Miscellaneous Examples of restriction Spaces

Benoît Ballenghien Benjamin Puyobro Burkhart Wolff

February 6, 2026

Abstract

In this session, a number of examples are provided to illustrate how the `Restriction_Spaces` library works. The simple cases are, of course, covered: trivial construction, booleans, integers, option type, and so on. More elaborate situations are also covered, such as formal series and a trace model of the CSP process algebra.

Contents

1 Trivial Construction

Restriction instance for any type.

typedef *'a type'* = ‹*UNIV* :: *'a set*› **by** *auto*

instantiation *type'* :: (*type*) *restriction*
begin

lift-definition *restriction-type'* :: ‹*'a type'* ⇒ *nat* ⇒ *'a type'*›
is ‹λ*x n*. *if n* = *0 then undefined else x*› **.**

instance by (*intro-classes*, *transfer*, *simp add*: *min-def*)

end

lemma *restriction-type'-0-is-undefined* [*simp*] :
‹*x* ↓ *0* = *undefined*› **for** *x* :: ‹*'a type'*› **by** *transfer simp*

instance *type'* :: (*type*) *restriction-space*
by (*intro-classes*, *simp*, *transfer*, *auto*)

lemma *restriction-tendsto-type'-iff* :
‹*σ* −↓→ Σ ⟷ (∃ *n0*. ∀ *n*≥*n0*. *σ n* = Σ)› **for** Σ :: ‹*'a type'*›
by (*simp add*: *restriction-tendsto-def*, *transfer*, *auto*)

lemma *restriction-chain-type'-iff* :
‹*chain*↓ *σ* ⟷ *σ 0* = *undefined* ∧ (∀ *n*≥*Suc 0*. *σ n* = *σ* (*Suc 0*))›
for *σ* :: ‹*nat* ⇒ *'a type'*›
by (*simp add*: *restriction-chain-def-ter*, *transfer*, *simp*)
(*safe*, (*simp-all*)[*3*], *metis Suc-le-D Suc-le-eq zero-less-Suc*)

instance *type'* :: (*type*) *complete-restriction-space*
by *intro-classes*
(*auto simp add*: *restriction-chain-type'-iff restriction-convergent-def*
restriction-tendsto-type'-iff)

2 Booleans

Restriction instance for *bool*.

instantiation *bool* :: *restriction*
begin

definition *restriction-bool* :: ‹*bool* ⇒ *nat* ⇒ *bool*›
  where ‹*b* ↓ *n* ≡ *if n* = *0 then False else b*›

instance by (*intro-classes*) (*auto simp add*: *restriction-bool-def*)
end

lemma *restriction-bool-0-is-False* [*simp*] : ‹*b* ↓ *0* = *False*›
  by (*simp add*: *restriction-bool-def*)

Restriction space instance for *bool*.

instance *bool* :: *restriction-space*
  by *intro-classes* (*simp-all add*: *restriction-bool-def gt-ex*)

Complete Restriction space instance for *bool*.

lemma *restriction-tendsto-bool-iff* :
  ‹*σ* −↓→ Σ ⟷ (∃ *n*. ∀ *k*≥*n*. *σ k* = Σ)› **for** Σ :: *bool*
  unfolding *restriction-tendsto-def*
  by (*auto simp add*: *restriction-bool-def*)

instance *bool* :: *complete-restriction-space*
proof *intro-classes*
  fix *σ* :: ‹*nat* ⇒ *bool*› **assume** ‹*chain*↓ *σ*›
  hence ‹(∀ *n*>*0*. ¬ *σ n*) ∨ (∀ *n*>*0*. *σ n*)›
   by (*simp add*: *restriction-chain-def restriction-bool-def split*: *if-split-asm*)
    (*metis One-nat-def Zero-not-Suc gr0-conv-Suc nat-induct-non-zero zero-induct*)
  hence ‹*σ* −↓→ *False* ∨ *σ* −↓→ *True*›
   by (*metis* (*full-types*) *gt-ex order.strict-trans2 restriction-tendsto-def*)
  thus ‹*convergent*↓ *σ*›
    using *restriction-convergentI* **by** *blast*
qed

lemma *restriction-cont-imp-restriction-adm* :
  ‹*cont*↓ *P* ⟹ *adm*↓ *P*› **for** *P* :: ‹*'a* :: *restriction-space* ⇒ *bool*›
 unfolding *restriction-adm-def restriction-cont-on-def restriction-cont-at-def*
  by (*auto simp add*: *restriction-tendsto-bool-iff*)

lemma *restriction-compact-bool* : ‹*compact*↓ (*UNIV* :: *bool set*)›
  by (*simp add*: *finite-imp-restriction-compact*)

3 Naturals

Restriction instance for *nat*.

instantiation *nat* :: *restriction*
begin

definition *restriction-nat* :: ‹*nat* ⇒ *nat* ⇒ *nat*›
  where ‹*x* ↓ *n* ≡ *if x* ≤ *n then x else n*›

instance by *intro-classes* (*simp add*: *restriction-nat-def*)

end

lemma *restriction-nat-0-is-0* [*simp*] : ‹*x* ↓ *0* = (*0* :: *nat*)›
  by (*simp add*: *restriction-nat-def*)

Restriction Space instance for *nat*.

instance *nat* :: *restriction-space*
 by *intro-classes* (*use nat-le-linear* **in** ‹*auto simp add*: *restriction-nat-def*›)

Constructive Suc

lemma *constructive-Suc* : ‹*constructive Suc*›
proof (*rule constructiveI*)
  show ‹*x* ↓ *n* = *y* ↓ *n* ⟹ *Suc x* ↓ *Suc n* = *Suc y* ↓ *Suc n*› **for** *x y n*
    by (*simp add*: *restriction-nat-def split*: *if-split-asm*)
qed

Non too destructive pred

lemma *non-too-destructive-pred* : ‹*non-too-destructive nat.pred*›
proof (*rule non-too-destructiveI*)
  show ‹*x* ↓ *Suc n* = *y* ↓ *Suc n* ⟹ *nat.pred x* ↓ *n* = *nat.pred y* ↓ *n*›
for *x y n*
    by (*cases x*; *cases y*) (*simp-all add*: *restriction-nat-def split*: *if-split-asm*)
qed

Restriction shift plus

lemma *restriction-shift-plus* : ‹*restriction-shift* (λ*x*. *x* + *k*) (*int k*)›
proof (*intro restriction-shiftI*)
  show ‹*x* ↓ *n* = *y* ↓ *n* ⟹ *x* + *k* ↓ *nat* (*int n* + *int k*) = *y* + *k* ↓ *nat* (*int n* + *int k*)› **for** *x y n*
    by (*simp add*: *restriction-nat-def nat-int-add split*: *if-split-asm*)
qed

lemma ‹*restriction-shift* (λ*x*. *k* + *x*) (*int k*)›
  by (*simp add*: *add.commute restriction-shift-plus*)

— In particular, constructive if $1 < k$.

4 Integers

instantiation *int* :: *restriction*
begin

definition *restriction-int* :: ‹*int* ⇒ *nat* ⇒ *int*›
  where ‹$x \downarrow n \equiv$ *if* $|x| \leq$ *int n then x else if* $0 \leq x$ *then int n else* − *int n*›

instance by *intro-classes* (*simp add*: *restriction-int-def min-def*)

end

instance *int* :: *restriction-space*
  by (*intro-classes*, *simp-all add*: *restriction-int-def*)
    (*metis le-eq-less-or-eq linorder-not-less nat-le-iff*)

lemma *restriction-int-0-is-0* [*simp*] : ‹$x \downarrow 0 = (0 ::$ *int*)›
  by (*simp add*: *restriction-int-def*)

Restriction shift plus

lemma *restriction-shift-on-pos-plus* : ‹*restriction-shift-on* ($\lambda x.\ x + k$) k $\{x.\ 0 \leq x\}$›
  by (*intro restriction-shift-onI*)
    (*simp add*: *restriction-int-def split*: *if-split-asm*)

lemma *restriction-shift-on-neg-minus* : ‹*restriction-shift-on* ($\lambda x.\ x - k$) k $\{x.\ x \leq 0\}$›
  by (*intro restriction-shift-onI*)
    (*simp add*: *restriction-int-def split*: *if-split-asm*)

5 Option Type

5.1 Restriction option type

instantiation *option* :: (*restriction*) *restriction*

begin

definition *restriction-option* :: ‹*'a option ⇒ nat ⇒ 'a option*›
  where ‹*x ↓ n ≡ if n = 0 then None else map-option* (λ*a. a ↓ n*) *x*›

instance
  by *intro-classes*
   (*simp add*: *restriction-option-def option.map-comp comp-def min-def*)

end

lemma *restriction-option-0-is-None* [*simp*] : ‹*x ↓ 0 = None*›
  by (*simp add*: *restriction-option-def*)

lemma *restriction-option-None* [*simp*] : ‹*None ↓ n = None*›
  by (*simp add*: *restriction-option-def*)

lemma *restriction-option-Some* [*simp*] : ‹*Some x ↓ n = (if n = 0 then None else Some* (*x ↓ n*))›
  by (*simp add*: *restriction-option-def*)

lemma *restriction-option-eq-None-iff* : ‹*x ↓ n = None* ⟷ *n = 0* ∨ *x = None*›
  by (*cases x*) *simp-all*

lemma *restriction-option-eq-Some-iff* : ‹*x ↓ n = Some y* ⟷ *n ≠ 0* ∧ *x ≠ None* ∧ *y = the x ↓ n*›
  by (*cases x*) *auto*

5.2 Restriction space option type

instance *option* :: (*restriction-space*) *restriction-space*
proof *intro-classes*
  show ‹*x ↓ 0 = y ↓ 0*› **for** *x y* :: ‹*'a option*› **by** *simp*
next
  show ‹*x ≠ y* ⟹ ∃ *n. x ↓ n ≠ y ↓ n*› **for** *x y* :: ‹*'a option*›
    by (*cases x*; *cases y*, *simp-all add*: *gt-ex*)
     (*metis bot-nat-0.not-eq-extremum ex-not-restriction-related restriction-0-related*)
qed

5.3 Complete restriction space option type

lemma *option-restriction-chainE* :
  fixes *σ* :: ‹*nat ⇒ 'a* :: *restriction-space option*› **assumes** ‹*chain*$_{\downarrow}$ *σ*›
  obtains ‹*σ* = (λ*n. None*)›
  | *σ'* **where** ‹*chain*$_{\downarrow}$ *σ'*› **and** ‹*σ* = (λ*n. if n = 0 then None else Some* (*σ' n*))›
proof −
  from ‹*chain*$_{\downarrow}$ *σ*› **consider** ‹∀ *n. σ n = None*› | ‹∀ *n*>*0. σ n ≠ None*›

```
    by (metis bot-nat-0.not-eq-extremum linorder-neqE-nat
        restriction-chain-def-bis restriction-option-eq-None-iff)
  thus thesis
  proof cases
    from that(1) show ‹∀ n. σ n = None ⟹ thesis› by fast
  next
    define σ' where ‹σ' n ≡ if n = 0 then undefined ↓ 0 else the (σ
n)› for n
    assume ‹∀ n>0. σ n ≠ None›
   with ‹chain↓ σ› have ‹chain↓ σ'› ‹σ = (λn. if n = 0 then undefined
↓ 0 else Some (σ' n))›
      by (simp-all add: σ'-def restriction-chain-def)
        (metis option.sel restriction-option-eq-Some-iff,
             metis σ'-def bot-nat-0.not-eq-extremum option.sel restric-
tion-option-0-is-None)
    with that(2) show thesis by force
  qed
qed


lemma non-destructive-Some : ‹non-destructive Some›
  by (simp add: non-destructiveI)

lemma restriction-cont-Some : ‹cont↓ (Some :: 'a :: restriction-space
⇒ 'a option)›
  by (rule restriction-shift-imp-restriction-cont[where k = 0])
    (simp add: restriction-shiftI)


instance option :: (complete-restriction-space) complete-restriction-space
proof intro-classes
  show ‹chain↓ σ ⟹ convergent↓ σ› for σ :: ‹nat ⇒ 'a option›
  proof (elim option-restriction-chainE)
    show ‹σ = (λn. None) ⟹ convergent↓ σ› by simp
  next
    fix σ' assume ‹chain↓ σ'› and σ-def : ‹σ = (λn. if n = 0 then
None else Some (σ' n))›
     from ‹chain↓ σ'› have ‹convergent↓ σ'› by (simp add: restric-
tion-chain-imp-restriction-convergent)
   hence ‹convergent↓ (λn. σ' (n + 1))› by (unfold restriction-convergent-shift-iff)
    then obtain Σ' where ‹(λn. σ' (n + 1)) −↓→ Σ'› by (blast dest:
restriction-convergentD')
    hence ‹(λn. Some (σ' (n + 1))) −↓→ Some Σ'› by (fact restric-
tion-contD[OF restriction-cont-Some])
     hence ‹convergent↓ (λn. Some (σ' (n + 1)))› by (blast intro:
restriction-convergentI)
    hence ‹convergent↓ (λn. σ (n + 1))› by (simp add: σ-def)
   thus ‹convergent↓ σ› using restriction-convergent-shift-iff by blast
  qed
```

qed

6 Lists

List is a restriction space using *take* as the restriction function

instantiation *list* :: (*type*) *restriction*
begin

definition *restriction-list* :: ‹*'a list ⇒ nat ⇒ 'a list*›
  where ‹*L ↓ n ≡ take n L*›

instance by *intro-classes* (*simp add*: *restriction-list-def min.commute*)

end

instance *list* :: (*type*) *order-restriction-space*
proof *intro-classes*
  show ‹*L ↓ 0 ≤ M ↓ 0*› **for** *L M* :: ‹*'a list*›
    by (*simp add*: *restriction-list-def*)
next
  show ‹*L ≤ M ⟹ L ↓ n ≤ M ↓ n*› **for** *L M* :: ‹*'a list*› **and** *n*
    unfolding *restriction-list-def*
    by (*metis less-eq-list-def prefix-def take-append*)
next
  show ‹*¬ L ≤ M ⟹ ∃ n. ¬ L ↓ n ≤ M ↓ n*› **for** *M L* :: ‹*'a list*›
    unfolding *restriction-list-def*
    by (*metis linorder-linear take-all-iff*)
qed

lemma ‹*OFCLASS*(*'a list*, *restriction-space-class*)› **..**

Of course, this space is not complete. We prove this with by exhibiting a counter-example.

notepad begin
  define *σ* :: ‹*nat ⇒ 'a list*›
    where ‹*σ n* = *replicate n undefined*› **for** *n*

  have ‹*chain*$_{\downarrow}$ *σ*›
    by (*intro restriction-chainI ext*)
      (*simp add*: *σ-def restriction-list-def flip*: *replicate-append-same*)

  hence ‹*∄ Σ. σ −↓→ Σ*›
   by (*metis σ-def convergent-restriction-chain-imp-ex1 length-replicate*
      lessI nat-less-le restriction-convergentI restriction-list-def take-all)

end

7 Binary Trees

datatype *'a ex-tree = tip | node ‹'a ex-tree› 'a ‹'a ex-tree›*

instantiation *ex-tree* :: (*type*) *restriction*
begin

fun *restriction-ex-tree* :: *‹'a ex-tree ⇒ nat ⇒ 'a ex-tree›*
  where *‹tip ↓ n = tip›*
  | *‹(node l val r) ↓ 0 = tip›*
  | *‹(node l val r) ↓ Suc n = node (l ↓ n) val (r ↓ n)›*

lemma *restriction-ex-tree-0-is-tip* [*simp*] : *‹T ↓ 0 = tip›*
  using *restriction-ex-tree.elims* **by** *blast*

instance
proof *intro-classes*
  show *‹T ↓ n ↓ m = T ↓ min n m›* **for** *T* :: *‹'a ex-tree›* **and** *n m*
  proof (*induct n arbitrary*: *T m*)
    show *‹T ↓ 0 ↓ m = T ↓ min 0 m›* **for** *T* :: *‹'a ex-tree›* **and** *m* **by** *simp*
  next
    fix *T* :: *‹'a ex-tree›* **and** *m n* **assume** *hyp* : *‹T ↓ n ↓ m = T ↓ min n m›* **for** *T* :: *‹'a ex-tree›* **and** *m*
    show *‹T ↓ Suc n ↓ m = T ↓ min (Suc n) m›*
      by (*cases T*; *cases m*, *simp-all add*: *hyp*)
  qed
qed

end

lemma *size-le-imp-restriction-ex-tree-eq-self* :
  ‹size x ≤ n ⟹ x ↓ n = x› **for** *x* :: *‹'a ex-tree›*
  by (*induct rule*: *restriction-ex-tree.induct*) *simp-all*

lemma *restriction-ex-tree-eqI* :
  ‹(⋀i. x ↓ i = y ↓ i) ⟹ x = y› **for** *x y* :: *‹'a ex-tree›*
  by (*metis linorder-linear size-le-imp-restriction-ex-tree-eq-self*)

lemma *restriction-ex-tree-eqI-optimized* :
‹($\bigwedge i.\ i \leq max\ (size\ x)\ (size\ y) \Longrightarrow x \downarrow i = y \downarrow i) \Longrightarrow x = y$› **for** x y :: ‹$'a$ *ex-tree*›
by (*metis max.cobounded1 max.cobounded2 order-eq-refl size-le-imp-restriction-ex-tree-eq-self*)

instance *ex-tree* :: (*type*) *restriction-space*
by (*intro-classes*, *simp*)
(*use restriction-ex-tree-eqI-optimized* **in** *blast*)

8 Decimals of a Number

typedef (**overloaded**) $'a$ :: *zero decimals* = ‹{σ :: *nat* ⇒ $'a$. σ *0* = *0*}›
morphisms *from-decimals to-decimals* **by** *auto*

setup-lifting *type-definition-decimals*

declare *from-decimals* [*simp*] *to-decimals-cases*[*simp*]
to-decimals-inject[*simp*] *to-decimals-inverse* [*simp*]

declare *from-decimals-inject* [*simp*]
from-decimals-inverse [*simp*]

lemmas *to-decimals-inject-simplified* [*simp*] = *to-decimals-inject* [*simplified*]
and *to-decimals-inverse-simplified*[*simp*] = *to-decimals-inverse*[*simplified*]

lemmas *to-decimals-induct-simplified* = *to-decimals-induct*[*simplified*]
and *to-decimals-cases-simplified* = *to-decimals-cases* [*simplified*]
and *from-decimals-induct-simplified* = *from-decimals-induct*[*simplified*]
and *from-decimals-cases-simplified* = *from-decimals-cases* [*simplified*]

instantiation *decimals* :: (*zero*) *restriction*
begin

lift-definition *restriction-decimals* :: ‹$'a$ *decimals* ⇒ *nat* ⇒ $'a$ *decimals*›
is ‹$\lambda\sigma\ m\ n.$ *if* $n \leq m$ *then* $\sigma\ n$ *else 0*› **by** *simp*

instance by (*intro-classes*, *transfer*, *rule ext*, *simp*)

end

instance *decimals* :: (*zero*) *restriction-space*
by (*intro-classes*; *transfer*, *auto*)
(*metis* (*no-types*, *lifting*) *ext order-refl*)

lemma *restriction-decimals-eq-iff* :
‹$x \downarrow n = y \downarrow n \longleftrightarrow (\forall i{\leq}n.$ *from-decimals* $x\ i =$ *from-decimals* $y\ i)$›
by *transfer meson*

lemma *restriction-decimals-eqI* :
‹$(\bigwedge i.\ i \leq n \Longrightarrow$ *from-decimals* $x\ i =$ *from-decimals* $y\ i) \Longrightarrow x \downarrow n = y \downarrow n$›
by (*simp add*: *restriction-decimals-eq-iff*)

lemma *restriction-decimals-eqD* :
‹$x \downarrow n = y \downarrow n \Longrightarrow i \leq n \Longrightarrow$ *from-decimals* $x\ i =$ *from-decimals* $y\ i$›
by (*simp add*: *restriction-decimals-eq-iff*)

This space is actually complete.

instance *decimals* :: (*zero*) *complete-restriction-space*
proof (*intro-classes*, *rule restriction-convergentI*)
fix σ :: ‹*nat* ⇒ *'a decimals*› **assume** ‹$chain_{\downarrow}\ \sigma$›
let *?Σ* = ‹*to-decimals* (λ*n*. *from-decimals* (σ *n*) *n*)›
have ‹*?Σ* $\downarrow n = \sigma\ n$› **for** *n*
proof (*subst restricted-restriction-chain-is*[*OF* ‹$chain_{\downarrow}\ \sigma$›, *symmetric*],
rule restriction-decimals-eqI)
fix *i* **assume** ‹$i \leq n$›
from *restriction-chain-def-ter*
[*THEN iffD1*, *OF* ‹*restriction-chain* σ›, *rule-format*, *OF* ‹$i \leq n$›]
show ‹*from-decimals ?Σ i* = *from-decimals* (σ *n*) *i*›
by (*subst to-decimals-inverse-simplified*, *use from-decimals* **in** *blast*)
(*metis dual-order.refl restriction-decimals.rep-eq*)
qed
thus ‹*restriction-chain* $\sigma \Longrightarrow \sigma \multimap\!\!\downarrow\!\!\to$ *?Σ*›
proof −
have ‹(↓) (*to-decimals* (λ*n*. *from-decimals* (σ *n*) *n*)) = σ›
using ‹$\bigwedge n.$ *to-decimals* (λ*n*. *from-decimals* (σ *n*) *n*) $\downarrow n = \sigma\ n$›
by *force*
then show *?thesis*
by (*metis restriction-tendsto-restrictions*)
qed
qed

typedef *nat-0-9* = ‹{*0*.. *9*::*nat*}›
  morphisms *from-nat-0-9 to-nat-0-9* **by** *auto*

setup-lifting *type-definition-nat-0-9*

instantiation *nat-0-9* :: *zero*
begin

lift-definition *zero-nat-0-9* :: *nat-0-9* **is** *0* **by** *simp*

instance ..

end

instantiation *nat-0-9* :: *one*
begin

lift-definition *one-nat-0-9* :: *nat-0-9* **is** *1* **by** *simp*

instance ..

end

lift-definition *update-nth-decimal* :: ‹[*nat-0-9 decimals*, *nat*, *nat*] ⇒ *nat-0-9 decimals*›
  is ‹λ*s index value*. *if index* = *0* ∨ *9* < *value then from-decimals s*
    else (*from-decimals s*)(*index* := *to-nat-0-9 value*)›
  using *from-decimals* **by** *auto*

lemma *no-update-nth-decimal* [*simp*] :
  ‹*index* = *0* ⟹ *update-nth-decimal s index val* = *s*›
  ‹*9* < *val* ⟹ *update-nth-decimal s index val* = *s*›
  by (*simp-all add*: *update-nth-decimal.abs-eq*)

lemma *non-destructive-update-nth-decimal* : ‹*non-destructive update-nth-decimal*›
proof (*rule non-destructiveI*)
  show ‹*update-nth-decimal x* ↓ *n* = *update-nth-decimal y* ↓ *n*› **if** ‹*x* ↓ *n* = *y* ↓ *n*› **for** *x y n*
  proof (*unfold restriction-fun-def*, *intro ext restriction-decimals-eqI*)
    fix *index val i* **assume** ‹*i* ≤ *n*›

```
    from restriction-decimals-eqD[OF ‹x ↓ n = y ↓ n› ‹i ≤ n›]
    show ‹from-decimals (update-nth-decimal x index val) i =
          from-decimals (update-nth-decimal y index val) i›
      by (simp add: update-nth-decimal.rep-eq)
  qed
qed


lift-definition shift-decimal-right :: ‹nat-0-9 decimals ⇒ nat-0-9 dec-
imals›
  is ‹λs n. case n of 0 ⇒ to-nat-0-9 0 | Suc n' ⇒ from-decimals s n'›
  by (simp add: zero-nat-0-9-def)


lemma constructive-shift-decimal-right : ‹constructive shift-decimal-right›
proof (rule constructiveI)
  show ‹shift-decimal-right x ↓ Suc n = shift-decimal-right y ↓ Suc n›
if ‹x ↓ n = y ↓ n› for x y n
  proof (intro restriction-decimals-eqI)
    fix index val i assume ‹i ≤ Suc n›
    hence ‹i − 1 ≤ n› by simp
    from restriction-decimals-eqD[OF ‹x ↓ n = y ↓ n› ‹i − 1 ≤ n›]
      show ‹from-decimals (shift-decimal-right x) i = from-decimals
(shift-decimal-right y) i›
      by (simp add: shift-decimal-right.rep-eq Nitpick.case-nat-unfold)
  qed
qed


lift-definition shift-decimal-left :: ‹nat-0-9 decimals ⇒ nat-0-9 deci-
mals›
  is ‹λs n. if n = 0 then to-nat-0-9 0 else from-decimals s (Suc n)›
  by (simp add: zero-nat-0-9-def)

lemma non-too-destructive-shift-decimal-left : ‹non-too-destructive shift-decimal-left›
proof (rule non-too-destructiveI)
  show ‹shift-decimal-left x ↓ n = shift-decimal-left y ↓ n› if ‹x ↓ Suc
n = y ↓ Suc n› for x y n
  proof (intro restriction-decimals-eqI)
    fix index val i assume ‹i ≤ n›
    hence ‹Suc i ≤ Suc n› by simp
    from restriction-decimals-eqD[OF ‹x ↓ Suc n = y ↓ Suc n› ‹Suc i
≤ Suc n›]
   show ‹from-decimals (shift-decimal-left x) i = from-decimals (shift-decimal-left
y) i›
      by (simp add: shift-decimal-left.rep-eq)
  qed
qed
```

lemma *restriction-fix-shift-decimal-right* : ‹(*υ x. shift-decimal-right x*) = *to-decimals* (*λ-. 0*)›
proof (*rule restriction-fix-unique*)
  show ‹*constructive shift-decimal-right*›
    by (*fact constructive-shift-decimal-right*)
next
  show ‹*shift-decimal-right* (*to-decimals* (*λ-. 0*)) = *to-decimals* (*λ-. 0*)›
    by (*simp add*: *shift-decimal-right.abs-eq*)
    (*metis nat.case-eq-if to-decimals-inverse-simplified zero-nat-0-9-def*)
qed

Example of a predicate that is not admissible.

lemma *one-in-decimals-not-admissible* :
  defines *P-def*: ‹*P* ≡ *λx.* (*1* :: *nat-0-9*) ∈ *range* (*from-decimals x*)›
  shows ‹¬ *adm*$_{\downarrow}$ *P*›
proof (*rule ccontr*)
  assume ∗ : ‹¬ ¬ *adm*$_{\downarrow}$ *P*›

  define *x* **where** ‹*x* ≡ *to-decimals* (*λn. if n = 0 then 0 else 1* :: *nat-0-9*)›

  have ‹*P* (*υ x. shift-decimal-right x*)›
  proof (*induct rule*: *restriction-fix-ind*)
   show ‹*constructive shift-decimal-right*› **by** (*fact constructive-shift-decimal-right*)
  next
    from ∗ **show** ‹*adm*$_{\downarrow}$ *P*› **by** *simp*
  next
    show ‹*P x*› **by** (*auto simp add*: *P-def x-def image-iff*)
  next
    show ‹*P x* ⟹ *P* (*shift-decimal-right x*)› **for** *x*
      by (*simp add*: *P-def image-def shift-decimal-right.rep-eq*)
        (*metis old.nat.simps*(*5*))
  qed
  moreover have ‹¬ *P* (*υ x. shift-decimal-right x*)›
    by (*simp add*: *P-def restriction-fix-shift-decimal-right*)
      (*transfer*, *simp*)
  ultimately show *False* **by** *blast*
qed

9 Trace Model of CSP

In the AFP one can already find `HOL-CSP`, a shallow embedding of the failure-divergence model of denotational semantics proposed by Hoare, Roscoe and Brookes in the eighties. Here, we

simplify the example by restraining ourselves to a trace model.

9.1 Prerequisites

datatype *'a event* = *ev* (*of-ev* : *'a*) | *tick* (‹✓›)

type-synonym *'a trace* = ‹*'a event list*›

definition *tickFree* :: ‹*'a trace* ⇒ *bool*› (‹*tF*›)
where ‹*tickFree t* ≡ ✓ ∉ *set t*›

definition *front-tickFree* :: ‹*'a trace* ⇒ *bool*› (‹*ftF*›)
where ‹*front-tickFree s* ≡ *s* = [] ∨ *tickFree* (*tl* (*rev s*))›

lemma *tickFree-Nil* [*simp*] : ‹*tF* []›
and *tickFree-Cons-iff* [*simp*] : ‹*tF* (*a* # *t*) ⟷ *a* ≠ ✓ ∧ *tF t*›
and *tickFree-append-iff* [*simp*] : ‹*tF* (*s* @ *t*) ⟷ *tF s* ∧ *tF t*›
and *tickFree-rev-iff* [*simp*] : ‹*tF* (*rev t*) ⟷ *tF t*›
and *non-tickFree-tick* [*simp*] : ‹¬ *tF* [✓]›
by (*auto simp add*: *tickFree-def*)

lemma *tickFree-iff-is-map-ev* : ‹*tF t* ⟷ (∃ *u*. *t* = *map ev u*)›
by (*metis event.collapse event.distinct*(*1*) *ex-map-conv tickFree-def*)

lemma *front-tickFree-Nil* [*simp*] : ‹*ftF* []›
and *front-tickFree-single*[*simp*] : ‹*ftF* [*a*]›
by (*simp-all add*: *front-tickFree-def*)

lemma *tickFree-tl* : ‹*tF s* ⟹ *tF* (*tl s*)›
by (*cases s*) *simp-all*

lemma *non-tickFree-imp-not-Nil*: ‹¬ *tF s* ⟹ *s* ≠ []›
using *tickFree-Nil* **by** *blast*

lemma *tickFree-butlast*: ‹*tF s* ⟷ *tF* (*butlast s*) ∧ (*s* ≠ [] ⟶ *last s* ≠ ✓)›
by (*induct s*) *simp-all*

lemma *front-tickFree-iff-tickFree-butlast*: ‹*ftF s* ⟷ *tF* (*butlast s*)›
by (*induct s*) (*auto simp add*: *front-tickFree-def*)

lemma *front-tickFree-Cons-iff*: ‹*ftF* (*a* # *s*) ⟷ *s* = [] ∨ *a* ≠ ✓ ∧ *ftF s*›
by (*simp add*: *front-tickFree-iff-tickFree-butlast*)

lemma *front-tickFree-append-iff*:

‹*ftF* (*s* @ *t*) ⟷ (*if t* = [] *then ftF s else tF s* ∧ *ftF t*)›
by (*simp add*: *butlast-append front-tickFree-iff-tickFree-butlast*)

lemma *tickFree-imp-front-tickFree* [*simp*] : ‹*tF s* ⟹ *ftF s*›
by (*simp add*: *front-tickFree-def tickFree-tl*)

lemma *front-tickFree-charn*: ‹*ftF s* ⟷ *s* = [] ∨ (∃ *a t*. *s* = *t* @ [*a*] ∧ *tF t*)›
by (*cases s rule*: *rev-cases*) (*simp-all add*: *front-tickFree-def*)

lemma *nonTickFree-n-frontTickFree*: ‹¬ *tF s* ⟹ *ftF s* ⟹ ∃ *t r*. *s* = *t* @ [✓]›
by (*metis front-tickFree-charn tickFree-Cons-iff*
tickFree-Nil tickFree-append-iff)

lemma *front-tickFree-dw-closed* : ‹*ftF* (*s* @ *t*) ⟹ *ftF s*›
by (*metis front-tickFree-append-iff tickFree-imp-front-tickFree*)

lemma *front-tickFree-append*: ‹*tF s* ⟹ *ftF t* ⟹ *ftF* (*s* @ *t*)›
by (*simp add*: *front-tickFree-append-iff*)

lemma *tickFree-imp-front-tickFree-snoc*: ‹*tF s* ⟹ *ftF* (*s* @ [*a*])›
by (*simp add*: *front-tickFree-append*)

lemma *front-tickFree-nonempty-append-imp*: ‹*ftF* (*t* @ *r*) ⟹ *r* ≠ [] ⟹ *tF t* ∧ *ftF r*›
by (*simp add*: *front-tickFree-append-iff*)

lemma *tickFree-map-ev* [*simp*] : ‹*tF* (*map ev t*)›
by (*induct t*) *simp-all*

lemma *tickFree-map-ev-comp* [*simp*] : ‹*tF* (*map* (*ev* ∘ *f*) *t*)›
by (*metis list.map-comp tickFree-map-ev*)

lemma *front-tickFree-map-map-event-iff* :
‹*ftF* (*map* (*map-event f*) *t*) ⟷ *ftF t*›
by (*induct t*) (*simp-all add*: *front-tickFree-Cons-iff*)

definition *is-process* :: ‹'*a trace set* ⇒ *bool*›
where ‹*is-process T* ≡ [] ∈ *T* ∧ (∀ *t*. *t* ∈ *T* ⟶ *ftF t*) ∧ (∀ *t u*. *t* @ *u* ∈ *T* ⟶ *t* ∈ *T*)›

typedef '*a process* = ‹{ *T* :: '*a trace set*. *is-process T*}›
morphisms *Traces to-process*
proof (*rule exI*)

show ‹{[]} ∈ {*T*. *is-process T*}›
by (*simp add*: *is-process-def front-tickFree-def*)
qed

setup-lifting *type-definition-process*

notation *Traces* (‹$\mathcal{T}$›)

lemma *is-process-inv* :
‹[] ∈ $\mathcal{T}$ *P* ∧ (∀ *t*. *t* ∈ $\mathcal{T}$ *P* ⟶ *ftF t*) ∧ (∀ *t u*. *t* @ *u* ∈ $\mathcal{T}$ *P* ⟶ *t* ∈ $\mathcal{T}$ *P*)›
by (*metis is-process-def mem-Collect-eq to-process-cases to-process-inverse*)

lemma *Nil-elem-T* : ‹[] ∈ $\mathcal{T}$ *P*›
and *front-tickFree-T* : ‹*t* ∈ $\mathcal{T}$ *P* ⟹ *ftF t*›
and *T-dw-closed* : ‹*t* @ *u* ∈ $\mathcal{T}$ *P* ⟹ *t* ∈ $\mathcal{T}$ *P*›
by (*simp-all add*: *is-process-inv*)

lemma *process-eq-spec* : ‹*P* = *Q* ⟷ $\mathcal{T}$ *P* = $\mathcal{T}$ *Q*›
by *transfer simp*

9.2 First Processes

lift-definition *BOT* :: ‹'*a process*› **is** ‹{*t*. *ftF t*}›
by (*auto simp add*: *is-process-def front-tickFree-append-iff*)

lemma *T-BOT* : ‹$\mathcal{T}$ *BOT* = {*t*. *ftF t*}›
by (*simp add*: *BOT.rep-eq*)

lift-definition *SKIP* :: ‹'*a process*› **is** ‹{[], [✓]}›
by (*simp add*: *is-process-def append-eq-Cons-conv*)

lemma *T-SKIP* : ‹$\mathcal{T}$ *SKIP* = {[], [✓]}›
by (*simp add*: *SKIP.rep-eq*)

lift-definition *STOP* :: ‹'*a process*› **is** ‹{[]}›
by (*simp add*: *is-process-def*)

lemma *T-STOP* : ‹$\mathcal{T}$ *STOP* = {[]}›
by (*simp add*: *STOP.rep-eq*)

lift-definition *Sup-processes* ::
‹(*nat* ⇒ '*a process*) ⇒ '*a process*› **is** ‹λσ. ⋂ *i*. $\mathcal{T}$ (σ *i*)›
proof −

show ‹*is-process* $(\bigcap i.\ \mathcal{T}\ (\sigma\ i))$› **for** σ :: ‹*nat* ⇒ *'a process*›
proof (*unfold is-process-def*, *intro conjI allI impI*)
show ‹[] ∈ $(\bigcap i.\ \mathcal{T}\ (\sigma\ i))$› **by** (*simp add*: *Nil-elem-T*)
next
show ‹$t \in (\bigcap i.\ \mathcal{T}\ (\sigma\ i)) \Longrightarrow$ *ftF t*› **for** *t*
by (*auto intro*: *front-tickFree-T*)
next
show ‹t @ $u \in (\bigcap i.\ \mathcal{T}\ (\sigma\ i)) \Longrightarrow t \in (\bigcap i.\ \mathcal{T}\ (\sigma\ i))$› **for** *t u*
by (*auto intro*: *T-dw-closed*)
qed
qed

lemma *T-Sup-processes* : ‹$\mathcal{T}$ (*Sup-processes* σ) = $(\bigcap i.\ \mathcal{T}\ (\sigma\ i))$›
by (*simp add*: *Sup-processes.rep-eq*)

9.3 Instantiations

instantiation *process* :: (*type*) *order*
begin

definition *less-eq-process* :: ‹*'a process* ⇒ *'a process* ⇒ *bool*›
where ‹$P \leq Q \equiv \mathcal{T}\ Q \subseteq \mathcal{T}\ P$›

definition *less-process* :: ‹*'a process* ⇒ *'a process* ⇒ *bool*›
where ‹$P < Q \equiv \mathcal{T}\ Q \subset \mathcal{T}\ P$›

instance
by *intro-classes*
(*auto simp add*: *less-eq-process-def less-process-def process-eq-spec*)

end

instantiation *process* :: (*type*) *order-restriction-space*
begin

lift-definition *restriction-process* :: ‹*'a process* ⇒ *nat* ⇒ *'a process*›
is ‹$\lambda P\ n.\ \mathcal{T}\ P \cup \{t$ @ $u\ |t\ u.\ t \in \mathcal{T}\ P \wedge$ *length* $t = n \wedge$ *tF* $t \wedge$ *ftF* $u\}$›
proof −
show ‹*?thesis P n*› (**is** ‹*is-process ?t*›) **for** *P n*
proof (*unfold is-process-def*, *intro conjI allI impI*)
show ‹[] ∈ *?t*› **by** (*simp add*: *Nil-elem-T*)
next
show ‹$t \in$ *?t* ⟹ *ftF t*› **for** *t*
by (*auto simp add*: *front-tickFree-append-iff front-tickFree-T*)
next
fix *t u* **assume** ‹t @ $u \in$ *?t*›
then consider ‹t @ $u \in \mathcal{T}\ P$›

```
    | (@) t' u' where ‹t @ u = t' @ u'› ‹t' ∈ 𝒯 P›
      ‹length t' = n› ‹tF t'› ‹ftF u'› by blast
  thus ‹t ∈ ?t›
  proof cases
    from T-dw-closed show ‹t @ u ∈ 𝒯 P ⟹ t ∈ ?t› by blast
  next
    case @
    show ‹t ∈ ?t›
    proof (cases ‹length t ≤ length t'›)
      assume ‹length t ≤ length t'›
      with @(1) obtain t'' where ‹t' = t @ t''›
        by (metis append-eq-append-conv-if append-eq-conv-conj)
      with @(2) T-dw-closed show ‹t ∈ ?t› by blast
    next
      assume ‹¬ length t ≤ length t'›
      hence ‹length t' ≤ length t› by simp
      with @(1,4,5) ‹¬ length t ≤ length t'›
      obtain t'' where ‹t = t' @ t''› ‹ftF t''›
      by (metis append-eq-conv-conj drop-eq-Nil front-tickFree-append
          front-tickFree-dw-closed front-tickFree-nonempty-append-imp
            take-all-iff take-append)
      with @(2,3,4) show ‹t ∈ ?t› by blast
    qed
  qed
 qed
qed

lemma T-restriction-process :
 ‹𝒯 (P ↓ n) = 𝒯 P ∪ {t @ u |t u. t ∈ 𝒯 P ∧ length t = n ∧ tF t ∧
ftF u}›
 by (simp add: restriction-process.rep-eq)


lemma restriction-process-0 [simp] : ‹P ↓ 0 = BOT›
 by transfer (auto simp add: front-tickFree-T Nil-elem-T)

lemma T-restriction-processE :
 ‹t ∈ 𝒯 (P ↓ n) ⟹
  (t ∈ 𝒯 P ⟹ length t ≤ n ⟹ thesis) ⟹
  (⋀u v. t = u @ v ⟹ u ∈ 𝒯 P ⟹ length u = n ⟹ tF u ⟹ ftF
v ⟹ thesis) ⟹
  thesis›
 by (simp add: T-restriction-process)
   (metis (no-types) T-dw-closed append-take-drop-id drop-eq-Nil
     front-tickFree-T front-tickFree-nonempty-append-imp length-take
min-def)


instance
```

proof *intro-classes*
show ‹$P \downarrow 0 \leq Q \downarrow 0$› **for** $P\ Q$ :: ‹'a *process*› **by** *simp*
next
show ‹$P \downarrow n \downarrow m = P \downarrow min\ n\ m$› **for** P :: ‹'a *process*› **and** $n\ m$
proof (*subst process-eq-spec*, *intro subset-antisym subsetI*)
show ‹$t \in \mathcal{T}\ (P \downarrow n \downarrow m) \Longrightarrow t \in \mathcal{T}\ (P \downarrow min\ n\ m)$› **for** t
by (*elim T-restriction-processE*)
(*auto simp add*: *T-restriction-process intro*: *front-tickFree-append*)
next
show ‹$t \in \mathcal{T}\ (P \downarrow min\ n\ m) \Longrightarrow t \in \mathcal{T}\ (P \downarrow n \downarrow m)$› **for** t
by (*elim T-restriction-processE*)
(*auto simp add*: *T-restriction-process min-def split*: *if-split-asm*)
qed
next
show ‹$P \leq Q \Longrightarrow P \downarrow n \leq Q \downarrow n$› **for** $P\ Q$ :: ‹'a *process*› **and** n
by (*auto simp add*: *less-eq-process-def T-restriction-process*)
next
fix $P\ Q$ :: ‹'a *process*› **assume** ‹$\neg\ P \leq Q$›
then obtain t **where** ‹$t \in \mathcal{T}\ Q$› ‹$t \notin \mathcal{T}\ P$›
unfolding *less-eq-process-def* **by** *blast*
hence ‹$t \in \mathcal{T}\ (Q \downarrow Suc\ (length\ t)) \wedge t \notin \mathcal{T}\ (P \downarrow Suc\ (length\ t))$›
by (*auto simp add*: *T-restriction-process*)
hence ‹$\neg\ P \downarrow Suc\ (length\ t) \leq Q \downarrow Suc\ (length\ t)$›
unfolding *less-eq-process-def* **by** *blast*
thus ‹$\exists\, n.\ \neg\ P \downarrow n \leq Q \downarrow n$› **..**
qed

Of course, we recover the structure of *restriction-space*.

lemma ‹*OFCLASS* ('a *process*, *restriction-space-class*)›
by *intro-classes*

end

lemma *restricted-Sup-processes-is* :
‹$(\lambda n.\ Sup\text{-}processes\ \sigma \downarrow n) = \sigma$› **if** ‹*restriction-chain* σ›
proof (*subst* (*2*) *restricted-restriction-chain-is*
[*OF* ‹*restriction-chain* σ›, *symmetric*], *rule ext*)
fix n
have ‹$length\ t \leq n \Longrightarrow t \in \mathcal{T}\ (\sigma\ n) \longleftrightarrow (\forall\, i.\ t \in \mathcal{T}\ (\sigma\ i))$› **for** $t\ n$
proof *safe*
show ‹$t \in \mathcal{T}\ (\sigma\ i)$› **if** ‹$length\ t \leq n$› ‹$t \in \mathcal{T}\ (\sigma\ n)$› **for** i
proof (*cases* ‹$i \leq n$›)
from *restriction-chain-def-ter*[*THEN iffD1*, *OF* ‹*restriction-chain* σ›]
show ‹$i \leq n \Longrightarrow t \in \mathcal{T}\ (\sigma\ i)$›
by (*metis* (*lifting*) ‹$t \in \mathcal{T}\ (\sigma\ n)$› *T-restriction-process Un-iff*)
next
from ‹$length\ t \leq n$› ‹$t \in \mathcal{T}\ (\sigma\ n)$› **show** ‹$\neg\ i \leq n \Longrightarrow t \in \mathcal{T}\ (\sigma$

```
i)›
      by (induct n, simp-all add: Nil-elem-T)
       (metis (no-types) T-restriction-processE
         append-eq-conv-conj linorder-linear take-all-iff
         restriction-chain-def-ter[THEN iffD1, OF ‹restriction-chain
σ›])
   qed
  next
   show ‹∀ i. t ∈ 𝒯 (σ i) ⟹ t ∈ 𝒯 (σ n)› by simp
  qed
  hence ∗ : ‹length t ≤ n ⟹ t ∈ 𝒯 (σ n) ⟷ t ∈ 𝒯 (Sup-processes
σ)› for t n
   by (simp add: T-Sup-processes)

  show ‹Sup-processes σ ↓ n = σ n ↓ n› for n
  proof (subst process-eq-spec, intro subset-antisym subsetI)
   show ‹t ∈ 𝒯 (Sup-processes σ ↓ n) ⟹ t ∈ 𝒯 (σ n ↓ n)› for t
   proof (elim T-restriction-processE)
    show ‹t ∈ 𝒯 (Sup-processes σ) ⟹ length t ≤ n ⟹ t ∈ 𝒯 (σ n
↓ n)›
      by (simp add: ∗ T-restriction-process)
   next
    show ‹⟦t = u @ v; u ∈ 𝒯 (Sup-processes σ); length u = n; tF u;
ftF v⟧
        ⟹ t ∈ 𝒯 (σ n ↓ n)› for u v
      by (auto simp add: ∗ T-restriction-process)
   qed
  next
   from ∗ show ‹t ∈ 𝒯 (σ n ↓ n) ⟹ t ∈ 𝒯 (Sup-processes σ ↓ n)›
for t
    by (elim T-restriction-processE)
     (auto simp add: T-restriction-process)
  qed
qed


instance process :: (type) complete-restriction-space
proof intro-classes
  show ‹restriction-convergent σ› if ‹restriction-chain σ›
   for σ :: ‹nat ⇒ 'a process›
  proof (rule restriction-convergentI)
   have ‹σ = (λn. Sup-processes σ ↓ n)›
    by (simp add: restricted-Sup-processes-is ‹restriction-chain σ›)
   moreover have ‹(λn. Sup-processes σ ↓ n) −↓→ Sup-processes σ›
    by (fact restriction-tendsto-restrictions)
   ultimately show ‹σ −↓→ Sup-processes σ› by simp
  qed
qed
```

9.4 Operators

lift-definition *Choice* :: ‹*'a process* ⇒ *'a process* ⇒ *'a process*› (**infixl** ‹□› *82*)
is ‹*λP Q*. *T P* ∪ *T Q*›
by (*auto simp add*: *is-process-def Nil-elem-T front-tickFree-T intro*: *T-dw-closed*)

lemma *T-Choice* : ‹*T* (*P* □ *Q*) = *T P* ∪ *T Q*›
by (*simp add*: *Choice.rep-eq*)

lift-definition *GlobalChoice* :: ‹[*'b set*, *'b* ⇒ *'a process*] ⇒ *'a process*›
is ‹*λA P*. *if A* = {} *then* {[]} *else* ⋃ *a*∈*A*. *T* (*P a*)›
by (*auto simp add*: *is-process-def Nil-elem-T front-tickFree-T intro*: *T-dw-closed*)

syntax -*GlobalChoice* :: ‹[*pttrn*, *'b set*, *'a process*] ⇒ *'a process*›
(‹(*3*□((-)/∈(-))./ (-))› [*78*,*78*,*77*] *77*)
syntax-consts -*GlobalChoice* ⇌ *GlobalChoice*
translations □ *a* ∈ *A*. *P* ⇌ *CONST GlobalChoice A* (*λa*. *P*)

lemma *T-GlobalChoice* : ‹*T* (□*a* ∈ *A*. *P a*) = (*if A* = {} *then* {[]} *else* ⋃ *a*∈*A*. *T* (*P a*))›
by (*simp add*: *GlobalChoice.rep-eq*)

lift-definition *Seq* :: ‹*'a process* ⇒ *'a process* ⇒ *'a process*› (**infixl** ‹;› *74*)
is ‹*λP Q*. {*t* ∈ *T P*. *tF t*} ∪ {*t* @ *u* |*t u*. *t* @ [✓] ∈ *T P* ∧ *u* ∈ *T Q*}›
by (*auto simp add*: *is-process-def Nil-elem-T append-eq-append-conv2 intro*: *T-dw-closed*)
(*metis front-tickFree-T front-tickFree-append-iff*
front-tickFree-dw-closed not-Cons-self,
meson front-tickFree-append-iff is-process-inv snoc-eq-iff-butlast)

lemma *T-Seq* : ‹*T* (*P* ; *Q*) = {*t* ∈ *T P*. *tF t*} ∪ {*t* @ *u* |*t u*. *t* @ [✓] ∈ *T P* ∧ *u* ∈ *T Q*}›
by (*simp add*: *Seq.rep-eq*)

lift-definition *Renaming* :: ‹[*'a process*, *'a* ⇒ *'b*] ⇒ *'b process*›
is ‹*λP f*. {*map* (*map-event f*) *u* |*u*. *u* ∈ *T P*}›
by (*auto simp add*: *is-process-def Nil-elem-T front-tickFree-map-map-event-iff*
front-tickFree-T append-eq-map-conv intro: *T-dw-closed*)

lemma *T-Renaming* : ‹*T* (*Renaming P f*) = {*map* (*map-event f*) *u* |*u*. *u* ∈ *T P*}›
by (*simp add*: *Renaming.rep-eq*)

lift-definition *Mprefix* :: ‹['a *set*, 'a ⇒ 'a *process*] ⇒ 'a *process*›
  is ‹λ*A P*. *insert* [] {*ev a* # *t* |*a t*. *a* ∈ *A* ∧ *t* ∈ 𝒯 (*P a*)}›
  by (*auto simp add*: *is-process-def front-tickFree-Cons-iff*
    front-tickFree-T append-eq-Cons-conv intro: *T-dw-closed*)

syntax -*Mprefix* :: ‹[*pttrn*, 'a *set*, 'a *process*] ⇒ 'a *process*›
  (‹(3□((-)/∈(-))/ → (-))› [78,78,77] 77)
syntax-consts -*Mprefix* ⇌ *Mprefix*
translations □*a*∈*A* → *P* ⇌ *CONST Mprefix A* (λ*a*. *P*)

lemma *T-Mprefix* : ‹𝒯 (□*a* ∈ *A* → *P a*) = *insert* [] {*ev a* # *t* |*a t*. *a* ∈ *A* ∧ *t* ∈ 𝒯 (*P a*)}›
  by (*simp add*: *Mprefix.rep-eq*)

fun *setinterleaving* :: ‹'a *trace* × 'a *set* × 'a *trace* ⇒ 'a *trace set*›
  where *Nil-setinterleaving-Nil* : ‹*setinterleaving* ([], *A*, []) = {[]}›

| *ev-setinterleaving-Nil* :
  ‹*setinterleaving* (*ev a* # *u*, *A*, []) =
      (*if a* ∈ *A then* {} *else* {*ev a* # *t*| *t*. *t* ∈ *setinterleaving* (*u*, *A*, [])})›
| *tick-setinterleaving-Nil* : ‹*setinterleaving* (✓ # *u*, *A*, []) = {}›

| *Nil-setinterleaving-ev* :
  ‹*setinterleaving* ([], *A*, *ev b* # *v*) =
      (*if b* ∈ *A then* {} *else* {*ev b* # *t*| *t*. *t* ∈ *setinterleaving* ([], *A*, *v*)})›
| *Nil-setinterleaving-tick* : ‹*setinterleaving* ([], *A*, ✓ # *v*) = {}›

| *ev-setinterleaving-ev* :
  ‹*setinterleaving* (*ev a* # *u*, *A*, *ev b* # *v*) =
      (*if a* ∈ *A*
       then *if b* ∈ *A*
            then *if a* = *b*
                 then {*ev a* # *t* |*t*. *t* ∈ *setinterleaving* (*u*, *A*, *v*)}
                 else {}
            else {*ev b* # *t* |*t*. *t* ∈ *setinterleaving* (*ev a* # *u*, *A*, *v*)}
        else *if b* ∈ *A then* {*ev a* # *t* |*t*. *t* ∈ *setinterleaving* (*u*, *A*, *ev b* # *v*)}
            else {*ev a* # *t* |*t*. *t* ∈ *setinterleaving* (*u*, *A*, *ev b* # *v*)} ∪
                 {*ev b* # *t* |*t*. *t* ∈ *setinterleaving* (*ev a* # *u*, *A*, *v*)})›
| *ev-setinterleaving-tick* :

‹*setinterleaving* (*ev a* # *u*, *A*, ✓ # *v*) =
(*if a* ∈ *A then* {} *else* {*ev a* # *t* |*t*. *t* ∈ *setinterleaving* (*u*, *A*, ✓ # *v*)})›
| *tick-setinterleaving-ev* :
‹*setinterleaving* (✓ # *u*, *A*, *ev b* # *v*) =
(*if b* ∈ *A then* {} *else* {*ev b* # *t* |*t*. *t* ∈ *setinterleaving* (✓ # *u*, *A*, *v*)})›
| *tick-setinterleaving-tick* :
‹*setinterleaving* (✓ # *u*, *A*, ✓ # *v*) = {✓ # *t* |*t*. *t* ∈ *setinterleaving* (*u*, *A*, *v*)}›

lemmas *setinterleaving-induct*
[*case-names Nil-setinterleaving-Nil ev-setinterleaving-Nil tick-setinterleaving-Nil Nil-setinterleaving-ev Nil-setinterleaving-tick ev-setinterleaving-ev ev-setinterleaving-tick tick-setinterleaving-ev tick-setinterleaving-tick*]
=
setinterleaving.induct

lemma *Cons-setinterleaving-Nil* :
‹*setinterleaving* (*e* # *u*, *A*, []) =
(*case e of ev a* ⇒ (*if a* ∈ *A then* {}
else {*ev a* # *t* |*t*. *t* ∈ *setinterleaving* (*u*, *A*, [])})
| ✓ ⇒ {})›
by (*cases e*) *simp-all*

lemma *Nil-setinterleaving-Cons* :
‹*setinterleaving* ([], *A*, *e* # *v*) =
(*case e of ev a* ⇒ (*if a* ∈ *A then* {}
else {*ev a* # *t* |*t*. *t* ∈ *setinterleaving* ([], *A*, *v*)})
| ✓ ⇒ {})›
by (*cases e*) *simp-all*

lemma *Cons-setinterleaving-Cons* :
‹*setinterleaving* (*e* # *u*, *A*, *f* # *v*) =
(*case e of ev a* ⇒
(*case f of ev b* ⇒
if a ∈ *A*
then *if b* ∈ *A*
then *if a* = *b*
then {*ev a* # *t* |*t*. *t* ∈ *setinterleaving* (*u*, *A*, *v*)}
else {}
else {*ev b* # *t* |*t*. *t* ∈ *setinterleaving* (*ev a* # *u*, *A*, *v*)}
else *if b* ∈ *A then* {*ev a* # *t* |*t*. *t* ∈ *setinterleaving* (*u*, *A*, *ev b* # *v*)}
else {*ev a* # *t* |*t*. *t* ∈ *setinterleaving* (*u*, *A*, *ev b* # *v*)} ∪
{*ev b* # *t* |*t*. *t* ∈ *setinterleaving* (*ev a* # *u*, *A*, *v*)}

$| \checkmark \Rightarrow \quad \mathit{if}\ a \in A\ \mathit{then}\ \{\}$
$\mathit{else}\ \{\mathit{ev}\ a\ \#\ t\ |t.\ t \in \mathit{setinterleaving}\ (u,\ A,\ \checkmark\ \#\ v)\}\big)$
$| \checkmark \Rightarrow$
$(\mathit{case}\ f\ \mathit{of}\ \mathit{ev}\ b \Rightarrow \quad \mathit{if}\ b \in A\ \mathit{then}\ \{\}$
$\mathit{else}\ \{\mathit{ev}\ b\ \#\ t|\ t.\ t \in \mathit{setinterleaving}\ (\checkmark\ \#\ u,\ A,\ v)\}$
$| \checkmark \Rightarrow \{\checkmark\ \#\ t\ |t.\ t \in \mathit{setinterleaving}\ (u,\ A,\ v)\}))\rangle$
by (*cases e*; *cases f*) *simp-all*

lemmas *setinterleaving-simps* =
Cons-setinterleaving-Nil Nil-setinterleaving-Cons Cons-setinterleaving-Cons

abbreviation *setinterleaves* ::
$\langle['a\ \mathit{trace},\ 'a\ \mathit{trace},\ 'a\ \mathit{trace},\ 'a\ \mathit{set}] \Rightarrow \mathit{bool}\rangle$
$(\langle(\text{- }/(\mathit{setinterleaves})/\ '(()'(\text{-, -}')(),\ \text{-}'))\rangle\ [63,0,0,0]\ 64)$
where $\langle t\ \mathit{setinterleaves}\ ((u,\ v),\ A) \equiv t \in \mathit{setinterleaving}\ (u,\ A,\ v)\rangle$

lemma *tickFree-setinterleaves-iff* :
$\langle t\ \mathit{setinterleaves}\ ((u,\ v),\ A) \Longrightarrow tF\ t \longleftrightarrow tF\ u \wedge tF\ v\rangle$
by (*induct* $\langle(u,\ A,\ v)\rangle$ *arbitrary*: *t u v rule*: *setinterleaving-induct*)
(*auto split*: *if-split-asm*)

lemma *setinterleaves-tickFree-imp* :
$\langle tF\ u \vee tF\ v \Longrightarrow t\ \mathit{setinterleaves}\ ((u,\ v),\ A) \Longrightarrow tF\ t \wedge tF\ u \wedge tF\ v\rangle$
by (*elim disjE*; *induct* $\langle(u,\ A,\ v)\rangle$ *arbitrary*: *t u v rule*: *setinterleaving-induct*)
(*auto split*: *if-split-asm*)

lemma *setinterleaves-NilL-iff* :
$\langle t\ \mathit{setinterleaves}\ (([],\ v),\ A) \longleftrightarrow$
$tF\ v \wedge \mathit{set}\ v \cap \mathit{ev}\ `\ A = \{\} \wedge t = \mathit{map}\ \mathit{ev}\ (\mathit{map}\ \mathit{of\text{-}ev}\ v)\rangle$
by (*induct* $\langle([]\ ::\ 'a\ \mathit{trace},\ A,\ v)\rangle$ *arbitrary*: *t v rule*: *setinterleaving-induct*)
(*auto split*: *if-split-asm*)

lemma *setinterleaves-NilR-iff* :
$\langle t\ \mathit{setinterleaves}\ ((u,\ []),\ A) \longleftrightarrow$
$tF\ u \wedge \mathit{set}\ u \cap \mathit{ev}\ `\ A = \{\} \wedge t = \mathit{map}\ \mathit{ev}\ (\mathit{map}\ \mathit{of\text{-}ev}\ u)\rangle$
by (*induct* $\langle(u,\ A,\ []\ ::\ 'a\ \mathit{trace})\rangle$
arbitrary: *t u rule*: *setinterleaving-induct*)
(*auto split*: *if-split-asm*)

lemma *Nil-setinterleaves* :
‹[] *setinterleaves* ((*u*, *v*), *A*) ⟹ *u* = [] ∧ *v* = []›
by (*induct* ‹(*u*, *A*, *v*)› *arbitrary*: *u v rule*: *setinterleaving-induct*)
(*simp-all split*: *if-split-asm*)

lemma *front-tickFree-setinterleaves-iff* :
‹*t setinterleaves* ((*u*, *v*), *A*) ⟹ *ftF t* ⟷ *ftF u* ∧ *ftF v*›
proof (*induct* ‹(*u*, *A*, *v*)› *arbitrary*: *t u v rule*: *setinterleaving-induct*)
case (*tick-setinterleaving-tick u v*) **thus** *?case*
by (*simp add*: *split*: *if-split-asm*)
(*metis Nil-setinterleaves Nil-setinterleaving-Nil front-tickFree-Cons-iff singletonD*)
qed (*simp add*: *setinterleaves-NilL-iff setinterleaves-NilR-iff split*: *if-split-asm*;
metis event.simps(*3*) *front-tickFree-Cons-iff front-tickFree-Nil*)+

lemma *setinterleaves-snoc-notinL* :
‹*t setinterleaves* ((*u*, *v*), *A*) ⟹ *a* ∉ *A* ⟹
t @ [*ev a*] *setinterleaves* ((*u* @ [*ev a*], *v*), *A*)›
by (*induct* ‹(*u*, *A*, *v*)› *arbitrary*: *t u v rule*: *setinterleaving-induct*)
(*auto split*: *if-split-asm*)

lemma *setinterleaves-snoc-notinR* :
‹*t setinterleaves* ((*u*, *v*), *A*) ⟹ *a* ∉ *A* ⟹
t @ [*ev a*] *setinterleaves* ((*u*, *v* @ [*ev a*]), *A*)›
by (*induct* ‹(*u*, *A*, *v*)› *arbitrary*: *t u v rule*: *setinterleaving-induct*)
(*auto split*: *if-split-asm*)

lemma *setinterleaves-snoc-inside* :
‹*t setinterleaves* ((*u*, *v*), *A*) ⟹ *a* ∈ *A* ⟹
t @ [*ev a*] *setinterleaves* ((*u* @ [*ev a*], *v* @ [*ev a*]), *A*)›
by (*induct* ‹(*u*, *A*, *v*)› *arbitrary*: *t u v rule*: *setinterleaving-induct*)
(*auto split*: *if-split-asm*)

lemma *setinterleaves-snoc-tick* :
‹*t setinterleaves* ((*u*, *v*), *A*) ⟹ *t* @ [✓] *setinterleaves* ((*u* @ [✓], *v* @ [✓]), *A*)›
by (*induct* ‹(*u*, *A*, *v*)› *arbitrary*: *t u v rule*: *setinterleaving-induct*)
(*auto split*: *if-split-asm*)

lemma *Cons-tick-setinterleavesE* :
‹✓ # *t setinterleaves* ((*u*, *v*), *A*) ⟹
(⋀*u*′ *v*′ *r s*. ⟦*u* = ✓ # *u*′; *v* = ✓ # *v*′; *t setinterleaves* ((*u*′, *v*′), *A*)⟧
⟹ *thesis*) ⟹ *thesis*›
by (*induct* ‹(*u*, *A*, *v*)› *arbitrary*: *t u v rule*: *setinterleaving-induct*)

(*simp-all split*: *if-split-asm*)

lemma *Cons-ev-setinterleavesE* :
‹*ev a # t setinterleaves* ((*u*, *v*), *A*) ⟹
(⋀*u'*. *a* ∉ *A* ⟹ *u* = *ev a # u'* ⟹ *t setinterleaves* ((*u'*, *v*), *A*) ⟹ *thesis*) ⟹
(⋀*v'*. *a* ∉ *A* ⟹ *v* = *ev a # v'* ⟹ *t setinterleaves* ((*u*, *v'*), *A*) ⟹ *thesis*) ⟹
(⋀*u' v'*. *a* ∈ *A* ⟹ *u* = *ev a # u'* ⟹ *v* = *ev a # v'* ⟹
t setinterleaves ((*u'*, *v'*), *A*) ⟹ *thesis*) ⟹ *thesis*›
proof (*induct* ‹(*u*, *A*, *v*)› *arbitrary*: *u v t rule*: *setinterleaving-induct*)
case *Nil-setinterleaving-Nil* **thus** *?case* **by** *simp*
next
case (*ev-setinterleaving-Nil b u*)
from *ev-setinterleaving-Nil.prems*(*1*) **show** *?case*
by (*simp add*: *ev-setinterleaving-Nil.prems*(*2*) *split*: *if-split-asm*)
next
case (*tick-setinterleaving-Nil r u*) **thus** *?case* **by** *simp*
next
case (*Nil-setinterleaving-ev c v*)
from *Nil-setinterleaving-ev.prems*(*1*) **show** *?case*
by (*simp add*: *Nil-setinterleaving-ev.prems*(*3*) *split*: *if-split-asm*)
next
case (*Nil-setinterleaving-tick s v*) **thus** *?case* **by** *simp*
next
case (*ev-setinterleaving-ev b u c v*)
from *ev-setinterleaving-ev.prems*(*1*) **show** *?case*
by (*simp add*: *ev-setinterleaving-ev.prems*(*2*, *3*, *4*) *split*: *if-split-asm*)
(*use ev-setinterleaving-ev.prems*(*2*, *3*) **in** *presburger*)
next
case (*ev-setinterleaving-tick b u s v*)
from *ev-setinterleaving-tick.prems*(*1*) **show** *?case*
by (*simp add*: *ev-setinterleaving-tick.prems*(*2*) *split*: *if-split-asm*)
next
case (*tick-setinterleaving-ev r u c v*)
from *tick-setinterleaving-ev.prems*(*1*) **show** *?case*
by (*simp add*: *tick-setinterleaving-ev.prems*(*3*) *split*: *if-split-asm*)
next
case (*tick-setinterleaving-tick u v*) **thus** *?case* **by** *simp*
qed

lemma *rev-setinterleaves-rev-rev-iff* :
‹*rev t setinterleaves* ((*rev u*, *rev v*), *A*)
⟷ *t setinterleaves* ((*u*, *v*), *A*)›
proof (*rule iffI*)
show ‹*t setinterleaves* ((*u*, *v*), *A*) ⟹
rev t setinterleaves ((*rev u*, *rev v*), *A*)› **for** *t u v*
by (*induct* ‹(*u*, *A*, *v*)› *arbitrary*: *t u v rule*: *setinterleaving-induct*)

(*auto simp add*: *setinterleaves-snoc-notinL setinterleaves-snoc-notinR*
setinterleaves-snoc-inside setinterleaves-snoc-tick split: *if-split-asm*)
from *this*[*of* ‹*rev t*› ‹*rev u*› ‹*rev v*›, *simplified*]
show ‹*rev t setinterleaves* ((*rev u*, *rev v*), *A*) ⟹
t setinterleaves ((*u*, *v*), *A*)› **.**
qed

lemma *setinterleaves-preserves-ev-notin-set* :
‹⟦*ev a* ∉ *set u*; *ev a* ∉ *set v*; *t setinterleaves* ((*u*, *v*), *A*)⟧ ⟹ *ev a* ∉ *set t*›
by (*induct* ‹(*u*, *A*, *v*)› *arbitrary*: *t u v rule*: *setinterleaving-induct*)
(*auto split*: *if-split-asm*)

lemma *setinterleaves-preserves-ev-inside-set* :
‹⟦*ev a* ∈ *set u*; *ev a* ∈ *set v*; *t setinterleaves* ((*u*, *v*), *A*)⟧ ⟹ *ev a* ∈ *set t*›
proof (*induct* ‹(*u*, *A*, *v*)› *arbitrary*: *t u v rule*: *setinterleaving-induct*)
case *Nil-setinterleaving-Nil*
then show *?case* **by** *simp*
next
case (*ev-setinterleaving-Nil a u*)
then show *?case* **by** *simp*
next
case (*tick-setinterleaving-Nil u*)
then show *?case* **by** *simp*
next
case (*Nil-setinterleaving-ev b v*)
then show *?case* **by** *simp*
next
case (*Nil-setinterleaving-tick v*)
then show *?case* **by** *simp*
next
case (*ev-setinterleaving-ev a u b v*)
from *ev-setinterleaving-ev.prems* **show** *?case*
by (*simp-all split*: *if-split-asm*)
(*insert ev-setinterleaving-ev.hyps*; *metis list.set-intros*(*1*,*2*))+
next
case (*ev-setinterleaving-tick a u v*)
then show *?case* **by** (*auto split*: *if-split-asm*)
next
case (*tick-setinterleaving-ev u b v*)
then show *?case* **by** (*auto split*: *if-split-asm*)
next
case (*tick-setinterleaving-tick u v*)
then show *?case* **by** *auto*
qed

lemma *ev-notin-both-sets-imp-empty-setinterleaving* :
‹⟦*ev a* ∈ *set u* ∧ *ev a* ∉ *set v* ∨ *ev a* ∉ *set u* ∧ *ev a* ∈ *set v*; *a* ∈ *A*⟧
⟹
setinterleaving (*u*, *A*, *v*) = {}›
by (*induct* ‹(*u*, *A*, *v*)› *arbitrary*: *u v rule*: *setinterleaving-induct*)
(*simp-all*, *safe*, *auto*)

lemma *append-setinterleaves-imp* :
‹*t setinterleaves* ((*u*, *v*), *A*) ⟹ *t'* ≤ *t* ⟹
∃ *u'* ≤ *u*. ∃ *v'* ≤ *v*. *t' setinterleaves* ((*u'*, *v'*), *A*)›
proof (*induct* ‹(*u*, *A*, *v*)› *arbitrary*: *t u v t' rule*: *setinterleaving-induct*)
case *Nil-setinterleaving-Nil* **thus** *?case* **by** *auto*
next
case (*ev-setinterleaving-Nil a u*)
from *ev-setinterleaving-Nil.prems*
obtain *w w'* **where** ‹*a* ∉ *A*› ‹*t* = *ev a* # *w*› ‹*t'* = [] ∨ *t'* = *ev a* # *w'*›
‹*w'* ≤ *w*› ‹*w setinterleaves* ((*u*, []), *A*)›
by (*simp split*: *if-split-asm*) (*metis* (*no-types*) *Prefix-Order.prefix-Cons Nil-prefix*)
from ‹*t'* = [] ∨ *t'* = *ev a* # *w'*› **show** *?case*
proof (*elim disjE*)
show ‹*t'* = [] ⟹ *?case*› **by** *auto*
next
assume ‹*t'* = *ev a* # *w'*›
from *ev-setinterleaving-Nil.hyps*[*OF* ‹*a* ∉ *A*› ‹*w setinterleaves* ((*u*, []), *A*)› ‹*w'* ≤ *w*›]
obtain *u' v'* **where** ‹*u'* ≤ *u* ∧ *v'* ≤ [] ∧ *w' setinterleaves* ((*u'*, *v'*), *A*)› **by** *blast*
hence ‹*ev a* # *u'* ≤ *ev a* # *u* ∧ *v'* ≤ [] ∧ *t' setinterleaves* ((*ev a* # *u'*, *v'*), *A*)›
by (*auto simp add*: ‹*a* ∉ *A*› ‹*t'* = *ev a* # *w'*›)
thus *?case* **by** *blast*
qed
next
case (*tick-setinterleaving-Nil r u*) **thus** *?case* **by** *simp*
next
case (*Nil-setinterleaving-ev b v*)
from *Nil-setinterleaving-ev.prems*
obtain *w w'* **where** ‹*b* ∉ *A*› ‹*t* = *ev b* # *w*› ‹*t'* = [] ∨ *t'* = *ev b* # *w'*›
‹*w'* ≤ *w*› ‹*w setinterleaves* (([], *v*), *A*)›
by (*simp split*: *if-split-asm*) (*metis* (*no-types*) *Prefix-Order.prefix-Cons Nil-prefix*)
from ‹*t'* = [] ∨ *t'* = *ev b* # *w'*› **show** *?case*
proof (*elim disjE*)

show ‹$t' = [] \Longrightarrow$ *?case*› **by** *auto*
next
assume ‹$t' =$ *ev b # w'*›
from *Nil-setinterleaving-ev.hyps*[*OF* ‹$b \notin A$› ‹*w setinterleaves* (([], *v*), *A*)› ‹$w' \leq w$›]
obtain *u' v'* **where** ‹$u' \leq [] \wedge v' \leq v \wedge w'$ *setinterleaves* ((u', v'), *A*)› **by** *blast*
hence ‹$u' \leq [] \wedge$ *ev b # v'* $\leq$ *ev b # v* $\wedge$ t' *setinterleaves* ((u', *ev b # v'*), *A*)›
by (*auto simp add*: ‹$b \notin A$› ‹$t' =$ *ev b # w'*›)
thus *?case* **by** *blast*
qed
next
case (*Nil-setinterleaving-tick s v*) **thus** *?case* **by** *simp*
next
case (*ev-setinterleaving-ev a u b v*)
from *ev-setinterleaving-ev.prems*
consider ‹$t' = []$›
| (*both-in*) *w w'* **where** ‹$a \in A$› ‹$b \in A$› ‹$a = b$› ‹$t =$ *ev a # w*› ‹$t' =$ *ev a # w'*›
‹*w setinterleaves* ((u, v), A)› ‹$w' \leq w$›
| (*inR-mvL*) *w w'* **where** ‹$a \notin A$› ‹$b \in A$› ‹$t =$ *ev a # w*› ‹$t' =$ *ev a # w'*›
‹*w setinterleaves* ((u, *ev b # v*), A)› ‹$w' \leq w$›
| (*inL-mvR*) *w w'* **where** ‹$a \in A$› ‹$b \notin A$› ‹$t =$ *ev b # w*› ‹$t' =$ *ev b # w'*›
‹*w setinterleaves* ((*ev a # u*, v), A)› ‹$w' \leq w$›
| (*notin-mvL*) *w w'* **where** ‹$a \notin A$› ‹$b \notin A$› ‹$t =$ *ev a # w*› ‹$t' =$ *ev a # w'*›
‹*w setinterleaves* ((u, *ev b # v*), A)› ‹$w' \leq w$›
| (*notin-mvR*) *w w'* **where** ‹$a \notin A$› ‹$b \notin A$› ‹$t =$ *ev b # w*› ‹$t' =$ *ev b # w'*›
‹*w setinterleaves* ((*ev a # u*, v), A)› ‹$w' \leq w$›
by (*cases t'*) (*auto split*: *if-split-asm*)
thus *?case*
proof *cases*
from *Nil-setinterleaving-Nil* **show** ‹$t' = [] \Longrightarrow$ *?thesis*› **by** *blast*
next
case *both-in*
from *ev-setinterleaving-ev(1)*[*OF both-in*(*1−3*, *6−7*)]
obtain *u' v'* **where** ‹$u' \leq u \wedge v' \leq v \wedge w'$ *setinterleaves* ((u', v'), *A*)› **by** *blast*
hence ‹*ev a # u'* $\leq$ *ev a # u* $\wedge$ *ev b # v'* $\leq$ *ev b # v* $\wedge$ t' *setinterleaves* ((*ev a # u'*, *ev b # v'*), *A*)›
by (*auto simp add*: *both-in*(*2*, *3*) ‹$t' =$ *ev a # w'*›)
thus *?thesis* **by** *blast*
next
case *inR-mvL*
from *ev-setinterleaving-ev(3)*[*OF inR-mvL*(*1*, *2*, *5*, *6*)]

obtain u' v' **where** ‹$u' \leq u \wedge v' \leq ev\ b\ \#\ v \wedge w'$ *setinterleaves* $((u', v'), A)$› **by** *blast*
hence ‹$ev\ a\ \#\ u' \leq ev\ a\ \#\ u \wedge v' \leq ev\ b\ \#\ v \wedge t'$ *setinterleaves* $((ev\ a\ \#\ u', v'), A)$›
by (*cases* v') (*auto simp add*: *inR-mvL*(*1*, *4*))
thus *?thesis* **by** *blast*
next
case *inL-mvR*
from *ev-setinterleaving-ev*(*2*)[*OF inL-mvR*(*1*, *2*, *5*, *6*)]
obtain u' v' **where** ‹$u' \leq ev\ a\ \#\ u \wedge v' \leq v \wedge w'$ *setinterleaves* $((u', v'), A)$› **by** *blast*
hence ‹$u' \leq ev\ a\ \#\ u \wedge ev\ b\ \#\ v' \leq ev\ b\ \#\ v \wedge t'$ *setinterleaves* $((u', ev\ b\ \#\ v'), A)$›
by (*cases* u') (*auto simp add*: *inL-mvR*(*2*, *4*))
thus *?thesis* **by** *blast*
next
case *notin-mvL*
from *ev-setinterleaving-ev*(*4*)[*OF notin-mvL*(*1*, *2*, *5*, *6*)]
obtain u' v' **where** ‹$u' \leq u \wedge v' \leq ev\ b\ \#\ v \wedge w'$ *setinterleaves* $((u', v'), A)$› **by** *blast*
hence ‹$ev\ a\ \#\ u' \leq ev\ a\ \#\ u \wedge v' \leq ev\ b\ \#\ v \wedge t'$ *setinterleaves* $((ev\ a\ \#\ u', v'), A)$›
by (*cases* v') (*auto simp add*: *notin-mvL*(*1*, *4*))
thus *?thesis* **by** *blast*
next
case *notin-mvR*
from *ev-setinterleaving-ev*(*5*)[*OF notin-mvR*(*1*, *2*, *5*, *6*)]
obtain u' v' **where** ‹$u' \leq ev\ a\ \#\ u \wedge v' \leq v \wedge w'$ *setinterleaves* $((u', v'), A)$› **by** *blast*
hence ‹$u' \leq ev\ a\ \#\ u \wedge ev\ b\ \#\ v' \leq ev\ b\ \#\ v \wedge t'$ *setinterleaves* $((u', ev\ b\ \#\ v'), A)$›
by (*cases* u') (*auto simp add*: *notin-mvR*(*2*, *4*))
thus *?thesis* **by** *blast*
qed
next
case (*ev-setinterleaving-tick a u v*)
from *ev-setinterleaving-tick.prems*
obtain w w' **where** ‹$a \notin A$› ‹$t = ev\ a\ \#\ w$› ‹$t' = [] \vee t' = ev\ a\ \#\ w'$›
‹$w' \leq w$› ‹w *setinterleaves* $((u, \checkmark\ \#\ v), A)$›
by (*simp split*: *if-split-asm*) (*metis* (*no-types*) *Prefix-Order.prefix-Cons Nil-prefix*)
from ‹$t' = [] \vee t' = ev\ a\ \#\ w'$› **show** *?case*
proof (*elim disjE*)
from *Nil-setinterleaving-Nil* **show** ‹$t' = [] \Longrightarrow$ *?case*› **by** *blast*
next
assume ‹$t' = ev\ a\ \#\ w'$›
from *ev-setinterleaving-tick.hyps*[*OF* ‹$a \notin A$› ‹w *setinterleaves* $((u, \checkmark\ \#\ v), A)$› ‹$w' \leq w$›]

```
    obtain u' v' where ‹u' ≤ u ∧ v' ≤ ✓ # v ∧ w' setinterleaves ((u',
v'), A)› by blast
     hence ‹ev a # u' ≤ ev a # u ∧ v' ≤ ✓ # v ∧ t' setinterleaves
((ev a # u', v'), A)›
       by (cases v') (simp-all add: ‹a ∉ A› ‹t' = ev a # w'›)
    thus ?case by blast
  qed
next
  case (tick-setinterleaving-ev u b v)
  from tick-setinterleaving-ev.prems
  obtain w w' where ‹b ∉ A› ‹t = ev b # w› ‹t' = [] ∨ t' = ev b #
w'›
    ‹w' ≤ w› ‹w setinterleaves ((✓ # u, v), A)›
   by (simp split: if-split-asm) (metis (no-types) Prefix-Order.prefix-Cons
Nil-prefix)
  from ‹t' = [] ∨ t' = ev b # w'› show ?case
  proof (elim disjE)
    from Nil-setinterleaving-Nil show ‹t' = [] ⟹ ?case› by blast
  next
    assume ‹t' = ev b # w'›
    from tick-setinterleaving-ev.hyps[OF ‹b ∉ A› ‹w setinterleaves ((✓
# u, v), A)› ‹w' ≤ w›]
    obtain u' v' where ‹u' ≤ ✓ # u ∧ v' ≤ v ∧ w' setinterleaves ((u',
v'), A)› by blast
     hence ‹u' ≤ ✓ # u ∧ ev b # v' ≤ ev b # v ∧ t' setinterleaves
((u', ev b # v'), A)›
       by (cases u') (simp-all add: ‹b ∉ A› ‹t' = ev b # w'›)
    thus ?case by blast
  qed
next
  case (tick-setinterleaving-tick u v)
  from tick-setinterleaving-tick.prems obtain w w'
    where ‹t = ✓ # w› ‹t' = [] ∨ t' = ✓ # w'›
      ‹w' ≤ w› ‹w setinterleaves ((u, v), A)›
    by (cases t') (auto split: option.split-asm)
  from ‹t' = [] ∨ t' = ✓ # w'› show ?case
  proof (elim disjE)
    from Nil-setinterleaving-Nil show ‹t' = [] ⟹ ?case› by blast
  next
    assume ‹t' = ✓ # w'›
    from tick-setinterleaving-tick.hyps
      [OF ‹w setinterleaves ((u, v), A)› ‹w' ≤ w›]
    obtain u' v' where ‹u' ≤ u ∧ v' ≤ v ∧ w' setinterleaves ((u', v'),
A)› by blast
    hence ‹✓ # u' ≤ ✓ # u ∧ ✓ # v' ≤ ✓ # v ∧
          t' setinterleaves ((✓ # u', ✓ # v'), A)›
      by (simp add: ‹t' = ✓ # w'›)
    thus ?case by blast
  qed
```

qed

lift-definition *Sync* :: ‹'*a process* ⇒ '*a set* ⇒ '*a process* ⇒ '*a process*›
(‹(*3*(- ⟦-⟧/ -))› [*70*, *0*, *71*] *70*)
is ‹λ*P A Q*. {*t*. ∃ *t-P t-Q*. *t-P* ∈ 𝒯 *P* ∧ *t-Q* ∈ 𝒯 *Q* ∧ *t setinterleaves* ((*t-P*, *t-Q*), *A*)}›
proof −
show ‹*?thesis P A Q*› (**is** ‹*is-process ?t*›) **for** *P A Q*
proof (*unfold is-process-def*, *intro conjI allI impI*)
from *Nil-elem-T Nil-setinterleaving-Nil* **show** ‹[] ∈ *?t*› **by** *blast*
next
fix *t* **assume** ‹*t* ∈ *?t*›
then obtain *t-P t-Q* **where** ‹*t-P* ∈ 𝒯 *P*› ‹*t-Q* ∈ 𝒯 *Q*›
‹*t setinterleaves* ((*t-P*, *t-Q*), *A*)› **by** *blast*
from ‹*t-P* ∈ 𝒯 *P*› ‹*t-Q* ∈ 𝒯 *Q*› *front-tickFree-T*
have ‹*ftF t-P*› ‹*ftF t-Q*› **by** *auto*
with ‹*t setinterleaves* ((*t-P*, *t-Q*), *A*)›
show ‹*ftF t*› **by** (*simp add*: *front-tickFree-setinterleaves-iff*)
next
fix *t u* **assume** ‹*t* @ *u* ∈ *?t*›
then obtain *t-P t-Q* **where** ‹*t-P* ∈ 𝒯 *P*› ‹*t-Q* ∈ 𝒯 *Q*›
‹*t* @ *u setinterleaves* ((*t-P*, *t-Q*), *A*)› **by** *blast*
from *this*(*3*) **obtain** *t-P′ t-P″ t-Q′ t-Q″*
where ‹*t-P* = *t-P′* @ *t-P″*› ‹*t-Q* = *t-Q′* @ *t-Q″*›
‹*t setinterleaves* ((*t-P′*, *t-Q′*), *A*)›
by (*meson Prefix-Order.prefixE Prefix-Order.prefixI append-setinterleaves-imp*)
from ‹*t-P* ∈ 𝒯 *P*› ‹*t-Q* ∈ 𝒯 *Q*› *this*(*1*, *2*) **have** ‹*t-P′* ∈ 𝒯 *P*› ‹*t-Q′* ∈ 𝒯 *Q*›
by (*auto intro*: *T-dw-closed*)
with ‹*t setinterleaves* ((*t-P′*, *t-Q′*), *A*)› **show** ‹*t* ∈ *?t*› **by** *blast*
qed
qed

lemma *T-Sync* :
‹𝒯 (*P* ⟦*A*⟧ *Q*) = {*t*. ∃ *t-P t-Q*. *t-P* ∈ 𝒯 *P* ∧ *t-Q* ∈ 𝒯 *Q* ∧ *t setinterleaves* ((*t-P*, *t-Q*), *A*)}›
by (*simp add*: *Sync.rep-eq*)

lift-definition *Interrupt* :: ‹'*a process* ⇒ '*a process* ⇒ '*a process*›
(**infixl** ‹△› *81*)
is ‹λ*P Q*. 𝒯 *P* ∪ {*t* @ *u* |*t u*. *t* ∈ 𝒯 *P* ∧ *tF t* ∧ *u* ∈ 𝒯 *Q*}›
proof −
show ‹*?thesis P Q*› (**is** ‹*is-process ?t*›) **for** *P Q*
proof (*unfold is-process-def*, *intro conjI allI impI*)
show ‹[] ∈ *?t*› **by** (*simp add*: *Nil-elem-T*)
next

show ‹$t \in ?t \Longrightarrow ftF\ t$› **for** t
by (*auto simp add*: *front-tickFree-append-iff intro*: *front-tickFree-T*)
next
show ‹$t\ @\ u \in ?t \Longrightarrow t \in ?t$› **for** $t\ u$
by (*auto simp add*: *append-eq-append-conv2 intro*: *T-dw-closed*)
qed
qed

9.5 Constructiveness

lemma *restriction-process-Mprefix* :
‹$\Box a \in A \rightarrow P\ a \downarrow n = (case\ n\ of\ 0 \Rightarrow BOT \mid Suc\ m \Rightarrow \Box a{\in}A \rightarrow (P\ a \downarrow m))$›
by (*auto simp add*: *process-eq-spec T-restriction-process T-Mprefix T-BOT*
Nil-elem-T nat.case-eq-if front-tickFree-Cons-iff front-tickFree-T)
(*metis Cons-eq-append-conv Suc-length-conv event.distinct*(*1*) *length-greater-0-conv list.size*(*3*) *nat.exhaust-sel tickFree-Cons-iff*)

lemma *constructive-Mprefix* [*simp*] :
‹*constructive* $(\lambda b.\ \Box a{\in}A \rightarrow f\ a\ b)$› **if** ‹$\bigwedge a.\ a \in A \Longrightarrow$ *non-destructive* $(f\ a)$›
proof −
have ‹$\Box a{\in}A \rightarrow f\ a\ b = \Box a{\in}A \rightarrow (if\ a \in A\ then\ f\ a\ b\ else\ STOP)$› **for** b
by (*auto simp add*: *process-eq-spec T-Mprefix*)
moreover have ‹*constructive* $(\lambda b.\ \Box a{\in}A \rightarrow (if\ a \in A\ then\ f\ a\ b\ else\ STOP))$›
proof (*rule constructive-comp-non-destructive*[*of* ‹$\lambda P.\ \Box a{\in}A \rightarrow P\ a$›])
show ‹*constructive* $(\lambda P.\ \Box a{\in}A \rightarrow P\ a)$›
by (*rule constructiveI*) (*simp add*: *restriction-process-Mprefix restriction-fun-def*)
next
show ‹*non-destructive* $(\lambda b\ a.\ if\ a \in A\ then\ f\ a\ b\ else\ STOP)$›
by (*simp add*: *non-destructive-fun-iff*, *intro allI non-destructive-if-then-else*)
(*simp-all add*: ‹$\bigwedge a.\ a \in A \Longrightarrow$ *non-destructive* $(f\ a)$› *non-destructiveI*)
qed
ultimately show ‹*constructive* $(\lambda b.\ \Box a{\in}A \rightarrow f\ a\ b)$› **by** *simp*
qed

9.6 Non Destructiveness

lemma *non-destructive-Choice* [*simp*] :
‹*non-destructive* $(\lambda x.\ f\ x\ \Box\ g\ x)$›
if ‹*non-destructive* f› ‹*non-destructive* g›
for $f\ g :: \langle 'a :: restriction \Rightarrow 'b\ process\rangle$›
proof −
have $* :$ ‹*non-destructive* $(\lambda(P,\ Q).\ P \Box Q :: 'b\ process)$›

proof (*rule order-non-destructiveI*, *clarify*)
fix $P\ Q\ P'\ Q' :: \langle 'b\ process\rangle$ **and** n
assume $\langle (P,\ Q) \downarrow n = (P',\ Q') \downarrow n\rangle$
hence $\langle P \downarrow n = P' \downarrow n\rangle\ \langle Q \downarrow n = Q' \downarrow n\rangle$
by (*simp-all add*: *restriction-prod-def*)
show $\langle P \Box Q \downarrow n \leq P' \Box Q' \downarrow n\rangle$
proof (*unfold less-eq-process-def*, *rule subsetI*)
show $\langle t \in \mathcal{T}\ (P' \Box Q' \downarrow n) \Longrightarrow t \in \mathcal{T}\ (P \Box Q \downarrow n)\rangle$ **for** t
proof (*elim T-restriction-processE*)
show $\langle t \in \mathcal{T}\ (P' \Box Q') \Longrightarrow length\ t \leq n \Longrightarrow t \in \mathcal{T}\ (P \Box Q \downarrow n)\rangle$
by (*simp add*: *T-restriction-process T-Choice*)
(*metis* (*lifting*) *T-restriction-process T-restriction-processE*
Un-iff $\langle P \downarrow n = P' \downarrow n\rangle\ \langle Q \downarrow n = Q' \downarrow n\rangle$)
next
show $\langle [\![t = u\ @\ v;\ u \in \mathcal{T}\ (P' \Box Q');\ length\ u = n;\ tF\ u;\ ftF\ v]\!] \Longrightarrow t \in \mathcal{T}\ (P \Box Q \downarrow n)\rangle$ **for** $u\ v$
by (*simp add*: *T-restriction-process T-Choice*)
(*metis* (*lifting*) *T-restriction-process T-restriction-processE*
Un-iff
$\langle P \downarrow n = P' \downarrow n\rangle\ \langle Q \downarrow n = Q' \downarrow n\rangle$ *append.right-neutral*
append-eq-conv-conj)
qed
qed
qed
have $** : \langle non\text{-}destructive\ (\lambda x.\ (f\ x,\ g\ x))\rangle$
by (*fact non-destructive-prod-codomain*[*OF that*])
from *non-destructive-comp-non-destructive*[*OF* $* **$, *simplified*]
show $\langle non\text{-}destructive\ (\lambda x.\ f\ x \Box g\ x)\rangle$ **.**
qed

lemma *restriction-process-GlobalChoice* :
$\langle \Box a \in A.\ P\ a \downarrow n = (if\ A = \{\}\ then\ case\ n\ of\ 0 \Rightarrow BOT \mid Suc\ m \Rightarrow STOP\ else\ \Box a \in A.\ (P\ a \downarrow n))\rangle$
by (*auto simp add*: *process-eq-spec T-restriction-process T-GlobalChoice T-BOT T-STOP*
split: *nat.split*)

lemma *non-destructive-GlobalChoice* [*simp*] :
$\langle non\text{-}destructive\ (\lambda b.\ \Box a{\in}A.\ f\ a\ b)\rangle$ **if** $\langle \bigwedge a.\ a \in A \Longrightarrow non\text{-}destructive\ (f\ a)\rangle$
proof −
have $\langle \Box a{\in}A.\ f\ a\ b = \Box a{\in}A.\ (if\ a \in A\ then\ f\ a\ b\ else\ STOP)\rangle$ **for** b
by (*auto simp add*: *process-eq-spec T-GlobalChoice*)
moreover have $\langle non\text{-}destructive\ (\lambda b.\ \Box a{\in}A.\ (if\ a \in A\ then\ f\ a\ b\ else\ STOP))\rangle$
proof (*rule non-destructive-comp-non-destructive*[*of* $\langle \lambda P.\ \Box a{\in}A.\ P$

a›])
show ‹*non-destructive* (λ*P*. □*a*∈*A*. *P a*)›
by (*rule non-destructiveI*) (*simp add*: *restriction-process-GlobalChoice restriction-fun-def*)
next
show ‹*non-destructive* (λ*b a*. *if a* ∈ *A then f a b else STOP*)›
by (*simp add*: *non-destructive-fun-iff*, *intro allI non-destructive-if-then-else*)
(*simp-all add*: ‹⋀*a*. *a* ∈ *A* ⟹ *non-destructive* (*f a*)› *non-destructiveI*)
qed
ultimately show ‹*non-destructive* (λ*b*. □*a*∈*A*. *f a b*)› **by** *simp*
qed

9.7 Examples

notepad begin
fix *A B* :: ‹'*b* ⇒ '*a set*›
define *P* :: ‹'*b* ⇒ '*a process*›
where ‹*P* ≡ υ *X*. (λ*s*. □ *a* ∈ *A s* → *X s* □ (□ *b* ∈ *B s* → *X s*))›
(**is** ‹*P* ≡ υ *X*. *?f X*›)
have ‹*P* = *?f P*›
by (*unfold P-def*, *subst restriction-fix-eq*) *simp-all*

end

lemma ‹*constructive* (λ*X* σ. □ *e* ∈ *f* σ → □ σ' ∈ *g* σ *e*. *X* σ')›
by *simp*

lemma *length-le-T-restriction-process-iff-T* :
‹*length t* ≤ *n* ⟹ *t* ∈ 𝒯 (*P* ↓ *n*) ⟷ *t* ∈ 𝒯 *P*›
by (*auto simp add*: *T-restriction-process*)

lemma *restriction-adm-notin-T* [*simp*] : ‹*adm*↓ (λ*a*. *t* ∉ 𝒯 *a*)›
proof (*rule restriction-admI*)
fix σ **and** Σ **assume** ‹σ −↓→ Σ› ‹⋀*n*. *t* ∉ 𝒯 (σ *n*)›
from ‹σ −↓→ Σ› **obtain** *n0* **where** ‹∀ *k*≥*n0*. Σ ↓ *length t* = σ *k* ↓ *length t*›
by (*blast dest*: *restriction-tendstoD*)
hence ‹∀ *k*≥*n0*. 𝒯 (Σ ↓ *length t*) = 𝒯 (σ *k* ↓ *length t*)› **by** *simp*
hence ‹∀ *k*≥*n0*. *t* ∈ 𝒯 Σ ⟷ *t* ∈ 𝒯 (σ *k*)›
by (*metis dual-order.refl length-le-T-restriction-process-iff-T*)
with ‹⋀*n*. *t* ∉ 𝒯 (σ *n*)› **show** ‹*t* ∉ 𝒯 Σ› **by** *blast*
qed

lemma *restriction-adm-in-T* [*simp*] : ‹$adm_{\downarrow}$ ($\lambda a.\ t \in \mathcal{T}\ a$)›
proof (*rule restriction-admI*)
fix σ **and** Σ **assume** ‹$\sigma\ -\downarrow\rightarrow \Sigma$› ‹$\bigwedge n.\ t \in \mathcal{T}\ (\sigma\ n)$›
from ‹$\sigma\ -\downarrow\rightarrow \Sigma$› **obtain** *n0* **where** ‹$\forall\, k{\geq}n0.\ \Sigma \downarrow$ *length t* $= \sigma\ k \downarrow$ *length t*›
by (*blast dest*: *restriction-tendstoD*)
hence ‹$\forall\, k{\geq}n0.\ \mathcal{T}\ (\Sigma \downarrow$ *length t*$) = \mathcal{T}\ (\sigma\ k \downarrow$ *length t*)› **by** *simp*
hence ‹$\forall\, k{\geq}n0.\ t \in \mathcal{T}\ \Sigma \longleftrightarrow t \in \mathcal{T}\ (\sigma\ k)$›
by (*metis dual-order.refl length-le-T-restriction-process-iff-T*)
with ‹$\bigwedge n.\ t \in \mathcal{T}\ (\sigma\ n)$› **show** ‹$t \in \mathcal{T}\ \Sigma$› **by** *blast*
qed

10 Formal power Series

instantiation *fps* ::({*comm-ring-1*}) *restriction-space* **begin**
definition *restriction-fps* :: *'a fps* ⇒ *nat* ⇒ *'a fps*
where ‹*restriction-fps a n* ≡ $\sum i{<}n$. *fps-const* (*fps-nth a i*)∗*fps-X*^*i*›

lemma *intersection-equality*:‹(*n*::*nat*) ≤ *m* ⟹ {..<*m*} ∩ {*i*. *i* < *n*} = {*i*. *i*<*n*}›
by *auto*

lemma *exist-noneq*:‹*x* ≠ *y* ⟹
∃ *n*. ($\sum i{\in}\{x.\ x < n\}$. *fps-const* (*fps-nth x i*) ∗ *fps-X* ^ *i*) ≠
($\sum i{\in}\{x.\ x < n\}$. *fps-const* (*fps-nth y i*) ∗ *fps-X* ^ *i*)› **for** *x y*::‹*'a fps*›
proof −
assume ‹*x*≠*y*›
then have ‹∃ *n*. (*n* = (*LEAST n*. *fps-nth x n* ≠ *fps-nth y n*))›
using *fps-nth-inject* **by** *blast*
then obtain *n* **where** ‹ (*n* = (*LEAST n*. *fps-nth x n* ≠ *fps-nth y n*))› **by** *blast*
then have ‹∀ *i*<*n*. *fps-nth x i* = *fps-nth y i*›
using *not-less-Least* **by** *blast*
then have *f0*:‹($\sum i{<}n$. *fps-const* (*fps-nth x i*) ∗ *fps-X* ^ *i*) = ($\sum i{<}n$. *fps-const* (*fps-nth y i*) ∗ *fps-X* ^ *i*)›
by(*auto*)
have *rule*:‹*a*≠*c* ⟹ *a*+*b* ≠ *c*+*b*› **for** *a b c*::*'a fps*
unfolding *fps-plus-def* **using** *fps-ext*
by(*auto simp*:*fun-eq-iff fps-ext Abs-fps-inverse fps-nth-inverse Abs-fps-inject*)

have ‹*fps-nth x n* ≠ *fps-nth y n*›
by (*metis* (*mono-tags*, *lifting*) *LeastI-ex* ‹*n* = (*LEAST n*. *fps-nth x n* ≠ *fps-nth y n*)› ‹*x* ≠ *y*› *fps-ext*)
then have ‹ ($\sum i{<}n$. *fps-const* (*fps-nth x i*) ∗ *fps-X* ^ *i*) + *fps-const* (*fps-nth x n*) ∗ *fps-X* ^ *n* ≠

$(\sum i{<}n.\ \mathit{fps\text{-}const}\ (\mathit{fps\text{-}nth}\ y\ i) * \mathit{fps\text{-}X} \hat{} i) + \mathit{fps\text{-}const}\ (\mathit{fps\text{-}nth}\ y\ n) * \mathit{fps\text{-}X} \hat{} n\rangle$
by (*metis* (*no-types*, *lifting*) *f0 add-left-imp-eq fps-nth-fps-const fps-shift-times-fps-X-power'*)
moreover have ‹ $(\sum i{\in}\{x.\ x < \mathit{Suc}\ n\}.\ \mathit{fps\text{-}const}\ (\mathit{fps\text{-}nth}\ z\ i) * \mathit{fps\text{-}X} \hat{} i)$
$= (\sum i{<}n.\ \mathit{fps\text{-}const}\ (\mathit{fps\text{-}nth}\ z\ i) * \mathit{fps\text{-}X} \hat{} i) + \mathit{fps\text{-}const}\ (\mathit{fps\text{-}nth}\ z\ n) * \mathit{fps\text{-}X} \hat{} n\rangle$ **for** $z{::}'a\ \mathit{fps}$
proof −
have $\forall n.\ \{..{<}n{::}\mathit{nat}\} = \{na.\ na < n\}$
by (*simp add*: *lessThan-def*)
then show *?thesis*
using *sum.lessThan-Suc* **by** *auto*
qed
ultimately show ‹$\exists n.\ (\sum i{\in}\{x.\ x < n\}.\ \mathit{fps\text{-}const}\ (\mathit{fps\text{-}nth}\ x\ i) * \mathit{fps\text{-}X} \hat{} i) \neq$
$(\sum i{\in}\{x.\ x < n\}.\ \mathit{fps\text{-}const}\ (\mathit{fps\text{-}nth}\ y\ i) * \mathit{fps\text{-}X} \hat{} i)$›
by(*auto intro*: *exI*[**where** $x{=}\langle \mathit{Suc}\ n\rangle$])
qed

instance
using *intersection-equality exist-noneq*
by *intro-classes*
(*auto cong*: *if-cong simp add*:
Collect-mono Int-absorb2 restriction-fps-def fps-sum-nth
if-distrib[**where** *f*=*fps-const*] *if-distrib*[**where** $f{=}\langle \lambda x.\ a * x\rangle$ **for** *a*]
if-distrib[**where** $f{=}\langle \lambda x.\ x * a\rangle$ **for** *a*] *lessThan-def sum.If-cases min-def*)

end

lemma *fps-sum-rep-nthb*: $\mathit{fps\text{-}nth}\ (\sum i{<}m.\ \mathit{fps\text{-}const}(a\ i){*}\mathit{fps\text{-}X}\hat{}i)\ n = (\mathit{if}\ n < m\ \mathit{then}\ a\ n\ \mathit{else}\ 0)$
by (*simp add*: *fps-sum-nth if-distrib cong del*: *if-weak-cong*)

lemma *restriction-eq-iff* :‹$a{\downarrow}n = b{\downarrow}n \longleftrightarrow (\forall i{<}n.\ \mathit{fps\text{-}nth}\ a\ i = \mathit{fps\text{-}nth}\ b\ i)$›
by (*auto simp*:*restriction-fps-def*)
(*metis* (*full-types*) *fps-sum-rep-nthb*)

lemma *restriction-eqI* :
‹$(\bigwedge i.\ i < n \Longrightarrow \mathit{fps\text{-}nth}\ x\ i = \mathit{fps\text{-}nth}\ y\ i) \Longrightarrow x \downarrow n = y \downarrow n$›
by (*simp add*: *restriction-eq-iff*)

lemma *restriction-eqI'* :
‹$(\bigwedge i.\ i \leq n \Longrightarrow \mathit{fps\text{-}nth}\ x\ i = \mathit{fps\text{-}nth}\ y\ i) \Longrightarrow x \downarrow n = y \downarrow n$›

by (*simp add*: *restriction-eq-iff*)

instantiation *fps* :: (*comm-ring-1*) *complete-restriction-space*
begin
instance
proof (*intro-classes*, *rule restriction-convergentI*)
fix σ :: ‹*nat* ⇒′*a fps*› **assume** *h*:‹*restriction-chain* σ›
have *h*′:‹∀ *n*. (∑ *i*<*n*. *fps-const* (*fps-nth* (σ (*Suc n*)) *i*) ∗ *fps-X* ^ *i*) = σ *n*›
using *h* **unfolding** *restriction-chain-def restriction-fps-def* **by** *auto*
let *?*Σ = ‹*Abs-fps* (λ*n*. *fps-nth* (σ (*Suc n*)) *n*)›
have ‹*?*Σ ↓ (*n*) = σ *n*› **for** *n*
proof (*subst restricted-restriction-chain-is*[*OF* ‹*restriction-chain* σ›, *symmetric*],
rule restriction-eqI)
fix *i* **assume** ‹*i* < *n*›
then have ‹*i* ≤ *n*› **by** *auto*
from *restriction-chain-def-ter*
[*THEN iffD1*, *OF* ‹*restriction-chain* σ›, *rule-format*, *OF* ‹*i* ≤ *n*›]
show ‹*fps-nth* (*Abs-fps* (λ*n*. *fps-nth* (σ (*Suc n*)) *n*)) *i* = *fps-nth* (σ *n*) *i*›
by (*subst Abs-fps-inverse*, *use Abs-fps-inject restriction-fps-def* **in** *blast*)
(*smt* (*verit*, *ccfv-threshold*) *Suc-leI* ‹*i* < *n*› *h le-refl lessI restriction-eq-iff*
restriction-chain-def restriction-chain-def-ter)
qed
thus ‹*restriction-chain* σ ⟹ σ −↓→ *?*Σ›
proof −
have (↓) (*Abs-fps* (λ*n*. *fps-nth* (σ (*Suc n*)) *n*)) = σ
using ‹⋀*n*. *Abs-fps* (λ*n*. *fps-nth* (σ (*Suc n*)) *n*) ↓ *n* = σ *n*› **by** *force*
then show *?thesis*
by (*metis restriction-tendsto-restrictions*)
qed
qed

end