16. RSL: The Raise Specification Language

16.1. Type Expressions

- Type expressions are expressions whose value are type, that is,
- possibly infinite sets of values (of "that" type).

16.1.1. Atomic Types

- Atomic types have (atomic) values.
- That is, values which we consider to have no proper constituent (sub-)values,
- i.e., cannot, to us, be meaningfully "taken apart".

type

[1] **Bool** **true**, **false**
[2] **Int** ... , −2, −2, 0, 1, 2, ...
[3] **Nat** 0, 1, 2, ...
[4] **Real** ..., −5.43, −1.0, 0.0, 1.23· · · , 2,7182· · · , 3,1415· · · , 4.56, ...
[5] **Char** ”a”, ”b”, ..., ”0”, ...
[6] **Text** ”abracadabra”

16.1.2. Composite Types

- Composite types have composite values.
  - That is, values which we consider to have proper constituent (sub-)values,
  - i.e., can be meaningfully "taken apart".
- There are two ways of expressing composite types:
  - either explicitly, using concrete type expressions,
  - or implicitly, using sorts (i.e., abstract types) and observer functions.

16.1.2.1 Concrete Composite Types

[7] A-set
[8] A-infset
[9] A × B × ... × C
[10] A*
[11] A^ω
[12] A
[13] A
[14] A → B
[15] (A)
[16] A | B | ... | C
[17] mk_id(sel_a:A,...,sel_b:B)
[18] sel_a:A ... sel_b:B

16.1.2.2 Sorts and Observer Functions

type
A, B, C, ..., D
value
obs_B: A → B, obs_C: A → C, ..., obs_D: A → D

- The above expresses
  - that values of type A
  - are composed from at least three values —
  - and these are of type B, C, ..., and D.
- A concrete type definition corresponding to the above
  - presupposing material of the next section

type
B, C, ..., D
A = B × C × ... × D

16.2. Type Definitions

16.2.1. Concrete Types

- Types can be concrete
- in which case the structure of the type is specified by type expressions:

type
A = Type_expr

- Schematic type definitions:

[1] Type_name = Type_expr /∗ without |s or subtypes ∗/
[2] Type_name = Type_expr_1 | Type_expr_2 | ... | Type_expr_n
[3] Type_name ==
mk_id_1(s_a1:Type_name_a1,...,s_ai:Type_name_ai) |
... |
mk_id_n(s_z1:Type_name_z1,...,s_zk:Type_name_zk)
[4] Type_name :: sel_a:Type_name_a ... sel_z:Type_name_z
[5] Type_name = {| v:Type_name′ · $\mathcal{P}$(v) |}

- where a form of [2–3] is provided by combining the types:

Type_name = A | B | ... | Z
A == mk_id_1(s_a1:A_1,...,s_ai:A_i)
B == mk_id_2(s_b1:B_1,...,s_bj:B_j)
...
Z == mk_id_n(s_z1:Z_1,...,s_zk:Z_k)

axiom
∀ a1:A_1, a2:A_2, ..., ai:Ai ·
  s_a1(mk_id_1(a1,a2,...,ai))=a1 ∧ s_a2(mk_id_1(a1,a2,...,ai))=a2 ∧
  ... ∧ s_ai(mk_id_1(a1,a2,...,ai))=ai ∧
∀ a:A · **let** mk_id_1(a1′,a2′,...,ai′) = a **in**
  a1′ = s_a1(a) ∧ a2′ = s_a2(a) ∧ ... ∧ ai′ = s_ai(a) **end**

16.2.2. Subtypes

- In RSL, each type represents a set of values. Such a set can be delimited by means of predicates.
- The set of values b which have type B and which satisfy the predicate $\mathcal{P}$, constitute the subtype A:

type

$$A = \{|\ b{:}B \cdot \mathcal{P}(b)\ |\}$$

16.2.3. Sorts — Abstract Types

- Types can be (abstract) sorts
- in which case their structure is not specified:

type
A, B, ..., C

16.3. The RSL Predicate Calculus

16.3.1. Propositional Expressions

- Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values (**true** or **false** [or **chaos**]).
- Then:

false, **true**
a, b, ..., c ∼a, a∧b, a∨b, a⇒b, a=b, a≠b

- are propositional expressions having Boolean values.
- ∼, ∧, ∨, ⇒, = and ≠ are Boolean connectives (i.e., operators).
- They can be read as: *not*, *and*, *or*, *if then* (or *implies*), *equal* and *not equal*.

16.3.2. Simple Predicate Expressions

- Let identifiers (or propositional expressions) a, b, ..., c designate Boolean values,
- let x, y, ..., z (or term expressions) designate non-Boolean values
- and let i, j, . . ., k designate number values,
- then:

false, **true**
a, b, ..., c
∼a, a∧b, a∨b, a⇒b, a=b, a≠b
x=y, x≠y,
i<j, i≤j, i≥j, i≠j, i≥j, i>j

- are simple predicate expressions.

16.3.3. Quantified Expressions

- Let X, Y, ..., C be type names or type expressions,
- and let $\mathcal{P}(x)$, $\mathcal{Q}(y)$ and $\mathcal{R}(z)$ designate predicate expressions in which x, y and z are free.
- Then:

$\forall$ x:X · $\mathcal{P}(x)$
$\exists$ y:Y · $\mathcal{Q}(y)$
$\exists$! z:Z · $\mathcal{R}(z)$

- are quantified expressions — also being predicate expressions.

16.4. Concrete RSL Types: Values and Operations

16.4.1. Arithmetic

type
  Nat, Int, Real
value
  $+,-,*$: $\mathbf{Nat}\times\mathbf{Nat}\to\mathbf{Nat} \mid \mathbf{Int}\times\mathbf{Int}\to\mathbf{Int} \mid \mathbf{Real}\times\mathbf{Real}\to\mathbf{Real}$
  $/$: $\mathbf{Nat}\times\mathbf{Nat}\xrightarrow{\sim}\mathbf{Nat} \mid \mathbf{Int}\times\mathbf{Int}\xrightarrow{\sim}\mathbf{Int} \mid \mathbf{Real}\times\mathbf{Real}\xrightarrow{\sim}\mathbf{Real}$
  $<,\leq,=,\neq,\geq,>$ $(\mathbf{Nat}|\mathbf{Int}|\mathbf{Real}) \to (\mathbf{Nat}|\mathbf{Int}|\mathbf{Real})$

16.4.2. Set Expressions

16.4.2.1 Set Enumerations

Let the below a's denote values of type A, then the below designate simple set enumerations:

$\{\{\}, \{a\}, \{e_1,e_2,...,e_n\}, ...\} \in$ A-**set**
$\{\{\}, \{a\}, \{e_1,e_2,...,e_n\}, ..., \{e_1,e_2,...\}\} \in$ A-**infset**

16.4.2.2 Set Comprehension

- The expression, last line below, to the right of the ≡, expresses set comprehension.
- The expression "builds" the set of values satisfying the given predicate.
- It is abstract in the sense that it does not do so by following a concrete algorithm.

type
  A, B
  P = A → **Bool**
  Q = A $\xrightarrow{\sim}$ B
value
  comprehend: A-**infset** × P × Q → B-**infset**
  comprehend(s,P,Q) ≡ { Q(a) | a:A · a ∈ s ∧ P(a)}

16.4.3. Cartesian Expressions

16.4.3.1 Cartesian Enumerations

- Let e range over values of Cartesian types involving A, B, ..., C,
- then the below expressions are simple Cartesian enumerations:

type
  A, B, ..., C
  A × B × ... × C
value
  (e1,e2,...,en)

16.4.4. List Expressions

16.4.4.1 List Enumerations

- Let a range over values of type A,
- then the below expressions are simple list enumerations:

$$\{\langle\rangle, \langle e\rangle, ..., \langle e1,e2,...,en\rangle, ...\} \in A^*$$
$$\{\langle\rangle, \langle e\rangle, ..., \langle e1,e2,...,en\rangle, ..., \langle e1,e2,...,en,...\rangle, ...\} \in A^\omega$$

$$\langle\ a\_i\ ..\ a\_j\ \rangle$$

- The last line above assumes a_i and a_j to be integer-valued expressions.
- It then expresses the set of integers from the value of e_i to and including the value of e_j.
- If the latter is smaller than the former, then the list is empty.

16.4.4.2 List Comprehension

- The last line below expresses list comprehension.

type
  A, B, P = A → **Bool**, Q = A $\xrightarrow{\sim}$ B
value
  comprehend: A^{ω} × P × Q $\xrightarrow{\sim}$ B^{ω}
  comprehend(l,P,Q) ≡
    ⟨ Q(l(i)) | i **in** ⟨1..**len** l⟩ · P(l(i))⟩

16.4.5. Map Expressions

16.4.5.1 Map Enumerations

- Let (possibly indexed) u and v range over values of type $T1$ and $T2$, respectively,
- then the below expressions are simple map enumerations:

type
  T1, T2
  M = T1 $\overrightarrow{m}$ T2
value
  u,u1,u2,...,un:T1, v,v1,v2,...,vn:T2
  [ ], [u↦v], ..., [u1↦v1,u2↦v2,...,un↦vn] ∀ ∈ M

16.4.5.2 Map Comprehension

- The last line below expresses map comprehension:

type
  U, V, X, Y
  $M = U \xrightarrow[m]{} V$
  $F = U \xrightarrow{\sim} X$
  $G = V \xrightarrow{\sim} Y$
  $P = U \to \mathbf{Bool}$
value
  comprehend: $M \times F \times G \times P \to (X \xrightarrow[m]{} Y)$
  comprehend(m,F,G,P) $\equiv$
    $[\ F(u) \mapsto G(m(u)) \mid u{:}U \cdot u \in \mathbf{dom}\ m \wedge P(u)\ ]$

16.4.6. Set Operations

16.4.6.1 Set Operator Signatures

value

209 $\in$: A × A-**infset** → **Bool**
210 $\notin$: A × A-**infset** → **Bool**
211 $\cup$: A-**infset** × A-**infset** → A-**infset**
212 $\cup$: (A-**infset**)-**infset** → A-**infset**
213 $\cap$: A-**infset** × A-**infset** → A-**infset**
214 $\cap$: (A-**infset**)-**infset** → A-**infset**
215 $\setminus$: A-**infset** × A-**infset** → A-**infset**
216 $\subset$: A-**infset** × A-**infset** → **Bool**
217 $\subseteq$: A-**infset** × A-**infset** → **Bool**
218 $=$: A-**infset** × A-**infset** → **Bool**
219 $\neq$: A-**infset** × A-**infset** → **Bool**
220 **card**: A-**infset** $\xrightarrow{\sim}$ **Nat**

16.4.6.2 Set Examples

examples

$a \in \{a,b,c\}$
$a \notin \{\}$, $a \notin \{b,c\}$
$\{a,b,c\} \cup \{a,b,d,e\} = \{a,b,c,d,e\}$
$\cup\{\{a\},\{a,b\},\{a,d\}\} = \{a,b,d\}$
$\{a,b,c\} \cap \{c,d,e\} = \{c\}$
$\cap\{\{a\},\{a,b\},\{a,d\}\} = \{a\}$
$\{a,b,c\} \setminus \{c,d\} = \{a,b\}$
$\{a,b\} \subset \{a,b,c\}$
$\{a,b,c\} \subseteq \{a,b,c\}$
$\{a,b,c\} = \{a,b,c\}$
$\{a,b,c\} \neq \{a,b\}$
card $\{\} = 0$, **card** $\{a,b,c\} = 3$

16.4.6.3 Informal Explication

209. $\in$: The membership operator expresses that an element is a member of a set.
210. $\notin$: The nonmembership operator expresses that an element is not a member of a set.
211. $\cup$: The infix union operator. When applied to two sets, the operator gives the set whose members are in either or both of the two operand sets.
212. $\cup$: The distributed prefix union operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
213. $\cap$: The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
214. $\cap$: The prefix distributed intersection operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.

215. \: The set complement (or set subtraction) operator. When applied to two sets, the operator gives the set whose members are those of the left operand set which are not in the right operand set.

216. $\subseteq$: The proper subset operator expresses that all members of the left operand set are also in the right operand set.

217. $\subset$: The proper subset operator expresses that all members of the left operand set are also in the right operand set, and that the two sets are not identical.

218. $=$: The equal operator expresses that the two operand sets are identical.

219. $\neq$: The nonequal operator expresses that the two operand sets are *not* identical.

220. **card**: The cardinality operator gives the number of elements in a finite set.

16.4.6.4 Set Operator Definitions

```
value
  s′ ∪ s″ ≡ { a | a:A · a ∈ s′ ∨ a ∈ s″ }
  s′ ∩ s″ ≡ { a | a:A · a ∈ s′ ∧ a ∈ s″ }
  s′ \ s″ ≡ { a | a:A · a ∈ s′ ∧ a ∉ s″ }
  s′ ⊆ s″ ≡ ∀ a:A · a ∈ s′ ⇒ a ∈ s″
  s′ ⊂ s″ ≡ s′ ⊆ s″ ∧ ∃ a:A · a ∈ s″ ∧ a ∉ s′
  s′ = s″ ≡ ∀ a:A · a ∈ s′ ≡ a ∈ s″ ≡ s⊆s′ ∧ s′⊆s
  s′ ≠ s″ ≡ s′ ∩ s″ ≠ {}
  card s ≡
    if s = {} then 0 else
    let a:A · a ∈ s in 1 + card (s \ {a}) end end
    pre s /* is a finite set */
  card s ≡ chaos /* tests for infinity of s */
```

16.4.7. Cartesian Operations

```
type
  A, B, C
  g0: G0 = A × B × C
  g1: G1 = ( A × B × C )
  g2: G2 = ( A × B ) × C
  g3: G3 = A × ( B × C )

value
  va:A, vb:B, vc:C, vd:D
  (va,vb,vc):G0,
  (va,vb,vc):G1
  ((va,vb),vc):G2
  (va3,(vb3,vc3)):G3

decomposition expressions
  let (a1,b1,c1) = g0,
      (a1′,b1′,c1′) = g1 in .. end
  let ((a2,b2),c2) = g2 in .. end
  let (a3,(b3,c3)) = g3 in .. end
```

16.4.8. List Operations

16.4.8.1 List Operator Signatures

value

hd: $\mathrm{A}^{\omega} \xrightarrow{\sim} \mathrm{A}$

tl: $\mathrm{A}^{\omega} \xrightarrow{\sim} \mathrm{A}^{\omega}$

len: $\mathrm{A}^{\omega} \xrightarrow{\sim} \mathbf{Nat}$

inds: $\mathrm{A}^{\omega} \rightarrow \mathbf{Nat\text{-}infset}$

elems: $\mathrm{A}^{\omega} \rightarrow \mathrm{A}\text{-}\mathbf{infset}$

.(.): $\mathrm{A}^{\omega} \times \mathbf{Nat} \xrightarrow{\sim} \mathrm{A}$

$\widehat{\ }$: ~~$\mathrm{A}^{\omega} \times \mathrm{A}^{\omega} \rightarrow \mathrm{A}^{\omega}$~~ **Bobl**

16.4.8.2 List Operation Examples

examples

hd⟨a1,a2,...,am⟩=a1

tl⟨a1,a2,...,am⟩=⟨a2,...,am⟩

len⟨a1,a2,...,am⟩=m

inds⟨a1,a2,...,am⟩={1,2,...,m}

elems⟨a1,a2,...,am⟩={a1,a2,...,am}

⟨a1,a2,...,am⟩(i)=ai

⟨a,b,c⟩⌢⟨a,b,d⟩ = ⟨a,b,c,a,b,d⟩

⟨a,b,c⟩=⟨a,b,c⟩

⟨a,b,c⟩ ≠ ⟨a,b,d⟩

16.4.8.3 Informal Explication

- **hd**: Head gives the first element in a nonempty list.
- **tl**: Tail gives the remaining list of a nonempty list when Head is removed.
- **len**: Length gives the number of elements in a finite list.
- **inds**: Indices give the set of indices from **1** to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems**: Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$: Indexing with a natural number, i larger than 0, into a list ℓ having a number of elements larger than or equal to i, gives the ith element of the list.

- $\widehat{}$: Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.
- $=$: The equal operator expresses that the two operand lists are identical.
- $\neq$: The nonequal operator expresses that the two operand lists are *not* identical.

The operations can also be defined as follows:

16.4.8.4 List Operator Definitions

```
value
  is_finite_list: A^ω → Bool

  len q ≡
    case is_finite_list(q) of
      true → if q = ⟨⟩ then 0 else 1 + len tl q end,
      false → chaos end

  inds q ≡
    case is_finite_list(q) of
      true → { i | i:Nat · 1 ≤ i ≤ len q },
      false → { i | i:Nat · i≠0 } end

  elems q ≡ { q(i) | i:Nat · i ∈ inds q }
```

```
q(i) ≡
  if i=1
    then
      if q≠⟨⟩
        then let a:A,q′:Q · q=⟨a⟩⌢q′ in a end
        else chaos end
    else q(i−1) end

fq ⌢ iq ≡
    ⟨ if 1 ≤ i ≤ len fq then fq(i) else iq(i − len fq) end
     | i:Nat · if len iq≠chaos then i ≤ len fq+len end ⟩
  pre is_finite_list(fq)

iq′ = iq″ ≡
  inds iq′ = inds iq″ ∧ ∀ i:Nat · i ∈ inds iq′ ⇒ iq′(i) = iq″(i)

iq′ ≠ iq″ ≡ ∼(iq′ = iq″)
```

16.4.9. Map Operations

16.4.9.1 Map Operator Signatures and Map Operation Examples

value

$$\mathrm{m(a)}: \mathrm{M} \to \mathrm{A} \xrightarrow{\sim} \mathrm{B},\ \mathrm{m(a)} = \mathrm{b}$$

dom: M → A-**infset** [domain of map]
  dom [a1↦b1,a2↦b2,...,an↦bn] = {a1,a2,...,an}

rng: M → B-**infset** [range of map]
  rng [a1↦b1,a2↦b2,...,an↦bn] = {b1,b2,...,bn}

†: M × M → M [override extension]
  [a↦b,a′↦b′,a″↦b″] † [a′↦b″,a″↦b′] = [a↦b,a′↦b″,a″↦b′]

$\cup$: M $\times$ M $\to$ M [merge $\cup$]
[a$\mapsto$b,a′$\mapsto$b′,a″$\mapsto$b″] $\cup$ [a‴$\mapsto$b‴] = [a$\mapsto$b,a′$\mapsto$b′,a″$\mapsto$b″,a‴$\mapsto$b‴]

$\backslash$: M $\times$ A-**infset** $\to$ M [restriction by]
[a$\mapsto$b,a′$\mapsto$b′,a″$\mapsto$b″]$\backslash${a} = [a′$\mapsto$b′,a″$\mapsto$b″]

/: M $\times$ A-**infset** $\to$ M [restriction to]
[a$\mapsto$b,a′$\mapsto$b′,a″$\mapsto$b″]/{a′,a″} = [a′$\mapsto$b′,a″$\mapsto$b″]

=,$\neq$: M $\times$ M $\to$ **Bool**

$\circ$: (A $\overrightarrow{m}$ B) $\times$ (B $\overrightarrow{m}$ C) $\to$ (A $\overrightarrow{m}$ C) [composition]
[a$\mapsto$b,a′$\mapsto$b′] $\circ$ [b$\mapsto$c,b′$\mapsto$c′,b″$\mapsto$c″] = [a$\mapsto$c,a′$\mapsto$c′]

16.4.9.2 Map Operation Explication

- $m(a)$: Application gives the element that a maps to in the map m.
- **dom**: Domain/Definition Set gives the set of values which *maps to* in a map.
- **rng**: Range/Image Set gives the set of values which *are mapped to* in a map.
- †: Override/Extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some "pairings" of the right operand map.
- ∪: Merge. When applied to two operand maps, it gives a merge of these maps.
- \: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements that are not in the right operand set.

- /: Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements of the right operand set.
- =: The equal operator expresses that the two operand maps are identical.
- ≠: The nonequal operator expresses that the two operand maps are *not* identical.
- °: Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map, m_1, to the range elements of the right operand map, m_2, such that if a is in the definition set of m_1 and maps into b, and if b is in the definition set of m_2 and maps into c, then a, in the composition, maps into c.

16.4.9.3 Map Operation Redefinitions

```
value
  rng m ≡ { m(a) | a:A • a ∈ dom m }

  m1 † m2 ≡
    [ a↦b | a:A,b:B •
      a ∈ dom m1 \ dom m2 ∧ b=m1(a) ∨ a ∈ dom m2 ∧ b=m2(a) ]

  m1 ∪ m2 ≡ [ a↦b | a:A,b:B •
            a ∈ dom m1 ∧ b=m1(a) ∨ a ∈ dom m2 ∧ b=m2(a) ]

  m \ s ≡ [ a↦m(a) | a:A • a ∈ dom m \ s ]
  m / s ≡ [ a↦m(a) | a:A • a ∈ dom m ∩ s ]

  m1 = m2 ≡
    dom m1 = dom m2 ∧ ∀ a:A • a ∈ dom m1 ⇒ m1(a) = m2(a)
  m1 ≠ m2 ≡ ∼(m1 = m2)

  m°n ≡
    [ a↦c | a:A,c:C • a ∈ dom m ∧ c = n(m(a)) ]
    pre rng m ⊆ dom n
```

16.5. λ-Calculus + Functions

16.5.1. The λ-Calculus Syntax

type /∗ A BNF Syntax: ∗/
  ⟨L⟩ ::= ⟨V⟩ | ⟨F⟩ | ⟨A⟩ | (⟨A⟩)
  ⟨V⟩ ::= /∗ variables, i.e. identifiers ∗/
  ⟨F⟩ ::= λ⟨V⟩ · ⟨L⟩
  ⟨A⟩ ::= (⟨L⟩⟨L⟩)
value /∗ Examples ∗/
  ⟨L⟩: e, f, a, ...
  ⟨V⟩: x, ...
  ⟨F⟩: λ x · e, ...
  ⟨A⟩: f a, (f a), f(a), (f)(a), ...

16.5.2. Free and Bound Variables

Let x, y be variable names and e, f be λ-expressions.

- $\langle$V$\rangle$: Variable x is free in x.
- $\langle$F$\rangle$: x is free in $\lambda y \cdot e$ if $x \neq y$ and x is free in e.
- $\langle$A$\rangle$: x is free in $f(e)$ if it is free in either f or e (i.e., also in both).

16.5.3. Substitution

- **subst**([N/x]x) ≡ N;
- **subst**([N/x]a) ≡ a,

  for all variables a≠ x;
- **subst**([N/x](P Q)) ≡ (**subst**([N/x]P) **subst**([N/x]Q));
- **subst**([N/x](λx·P)) ≡ λ y·P;
- **subst**([N/x](λ y·P)) ≡ λy· **subst**([N/x]P),

  if x≠y and y is not free in N or x is not free in P;
- **subst**([N/x](λy·P)) ≡ λz·**subst**([N/z]**subst**([z/y]P)),

  if y≠x and y is free in N and x is free in P
  (where z is not free in (N P)).

16.5.4. α-Renaming and β-Reduction

- α-renaming: λx·M

  If x, y are distinct variables then replacing x by y in λx·M results in λy·**subst**([y/x]M). We can rename the formal parameter of a λ-function expression provided that no free variables of its body M thereby become bound.

- β-reduction: (λx·M)(N)

  All free occurrences of x in M are replaced by the expression N provided that no free variables of N thereby become bound in the result. (λx·M)(N) ≡ **subst**([N/x]M)

16.5.5. Function Signatures

For sorts we may want to postulate some functions:

type
  A, B, C
value
  obs_B: A → B,
  obs_C: A → C,
  gen_A: B×C → A

16.5.6. Function Definitions

Functions can be defined explicitly:

value
  f: Arguments → Result
  f(args) ≡ DValueExpr

  g: Arguments $\xrightarrow{\sim}$ Result
  g(args) ≡ ValueAndStateChangeClause
  pre P(args)

Or functions can be defined implicitly:

value
  f: Arguments → Result
  f(args) **as** result
  post P1(args,result)

  g: Arguments $\xrightarrow{\sim}$ Result
  g(args) **as** result
  pre P2(args)
  post P3(args,result)

16.6. Other Applicative Expressions

16.6.1. Simple let Expressions

Simple (i.e., nonrecursive) **let** expressions:

$$\textbf{let}\ \mathrm{a} = \mathcal{E}_d\ \textbf{in}\ \mathcal{E}_b(\mathrm{a})\ \textbf{end}$$

is an "expanded" form of:

$$(\lambda \mathrm{a}.\mathcal{E}_b(\mathrm{a}))(\mathcal{E}_d)$$

16.6.2. Recursive let Expressions

Recursive **let** expressions are written as:

$$\textbf{let}\ \mathrm{f} = \lambda \mathrm{a{:}A} \cdot \mathrm{E(f)}\ \textbf{in}\ \mathrm{B(f,a)}\ \textbf{end}$$

is "the same" as:

$$\textbf{let}\ \mathrm{f} = \mathbf{Y}\mathrm{F}\ \textbf{in}\ \mathrm{B(f,a)}\ \textbf{end}$$

where:

$$\mathrm{F} \equiv \lambda \mathrm{g} \cdot \lambda \mathrm{a} \cdot (\mathrm{E(g)}) \text{ and } \mathrm{YF} = \mathrm{F(YF)}$$

16.6.3. Predicative let Expressions

Predicative **let** expressions:

let a:A · $\mathcal{P}$(a) **in** $\mathcal{B}$(a) **end**

express the selection of a value a of type A which satisfies a predicate $\mathcal{P}$(a) for evaluation in the body $\mathcal{B}$(a).

16.6.4. Pattern and "Wild Card" let Expressions

Patterns and *wild cards* can be used:

let {a} ∪ s = set **in** ... **end**
let {a,_} ∪ s = set **in** ... **end**

let (a,b,...,c) = cart **in** ... **end**
let (a,_,...,c) = cart **in** ... **end**

let ⟨a⟩⌢ℓ = list **in** ... **end**
let ⟨a,_,b⟩⌢ℓ = list **in** ... **end**

let [a↦b] ∪ m = map **in** ... **end**
let [a↦b,_] ∪ m = map **in** ... **end**

16.6.5. Conditionals

if b_expr **then** c_expr **else** a_expr
end

if b_expr **then** c_expr **end** ≡ /∗ same as: ∗/
  if b_expr **then** c_expr **else skip end**

if b_expr_1 **then** c_expr_1
elsif b_expr_2 **then** c_expr_2
elsif b_expr_3 **then** c_expr_3
...
elsif b_expr_n **then** c_expr_n **end**

case expr **of**
  choice_pattern_1 → expr_1,
  choice_pattern_2 → expr_2,
  ...
  choice_pattern_n_or_wild_card → expr_n
end

16.6.6. Operator/Operand Expressions

⟨Expr⟩ ::=
⟨Prefix_Op⟩ ⟨Expr⟩
| ⟨Expr⟩ ⟨Infix_Op⟩ ⟨Expr⟩
| ⟨Expr⟩ ⟨Suffix_Op⟩
| ...

⟨Prefix_Op⟩ ::=
− | ∼ | ∪ | ∩ | **card** | **len** | **inds** | **elems** | **hd** | **tl** | **dom** | **rng**

⟨Infix_Op⟩ ::=
= | ≠ | ≡ | + | − | ∗ | ↑ | / | < | ≤ | ≥ | > | ∧ | ∨ | ⇒
| ∈ | ∉ | ∪ | ∩ | \ | ⊂ | ⊆ | ⊇ | ⊃ | ⌢ | † | ◦

⟨Suffix_Op⟩ ::= !

16.7. Imperative Constructs

16.7.1. Statements and State Changes

```
  Unit
value
  stmt: Unit → Unit
  stmt()
```

- Statements accept no arguments.
- Statement execution changes the state (of declared variables).
- **Unit** → **Unit** designates a function from states to states.
- Statements, **stmt**, denote state-to-state changing functions.
- Writing () as "only" arguments to a function "means" that () is an argument of type **Unit**.

16.7.2. Variables and Assignment

0. **variable** v:Type := expression
1. v := expr

16.7.3. Statement Sequences and skip

2. **skip**
3. stm_1;stm_2;...;stm_n

16.7.4. Imperative Conditionals

4. **if** expr **then** stm_c **else** stm_a **end**
5. **case** e **of**: p_1→S_1(p_1),...,p_n→S_n(p_n) **end**

16.7.5. Iterative Conditionals

6. **while** expr **do** stm **end**
7. **do** stmt **until** expr **end**

16.7.6. Iterative Sequencing

8. **for** e **in** list_expr · P(b) **do** S(b) **end**

16.8. Process Constructs

16.8.1. Process Channels

Let A and B stand for two types of (channel) messages and i:KIdx for channel array indexes, then:

channel c:A
channel { k[i]:B · i:KIdx }

16.8.2. Process Composition

- Let P and Q stand for names of process functions,
- i.e., of functions which express willingness to engage in input and/or output events,
- thereby communicating over declared channels.
- Let P() and Q stand for process expressions, then:

P ‖ Q	Parallel composition
P ▯ Q	Nondeterministic external choice (either/or)
P ⨅ Q	Nondeterministic internal choice (either/or)
P ⫴ Q	Interlock parallel composition

16.8.3. Input/Output Events

Let c, k[i] and e designate channels of type A and B, then:

c ?, k[i] ? Input
c ! e, k[i] ! e Output

- expresses the willingness of a process to engage in an event that
  - "reads" an input, respectively
  - "writes" an output.

16.8.4. Process Definitions

The below signatures are just examples. They emphasise that process functions must somehow express, in their signature, via which channels they wish to engage in input and output events.

value
  P: **Unit** → **in** c **out** k[i]
  Unit
  Q: i:KIdx → **out** c **in** k[i] **Unit**

  P() ≡ ... c ? ... k[i] ! e ...
  Q(i) ≡ ... k[i] ? ... c ! e ...

The process function definitions (i.e., their bodies) express possible events.

16.9. Simple RSL Specifications

type

...

variable

...

channel

...

value

...

axiom

...