Miscellaneous Examples of restriction Spaces

Benoît Ballenghien Benjamin Puyobro Burkhart Wolff

September 1, 2025

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* :
  ‹$chain_{\downarrow}$ 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 ‹$b \downarrow n \equiv$ *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*] : ‹$b \downarrow 0$ = *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* :
  ‹$\sigma \ -\!\downarrow\!\rightarrow \ \Sigma \longleftrightarrow (\exists\, n.\ \forall\, k \geq n.\ \sigma\ k = \Sigma)$› **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** ‹$chain_{\downarrow}\ \sigma$›
  hence ‹$(\forall\, n{>}0.\ \neg\ \sigma\ n) \vee (\forall\, n{>}0.\ \sigma\ n)$›
   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 ‹$\sigma \ -\!\downarrow\!\rightarrow$ *False* $\vee\ \sigma \ -\!\downarrow\!\rightarrow$ *True*›
   by (*metis* (*full-types*) *gt-ex order.strict-trans2 restriction-tendsto-def*)
  thus ‹$convergent_{\downarrow}\ \sigma$›
    using *restriction-convergentI* **by** *blast*
qed

lemma *restriction-cont-imp-restriction-adm* :
  ‹$cont_{\downarrow}\ P \Longrightarrow adm_{\downarrow}\ P$› **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* : ‹$compact_{\downarrow}$ (*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 ‹$x \downarrow n \equiv$ *if* $x \leq n$ *then* x *else* n›

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

end

lemma *restriction-nat-0-is-0* [*simp*] : ‹$x \downarrow 0 = (0 :: nat)$›
 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 ‹$x \downarrow n = y \downarrow n \Longrightarrow Suc\ x \downarrow Suc\ n = Suc\ y \downarrow Suc\ n$› **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 ‹$x \downarrow Suc\ n = y \downarrow Suc\ n \Longrightarrow nat.pred\ x \downarrow n = nat.pred\ y \downarrow n$›
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* ($\lambda x.\ x + k$) (*int k*)›
proof (*intro restriction-shiftI*)
 show ‹$x \downarrow n = y \downarrow n \Longrightarrow x + k \downarrow nat\ (int\ n + int\ k) = y + k \downarrow nat\ (int\ n + int\ k)$› **for** $x\ y\ n$
  by (*simp add*: *restriction-nat-def nat-int-add split*: *if-split-asm*)
qed

lemma ‹*restriction-shift* ($\lambda x.\ k + x$) (*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* :: ‹$int \Rightarrow nat \Rightarrow int$›
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 ‹$x \downarrow n \equiv$ *if* $n = 0$ *then None else map-option* ($\lambda a.\ a \downarrow n$) 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*] : ‹$x \downarrow 0 =$ *None*›
  by (*simp add*: *restriction-option-def*)

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

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

lemma *restriction-option-eq-None-iff* : ‹$x \downarrow n =$ *None* $\longleftrightarrow n = 0 \lor x =$ *None*›
  by (*cases* x) *simp-all*

lemma *restriction-option-eq-Some-iff* : ‹$x \downarrow n =$ *Some* $y \longleftrightarrow n \neq 0 \land x \neq$ *None* $\land y =$ *the* $x \downarrow n$›
  by (*cases* x) *auto*

5.2 Restriction space option type

instance *option* :: (*restriction-space*) *restriction-space*
proof *intro-classes*
  show ‹$x \downarrow 0 = y \downarrow 0$› **for** $x\ y$ :: ‹$'a$ *option*› **by** *simp*
next
  show ‹$x \neq y \Longrightarrow \exists n.\ x \downarrow n \neq y \downarrow n$› **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** ‹$chain_{\downarrow}\ \sigma$›
  obtains ‹$\sigma = (\lambda n.$ *None*$)$›
  | σ' **where** ‹$chain_{\downarrow}\ \sigma'$› **and** ‹$\sigma = (\lambda n.$ *if* $n = 0$ *then None else Some* $(\sigma'\ n))$›
proof −
  from ‹$chain_{\downarrow}\ \sigma$› **consider** ‹$\forall n.\ \sigma\ n =$ *None*› | ‹$\forall n{>}0.\ \sigma\ n \neq$ *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** ‹$\forall$ *n*. σ *n* = *None* $\Longrightarrow$ *thesis*› **by** *fast*
next
define σ' **where** ‹σ' *n* $\equiv$ *if n* = *0 then undefined* $\downarrow$ *0 else the* (σ *n*)› **for** *n*
assume ‹$\forall$ *n*>*0*. σ *n* $\neq$ *None*›
with ‹*chain*$_\downarrow$ σ› **have** ‹*chain*$_\downarrow$ σ'› ‹σ = (λ*n*. *if n* = *0 then undefined* $\downarrow$ *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 restriction-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*$_\downarrow$ (*Some* :: *'a* :: *restriction-space* $\Rightarrow$ *'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*$_\downarrow$ σ $\Longrightarrow$ *convergent*$_\downarrow$ σ› **for** σ :: ‹*nat* $\Rightarrow$ *'a option*›
proof (*elim option-restriction-chainE*)
show ‹σ = (λ*n*. *None*) $\Longrightarrow$ *convergent*$_\downarrow$ σ› **by** *simp*
next
fix σ' **assume** ‹*chain*$_\downarrow$ σ'› **and** σ*-def* : ‹σ = (λ*n*. *if n* = *0 then None else Some* (σ' *n*))›
from ‹*chain*$_\downarrow$ σ'› **have** ‹*convergent*$_\downarrow$ σ'› **by** (*simp add*: *restriction-chain-imp-restriction-convergent*)
hence ‹*convergent*$_\downarrow$ (λ*n*. σ' (*n* + *1*))› **by** (*unfold restriction-convergent-shift-iff*)
then obtain Σ' **where** ‹(λ*n*. σ' (*n* + *1*)) $-\downarrow\rightarrow$ Σ'› **by** (*blast dest*: *restriction-convergentD'*)
hence ‹(λ*n*. *Some* (σ' (*n* + *1*))) $-\downarrow\rightarrow$ *Some* Σ'› **by** (*fact restriction-contD*[*OF restriction-cont-Some*])
hence ‹*convergent*$_\downarrow$ (λ*n*. *Some* (σ' (*n* + *1*)))› **by** (*blast intro*: *restriction-convergentI*)
hence ‹*convergent*$_\downarrow$ (λ*n*. σ (*n* + *1*))› **by** (*simp add*: σ*-def*)
thus ‹*convergent*$_\downarrow$ 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 ‹$L \downarrow n \equiv$ *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 ‹$L \downarrow 0 \leq M \downarrow 0$› **for** *L M* :: ‹*'a list*›
    by (*simp add*: *restriction-list-def*)
next
  show ‹$L \leq M \Longrightarrow L \downarrow n \leq M \downarrow n$› **for** *L M* :: ‹*'a list*› **and** *n*
    unfolding *restriction-list-def*
    by (*metis less-eq-list-def prefix-def take-append*)
next
  show ‹$\neg\ L \leq M \Longrightarrow \exists n.\ \neg\ L \downarrow n \leq M \downarrow n$› **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 ‹$\sigma\ n$ = *replicate n undefined*› **for** *n*

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

  hence ‹$\nexists \Sigma.\ \sigma \dashrightarrow_{\downarrow} \Sigma$›
   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* :
‹($\bigwedge i.\ i \le max\ (size\ x)\ (size\ y) \Longrightarrow x \downarrow i = \ y \downarrow i) \Longrightarrow x = y$› **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* = ‹$\{\sigma :: nat \Rightarrow 'a.\ \sigma\ 0 = 0\}$›
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* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *decimals*›
is ‹$\lambda\sigma\ m\ n.\ if\ n \le m\ then\ \sigma\ n\ else\ 0$› **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* :
  ‹$x \downarrow n = y \downarrow n \longleftrightarrow (\forall\, i{\le}n.\ \textit{from-decimals}\ x\ i = \textit{from-decimals}\ y\ i)$›
  by *transfer meson*

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

lemma *restriction-decimals-eqD* :
  ‹$x \downarrow n = y \downarrow n \Longrightarrow i \le n \Longrightarrow \textit{from-decimals}\ x\ i = \textit{from-decimals}\ y\ i$›
  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 :: ‹$nat \Rightarrow$ *'a decimals*› **assume** ‹$chain_{\downarrow}\ \sigma$›
  let *?Σ* = ‹*to-decimals* ($\lambda n.$ *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** ‹$i \le n$›
    from *restriction-chain-def-ter*
      [*THEN iffD1*, *OF* ‹*restriction-chain* σ›, *rule-format*, *OF* ‹$i \le n$›]
    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* $\sigma \Longrightarrow \sigma \dashv\!\downarrow\!\rightarrow$ *?Σ*›
  proof –
    have ‹($\downarrow$) (*to-decimals* ($\lambda n.$ *from-decimals* (σ n) n)) = σ›
      using ‹$\bigwedge n.$ *to-decimals* ($\lambda n.$ *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 ‹*λs 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* ‹$x \downarrow n = y \downarrow n$› ‹$i \leq n$›]
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 decimals*›
is ‹$\lambda s\ n.$ *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* $x \downarrow$ *Suc* n = *shift-decimal-right* $y \downarrow$ *Suc* n›
if ‹$x \downarrow n = y \downarrow n$› **for** x y n
proof (*intro restriction-decimals-eqI*)
fix *index val* i **assume** ‹$i \leq$ *Suc* n›
hence ‹$i - 1 \leq n$› **by** *simp*
from *restriction-decimals-eqD*[*OF* ‹$x \downarrow n = y \downarrow n$› ‹$i - 1 \leq n$›]
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 decimals*›
is ‹$\lambda s\ n.$ *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* $x \downarrow n$ = *shift-decimal-left* $y \downarrow n$› **if** ‹$x \downarrow$ *Suc* $n = y \downarrow$ *Suc* n› **for** x y n
proof (*intro restriction-decimals-eqI*)
fix *index val* i **assume** ‹$i \leq n$›
hence ‹*Suc* $i \leq$ *Suc* n› **by** *simp*
from *restriction-decimals-eqD*[*OF* ‹$x \downarrow$ *Suc* $n = y \downarrow$ *Suc* n› ‹*Suc* $i \leq$ *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*: ‹$P \equiv \lambda x$. (*1* :: *nat-0-9*) $\in$ *range* (*from-decimals x*)›
  shows ‹$\neg$ $adm_{\downarrow}$ *P*›
proof (*rule ccontr*)
  assume $*$: ‹$\neg$ $\neg$ $adm_{\downarrow}$ *P*›

  define *x* **where** ‹$x \equiv$ *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 ‹$P\ x \Longrightarrow P$ (*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 ‹$\neg$ *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 ‹$\{[]\} \in \{T.\ \textit{is-process}\ T\}$›
  by (*simp add*: *is-process-def front-tickFree-def*)
qed

setup-lifting *type-definition-process*

notation *Traces* (‹$\mathcal{T}$›)

lemma *is-process-inv* :
‹$[] \in \mathcal{T}\ P \land (\forall\, t.\ t \in \mathcal{T}\ P \longrightarrow \textit{ftF}\ t) \land (\forall\, t\ u.\ t\ @\ u \in \mathcal{T}\ P \longrightarrow t \in \mathcal{T}\ P)$›
by (*metis is-process-def mem-Collect-eq to-process-cases to-process-inverse*)

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

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

9.2 First Processes

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

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

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

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

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

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

lift-definition *Sup-processes* ::
‹$(\textit{nat} \Rightarrow {}'a\ \textit{process}) \Rightarrow {}'a\ \textit{process}$› **is** ‹$\lambda\sigma.\ \bigcap i.\ \mathcal{T}\ (\sigma\ i)$›
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 ‹[] $\in (\bigcap i.\ \mathcal{T}\ (\sigma\ i))$› **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* $\Longrightarrow$ *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'› ‹$t' \in \mathcal{T}$ P›
‹*length* t' = n› ‹*tF* t'› ‹*ftF* u'› **by** *blast*
thus ‹$t \in$ *?t*›
proof *cases*
from *T-dw-closed* **show** ‹t @ $u \in \mathcal{T}$ $P \Longrightarrow t \in$ *?t*› **by** *blast*
next
case @
show ‹$t \in$ *?t*›
proof (*cases* ‹*length* $t \leq$ *length* t'›)
assume ‹*length* $t \leq$ *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** ‹$t \in$ *?t*› **by** *blast*
next
assume ‹$\neg$ *length* $t \leq$ *length* t'›
hence ‹*length* $t' \leq$ *length* t› **by** *simp*
with @(*1*,*4*,*5*) ‹$\neg$ *length* $t \leq$ *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** ‹$t \in$ *?t*› **by** *blast*
qed
qed
qed
qed

lemma *T-restriction-process* :
‹$\mathcal{T}$ ($P \downarrow n$) = $\mathcal{T}$ $P \cup \{t$ @ u |t u. $t \in \mathcal{T}$ $P \wedge$ *length* $t = n \wedge$ *tF* $t \wedge$ *ftF* $u\}$›
by (*simp add*: *restriction-process.rep-eq*)

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

lemma *T-restriction-processE* :
‹$t \in \mathcal{T}$ ($P \downarrow n$) $\Longrightarrow$
($t \in \mathcal{T}$ $P \Longrightarrow$ *length* $t \leq n \Longrightarrow$ *thesis*) $\Longrightarrow$
($\bigwedge u$ v. $t = u$ @ $v \Longrightarrow u \in \mathcal{T}$ $P \Longrightarrow$ *length* $u = n \Longrightarrow$ *tF* $u \Longrightarrow$ *ftF* $v \Longrightarrow$ *thesis*) $\Longrightarrow$
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 ↓ 0 ≤ Q ↓ 0› for P Q :: ‹'a process› by simp
next
  show ‹P ↓ n ↓ m = P ↓ min n m› for P :: ‹'a process› and n m
  proof (subst process-eq-spec, intro subset-antisym subsetI)
    show ‹t ∈ 𝒯 (P ↓ n ↓ m) ⟹ t ∈ 𝒯 (P ↓ min n m)› for t
      by (elim T-restriction-processE)
      (auto simp add: T-restriction-process intro: front-tickFree-append)
  next
    show ‹t ∈ 𝒯 (P ↓ min n m) ⟹ t ∈ 𝒯 (P ↓ n ↓ m)› for t
      by (elim T-restriction-processE)
        (auto simp add: T-restriction-process min-def split: if-split-asm)
  qed
next
  show ‹P ≤ Q ⟹ P ↓ n ≤ Q ↓ 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 ‹¬ P ≤ Q›
  then obtain t where ‹t ∈ 𝒯 Q› ‹t ∉ 𝒯 P›
    unfolding less-eq-process-def by blast
  hence ‹t ∈ 𝒯 (Q ↓ Suc (length t)) ∧ t ∉ 𝒯 (P ↓ Suc (length t))›
    by (auto simp add: T-restriction-process)
  hence ‹¬ P ↓ Suc (length t) ≤ Q ↓ Suc (length t)›
    unfolding less-eq-process-def by blast
  thus ‹∃ n. ¬ P ↓ n ≤ Q ↓ 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 :
  ‹(λn. Sup-processes σ ↓ n) = σ› if ‹restriction-chain σ›
proof (subst (2) restricted-restriction-chain-is
    [OF ‹restriction-chain σ›, symmetric], rule ext)
  fix n
  have ‹length t ≤ n ⟹ t ∈ 𝒯 (σ n) ⟷ (∀ i. t ∈ 𝒯 (σ i))› for t n
  proof safe
    show ‹t ∈ 𝒯 (σ i)› if ‹length t ≤ n› ‹t ∈ 𝒯 (σ n)› for i
    proof (cases ‹i ≤ n›)
     from restriction-chain-def-ter[THEN iffD1, OF ‹restriction-chain
σ›]
      show ‹i ≤ n ⟹ t ∈ 𝒯 (σ i)›
        by (metis (lifting) ‹t ∈ 𝒯 (σ n)› T-restriction-process Un-iff)
    next
      from ‹length t ≤ n› ‹t ∈ 𝒯 (σ n)› show ‹¬ i ≤ n ⟹ t ∈ 𝒯 (σ
```

```
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 ‹$\lambda P\ Q.\ \mathcal{T}\ P \cup \mathcal{T}\ Q$›
by (*auto simp add*: *is-process-def Nil-elem-T front-tickFree-T intro*: *T-dw-closed*)

lemma *T-Choice* : ‹$\mathcal{T}\ (P \,\square\, Q) = \mathcal{T}\ P \cup \mathcal{T}\ Q$›
by (*simp add*: *Choice.rep-eq*)

lift-definition *GlobalChoice* :: ‹[′*b set*, ′*b* ⇒ ′*a process*] ⇒ ′*a process*›
is ‹$\lambda A\ P.$ *if* $A = \{\}$ *then* $\{[]\}$ *else* $\bigcup a{\in}A.\ \mathcal{T}\ (P\ a)$›
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* (λ*a*. *P*)

lemma *T-GlobalChoice* : ‹$\mathcal{T}\ (\square a \in A.\ P\ a) =$ (*if* $A = \{\}$ *then* $\{[]\}$ *else* $\bigcup a{\in}A.\ \mathcal{T}\ (P\ a)$)›
by (*simp add*: *GlobalChoice.rep-eq*)

lift-definition *Seq* :: ‹′*a process* ⇒ ′*a process* ⇒ ′*a process*› (**infixl** ‹**;**› *74*)
is ‹$\lambda P\ Q.\ \{t \in \mathcal{T}\ P.\ tF\ t\} \cup \{t\ @\ u\ |t\ u.\ t\ @\ [✓] \in \mathcal{T}\ P \wedge u \in \mathcal{T}\ Q\}$›
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* : ‹$\mathcal{T}\ (P\ \mathbf{;}\ Q) = \{t \in \mathcal{T}\ P.\ tF\ t\} \cup \{t\ @\ u\ |t\ u.\ t\ @\ [✓] \in \mathcal{T}\ P \wedge u \in \mathcal{T}\ Q\}$›
by (*simp add*: *Seq.rep-eq*)

lift-definition *Renaming* :: ‹[′*a process*, ′*a* ⇒ ′*b*] ⇒ ′*b process*›
is ‹$\lambda P\ f.$ {*map* (*map-event f*) *u* |*u*. $u \in \mathcal{T}\ P$}›
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* : ‹$\mathcal{T}$ (*Renaming P f*) = {*map* (*map-event f*) *u* |*u*. $u \in \mathcal{T}\ P$}›
by (*simp add*: *Renaming.rep-eq*)

lift-definition *Mprefix* :: ‹['a set, 'a ⇒ 'a process] ⇒ 'a process›
is ‹$\lambda A\ P.$ *insert* [] {*ev a* # *t* |*a t*. $a \in A \wedge t \in \mathcal{T}$ (*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* : ‹$\mathcal{T}$ (□*a* ∈ *A* → *P a*) = *insert* [] {*ev a* # *t* |*a t*. *a* ∈ *A* ∧ *t* ∈ $\mathcal{T}$ (*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*)}

| ✓ ⇒ *if* $a \in A$ *then* {}
else {*ev a* # *t* |*t*. *t* ∈ *setinterleaving* (*u*, *A*, ✓ # *v*)})
| ✓ ⇒
(*case f of ev b* ⇒ *if* $b \in A$ *then* {}
else {*ev b* # *t*| *t*. *t* ∈ *setinterleaving* (✓ # *u*, *A*, *v*)}
| ✓ ⇒ {✓ # *t* |*t*. *t* ∈ *setinterleaving* (*u*, *A*, *v*)}))›
by (*cases e*; *cases f*) *simp-all*

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

abbreviation *setinterleaves* ::
‹[′*a trace*, ′*a trace*, ′*a trace*, ′*a set*] ⇒ *bool*›
(‹(- /(*setinterleaves*)/ ′(()′(-, -′)(), -′))› [*63*,*0*,*0*,*0*] *64*)
where ‹*t setinterleaves* ((*u*, *v*), *A*) ≡ *t* ∈ *setinterleaving* (*u*, *A*, *v*)›

lemma *tickFree-setinterleaves-iff* :
‹*t setinterleaves* ((*u*, *v*), *A*) ⟹ *tF t* ⟷ *tF u* ∧ *tF v*›
by (*induct* ‹(*u*, *A*, *v*)› *arbitrary*: *t u v rule*: *setinterleaving-induct*)
(*auto split*: *if-split-asm*)

lemma *setinterleaves-tickFree-imp* :
‹*tF u* ∨ *tF v* ⟹ *t setinterleaves* ((*u*, *v*), *A*) ⟹ *tF t* ∧ *tF u* ∧ *tF v*›
by (*elim disjE*; *induct* ‹(*u*, *A*, *v*)› *arbitrary*: *t u v rule*: *setinterleaving-induct*)
(*auto split*: *if-split-asm*)

lemma *setinterleaves-NilL-iff* :
‹*t setinterleaves* (([], *v*), *A*) ⟷
tF v ∧ *set v* ∩ *ev* ‘ *A* = {} ∧ *t* = *map ev* (*map of-ev v*)›
by (*induct* ‹([] :: ′*a trace*, *A*, *v*)› *arbitrary*: *t v rule*: *setinterleaving-induct*)
(*auto split*: *if-split-asm*)

lemma *setinterleaves-NilR-iff* :
‹*t setinterleaves* ((*u*, []), *A*) ⟷
tF u ∧ *set u* ∩ *ev* ‘ *A* = {} ∧ *t* = *map ev* (*map of-ev u*)›
by (*induct* ‹(*u*, *A*, [] :: ′*a trace*)›
arbitrary: *t u rule*: *setinterleaving-induct*)
(*auto split*: *if-split-asm*)

lemma *Nil-setinterleaves* :
‹[] *setinterleaves* ((*u*, *v*), *A*) $\Longrightarrow$ *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*) $\Longrightarrow$ *ftF t* $\longleftrightarrow$ *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*) $\Longrightarrow$ *a* ∉ *A* $\Longrightarrow$
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*) $\Longrightarrow$ *a* ∉ *A* $\Longrightarrow$
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*) $\Longrightarrow$ *a* ∈ *A* $\Longrightarrow$
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*) $\Longrightarrow$ *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*) $\Longrightarrow$
($\bigwedge$*u′ v′ r s*. ⟦*u* = ✓ # *u′*; *v* = ✓ # *v′*; *t setinterleaves* ((*u′*, *v′*), *A*)⟧ $\Longrightarrow$ *thesis*) $\Longrightarrow$ *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*) $\Longrightarrow$
t setinterleaves ((*u*, *v*), *A*)› **.**
qed

lemma *setinterleaves-preserves-ev-notin-set* :
‹⟦*ev a* $\notin$ *set u*; *ev a* $\notin$ *set v*; *t setinterleaves* ((*u*, *v*), *A*)⟧ $\Longrightarrow$ *ev a* $\notin$ *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* $\in$ *set u*; *ev a* $\in$ *set v*; *t setinterleaves* ((*u*, *v*), *A*)⟧ $\Longrightarrow$ *ev a* $\in$ *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* $((u, ✓ \# v), A)$›
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* $((u, ✓ \# v), A)$› ‹$w' \leq w$›]

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 # $u \wedge v' \leq$ ✓ # $v \wedge t'$ *setinterleaves* $(($*ev* a # u', $v'), A)$›
by (*cases* v') (*simp-all add*: ‹$a \notin A$› ‹$t' =$ *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** ‹$b \notin A$› ‹$t =$ *ev* b # w› ‹$t' = [] \vee t' =$ *ev* b # w'›
‹$w' \leq w$› ‹w *setinterleaves* ((✓ # u, v), A)›
by (*simp split*: *if-split-asm*) (*metis* (*no-types*) *Prefix-Order.prefix-Cons Nil-prefix*)
from ‹$t' = [] \vee t' =$ *ev* b # w'› **show** *?case*
proof (*elim disjE*)
from *Nil-setinterleaving-Nil* **show** ‹$t' = [] \Longrightarrow$ *?case*› **by** *blast*
next
assume ‹$t' =$ *ev* b # w'›
from *tick-setinterleaving-ev.hyps*[*OF* ‹$b \notin A$› ‹w *setinterleaves* ((✓ # u, v), A)› ‹$w' \leq w$›]
obtain u' v' **where** ‹$u' \leq$ ✓ # $u \wedge v' \leq v \wedge w'$ *setinterleaves* $((u', v'), A)$› **by** *blast*
hence ‹$u' \leq$ ✓ # $u \wedge$ *ev* b # $v' \leq$ *ev* b # $v \wedge t'$ *setinterleaves* $((u',$ *ev* b # $v'), A)$›
by (*cases* u') (*simp-all add*: ‹$b \notin A$› ‹$t' =$ *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› ‹$t' = [] \vee t' =$ ✓ # w'›
‹$w' \leq w$› ‹w *setinterleaves* $((u, v), A)$›
by (*cases* t') (*auto split*: *option.split-asm*)
from ‹$t' = [] \vee t' =$ ✓ # w'› **show** *?case*
proof (*elim disjE*)
from *Nil-setinterleaving-Nil* **show** ‹$t' = [] \Longrightarrow$ *?case*› **by** *blast*
next
assume ‹$t' =$ ✓ # w'›
from *tick-setinterleaving-tick.hyps*
[*OF* ‹w *setinterleaves* $((u, v), A)$› ‹$w' \leq w$›]
obtain u' v' **where** ‹$u' \leq u \wedge v' \leq v \wedge w'$ *setinterleaves* $((u', v'), A)$› **by** *blast*
hence ‹✓ # $u' \leq$ ✓ # $u \wedge$ ✓ # $v' \leq$ ✓ # $v \wedge$
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* :
‹$\square a \in A \to P\ a \downarrow n$ = (*case n of 0* $\Rightarrow$ *BOT* | *Suc m* $\Rightarrow$ $\square a {\in} A \to$ ($P\ a \downarrow m$))›
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* (λb. $\square a {\in} A \to f\ a\ b$)› **if** ‹$\bigwedge a$. $a \in A \Longrightarrow$ *non-destructive* ($f\ a$)›
proof −
have ‹$\square a {\in} A \to f\ a\ b = \square a {\in} A \to$ (*if* $a \in A$ *then* $f\ a\ b$ *else STOP*)› **for** b
by (*auto simp add*: *process-eq-spec T-Mprefix*)
moreover have ‹*constructive* (λb. $\square a {\in} A \to$ (*if* $a \in A$ *then* $f\ a\ b$ *else STOP*))›
proof (*rule constructive-comp-non-destructive*[*of* ‹λP. $\square a {\in} A \to P\ a$›])
show ‹*constructive* (λP. $\square a {\in} A \to P\ a$)›
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* (λb. $\square a {\in} A \to f\ a\ b$)› **by** *simp*
qed

9.6 Non Destructiveness

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

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

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

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

a›])
  show ‹$\mathit{non\text{-}destructive}\ (\lambda P.\ \Box a{\in}A.\ P\ a)$›
  by (*rule non-destructiveI*) (*simp add*: *restriction-process-GlobalChoice restriction-fun-def*)
 next
  show ‹$\mathit{non\text{-}destructive}\ (\lambda b\ a.\ \mathit{if}\ a \in A\ \mathit{then}\ f\ a\ b\ \mathit{else}\ \mathit{STOP})$›
  by (*simp add*: *non-destructive-fun-iff*, *intro allI non-destructive-if-then-else*)
   (*simp-all add*: ‹$\bigwedge a.\ a \in A \Longrightarrow \mathit{non\text{-}destructive}\ (f\ a)$› *non-destructiveI*)
 qed
 ultimately show ‹$\mathit{non\text{-}destructive}\ (\lambda b.\ \Box a{\in}A.\ f\ a\ b)$› **by** *simp*
qed

9.7 Examples

notepad begin
 fix $A\ B$ :: ‹$'b \Rightarrow 'a\ \mathit{set}$›
 define P :: ‹$'b \Rightarrow 'a\ \mathit{process}$›
  where ‹$P \equiv \upsilon\ X.\ (\lambda s.\ \Box\ a \in A\ s \to X\ s\ \Box\ (\Box\ b \in B\ s \to X\ s))$›
(**is** ‹$P \equiv \upsilon\ X.\ ?f\ X$›)
 have ‹$P = ?f\ P$›
  by (*unfold P-def*, *subst restriction-fix-eq*) *simp-all*

end

lemma ‹$\mathit{constructive}\ (\lambda X\ \sigma.\ \Box\ e \in f\ \sigma \to \Box\ \sigma' \in g\ \sigma\ e.\ X\ \sigma')$›
 by *simp*

lemma *length-le-T-restriction-process-iff-T* :
 ‹$\mathit{length}\ t \leq n \Longrightarrow t \in \mathcal{T}\ (P \downarrow n) \longleftrightarrow t \in \mathcal{T}\ P$›
 by (*auto simp add*: *T-restriction-process*)

lemma *restriction-adm-notin-T* [*simp*] : ‹$\mathit{adm}_{\downarrow}\ (\lambda a.\ t \notin \mathcal{T}\ a)$›
proof (*rule restriction-admI*)
 fix σ **and** Σ **assume** ‹$\sigma \ {-}{\downarrow}{\to}\ \Sigma$› ‹$\bigwedge n.\ t \notin \mathcal{T}\ (\sigma\ n)$›
 from ‹$\sigma \ {-}{\downarrow}{\to}\ \Sigma$› **obtain** *n0* **where** ‹$\forall k{\geq}n0.\ \Sigma \downarrow \mathit{length}\ t = \sigma\ k \downarrow \mathit{length}\ t$›
  by (*blast dest*: *restriction-tendstoD*)
 hence ‹$\forall k{\geq}n0.\ \mathcal{T}\ (\Sigma \downarrow \mathit{length}\ t) = \mathcal{T}\ (\sigma\ k \downarrow \mathit{length}\ t)$› **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 ‹$\bigwedge n.\ t \notin \mathcal{T}\ (\sigma\ n)$› **show** ‹$t \notin \mathcal{T}\ \Sigma$› **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** ‹σ −↓→ Σ› ‹$\bigwedge n.\ t \in \mathcal{T}\ (\sigma\ n)$›
  from ‹σ −↓→ Σ› **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*∈{*x*. *x* < *n*}. *fps-const* (*fps-nth x i*) ∗ *fps-X* ^ *i*) ≠
      ($\sum$ *i*∈{*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* ≠

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

end

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

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

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

lemma *restriction-eqI'* :
 ‹($\bigwedge i.\ i \leq n \Longrightarrow$ *fps-nth x i* = *fps-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* $\Rightarrow$'*a fps*› **assume** *h*:‹*restriction-chain* σ›
have *h'*:‹$\forall$ *n*. ($\sum$ *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 $\downarrow$ (*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* $\leq$ *n*› **by** *auto*
from *restriction-chain-def-ter*
[*THEN iffD1*, *OF* ‹*restriction-chain* σ›, *rule-format*, *OF* ‹*i* $\leq$ *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 $\Longrightarrow$ σ $-\downarrow\rightarrow$ *?*Σ›
proof −
have ($\downarrow$) (*Abs-fps* (λ*n*. *fps-nth* (σ (*Suc n*)) *n*)) = σ
using ‹$\bigwedge$*n*. *Abs-fps* (λ*n*. *fps-nth* (σ (*Suc n*)) *n*) $\downarrow$ *n* = σ *n*› **by** *force*
then show *?thesis*
by (*metis restriction-tendsto-restrictions*)
qed
qed

end