧
    (□ *t2* [*t ∗ k...*,*k − Suc 0*] ↓> *t1*. *s t2* = *ε*))
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*:
  ⟦ *0 < k*; *m ≠ ε* ⟧ ⟹
  ((*s ÷ᵢ k*) *t* = *m*) =
  (◇ *t1* [*...k − 1*] ⊕ (*t ∗ k*). *s t1* = *m* ∧
    (□ *t2* ([*...k − 1*] ⊕ (*t ∗ k*)) ↓> *t1*. *s t2* = *ε*))
by (*simp add*: *iT-add i-shrink-eq-Msg-iEx-iAll-cut-greater-conv*)

lemma *i-shrink-eq-Msg-iSince-conv*:
  ⟦ *0 < k*; *m ≠ ε* ⟧ ⟹
  ((*s ÷ᵢ k*) *t* = *m*) =
  (*s t2* = *ε*. *t2* 𝒮 *t1* [*t ∗ k...*,*k − 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*:
  ⟦ *0 < k*; *m ≠ ε* ⟧ ⟹
  ((*s ÷ᵢ k*) *t* = *m*) =
  (*s t2* = *ε*. *t2* 𝒮 *t1* [*...k − 1*] ⊕ (*t ∗ k*). *s t1* = *m*)
by (*simp add*: *iT-add i-shrink-eq-Msg-iSince-conv*)

lemma *iT-Mult-iAll-i-expand-nth-iff*:
$0 < k \Longrightarrow (\Box\ t\ (I \otimes k).\ P\ ((f \odot_i k)\ t)) = (\Box\ t\ I.\ P\ (f\ t))$
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 (*fastforce 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*:
$[\![\ 0 < k;\ \bigwedge t.\ event\ t = (t\ mod\ k = 0);\ t0 = t * k\ ]\!] \Longrightarrow$
$((s \div_i k)\ t = \varepsilon) =$
$(s\ t0 = \varepsilon \wedge (\bigcirc\ t'\ t0\ [0\ldots].\ (s\ t1 = \varepsilon.\ t1\ \mathcal{U}\ t2\ ([0\ldots] \oplus t').\ event\ t2)))$
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-inext*)
apply (*rule iffI*)
apply *simp*
apply (*rule-tac x=t * k + k* **in** *exI*)
apply *fastforce*
apply (*clarify elim*!: *dvdE*, *rename-tac x1 x2*)
apply (*case-tac x2 = Suc* (*t * k*))
apply (*simp add*: *mod-Suc*)
apply (*clarsimp elim*!: *dvdE*, *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*)
apply (*drule-tac y=q * k* **and** *m=k* **in** *less-mod-eq-imp-add-divisor-le*, *simp*)
apply *simp*
done

lemma *i-shrink-eq-Msg-iUntil-start-event-conv*:
$[\![\ 0 < k;\ m \neq \varepsilon;\ \bigwedge t.\ event\ t = (t\ mod\ k = 0);\ t0 = t * k\ ]\!] \Longrightarrow$
$((s \div_i k)\ t = m) = ($
$(s\ t0 = m \wedge (\bigcirc\ t'\ t0\ [0\ldots].\ (s\ t1 = \varepsilon.\ t1\ \mathcal{U}\ t2\ ([0\ldots] \oplus t').\ event\ t2))) \vee$
$(\bigcirc\ t'\ t0\ [0\ldots].\ (\neg\ event\ t1.\ t1\ \mathcal{U}\ t2\ ([0\ldots] \oplus t').\ ($
$s\ t2 = m \wedge \neg\ event\ t2 \wedge (\bigcirc\ t''\ t2\ [0\ldots].$
$(s\ t3 = \varepsilon.\ t3\ \mathcal{U}\ t4\ ([0\ldots] \oplus t'').\ event\ t4))))))$
apply (*case-tac k = Suc 0*)
apply (*simp add*: *iT-add iT-not-empty iNext-iff*)
apply (*drule neq-le-trans*[*OF not-sym*], *simp*)
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=Suc* (*t* ∗ *k* + *k* − *2*) **and** *s=t* ∗ *k* + *k* − *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* = *Suc* (*t* ∗ *k*))
apply *simp*
apply (*case-tac Suc* (*t* ∗ *k*) < *t* ∗ *k* + *k* − *Suc 0*)
apply (*clarsimp simp*: *mod-Suc*)
apply (*case-tac k* = *Suc* (*Suc 0*), *simp*)
apply *simp*
apply (*rule-tac x=t* ∗ *k* + *k* **in** *exI*)
apply *force*
apply *clarsimp*
apply (*subgoal-tac k* = *Suc* (*Suc 0*))
prefer *2*
apply *simp*
apply (*simp add*: *mod-Suc*)
apply (*simp add*: *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 Suc* (*t* ∗ *k*) ≤ *i*)
prefer *2*
apply (*rule ccontr*, *simp*)
apply *simp*
apply (*case-tac i* < *t* ∗ *k* + *k* − *Suc 0*)
apply *clarsimp*
apply (*subgoal-tac 0* < *i mod k*)
prefer *2*
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*

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*] *elim*!: *dvdE*, *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*] *elim*!: *dvdE*, *rename-tac q*)
apply (*subgoal-tac Suc* (*t* $*$ *k*) $<$ *t* $*$ *k* $+$ *k* $-$ *Suc 0*)
prefer *2*
apply *simp*
apply (*clarsimp elim*!: *dvdE*, *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* $\leq$ *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*
apply (*drule-tac x=i* **in** *le-imp-less-or-eq*, *erule disjE*)
apply *simp*
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 elim*!: *dvdE*)+
apply (*rename-tac q*)
apply (*simp add*: *mult.commute*[*of k*])
apply (*subgoal-tac Suc t* ≤ *q*)
prefer *2*
apply (*rule Suc-leI*)
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*:
⟦ *1 < k*; ⋀*t. event t* = (*t mod k* = *k* − *1*); *t0* = *t* ∗ *k* ⟧ ⟹
((*s* ÷*i k*) *t* = *ε*) =
(*s t0* = *ε* ∧ (○ *t′ t0* [*0*...]. (*s t1* = *ε*. *t1* 𝒰 *t2* ([*0*...] ⊕ *t′*). (*event t2* ∧ *s t2* = *ε*))))
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* − *Suc 0*)

prefer *2*
apply *simp*
apply (*rule iffI*)
apply *simp*
apply (*rule-tac x=t ∗ k + k − 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 ≤ Suc* (*t ∗ k*))
apply (*clarsimp simp*: *mod-Suc*)
apply (*drule-tac s=Suc 0* **in** *sym*, *drule-tac x=k − Suc 0* **and** *f=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 − Suc 0 ≤ 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 − 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*:
⟦ *1 < k*; *m ≠ ε*; ⋀*t*. *event t* = (*t mod k* = *k − 1*); *t0* = *t ∗ k* ⟧ ⟹
((*s ÷ᵢ k*) *t* = *m*) = (
(¬ *event t1*. *t1* 𝒰 *t2* ([*0*...] ⊕ *t0*). *event t2* ∧ *s t2* = *m*) ∨
(¬ *event t1*. *t1* 𝒰 *t2* ([*0*...] ⊕ *t0*). (¬ *event t2* ∧ *s t2* = *m* ∧ (
○ *t′ t2* [*0*...]. (*s t3* = *ε*. *t3* 𝒰 *t4* ([*0*...] ⊕ *t′*). *event t4* ∧ *s t4* = *ε*)))))
apply (*simp add*: *i-shrink-eq-Msg-iSince-conv split del*: *if-split*)
apply (*simp only*: *iTL-defs iT-add iT-cut-greater iT-cut-less Ball-def Bex-def iT-iff iFROM-inext*)

apply (*subgoal-tac* $t * k < t * k + k - Suc\ 0$)
 prefer *2*
 apply *simp*
apply (*rule iffI*)
 apply (*subgoal-tac* $\bigwedge x.\ t * k \leq x \Longrightarrow x < t * k + k - Suc\ 0 \Longrightarrow x\ mod\ k \neq k - Suc\ 0$)
  prefer *2*
  apply (*rule less-imp-neq*)
  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*: *if-split*)+
 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*: *if-split*)
 apply (*rule conjI*)
  apply (*rule-tac x=t * k + k − 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 ≤ 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 (*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 if-split-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 ≤ t2*)
prefer *2*
apply (*rule ccontr*)
apply (*simp add*: *linorder-not-le*)
apply (*drule-tac m=t2* **in** *less-imp-le-pred*)
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 (*fastforce 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 ≤ 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) ≤ 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*])
apply (*drule-tac n=Suc t1* **in** *between-imp-mod-le*[*of k − Suc 0 − Suc 0 k t*])
apply *simp-all*
apply (*simp add*: *iIN-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*)) **for** *A B*)

apply (*drule-tac x=t3* **in** *spec*)
apply (*subgoal-tac t3* $\leq$ *t2* $\Longrightarrow$ *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* $\leq$ *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*) $\leq$ *t2*)
prefer *2*
apply (*simp only*: *mult.commute*[*of t*])
apply (*rule mult-divisor-le-mod-ge-imp-ge*)
apply *simp-all*
done

end

5 AutoFocus message stream processing and temporal logic on intervals

theory *IL-AF-Stream-Exec*
imports *Main IL-AF-Stream AF-Stream-Exec*
begin

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

lemma *i-Exec-Stream-Pre-Post1-iAll*:
⟦ *result* = *i-Exec-Comp-Stream trans-fun input c*;
∀ *x-n c-n*. *P1 x-n* ∧ *P2 c-n* ⟶ *Q* (*trans-fun x-n c-n*) ⟧ ⟹
□ *t I*. (*P1* (*input t*) ∧ *P2* (*result*$^{\leftarrow\ c}$ *t*) ⟶ *Q* (*result t*))
by (*simp add*: *i-Exec-Stream-Pre-Post1*)

Direct relation between input and result after transition

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

lemma *i-Exec-Stream-Pre-Post3-iAll-iNext*:
⟦ *result* = *i-Exec-Comp-Stream trans-fun input c*;
∀ *x-n c-n*. *P c-n* ⟶ *Q x-n* (*trans-fun x-n c-n*);

$\forall\, t{\in}I.\ inext\ t\ I' = Suc\ t\ ]\!] \Longrightarrow$
$\Box\ t\ I.\ P\ (result\ t) \longrightarrow (\bigcirc\ t1\ t\ I'.\ Q\ (input\ t1)\ (result\ t1))$
by (*rule iallI*, *simp add*: *iNext-def i-Exec-Stream-Pre-Post2-Suc*)

lemma *i-Exec-Stream-Init-Pre-Post1-iAll-iNext*:
$[\![\ result = i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ input\ c;$
$\forall\, x\text{-}n\ c\text{-}n.\ P1\ x\text{-}n \wedge P2\ c\text{-}n \longrightarrow Q\ (trans\text{-}fun\ x\text{-}n\ c\text{-}n);$
$\forall\, t{\in}I.\ inext\ t\ I' = Suc\ t\ ]\!] \Longrightarrow$
$\Box\ t\ I.\ (P1\ (input\ t) \wedge P2\ (result\ t) \longrightarrow (\bigcirc\ t1\ t\ I'.\ Q\ (result\ t1)))$
by (*rule iallI*, *simp add*: *iNext-def i-Exec-Stream-Init-Pre-Post1*)

Direct relation between input and state before transition

lemma *i-Exec-Stream-Init-Pre-Post2-iAll-iNext*:
$[\![\ result = i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ input\ c;$
$\forall\, x\text{-}n\ c\text{-}n.\ P\ x\text{-}n\ c\text{-}n \longrightarrow Q\ (trans\text{-}fun\ x\text{-}n\ c\text{-}n);$
$\forall\, t{\in}I.\ inext\ t\ I' = Suc\ t\ ]\!] \Longrightarrow$
$\Box\ t\ I.\ (P\ (input\ t)\ (result\ t) \longrightarrow (\bigcirc\ t1\ t\ I'.\ Q\ (result\ t1)))$
by (*rule iallI*, *simp add*: *iNext-def i-Exec-Stream-Init-Pre-Post2*)

Relation between input and output

lemma *i-Exec-Stream-Init-Pre-Post3-iAll-iNext*:
$[\![\ result = i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ input\ c;$
$\forall\, x\text{-}n\ c\text{-}n.\ P\ c\text{-}n \longrightarrow Q\ x\text{-}n\ (trans\text{-}fun\ x\text{-}n\ c\text{-}n);$
$\forall\, t{\in}I.\ inext\ t\ I' = Suc\ t\ ]\!] \Longrightarrow$
$\Box\ t\ I.\ (P\ (result\ t) \longrightarrow (\bigcirc\ t1\ t\ I'.\ Q\ (input^{\leftarrow\ \varepsilon}\ t1)\ (result\ t1)))$
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*)
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.

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

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 ⟹
((*i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c*) *t* = *ε*) =
(□ *t1* [*Suc* (*t* ∗ *k*)...,*k* − *Suc 0*]. (*output-fun* ∘ *i-Exec-Comp-Stream-Init trans-fun* (*input* ⊙*i k*) *c*) *t1* = *ε*)
apply (*unfold i-Exec-Comp-Stream-Acc-Output-def*)
apply (*simp add*: *i-shrink-eq-NoMsg-iAll-conv i-Exec-Stream-Init-eq-i-Exec-Stream-Cons*)
apply (*rule-tac t*=[*Suc* (*t* ∗ *k*)...,*k* − *Suc 0*] **and** *s*=[*t* ∗ *k*...,*k* − *Suc 0*] ⊕ *1* **in** *subst*)
apply (*simp add*: *iIN-add*)
apply (*simp add*: *iT-Plus-iAll-conv*)
done

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iEx-iAll-cut-greater-conv*:
⟦ *0 < k*; *m* ≠ *ε*; *s* = (*output-fun* ∘ *i-Exec-Comp-Stream trans-fun* (*input* ⊙*i k*) *c*) ⟧ ⟹
((*i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c*) *t* = *m*) =
(◇ *t1* [*t* ∗ *k*...,*k* − *Suc 0*]. (*s t1* = *m* ∧
(□ *t2* [*t* ∗ *k*...,*k* − *Suc 0*] ↓> *t1* . *s t2* = *ε*)))
by (*simp add*: *i-Exec-Comp-Stream-Acc-Output-def i-shrink-eq-Msg-iEx-iAll-cut-greater-conv*)

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iEx-iAll-cut-greater-conv2*:
⟦ *0 < k*; *m* ≠ *ε*; *s* = (*output-fun* ∘ *i-Exec-Comp-Stream trans-fun* (*input* ⊙*i k*) *c*) ⟧ ⟹
((*i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c*) *t* = *m*) =
(◇ *t1* [...*k* − *Suc 0*] ⊕ (*t* ∗ *k*). (*s t1* = *m* ∧
(□ *t2* ([...*k* − *Suc 0*] ⊕ (*t* ∗ *k*)) ↓> *t1* . *s t2* = *ε*)))
by (*simp add*: *i-Exec-Comp-Stream-Acc-Output-def i-shrink-eq-Msg-iEx-iAll-cut-greater-conv2*)

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

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-Msg-iSince-conv*:
⟦ *0 < k*; *m* ≠ *ε*; *s* = (*output-fun* ∘ *i-Exec-Comp-Stream-Init trans-fun* (*input* ⊙*i k*) *c*) ⟧ ⟹
((*i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c*) *t* = *m*) =

(*s t2 = ε. t2 $\mathcal{S}$ t1 [Suc (t * k)...,k − 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*)...,*k* − *Suc 0*] **and** *s*=[*t* * *k*...,*k* − *Suc 0*] ⊕ *1* **in** *subst*)
apply (*simp add*: *iIN-add*)
apply (*simp add*: *iT-Plus-iSince-conv*)
done

lemma *i-Exec-Comp-Stream-Acc-Output--eq-iAll-iSince-conv*:
⟦ *0 < k*; *s* = (*output-fun* ∘ *i-Exec-Comp-Stream trans-fun* (*input* ⊙$_i$ *k*) *c*) ⟧ ⟹
((*i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c*) *t* = *m*) =
((*m* = *ε* ⟶ (□ *t1* [*t* * *k*...,*k* − *Suc 0*]. *s t1* = *ε*)) ∧
((*m* ≠ *ε* ⟶ (*s t2* = *ε*. *t2* $\mathcal{S}$ *t1* [*t* * *k*...,*k* − *Suc 0*]. *s t1* = *m*))))
apply (*case-tac m* = *ε*)
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*:
⟦ *0 < k*; *s* = (*output-fun* ∘ *i-Exec-Comp-Stream trans-fun* (*input* ⊙$_i$ *k*) *c*) ⟧ ⟹
((*i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c*) *t* = *m*) =
((*m* = *ε* ⟶ (□ *t1* [...*k* − *Suc 0*] ⊕ (*t* * *k*). *s t1* = *ε*)) ∧
((*m* ≠ *ε* ⟶ (*s t2* = *ε*. *t2* $\mathcal{S}$ *t1* [...*k* − *Suc 0*] ⊕ (*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.

lemma *i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-iAll-start-event-conv*:
⟦ *0 < k*; ⋀ *t*. *event t* = (*t mod k* = *0*); *t0* = *t* * *k*;
s = (*output-fun* ∘ *i-Exec-Comp-Stream trans-fun* (*input* ⊙$_i$ *k*) *c*) ⟧⟹
((*i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c*) *t* = *ε*) =
(*s t0* = *ε* ∧ (○ *t′ t0* [*0*...]. (*s t1* = *ε*. *t1* $\mathcal{U}$ *t2* [*0*...] ⊕ *t′*. *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*:
⟦ *0 < k*; ⋀ *t*. *event t* = ((*t* + *k* − *Suc 0*) *mod k* = *0*); *t0* = *Suc* (*t* * *k*);
s = (*output-fun* ∘ *i-Exec-Comp-Stream-Init trans-fun* (*input* ⊙$_i$ *k*) *c*) ⟧⟹
((*i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c*) *t* = *ε*) =
(*s t0* = *ε* ∧ (○ *t′ t0* [*0*...]. (*s t1* = *ε*. *t1* $\mathcal{U}$ *t2* [*0*...] ⊕ *t′*. *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*))...] **and** *s*=[*Suc* (*t***k*)...] ⊕ *Suc 0* **in** *subst*)
apply (*simp add*: *iFROM-add*)
apply (*simp add*: *iT-Plus-iUntil-conv*)
done

lemma *i-Exec-Comp-Stream-Acc-Output--Init--eq-NoMsg-iAll-start-event2-conv*:
⟦ *Suc 0 < k*; ⋀ *t. event t = (t mod k = Suc 0)*; *t0 = Suc (t * k)*;
s = (output-fun ∘ i-Exec-Comp-Stream-Init trans-fun (input ⊙ᵢ k) c) ⟧⟹
((*i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c*) *t = ε*) =
(*s t0 = ε* ∧ (○ *t′ t0* [*0...*]. (*s t1 = ε. t1* 𝒰 *t2* [*0...*] ⊕ *t′. event t2*)))
by (*simp add*: *i-Exec-Comp-Stream-Acc-Output--Init--eq-NoMsg-iAll-start-event-conv mod-eq-Suc-0-conv*)

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iUntil-start-event-conv*:
⟦ *0 < k*; *m ≠ ε*; ⋀*t. event t = (t mod k = 0)*; *t0 = t * k*;
s = (output-fun ∘ i-Exec-Comp-Stream trans-fun (input ⊙ᵢ k) c) ⟧ ⟹
((*i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c*) *t = m*) = (
(*s t0 = m* ∧ (○ *t′ t0* [*0...*]. (*s t1 = ε. t1* 𝒰 *t2* ([*0...*] ⊕ *t′*). *event t2*))) ∨
(○ *t′ t0* [*0...*]. (¬ *event t1. t1* 𝒰 *t2* ([*0...*] ⊕ *t′*). (
s t2 = m ∧ ¬ *event t2* ∧ (○ *t″ t2* [*0...*].
(*s t3 = ε. t3* 𝒰 *t4* ([*0...*] ⊕ *t″*). *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*:
⟦ *0 < k*; *m ≠ ε*; ⋀*t. event t = ((t + k − Suc 0) mod k = 0)*; *t0 = Suc (t * k)*;
s = (output-fun ∘ i-Exec-Comp-Stream-Init trans-fun (input ⊙ᵢ k) c) ⟧ ⟹
((*i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c*) *t = m*) = (
(*s t0 = m* ∧ (○ *t′ t0* [*0...*]. (*s t1 = ε. t1* 𝒰 *t2* ([*0...*] ⊕ *t′*). *event t2*))) ∨
(○ *t′ t0* [*0...*]. (¬ *event t1. t1* 𝒰 *t2* ([*0...*] ⊕ *t′*). (
s t2 = m ∧ ¬ *event t2* ∧ (○ *t″ t2* [*0...*].
(*s t3 = ε. t3* 𝒰 *t4* ([*0...*] ⊕ *t″*). *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*:
⟦ *Suc 0 < k*; *m ≠ ε*; ⋀*t. event t = (t mod k = Suc 0)*; *t0 = Suc (t * k)*;
s = (output-fun ∘ i-Exec-Comp-Stream-Init trans-fun (input ⊙ᵢ k) c) ⟧ ⟹
((*i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c*) *t = m*) = (
(*s t0 = m* ∧ (○ *t′ t0* [*0...*]. (*s t1 = ε. t1* 𝒰 *t2* ([*0...*] ⊕ *t′*). *event t2*))) ∨
(○ *t′ t0* [*0...*]. (¬ *event t1. t1* 𝒰 *t2* ([*0...*] ⊕ *t′*). (
s t2 = m ∧ ¬ *event t2* ∧ (○ *t″ t2* [*0...*].
(*s t3 = ε. t3* 𝒰 *t4* ([*0...*] ⊕ *t″*). *event t4*))))))
by (*simp add*: *i-Exec-Comp-Stream-Acc-Output--Init--eq-Msg-iUntil-start-event-conv mod-eq-Suc-0-conv*)

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

$((i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}Output\ k\ output\text{-}fun\ trans\text{-}fun\ input\ c)\ t = \varepsilon) =$
$(s\ t0 = \varepsilon \wedge (\bigcirc\ t'\ t0\ [0\ldots].\ (s\ t1 = \varepsilon.\ t1\ \mathcal{U}\ t2\ [0\ldots] \oplus t'.\ event\ t2 \wedge s\ t2 = \varepsilon)))$
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*:
$[\![\ Suc\ 0 < k;\ \bigwedge t.\ event\ t = (t\ mod\ k = 0);\ t0 = Suc\ (t * k);$
$s = (output\text{-}fun \circ i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Init\ trans\text{-}fun\ (input \odot_i k)\ c)\ ]\!]\Longrightarrow$
$((i\text{-}Exec\text{-}Comp\text{-}Stream\text{-}Acc\text{-}Output\ k\ output\text{-}fun\ trans\text{-}fun\ input\ c)\ t = \varepsilon) =$
$(s\ t0 = \varepsilon \wedge (\bigcirc\ t'\ t0\ [0\ldots].\ (s\ t1 = \varepsilon.\ t1\ \mathcal{U}\ t2\ [0\ldots] \oplus t'.\ event\ t2 \wedge s\ t2 = \varepsilon)))$
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*)*...*] ⊕ *1* **in** *subst*)
apply (*simp add*: *iFROM-add*)
apply (*simp add*: *iT-Plus-iUntil-conv*)
apply (*simp add*: *mod-eq-divisor-minus-Suc-0-conv*)
done

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

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)
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 # ε^(t1 − 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 ⊙i k ⇓ (t * k))
c)))))
```

apply (*subgoal-tac*
(□ *t1* [*t* ∗ *k*...,*k* − *Suc 0*]. *output-fun* (*f-Exec-Comp trans-fun* (*input* ⊙$_i$ *k* ⇓ *Suc t1*) *c*) = *NoMsg*))
prefer *2*
apply (*clarsimp simp*: *iIN-iff*, *rename-tac t1*)
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*)
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* ∀ *i* < *k*. *input* ⊙$_i$ *k* ⇑ *Suc* (*t* ∗ *k*) ⇓ *i* = *NoMsg*^{i})
prefer *2*
apply (*simp add*: *list-eq-iff i-expand-nth-if*)
apply (*rule iffI*)
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* $\Uparrow$ *Suc t2* $\Downarrow$ *i* **and** *s*=ε^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*)
apply *simp*
apply (*subgoal-tac t* $*$ *k* $<$ *Suc* (*t2* $+$ *i1*) $\wedge$ *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*+)
apply (*rule f-Exec-State-Idle-replicate-NoMsg-gr0-output*, *assumption*+)
done

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-imp*:
⟦ *0* $<$ *k*;
s = *i-Exec-Comp-Stream trans-fun* (*input* $\odot_i$ *k*) *c*;
t0 = *t* $*$ *k*;
t1 $\in$ [*0*..., *k* $-$ *Suc 0*] $\oplus$ *t0*;
State-Idle localState output-fun trans-fun (*localState* (*s t1*));
output-fun (*s t1*) $\neq$ ε ⟧ $\Longrightarrow$
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c t = *output-fun* (*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 add*: *iT-add iT-iff*, *erule conjE*)
apply (*simp only*: *i-Exec-Stream-Acc-Output-nth i-Exec-Stream-nth*)
apply (*rule-tac t=Suc t1* **and** *s*=*t* $*$ *k* $+$ (*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*) $\Downarrow$ (*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* $-$ *Suc 0* = *t1* $-$ *t* $*$ *k* $+$ *i*)
prefer *2*
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*])
apply (*subst State-Idle-append-replicate-NoMsg-output-last-message*)

apply (*simp only*: *f-Exec-append*[*symmetric*])
apply (*rule-tac t=input* ⇓ *t* ⊙$_f$ *k* @ *input t* # *NoMsg*$^{t1 - t * k}$ **and** *s=input* ⊙$_i$ *k* ⇓ *Suc t1* **in** *subst*)
apply (*subst i-expand-i-take-mult*[*symmetric*])
apply (*drule-tac t=input t* # *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* # *NoMsg*$^{t1 - t * k}$)
(*f-Exec-Comp trans-fun* (*input* ⇓ *t* ⊙$_f$ *k*) *c*) ≠ [])
prefer *2*
apply (*simp add*: *f-Exec-Stream-not-empty-conv*)
apply (*rule ssubst*[*OF last-message-Msg-eq-last*])
apply *simp*
apply (*subst map-last*, *simp*)
apply (*subst f-Exec-eq-f-Exec-Stream-last2*[*symmetric*], *simp*)
apply (*subst f-Exec-append*[*symmetric*])
apply (*rule-tac t=input* ⇓ *t* ⊙$_f$ *k* @ *input t* # *NoMsg*$^{t1 - t * k}$ **and** *s=input* ⊙$_i$ *k* ⇓ *Suc t1* **in** *subst*)
apply (*subst i-expand-i-take-mult*[*symmetric*])
apply (*rule-tac t=Suc t1* **and** *s=t* ∗ *k* + (*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

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 ≠ *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*) =
(◇ *t1* [*0*..., *k* − *Suc 0*] ⊕ *t0*. (
(*output-fun* (*s t1*) = *m* ∧ *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*
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*
⋀ *n1 n2*.
⟦ *n1* < *n2*; *n1* ∈ [*0*...,*k* − *Suc 0*] ⊕ *t* ∗ *k*; *n2* ∈ [*0*...,*k* − *Suc 0*] ⊕ *t* ∗ *k*;
State-Idle localState output-fun trans-fun (*localState* (*f-Exec-Comp trans-fun* (*input* ⊙*i k* ⇓ *Suc n1*) *c*));
output-fun (*f-Exec-Comp trans-fun* (*input* ⊙*i k* ⇓ *Suc n2*) *c*) ≠ *NoMsg* ⟧ ⟹ *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*)
apply (*subgoal-tac input* ⊙*i k* ⇑ *Suc n1* ⇓ *i* = *ε*^*i*)
prefer *2*
apply (*subst i-take-i-drop*)
apply (*rule-tac t=ε*^*i* **and** *s=ε*^*i* + *Suc n1* − *Suc n1* **in** *subst*, *simp*)
apply (*rule-tac t=t* **in** *i-expand-nth-interval-eq-replicate-NoMsg*)
apply (*simp add*: *iT-add iT-iff*)+
apply (*frule-tac c=f-Exec-Comp trans-fun* (*input* ⊙*i k* ⇓ *Suc n1*) *c* **and** *n=i*
in *f-Exec-State-Idle-replicate-NoMsg-gr0-output*)
apply (*fastforce 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*)))))
apply (*subst i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv2*, *assumption*+)
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* $\odot_i$ *k* $\Downarrow$ (*i* + *Suc t2'*) $\uparrow$ *Suc t2'* = ε^i)
prefer *2*
apply (*rule-tac t=*ε^i **and** *s=*$\varepsilon^{i + Suc\ t2' - Suc\ t2'}$ **in** *subst*, *simp*)
apply (*rule-tac t=t* **in** *i-expand-nth-interval-eq-replicate-NoMsg*)
apply *simp+*
apply (*drule-tac c=*(*f-Exec-Comp trans-fun* (*input* $\odot_i$ *k* $\Downarrow$ *Suc t2'*) *c*) **and** *n=i*
in *f-Exec-State-Idle-replicate-NoMsg-gr0-output*, *assumption*)
apply *simp*
apply (*fastforce 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* $\odot_i$ *k*) *c*;
t1 ∈ [*0*..., *k* − *Suc 0*] ⊕ *t0*;
output-fun (*s t1*) = *m*;
○ *t2 t1* [*0*...].
((*output-fun* (*s t3*) = *ε*. *t3* $\mathcal{U}$ *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*])

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* $\leq t * k + j$)
prefer *2*
apply *fastforce*
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* $\odot_i$ *k* $\Downarrow$ ($t * k +$ *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*
apply (*rule-tac t=t* **in** *ssubst*[*OF i-expand-nth-interval-eq-replicate-NoMsg*, *rule-format*], *simp+*)
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*
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*)
apply (*rule-tac m=t2 mod k* **in** *f-Exec-State-Idle-replicate-NoMsg-gr-output*, *assumption*)
apply (*simp add*: *minus-div-mult-eq-mod* [*symmetric*])
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*;
$\Box$ *t1* [*0*..., *k* $-$ *Suc 0*] $\oplus$ *t0*. $\neg$ (
State-Idle localState output-fun trans-fun (*localState* (*s t1*)) $\wedge$
output-fun (*s t1*) $\neq \varepsilon$) ⟧ $\Longrightarrow$
(*i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c t* = *m*) =
($\Diamond$ *t1* [*0*..., *k* $-$ *Suc 0*] $\oplus$ *t0*. (
(*output-fun* (*s t1*) = *m*) $\wedge$
($\bigcirc$ *t2 t1* [*0*...].

```
      ((output-fun (s t3) = ε. t3 𝒰 t4 ([0...] ⊕ t2).
      (output-fun (s t4) = ε ∧ 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)
 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
   □ t' ([0...] ⊕ (Suc (t * k + i))) ↓< (t * k + k).
   output-fun (f-Exec-Comp trans-fun (input ⊙i k ⇓ Suc t') c) = ε)
  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 ≤ t')
   prefer 2
   apply simp
  apply (rule-tac t=Suc t' and s=t * k + (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)
  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)
 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
   ¬ State-Idle localState output-fun trans-fun
       (localState (f-Exec-Comp trans-fun (input ⊙i k ⇓ (t * k + 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 for x)
 apply (drule-tac
   a=t * k + k and b=t * k + Suc (k − Suc 0) and
```

```
   P=λx. State-Idle localState output-fun trans-fun
          (localState (f-Exec-Comp trans-fun (input ⊙i k ⇓ x) c)) in back-subst,
simp)
  apply (simp only: i-expand-i-take-mult-Suc f-Exec-append)
 apply (frule-tac n=k − 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)
 apply (frule-tac i=i1 in less-diff-conv[THEN iffD1, rule-format])
 apply (drule-tac a=i1 and P=λx. (x < k − 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 + 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
  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 ∃ 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 for 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)
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)
 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 for 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)
   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: minus-div-mult-eq-mod [symmetric])
  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+
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* $\leq t * k + (k - Suc\ 0)$)
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* **for** *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* 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*))))))))
apply (*rule iffI*)
apply (*frule subst*[*OF i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-conv2*, **where** *P*=λ*x*. *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*)
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*)
apply (*simp only*: *i-expand-nth-interval-eq-replicate-NoMsg*[*of k t*, *OF - le-imp-less-Suc le-add2*])
apply (*drule-tac c=f-Exec-Comp trans-fun* (*input* $\odot_i$ *k* ⇓ *Suc t'*) *c* **and** *n=i* **in** *f-Exec-State-Idle-replicate-NoMsg-gr0-output*)
apply *simp+*
apply (*rule i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-imp*, *simp+*)
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* $\odot_i$ *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* $\mathcal{U}$ *t4* ([*0...*] ⊕ *t2*).
(*output-fun* (*s t4*) = *ε* ∧ *State-Idle localState output-fun trans-fun* (*localState* (*s t4*)))))))))
apply (*subst conj-disj-distribL*)
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*)
apply (*frule i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv2*[*THEN iffD1*, *OF Suc-lessD*], *assumption+*)
apply *fastforce*
apply (*elim iexE conjI*, *rename-tac t2*)

apply (*erule disjE*)
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+*)
apply (*rule-tac ?t1.0=t2* **in** *i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-imp2*, *simp+*)
apply (*rule ssubst*[*OF i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-conv2*[*OF - - - refl refl*]], *simp+*)
apply *fastforce*
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

lemma *i-Exec-Comp-Stream-Acc-Output--eq-iAll-iUntil-State-Idle-conv2*:
⟦ *Suc 0 < k*;
State-Idle localState output-fun trans-fun (
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input c t);
t0 = *t* ∗ *k*;
s = *i-Exec-Comp-Stream trans-fun* (*input* ⊙*i k*) *c* ⟧ ⟹
(*i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c t* = *m*) = (
(*m* = *ε* ⟶
(*output-fun* (*s t1*) = *ε*. *t1* 𝒰 *t2* ([*0*...] ⊕ *t0*). (
output-fun (*s t2*) = *ε* ∧ *State-Idle localState output-fun trans-fun* (*localState* (*s t2*))))) ∧
(*m* ≠ *ε* ⟶
(◇ *t1* [*0*..., *k* − *Suc 0*] ⊕ *t0*. (
output-fun (*s t1*) = *m* ∧
(*State-Idle localState output-fun trans-fun* (*localState* (*s t1*)) ∨
(○ *t2 t1* [*0*...].

```
      ((output-fun (s t3) = ε. t3 𝒰 t4 ([0...] ⊕ t2).
      (output-fun (s t4) = ε ∧ State-Idle localState output-fun trans-fun (localState
(s t4)))))))))))
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-conv2)
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)))) ∨
  ((¬ State-Idle localState output-fun trans-fun (localState (s t2))). t2 𝒰 t1 [0...]
⊕ t0.
   (output-fun (s t1) = m ∧
   (○ t3 t1 [0...].
     ((output-fun (s t4) = ε. t4 𝒰 t5 ([0...] ⊕ t3).
      (output-fun (s t5) = ε ∧ State-Idle localState output-fun trans-fun (localState
(s t5)))))))))
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)
 apply (frule i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv[THEN
iffD1, OF Suc-lessD], simp+)
 apply (erule disjE)
 apply (rule i-Exec-Comp-Stream-Acc-Output--eq-Msg-with-State-Idle-conv[THEN
iffD2], simp+)
 apply (rule-tac i-Exec-Comp-Stream-Acc-Output--eq-Msg-before-State-Idle-imp, simp+)
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 for 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 (fastforce 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

lemma i-Exec-Comp-Stream-Acc-Output--eq-iUntil-State-Idle-conv:
  ⟦ Suc 0 < k;
    State-Idle localState output-fun trans-fun (
      i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input c t);
    t0 = t * k;
    s = i-Exec-Comp-Stream trans-fun (input ⊙ᵢ k) c ⟧ ⟹
  (i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c t = m) = (
  (m = ε ⟶
    (output-fun (s t1) = ε. t1 𝒰 t2 ([0...] ⊕ t0). (
      output-fun (s t2) = ε ∧ State-Idle localState output-fun trans-fun (localState
(s t2))))) ∧
  (m ≠ ε ⟶
    (((¬ State-Idle localState output-fun trans-fun (localState (s t2))). t2 𝒰 t1 [0...]
⊕ t0.
      (output-fun (s t1) = m ∧
        (State-Idle localState output-fun trans-fun (localState (s t1)) ∨
        (○ t3 t1 [0...].
            ((output-fun (s t4) = ε. t4 𝒰 t5 ([0...] ⊕ t3).
                (output-fun (s t5) = ε ∧ State-Idle localState output-fun trans-fun
(localState (s t5))))))))))))
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.

corollary *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-iEx-imp1*:
⟦ *Suc 0 < k*;
State-Idle localState output-fun trans-fun (
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input c t);
m ≠ ε;
t0 = t ∗ k;
s = i-Exec-Comp-Stream trans-fun (*input* ⊙*ᵢ k*) *c*;
(◇ *t1* [*0*..., *k − Suc 0*] ⊕ *t0*. (
output-fun (*s t1*) = *m* ∧ *State-Idle localState output-fun trans-fun* (*localState* (*s t1*)))) ⟧ ⟹
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c t = m
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 ≠ 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
by (*blast intro*: *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv2'*[*THEN iffD2*])

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-iUntil-imp1*:
⟦ *Suc 0 < k*;
State-Idle localState output-fun trans-fun (
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input c t);
m ≠ ε;
t0 = t ∗ k;
s = i-Exec-Comp-Stream trans-fun (*input* ⊙*ᵢ k*) *c*;
(¬ *State-Idle localState output-fun trans-fun* (*localState* (*s t2*))). *t2* 𝒰 *t1* [*0*...] ⊕ *t0*.
(*output-fun* (*s t1*) = *m* ∧ *State-Idle localState output-fun trans-fun* (*localState* (*s t1*))) ⟧ ⟹
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c t = m
by (*blast intro*: *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv'*[*THEN iffD2*])

lemma *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-iUntil-imp2*:
⟦ *Suc 0 < k*;
State-Idle localState output-fun trans-fun (
i-Exec-Comp-Stream-Acc-LocalState k localState trans-fun input c t);
m ≠ ε;
t0 = t ∗ k;
s = i-Exec-Comp-Stream trans-fun (input ⊙ᵢ k) c;
(¬ State-Idle localState output-fun trans-fun (localState (s t2))). t2 𝒰 t1 [0...]
⊕ t0.
(output-fun (s t1) = m ∧
(○ t3 t1 [0...].
((output-fun (s t4) = ε. t4 𝒰 t5 ([0...] ⊕ t3).
(output-fun (s t5) = ε ∧ State-Idle localState output-fun trans-fun (localState
(s t5))))))) ⟧ ⟹
i-Exec-Comp-Stream-Acc-Output k output-fun trans-fun input c t = m
by (*blast intro*: *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv'*[*THEN iffD2*])

List of selected lemmas about output of accelerated components.

thm *i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-iAll-conv*
thm *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iEx-iAll-cut-greater-conv*
thm *i-Exec-Comp-Stream-Acc-Output--eq-Msg-iSince-conv*
thm *i-Exec-Comp-Stream-Acc-Output--eq-iAll-iSince-conv*

thm *i-Exec-Comp-Stream-Acc-Output--eq-NoMsg-State-Idle-conv*
thm *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv2*
thm *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv*
thm *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv2'*
thm *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-conv'*

thm *i-Exec-Comp-Stream-Acc-Output--eq-iAll-iUntil-State-Idle-conv2*
thm *i-Exec-Comp-Stream-Acc-Output--eq-iUntil-State-Idle-conv*

thm *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-iEx-imp1*
thm *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-iEx-imp2*
thm *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-iUntil-imp1*
thm *i-Exec-Comp-Stream-Acc-Output--eq-Msg-State-Idle-iUntil-imp2*

end