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*› ⟨*proof*⟩

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

lift-definition *restriction-type'* :: ‹*'a type'* ⇒ *nat* ⇒ *'a type'*›
  is ‹$\lambda x\ n.$ *if* $n = 0$ *then undefined else* x› ⟨*proof*⟩

instance ⟨*proof*⟩

end

lemma *restriction-type'-0-is-undefined* [*simp*] :
  ‹$x \downarrow 0 =$ *undefined*› **for** x :: ‹*'a type'*› ⟨*proof*⟩

instance *type'* :: (*type*) *restriction-space*
  ⟨*proof*⟩

lemma *restriction-tendsto-type'-iff* :
  ‹$\sigma \multimap\!\!\downarrow\!\!\to \Sigma \longleftrightarrow (\exists\, n0.\ \forall\, n \geq n0.\ \sigma\ n = \Sigma)$› **for** Σ :: ‹*'a type'*›
  ⟨*proof*⟩

lemma *restriction-chain-type'-iff* :
  ‹$chain_{\downarrow}\ \sigma \longleftrightarrow \sigma\ 0 = undefined \wedge (\forall\, n \geq Suc\ 0.\ \sigma\ n = \sigma\ (Suc\ 0))$›
  for σ :: ‹*nat* ⇒ *'a type'*›
  ⟨*proof*⟩

instance *type'* :: (*type*) *complete-restriction-space*
  ⟨*proof*⟩

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 ⟨*proof*⟩
end

lemma *restriction-bool-0-is-False* [*simp*] : ‹$b \downarrow 0 = False$›
⟨*proof*⟩

Restriction space instance for *bool*.

instance *bool* :: *restriction-space*
⟨*proof*⟩

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*
⟨*proof*⟩

instance *bool* :: *complete-restriction-space*
⟨*proof*⟩

lemma *restriction-cont-imp-restriction-adm* :
‹$cont_{\downarrow}\ P \Longrightarrow adm_{\downarrow}\ P$› **for** P :: ‹$'a$ :: *restriction-space* $\Rightarrow$ *bool*›
⟨*proof*⟩

lemma *restriction-compact-bool* : ‹$compact_{\downarrow}$ (*UNIV* :: *bool set*)›
⟨*proof*⟩

3 Naturals

Restriction instance for *nat*.

instantiation *nat* :: *restriction*
begin

definition *restriction-nat* :: ‹*nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat*›
where ‹$x \downarrow n \equiv$ *if* $x \leq n$ *then* x *else* n›

instance ⟨*proof*⟩

end

lemma *restriction-nat-0-is-0* [*simp*] : ‹$x \downarrow 0 = (0$ :: *nat*$)$›
⟨*proof*⟩

Restriction Space instance for *nat*.

instance *nat* :: *restriction-space*
⟨*proof*⟩

Constructive Suc

lemma *constructive-Suc* : ‹*constructive Suc*›
⟨*proof*⟩

Non too destructive pred

lemma *non-too-destructive-pred* : ‹*non-too-destructive nat.pred*›
⟨*proof*⟩

Restriction shift plus

lemma *restriction-shift-plus* : ‹*restriction-shift* ($\lambda x.\ x + k$) (*int k*)›
⟨*proof*⟩

lemma ‹*restriction-shift* ($\lambda x.\ k + x$) (*int k*)›
⟨*proof*⟩

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 ⟨*proof*⟩

end

instance *int* :: *restriction-space*
⟨*proof*⟩

lemma *restriction-int-0-is-0* [*simp*] : ‹$x \downarrow 0 = (0 :: int)$›
⟨*proof*⟩

Restriction shift plus

lemma *restriction-shift-on-pos-plus* : ‹*restriction-shift-on* ($\lambda x.\ x + k$) k $\{x.\ 0 \leq x\}$›
⟨*proof*⟩

lemma *restriction-shift-on-neg-minus* : ‹*restriction-shift-on* ($\lambda x.\ x - k$) k $\{x.\ x \leq 0\}$›
⟨*proof*⟩

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
  ⟨*proof*⟩

end

lemma *restriction-option-0-is-None* [*simp*] : ‹$x \downarrow 0 = None$›
  ⟨*proof*⟩

lemma *restriction-option-None* [*simp*] : ‹$None \downarrow n = None$›
  ⟨*proof*⟩

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

lemma *restriction-option-eq-None-iff* : ‹$x \downarrow n = None \longleftrightarrow n = 0 \lor x = None$›
  ⟨*proof*⟩

lemma *restriction-option-eq-Some-iff* : ‹$x \downarrow n = Some\ y \longleftrightarrow n \neq 0 \land x \neq None \land y = the\ x \downarrow n$›
  ⟨*proof*⟩

5.2 Restriction space option type

instance *option* :: (*restriction-space*) *restriction-space*
⟨*proof*⟩

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*⟩

lemma *non-destructive-Some* : ‹*non-destructive Some*›
  ⟨*proof*⟩

lemma *restriction-cont-Some* : ‹$cont_{\downarrow}$ (*Some* :: ′*a* :: *restriction-space* ⇒ ′*a option*)›
⟨*proof*⟩

instance *option* :: (*complete-restriction-space*) *complete-restriction-space*
⟨*proof*⟩

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 ⟨*proof*⟩

end

instance *list* :: (*type*) *order-restriction-space*
⟨*proof*⟩

lemma ‹*OFCLASS*(′*a list*, *restriction-space-class*)› ⟨*proof*⟩

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

notepad begin
⟨*proof*⟩

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*›
  ⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *size-le-imp-restriction-ex-tree-eq-self* :
  ‹*size x* ≤ *n* ⟹ *x* ↓ *n* = *x*› **for** *x* :: ‹$'a$ *ex-tree*›
  ⟨*proof*⟩

lemma *restriction-ex-tree-eqI* :
  ‹(⋀*i*. *x* ↓ *i* = *y* ↓ *i*) ⟹ *x* = *y*› **for** *x y* :: ‹$'a$ *ex-tree*›
  ⟨*proof*⟩

lemma *restriction-ex-tree-eqI-optimized* :
  ‹(⋀*i*. *i* ≤ *max* (*size x*) (*size y*) ⟹ *x* ↓ *i* = *y* ↓ *i*) ⟹ *x* = *y*› **for** *x y* :: ‹$'a$ *ex-tree*›
  ⟨*proof*⟩

instance *ex-tree* :: (*type*) *restriction-space*
  ⟨*proof*⟩

8 Decimals of a Number

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

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* ≤ *m then* σ *n else 0*› ⟨*proof*⟩

instance ⟨*proof*⟩

end

instance *decimals* :: (*zero*) *restriction-space*
⟨*proof*⟩

lemma *restriction-decimals-eq-iff* :
‹*x* ↓ *n* = *y* ↓ *n* ⟷ (∀ *i*≤*n*. *from-decimals x i* = *from-decimals y i*)›
⟨*proof*⟩

lemma *restriction-decimals-eqI* :
‹(⋀*i*. *i* ≤ *n* ⟹ *from-decimals x i* = *from-decimals y i*) ⟹ *x* ↓ *n* = *y* ↓ *n*›
⟨*proof*⟩

lemma *restriction-decimals-eqD* :
‹*x* ↓ *n* = *y* ↓ *n* ⟹ *i* ≤ *n* ⟹ *from-decimals x i* = *from-decimals y i*›
⟨*proof*⟩

This space is actually complete.

instance *decimals* :: (*zero*) *complete-restriction-space*
⟨*proof*⟩

typedef *nat-0-9* = ‹{*0*.. *9*::*nat*}›
  morphisms *from-nat-0-9 to-nat-0-9* ⟨*proof*⟩

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

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

lift-definition *zero-nat-0-9* :: *nat-0-9* **is** *0* ⟨*proof*⟩

instance ⟨*proof*⟩

end

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

lift-definition *one-nat-0-9* :: *nat-0-9* **is** *1* ⟨*proof*⟩

instance ⟨*proof*⟩

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*)›
  ⟨*proof*⟩

lemma *no-update-nth-decimal* [*simp*] :
  ‹*index* = *0* ⟹ *update-nth-decimal s index val* = *s*›
  ‹*9* < *val* ⟹ *update-nth-decimal s index val* = *s*›
  ⟨*proof*⟩

lemma *non-destructive-update-nth-decimal* : ‹*non-destructive update-nth-decimal*›
⟨*proof*⟩

lift-definition *shift-decimal-right* :: ‹*nat-0-9 decimals* ⇒ *nat-0-9 decimals*›
  is ‹λ*s n*. *case n of 0* ⇒ *to-nat-0-9 0* | *Suc n*′ ⇒ *from-decimals s n*′›
  ⟨*proof*⟩

lemma *constructive-shift-decimal-right* : ‹*constructive shift-decimal-right*›
⟨*proof*⟩

lift-definition *shift-decimal-left* :: ‹*nat-0-9 decimals* ⇒ *nat-0-9 decimals*›
is ‹λs n. *if* $n = 0$ *then to-nat-0-9 0 else from-decimals* s (*Suc* n)›
⟨*proof*⟩

lemma *non-too-destructive-shift-decimal-left* : ‹*non-too-destructive shift-decimal-left*›
⟨*proof*⟩

lemma *restriction-fix-shift-decimal-right* : ‹(υ x. *shift-decimal-right* x) = *to-decimals* (λ-. *0*)›
⟨*proof*⟩

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*⟩

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* $t \equiv$ ✓ $\notin$ *set* t›

definition *front-tickFree* :: ‹*'a trace* ⇒ *bool*› (‹*ftF*›)
where ‹*front-tickFree* $s \equiv s = []$ $\vee$ *tickFree* (*tl* (*rev* s))›

lemma *tickFree-Nil* [*simp*] : ‹*tF* []›
and *tickFree-Cons-iff* [*simp*] : ‹*tF* (a # t) $\longleftrightarrow$ $a \neq$ ✓ $\wedge$ *tF* t›

and *tickFree-append-iff* [*simp*] : ‹tF (s @ t) $\longleftrightarrow$ tF s $\wedge$ tF t›
and *tickFree-rev-iff* [*simp*] : ‹tF (*rev* t) $\longleftrightarrow$ tF t›
and *non-tickFree-tick* [*simp*] : ‹$\neg$ tF [✓]›
⟨*proof*⟩

lemma *tickFree-iff-is-map-ev* : ‹tF t $\longleftrightarrow$ ($\exists$ u. t = *map ev* u)›
⟨*proof*⟩

lemma *front-tickFree-Nil* [*simp*] : ‹ftF []›
and *front-tickFree-single*[*simp*] : ‹ftF [a]›
⟨*proof*⟩

lemma *tickFree-tl* : ‹tF s $\Longrightarrow$ tF (*tl* s)›
⟨*proof*⟩

lemma *non-tickFree-imp-not-Nil*: ‹$\neg$ tF s $\Longrightarrow$ $s \neq$ []›
⟨*proof*⟩

lemma *tickFree-butlast*: ‹tF s $\longleftrightarrow$ tF (*butlast* s) $\wedge$ ($s \neq$ [] $\longrightarrow$ *last* s $\neq$ ✓)›
⟨*proof*⟩

lemma *front-tickFree-iff-tickFree-butlast*: ‹ftF s $\longleftrightarrow$ tF (*butlast* s)›
⟨*proof*⟩

lemma *front-tickFree-Cons-iff*: ‹ftF (a # s) $\longleftrightarrow$ s = [] $\vee$ $a \neq$ ✓ $\wedge$ ftF s›
⟨*proof*⟩

lemma *front-tickFree-append-iff*:
‹ftF (s @ t) $\longleftrightarrow$ (*if* t = [] *then* ftF s *else* tF s $\wedge$ ftF t)›
⟨*proof*⟩

lemma *tickFree-imp-front-tickFree* [*simp*] : ‹tF s $\Longrightarrow$ ftF s›
⟨*proof*⟩

lemma *front-tickFree-charn*: ‹ftF s $\longleftrightarrow$ s = [] $\vee$ ($\exists$ a t. s = t @ [a] $\wedge$ tF t)›
⟨*proof*⟩

lemma *nonTickFree-n-frontTickFree*: ‹$\neg$ tF s $\Longrightarrow$ ftF s $\Longrightarrow$ $\exists$ t r. s = t @ [✓]›
⟨*proof*⟩

lemma *front-tickFree-dw-closed* : ‹ftF (s @ t) $\Longrightarrow$ ftF s›
⟨*proof*⟩

lemma *front-tickFree-append*: ‹$tF\ s \Longrightarrow ftF\ t \Longrightarrow ftF\ (s\ @\ t)$›
⟨*proof*⟩

lemma *tickFree-imp-front-tickFree-snoc*: ‹$tF\ s \Longrightarrow ftF\ (s\ @\ [a])$›
⟨*proof*⟩

lemma *front-tickFree-nonempty-append-imp*: ‹$ftF\ (t\ @\ r) \Longrightarrow r \neq [] \Longrightarrow tF\ t \wedge ftF\ r$›
⟨*proof*⟩

lemma *tickFree-map-ev* [*simp*] : ‹$tF\ (map\ ev\ t)$›
⟨*proof*⟩

lemma *tickFree-map-ev-comp* [*simp*] : ‹$tF\ (map\ (ev \circ f)\ t)$›
⟨*proof*⟩

lemma *front-tickFree-map-map-event-iff* :
‹$ftF\ (map\ (map\text{-}event\ f)\ t) \longleftrightarrow ftF\ t$›
⟨*proof*⟩

definition *is-process* :: ‹$'a\ trace\ set \Rightarrow bool$›
where ‹$is\text{-}process\ T \equiv [] \in T \wedge (\forall t.\ t \in T \longrightarrow ftF\ t) \wedge (\forall t\ u.\ t\ @\ u \in T \longrightarrow t \in T)$›

typedef $'a\ process$ = ‹$\{T :: 'a\ trace\ set.\ is\text{-}process\ T\}$›
morphisms *Traces to-process*
⟨*proof*⟩

setup-lifting *type-definition-process*

notation *Traces* (‹$\mathcal{T}$›)

lemma *is-process-inv* :
‹$[] \in \mathcal{T}\ P \wedge (\forall t.\ t \in \mathcal{T}\ P \longrightarrow ftF\ t) \wedge (\forall t\ u.\ t\ @\ u \in \mathcal{T}\ P \longrightarrow t \in \mathcal{T}\ P)$›
⟨*proof*⟩

lemma *Nil-elem-T* : ‹$[] \in \mathcal{T}\ P$›
and *front-tickFree-T* : ‹$t \in \mathcal{T}\ P \Longrightarrow ftF\ t$›
and *T-dw-closed* : ‹$t\ @\ u \in \mathcal{T}\ P \Longrightarrow t \in \mathcal{T}\ P$›
⟨*proof*⟩

lemma *process-eq-spec* : ‹$P = Q \longleftrightarrow \mathcal{T}\ P = \mathcal{T}\ Q$›
⟨*proof*⟩

9.2 First Processes

lift-definition *BOT* :: ‹$'a$ *process*› **is** ‹$\{t.\ ftF\ t\}$›
⟨*proof*⟩

lemma *T-BOT* : ‹$\mathcal{T}\ BOT = \{t.\ ftF\ t\}$›
⟨*proof*⟩

lift-definition *SKIP* :: ‹$'a$ *process*› **is** ‹$\{[], [\checkmark]\}$›
⟨*proof*⟩

lemma *T-SKIP* : ‹$\mathcal{T}\ SKIP = \{[], [\checkmark]\}$›
⟨*proof*⟩

lift-definition *STOP* :: ‹$'a$ *process*› **is** ‹$\{[]\}$›
⟨*proof*⟩

lemma *T-STOP* : ‹$\mathcal{T}\ STOP = \{[]\}$›
⟨*proof*⟩

lift-definition *Sup-processes* ::
‹$(nat \Rightarrow 'a\ process) \Rightarrow 'a\ process$› **is** ‹$\lambda\sigma.\ \bigcap i.\ \mathcal{T}\ (\sigma\ i)$›
⟨*proof*⟩

lemma *T-Sup-processes* : ‹$\mathcal{T}\ (Sup\text{-}processes\ \sigma) = (\bigcap i.\ \mathcal{T}\ (\sigma\ i))$›
⟨*proof*⟩

9.3 Instantiations

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

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

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

instance
⟨*proof*⟩

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*⟩

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* $u\}$›
⟨*proof*⟩

lemma *restriction-process-0* [*simp*] : ‹$P \downarrow 0 = BOT$›
⟨*proof*⟩

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›
⟨*proof*⟩

instance
⟨*proof*⟩

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

lemma ‹*OFCLASS* ($'a$ *process*, *restriction-space-class*)›
⟨*proof*⟩

end

lemma *restricted-Sup-processes-is* :
‹$(\lambda n.$ *Sup-processes* $\sigma \downarrow n) = \sigma$› **if** ‹*restriction-chain* σ›
⟨*proof*⟩

instance *process* :: (*type*) *complete-restriction-space*
⟨*proof*⟩

9.4 Operators

lift-definition *Choice* :: ‹$'a$ *process* ⇒ $'a$ *process* ⇒ $'a$ *process*› (**infixl** ‹□› *82*)
is ‹$\lambda P\ Q.\ \mathcal{T}\ P \cup \mathcal{T}\ Q$›
⟨*proof*⟩

lemma *T-Choice* : ‹$\mathcal{T}\ (P \square Q) = \mathcal{T}\ P \cup \mathcal{T}\ Q$›

⟨*proof*⟩

lift-definition *GlobalChoice* :: ‹['b set, 'b ⇒ 'a process] ⇒ 'a process›
is ‹$\lambda A\ P.\ if\ A = \{\}\ then\ \{[]\}\ else\ \bigcup a{\in}A.\ \mathcal{T}\ (P\ a)$›
⟨*proof*⟩

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

lemma *T-GlobalChoice* : ‹$\mathcal{T}\ (\Box a \in A.\ P\ a) = (if\ A = \{\}\ then\ \{[]\}\ else\ \bigcup a{\in}A.\ \mathcal{T}\ (P\ a))$›
⟨*proof*⟩

lift-definition *Seq* :: ‹'a process ⇒ 'a process ⇒ 'a process› (**infixl** ‹**;**› *74*)
is ‹$\lambda P\ Q.\ \{t \in \mathcal{T}\ P.\ tF\ t\} \cup \{t\ @\ u\ |t\ u.\ t\ @\ [✓] \in \mathcal{T}\ P \wedge u \in \mathcal{T}\ Q\}$›
⟨*proof*⟩

lemma *T-Seq* : ‹$\mathcal{T}\ (P\ \mathbf{;}\ Q) = \{t \in \mathcal{T}\ P.\ tF\ t\} \cup \{t\ @\ u\ |t\ u.\ t\ @\ [✓] \in \mathcal{T}\ P \wedge u \in \mathcal{T}\ Q\}$›
⟨*proof*⟩

lift-definition *Renaming* :: ‹['a process, 'a ⇒ 'b] ⇒ 'b process›
is ‹$\lambda P\ f.\ \{map\ (map\text{-}event\ f)\ u\ |u.\ u \in \mathcal{T}\ P\}$›
⟨*proof*⟩

lemma *T-Renaming* : ‹$\mathcal{T}\ (Renaming\ P\ f) = \{map\ (map\text{-}event\ f)\ u\ |u.\ u \in \mathcal{T}\ P\}$›
⟨*proof*⟩

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

syntax *-Mprefix* :: ‹[pttrn, 'a set, 'a process] ⇒ 'a process›
(‹(3□((-)/∈(-))/ → (-))› [78,78,77] 77)
syntax-consts *-Mprefix* ⇌ *Mprefix*
translations $\Box a{\in}A \to P$ ⇌ *CONST Mprefix A* ($\lambda a.\ P$)

lemma *T-Mprefix* : ‹$\mathcal{T}\ (\Box a \in A \to P\ a) = insert\ []\ \{ev\ a\ \#\ t\ |a\ t.\ a \in A \wedge t \in \mathcal{T}\ (P\ a)\}$›
⟨*proof*⟩

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

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

| *ev-setinterleaving-ev* :
‹*setinterleaving* (*ev a* # *u*, *A*, *ev b* # *v*) =
(*if* $a \in A$
then *if* $b \in A$
then *if* $a = b$
then {*ev a* # *t* |*t*. $t \in$ *setinterleaving* (*u*, *A*, *v*)}
else {}
else {*ev b* # *t* |*t*. $t \in$ *setinterleaving* (*ev a* # *u*, *A*, *v*)}
else *if* $b \in A$ *then* {*ev a* # *t* |*t*. $t \in$ *setinterleaving* (*u*, *A*, *ev b* # *v*)}
else {*ev a* # *t* |*t*. $t \in$ *setinterleaving* (*u*, *A*, *ev b* # *v*)} ∪
{*ev b* # *t* |*t*. $t \in$ *setinterleaving* (*ev a* # *u*, *A*, *v*)})›
| *ev-setinterleaving-tick* :
‹*setinterleaving* (*ev a* # *u*, *A*, ✓ # *v*) =
(*if* $a \in A$ *then* {} *else* {*ev a* # *t* |*t*. $t \in$ *setinterleaving* (*u*, *A*, ✓ # *v*)})›
| *tick-setinterleaving-ev* :
‹*setinterleaving* (✓ # *u*, *A*, *ev b* # *v*) =
(*if* $b \in A$ *then* {} *else* {*ev b* # *t* |*t*. $t \in$ *setinterleaving* (✓ # *u*, *A*, *v*)})›
| *tick-setinterleaving-tick* :
‹*setinterleaving* (✓ # *u*, *A*, ✓ # *v*) = {✓ # *t* |*t*. $t \in$ *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, [])})
            | ✓ ⇒ {})›
⟨proof⟩

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)})
            | ✓ ⇒ {})›
⟨proof⟩

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 ∈ A then {}
                    else {ev a # t |t. t ∈ setinterleaving (u, A, ✓ # v)})
           | ✓ ⇒
   (case f of ev b ⇒   if b ∈ A then {}
                       else {ev b # t| t. t ∈ setinterleaving (✓ # u, A, v)}
            | ✓ ⇒ {✓ # t |t. t ∈ setinterleaving (u, A, v)}))›
⟨proof⟩

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›
⟨proof⟩

lemma *setinterleaves-tickFree-imp* :
‹tF u ∨ tF v ⟹ t setinterleaves ((u, v), A) ⟹ tF t ∧ tF u ∧ tF v›
⟨proof⟩

lemma *setinterleaves-NilL-iff* :
‹t setinterleaves (([], v), A) ⟷
tF v ∧ set v ∩ ev ‘ A = {} ∧ t = map ev (map of-ev v)›
⟨proof⟩

lemma *setinterleaves-NilR-iff* :
‹t setinterleaves ((u, []), A) ⟷
tF u ∧ set u ∩ ev ‘ A = {} ∧ t = map ev (map of-ev u)›
⟨proof⟩

lemma *Nil-setinterleaves* :
‹[] setinterleaves ((u, v), A) ⟹ u = [] ∧ v = []›
⟨proof⟩

lemma *front-tickFree-setinterleaves-iff* :
‹t setinterleaves ((u, v), A) ⟹ ftF t ⟷ ftF u ∧ ftF v›
⟨proof⟩

lemma *setinterleaves-snoc-notinL* :
‹t setinterleaves ((u, v), A) ⟹ a ∉ A ⟹
t @ [ev a] setinterleaves ((u @ [ev a], v), A)›
⟨proof⟩

lemma *setinterleaves-snoc-notinR* :
‹t setinterleaves ((u, v), A) ⟹ a ∉ A ⟹
t @ [ev a] setinterleaves ((u, v @ [ev a]), A)›
⟨proof⟩

lemma *setinterleaves-snoc-inside* :

‹t *setinterleaves* $((u, v), A) \Longrightarrow a \in A \Longrightarrow$
t @ $[ev\ a]$ *setinterleaves* $((u$ @ $[ev\ a], v$ @ $[ev\ a]), A)$›
⟨*proof*⟩

lemma *setinterleaves-snoc-tick* :
‹t *setinterleaves* $((u, v), A) \Longrightarrow t$ @ [✓] *setinterleaves* $((u$ @ [✓], v @ [✓]), $A)$›
⟨*proof*⟩

lemma *Cons-tick-setinterleavesE* :
‹✓ # t *setinterleaves* $((u, v), A) \Longrightarrow$
$(\bigwedge u'\ v'\ r\ s.\ [\![u =$ ✓ # $u'; v =$ ✓ # $v'; t$ *setinterleaves* $((u', v'), A)]\!] \Longrightarrow thesis) \Longrightarrow thesis$›
⟨*proof*⟩

lemma *Cons-ev-setinterleavesE* :
‹$ev\ a$ # t *setinterleaves* $((u, v), A) \Longrightarrow$
$(\bigwedge u'.\ a \notin A \Longrightarrow u = ev\ a$ # $u' \Longrightarrow t$ *setinterleaves* $((u', v), A) \Longrightarrow thesis) \Longrightarrow$
$(\bigwedge v'.\ a \notin A \Longrightarrow v = ev\ a$ # $v' \Longrightarrow t$ *setinterleaves* $((u, v'), A) \Longrightarrow thesis) \Longrightarrow$
$(\bigwedge u'\ v'.\ a \in A \Longrightarrow u = ev\ a$ # $u' \Longrightarrow v = ev\ a$ # $v' \Longrightarrow$
t *setinterleaves* $((u', v'), A) \Longrightarrow thesis) \Longrightarrow thesis$›
⟨*proof*⟩

lemma *rev-setinterleaves-rev-rev-iff* :
‹$rev\ t$ *setinterleaves* $((rev\ u, rev\ v), A)$
$\longleftrightarrow t$ *setinterleaves* $((u, v), A)$›
⟨*proof*⟩

lemma *setinterleaves-preserves-ev-notin-set* :
‹$[\![ev\ a \notin set\ u; ev\ a \notin set\ v; t$ *setinterleaves* $((u, v), A)]\!] \Longrightarrow ev\ a \notin set\ t$›
⟨*proof*⟩

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*⟩

lemma *ev-notin-both-sets-imp-empty-setinterleaving* :
‹$[\![ev\ a \in set\ u \wedge ev\ a \notin set\ v \vee ev\ a \notin set\ u \wedge ev\ a \in set\ v; a \in A]\!]$

$\Longrightarrow$
setinterleaving $(u,\ A,\ v) = \{\}$›
⟨*proof*⟩

lemma *append-setinterleaves-imp* :
‹t *setinterleaves* $((u,\ v),\ A) \Longrightarrow t' \leq t \Longrightarrow$
$\exists\, u' \leq u.\ \exists\, v' \leq v.\ t'$ *setinterleaves* $((u',\ v'),\ A)$›
⟨*proof*⟩

lift-definition *Sync* :: ‹$'a$ *process* $\Rightarrow$ $'a$ *set* $\Rightarrow$ $'a$ *process* $\Rightarrow$ $'a$ *process*›
(‹(*3*(- ⟦-⟧/ -))› [*70*, *0*, *71*] *70*)
is ‹$\lambda P\ A\ Q.\ \{t.\ \exists\, t\text{-}P\ t\text{-}Q.\ t\text{-}P \in \mathcal{T}\ P \wedge t\text{-}Q \in \mathcal{T}\ Q \wedge t$ *setinterleaves* $((t\text{-}P,\ t\text{-}Q),\ A)\}$›
⟨*proof*⟩

lemma *T-Sync* :
‹$\mathcal{T}\ (P\ [\![A]\!]\ Q) = \{t.\ \exists\, t\text{-}P\ t\text{-}Q.\ t\text{-}P \in \mathcal{T}\ P \wedge t\text{-}Q \in \mathcal{T}\ Q \wedge t$ *setinterleaves* $((t\text{-}P,\ t\text{-}Q),\ A)\}$›
⟨*proof*⟩

lift-definition *Interrupt* :: ‹$'a$ *process* $\Rightarrow$ $'a$ *process* $\Rightarrow$ $'a$ *process*› (**infixl** ‹$\triangle$› *81*)
is ‹$\lambda P\ Q.\ \mathcal{T}\ P \cup \{t\ @\ u\ |t\ u.\ t \in \mathcal{T}\ P \wedge tF\ t \wedge u \in \mathcal{T}\ Q\}$›
⟨*proof*⟩

9.5 Constructiveness

lemma *restriction-process-Mprefix* :
‹$\square a \in A \rightarrow P\ a \downarrow n = ($*case* n *of* $0 \Rightarrow BOT \mid Suc\ m \Rightarrow \square a{\in}A \rightarrow (P\ a \downarrow m))$›
⟨*proof*⟩

lemma *constructive-Mprefix* [*simp*] :
‹*constructive* $(\lambda b.\ \square a{\in}A \rightarrow f\ a\ b)$› **if** ‹$\bigwedge a.\ a \in A \Longrightarrow$ *non-destructive* $(f\ a)$›
⟨*proof*⟩

9.6 Non Destructiveness

lemma *non-destructive-Choice* [*simp*] :
‹*non-destructive* $(\lambda x.\ f\ x \ \square\ g\ x)$›
if ‹*non-destructive* f› ‹*non-destructive* g›
for $f\ g$ :: ‹$'a$ :: *restriction* $\Rightarrow$ $'b$ *process*›
⟨*proof*⟩

lemma *restriction-process-GlobalChoice* :
‹$\square a \in A.\ P\ a \downarrow n = (if\ A = \{\}\ then\ case\ n\ of\ 0 \Rightarrow BOT \mid Suc\ m \Rightarrow STOP\ else\ \square a \in A.\ (P\ a \downarrow n))$›
⟨*proof*⟩

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*⟩

9.7 Examples

notepad begin
⟨*proof*⟩

end

lemma ‹*constructive* $(\lambda X\ \sigma.\ \square\ e \in f\ \sigma \to \square\ \sigma' \in g\ \sigma\ e.\ X\ \sigma')$›
⟨*proof*⟩

lemma *length-le-T-restriction-process-iff-T* :
‹$length\ t \le n \Longrightarrow t \in \mathcal{T}\ (P \downarrow n) \longleftrightarrow t \in \mathcal{T}\ P$›
⟨*proof*⟩

lemma *restriction-adm-notin-T* [*simp*] : ‹$adm_{\downarrow}\ (\lambda a.\ t \notin \mathcal{T}\ a)$›
⟨*proof*⟩

lemma *restriction-adm-in-T* [*simp*] : ‹$adm_{\downarrow}\ (\lambda a.\ t \in \mathcal{T}\ a)$›
⟨*proof*⟩

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*:‹$(n{::}nat) \le m \Longrightarrow \{..{<}m\} \cap \{i.\ i < n\} = \{i.\ i{<}n\}$›
⟨*proof*⟩

lemma *exist-noneq*:‹$x \neq y \Longrightarrow$
$\exists\, n.\ (\sum i{\in}\{x.\ x < n\}.\ \textit{fps-const}\ (\textit{fps-nth}\ x\ i) * \textit{fps-X}\ \hat{}\ i) \neq$
$(\sum i{\in}\{x.\ x < n\}.\ \textit{fps-const}\ (\textit{fps-nth}\ y\ i) * \textit{fps-X}\ \hat{}\ i)$› **for** $x\ y$::‹$'a$ *fps*›
⟨*proof*⟩

instance
⟨*proof*⟩

end

lemma *fps-sum-rep-nthb*: $\textit{fps-nth}\ (\sum i{<}m.\ \textit{fps-const}(a\ i){*}\textit{fps-X}\hat{}i)\ n = (\textit{if}\ n < m\ \textit{then}\ a\ n\ \textit{else}\ 0)$
⟨*proof*⟩

lemma *restriction-eq-iff* :‹$a{\downarrow}n = b{\downarrow}n \longleftrightarrow (\forall\, i{<}n.\ \textit{fps-nth}\ a\ i = \textit{fps-nth}\ b\ i)$›
⟨*proof*⟩

lemma *restriction-eqI* :
‹$(\bigwedge i.\ i < n \Longrightarrow \textit{fps-nth}\ x\ i = \textit{fps-nth}\ y\ i) \Longrightarrow x \downarrow n = y \downarrow n$›
⟨*proof*⟩

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

instantiation *fps* :: (*comm-ring-1*) *complete-restriction-space*
begin
instance
⟨*proof*⟩

end