A An RSL Primer 263

This is an ultra-short introduction to the RAISE Specification Language, RSL. Examples follow and expand on the examples of earlier sections.

A.1 Types

The reader is kindly asked to study first the decomposition of this section into its sub-parts and sub-sub-parts.

A.1.1 Type Expressions

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

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".

RSL has a number of *built-in* atomic types. There are the Booleans, integers, natural numbers, reals, characters, and texts. 264

Basic Types

type
[1] **Bool**
[2] **Int**
[3] **Nat**
[4] **Real**
[5] **Char**
[6] **Text**

1. The Boolean type of truth values **false** and **true**.
2. The integer type on integers ..., –2, –1, 0, 1, 2, ...
3. The natural number type of positive integer values 0, 1, 2, ...
4. The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period ("."), followed by a natural number (the fraction).
5. The character type of character values ″a″, ″b″, ...
6. The text type of character string values ″aa″, ″aaa″, ..., ″abc″, ...

265

Example 1 ..... **Basic Net Attributes:**

- For safe, uncluttered traffic, hubs and links can 'carry' a maximum of vehicles.
- Links have lengths. (We ignore hub (traversal) lengths.)
- One can calculate whether a link is a two-way link.

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

266

```
type
   MAX = Nat
   LEN = Real
   is_Two_Way_Link = Bool
value
   obs_Max: (H|L) → MAX
   obs_Len: L → LEN
   is_two_way_link: L → is_Two_Way_Link
   is_two_way_link(l) ≡ ∃ lσ:LΣ • lσ ∈ obs_HΣ(l)∧card lσ=2
```

. . . . . **End of Example 1**

Composite Types

267

Composite types have composite values. That is, values which we consider to have proper constituent (sub-)values, i.e., can, to us, be meaningfully "taken apart".

From these one can form type expressions: finite sets, infinite sets, Cartesian products, lists, maps, etc.

Let A, B and C be any type names or type expressions, then:

Composite Type Expressions

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

7. The set type of finite cardinality set values.
8. The set type of infinite and finite cardinality set values.
9. The Cartesian type of Cartesian values.
10. The list type of finite length list values.
11. The list type of infinite and finite length list values.
12. The map type of finite definition set map values.
13. The function type of total function values.
14. The function type of partial function values.
15. In (A) A is constrained to be:
    - either a Cartesian B × C × ... × D, in which case it is identical to type expression kind 9,
    - or not to be the name of a built-in type (cf., 1–6) or of a type, in which case the parentheses serve as simple delimiters, e.g., (A $\overrightarrow{m}$ B), or (A*)-**set**, or (A-**set**)list, or (A|B) $\overrightarrow{m}$ (C|D|(E $\overrightarrow{m}$ F)), etc.
16. The postulated disjoint union of types A, B, . . . , and C.

17. The record type of mk_id-named record values mk_id(av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.
18. The record type of unnamed record values (av,...,bv), where av, ..., bv, are values of respective types. The distinct identifiers sel_a, etc., designate selector functions.

268

Example 2 ..... **Composite Net Type Expressions:**

The type clauses of function signatures:

```
value
  f: A → B
```

often have the type expressions *A* and/or *B* be composite type expressions:

```
value
  obs_HIs: L → HI-set
  obs_LIs: H → LI-set
  obs_HΣ: H → HT-set
  set_HΣ: H × HΣ → H
```

269

Right-hand sides of type definitions often have composite type expressions:

```
type
  N = H-set × L-set
  HT = LI × HI × LI
  LT′ = HI × LI × HI
```

..... **End of Example 2**

A.1.2 Type Definitions

270

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

Type Definition

```
type
  A = Type_expr
schematic examples:
  A1 = B1-set, A2 = B1-infset
  A3 = B2 × C1 × D1
  B1 = E*, B2 = Eω
  C1 = F m→ G
  D1 = H → J, D2 = H ~→ J
  K = L | M
```


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

271

Example 3 . . . . . **Composite Net Types:**

There are many ways in which nets can be concretely modelled:

- **Sorts + Observers + Axioms:** First we show an example of type definitions without right-hand side, that is, of sort definitions.

  From a net one can observe many things.

  Of the things we focus on are the hubs and the links.

  A net contains two or more hubs and one or more links. Possibly other entities and net attributes may also be observable, but we shall not consider those here.

  type
  [sorts] N_α, H, L, HI, LI
  value
  obs_Hs: $\mathsf{N}_\alpha \to$ H-**set**
  obs_Ls: $\mathsf{N}_\alpha \to$ L-**set**
  axiom
  $\forall$ n:N_α • **card** obs_Hs(n)>0 $\Rightarrow$ **card** obs_Ls(n)$\geq$1 $\wedge$ ...

272

- **Cartesians + Wellformedness:** A net can be considered as a Cartesian of sets of two or more hubs and sets of one or more links.

  type
  [sorts] H, L
  N_β = H-**set** $\times$ L-**set**
  value
  wf_N_β: $\mathsf{N}_\beta \to$ **Bool**
  wf_N_β(hs,ls) $\equiv$ **card** hs>1 $\Rightarrow$ **card** ls>0 ....
  inject_N_β: $\mathsf{N}_\alpha \xrightarrow{\sim} \mathsf{N}_\beta$ **pre**: wf_N_β(hs,ls)
  inject_N_β(n_α) $\equiv$ (obs_Hs(n_α),obs_Ls(n_α))

273

- **Cartesians + Maps + Wellformedness:** Or a net can be described

  a as a triple of b-c-d:

  b hubs (modelled as a map from hub identfiers to hubs),

  c links (modelled as a map from link identfiers to links), and

  d a graph from hub h_i identifiers h_{i_i} to maps from identfiers l_{ij_i} of hub h_i connected links l_{ij} to the identfiers h_{j_i} of link connected hubs h_j.

274

type
[sorts] H, HI, L, LI
[a] N_γ = HUBS × LINKS × GRAPH
[b] HUBS = HI $\overrightarrow{m}$ H
[c] LINKS = LI $\overrightarrow{m}$ L
[d] GRAPH = HI $\overrightarrow{m}$ (LI −m> HI)

– [b,c] *hs:HUBS* and *ls:LINKS* are maps from hub (link) identifiers to hubs (links) where one can still observe these identfiers from these hubs (link).

- Example 12 on page 117 defines the well-formedness predicates for the above map types.

. . . . . **End of Example 3**

275

Variety of 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: 276

Record 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)

Types A, B, ..., Z are disjoint, i.e., shares no values, provided all mk_id_k are distinct and due to the use of the disjoint record type constructor ==.

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**

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

277

Example 4 . . . . . Net Record Types: Insert Links:

19. To a net one can insert a new link in either of three ways:
    a) Either the link is connected to two existing hubs — and the insert operation must therefore specify the new link and the identifiers of two existing hubs;
    b) or the link is connected to one existing hub and to a new hub — and the insert operation must therefore specify the new link, the identifier of an existing hub, and a new hub;
    c) or the link is connected to two new hubs — and the insert operation must therefore specify the new link and two new hubs.
    d) From the inserted link one must be able to observe identifier of respective hubs.
20. From a net one can remove a link.[13] The removal command specifies a link identifier.

278

```
type
  19    Insert == Ins(s_ins:Ins)
  19    Ins = 2xHubs | 1x1nH | 2nHs
  19a   2xHubs == 2oldH(s_hi1:HI,s_l:L,s_hi2:HI)
  19b   1x1nH == 1oldH1newH(s_hi:HI,s_l:L,s_h:H)
  19c   2nHs == 2newH(s_h1:H,s_l:L,s_h2:H)
  20    Remove == Rmv(s_li:LI)
axiom
  19d  ∀ 2oldH(hi′,l,hi″):Ins • hi′≠hi″ ∧ obs_LIs(l)={hi′,hi″} ∧
       ∀ 1old1newH(hi,l,h):Ins • obs_LIs(l)={hi,obs_HI(h)} ∧
       ∀ 2newH(h′,l,h″):Ins • obs_LIs(l)={obs_HI(h′),obs_HI(h″)}
```

RSL Explanation

- 19: The type clause **type** Ins = 2xHubs | 1x1nH | 2nHs introduces the type name Ins and defines it to be the union (|) type of values of either of three types: 2xHubs, 1x1nH and 2nHs.
  - 19a): The type clause **type** 2xHubs == 2oldH(s_hi1:HI, s_l:L, s_hi2:HI) defines the type 2xHubs to be the type of values of record type 2oldH(s_hi1:HI,s_l:L,s_hi2:HI), that is, Cartesian-like, or "tree"-like values with record (root) name 2oldH and with three sub-values, like branches of a tree, of types HI, L and HI. Given a value, cmd, of type 2xHubs, applying the selectors s_hi1, s_l and s_hi2 to cmd yield the corresponding sub-values.

[13] – provided that what remains is still a proper net

  - 19b): Reading of this type clause is left as exercise to the reader.
  - 19c): Reading of this type clause is left as exercise to the reader.
  - 19d): The axiom **axiom** has three predicate clauses, one for each category of Insert commands.
    - ◊ The first clause: ∀ 2oldH(hi′,l,hi″):Ins • hi′≠hi″ ∧ obs_HIs(l) = {hi′, hi″} reads as follows:
      - For all record structures, 2oldH(hi′,l,hi″), that is, values of type Insert (which in this case is the same as of type 2xHubs),
      - that is values which can be expressed as a record with root name 2oldH and with three sub-values ("freely") named hi′, l and hi″
      - (where these are bound to be of type HI, L and HI by the definition of 2xHubs),
      - the two hub identifiers hi′ and hi″ must be different,
      - and the hub identifiers observed from the new link, l, must be the two argument hub identifiers hi′ and hi″.
    - ◊ Reading of the second predicate clause is left as exercise to the reader.
    - ◊ Reading of the third predicate clause is left as exercise to the reader.

  The three types 2xHubs, 1x1nH and 2nHs are disjoint: no value in one of them is the same value as in any of the other merely due to the fact that the record names, 2oldH, 1oldH1newH and 2newH, are distinct. This is no matter what the "bodies" of their record structure is, and they are here also distinct: (s_hi1:HI,s_l:L,s_hi2:HI), (s_hi:HI,s_l:L,s_h:H), respectively (s_h1:H,s_l:L,s_h2:H).

- 20; The type clause **type** Remove == Rmv(s_li:LI)
  - (as for Items 19b) and 19c))
  - defines a type of record values, say rmv,
  - with record name Rmv and with a single sub-value, say li of type LI
  - where li can be selected from by rmv selector s_li.

End of RSL Explanation

Example 17 on page 128 presents the semantics functions for *int_Insert* and *int_Remove*.

. . . . . **End of Example 4**

Subtypes 279

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:

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

Subtypes

```
type
  A = {| b:B • P(b) |}
```

Example 5 ..... **Net Subtypes:**

In Example 3 on page 98 we gave three examples. For the first we gave an example, **Sorts + Observers + Axioms**, "purely" in terms of sets, see *Sorts — Abstract Types* below. For the second and third we gave concrete types in terms of Cartesians and Maps.

- In the **Sorts + Observers + Axioms** part of Example 3
  - a net was defined as a sort, and so were its hubs, links, hub identifiers and link identifiers;
  - axioms – making use of appropriate observer functions - make up the wellformedness condition on such nets.

We now redefine this as follows:

```
type
  [ sorts ] N′, H, L, HI, LI
            N = {|n:N′ • wf_N(n)|}
value
  wf_N: N′ → Bool
  wf_N(n) ≡
    ∀ n:N • card obs_Hs(n)≥0 ∧ card obs_Ls(n)≥0 ∧
    axioms 2.–3., 5.–6., and 8., (Page 13)
```

- In the **Cartesians + Wellformedness** part of Example 3
  - a net was a Cartesian of a set of hubs and a set of links
  - with the wellformedness that there were at least two hubs and at least one link
  - and that these were connected appropriately (treated as ...).

We now redefine this as follows:

```
type
  N′ = H-set × L-set
  N = {|n:N′ • wf_N(n)|}
```

- In the **Cartesians + Maps + Wellformedness** part of Example 3

– a net was a triple of hubs, links and a graph,
– each with their wellformednes predicates.

We now redefine this as follows:

285

```
type
  [ sorts ] L, H, LI, HI
  N′ = HUBS × LINKS × GRAPH
  N = {|(hs,ls,g):N′ • wf_HUBS(hs)∧wf_LINKS(ls)∧wf_GRAPH(g)(hs,ls)|}
  HUBS′ = HI →m H
  HUBS = {|hs:HUBS′ • wf_HUBS(hs)|}
  LINKS′ = LI → L
  LINKS = {|ls:LINKS′ • wf_LINKS(ls)|}
  GRAPH′ = HI →m (LI →m HI)
  GRAPH = {|g:GRAPH′ • wf_GRAPH(g)|}
value
  wf_GRAPH: GRAPH′ → (HUBS × LINKS) → Bool
  wf_GRAPH(g)(hs,ls) ≡ wf_N(hs,ls,g)
```

Example 12 on page 117 presents a definition of *wf_GRAPH*.

. . . . . **End of Example 5**

Sorts — Abstract Types 286

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

Sorts

```
type
  A, B, ..., C
```

287

Example 6 . . . . . **Net Sorts:**

In formula lines of Examples 3–5 we have indicated those **type** clauses which define *sorts*, by bracketed [sorts] literals.

. . . . . **End of Example 6**

A.2 Concrete RSL Types: Values and Operations 288

A.2.1 Arithmetic

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

Arithmetic

type
  Nat, **Int**, **Real**
value
  +,−,∗: **Nat**×**Nat**→**Nat** | **Int**×**Int**→**Int** | **Real**×**Real**→**Real**
  /: **Nat**×**Nat**$\xrightarrow{\sim}$**Nat** | **Int**×**Int**$\xrightarrow{\sim}$**Int** | **Real**×**Real**$\xrightarrow{\sim}$**Real**
  <,≤,=,≠,≥,> (**Nat**|**Int**|**Real**) × (**Nat**|**Int**|**Real**) → **Bool**

A.2.2 Set Expressions

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

Set Enumerations

{{}, {a}, {e$_1$,e$_2$,...,e$_n$}, ...} ⊆ A-**set**
{{}, {a}, {e$_1$,e$_2$,...,e$_n$}, ..., {e$_1$,e$_2$,...}} ⊆ A-**infset**

Example 7 ..... **Set Expressions over Nets:**
We now consider hubs to abstract cities, towns, villages, etcetera. Thus with hubs we can associate sets of citizens.

Let c:C stand for a citizen value c being an element in the type C of all such. Let g:G stand for any (group) of citizens, respectively the type of all such. Let s:S stand for any set of groups, respectively the type of all such. Two otherwise distinct groups are related to one another if they share at least one citizen, the liaisons. A network nw:NW is a set of groups such that for every group in the network one can always find another group with which it shares liaisons. Solely using the set data type and the concept of subtypes, we can model the above:

type
  C
  G′ = C-**set**, G = {| g:G′ • g≠{} |}
  S = G-**set**
  L′ = C-**set**, L = {| ℓ:L′ • ℓ≠{} |}
  NW′ = S, NW = {| s:S • wf_S(s) |}
value
  wf_S: S → **Bool**
  wf_S(s) ≡ ∀ g:G • g ∈ s ⇒ ∃ g′:G • g′ ∈ s ∧ share(g,g′)
  share: G×G → **Bool**
  share(g,g′) ≡ g≠g′ ∧ g ∩ g′ ≠ {}
  liaisons: G×G → L
  liaisons(g,g′) = g ∩ g′ **pre** share(g,g′)

292

Annotations: L stands for proper liaisons (of at least one liaison). G′, L′ and N′ are the "raw" types which are constrained to G, L and N. {| binding:type_expr • bool_expr |} is the general form of the subtype expression. For G and L we state the constraints "in-line", i.e., as direct part of the subtype expression. For NW we state the constraints by referring to a separately defined predicate. wf_S(s) expresses — through the auxiliary predicate — that s contains at least two groups and that any such two groups share at least one citizen. liaisons is a "truly" auxiliary function in that we have yet to "find an active need" for this function! 293

The idea is that citizens can be associated with more than one city, town, village, etc. (primary home, summer and/or winter house, working place, etc.). A group is now a set of citizens related by some "interest" (Rotary club membership, political party "grassroots", religion, et.). The reader is invited to define, for example, such functions as:The set of groups (or networks) which are represented in all hubs [or in only one hub]. The set of hubs whose citizens partake in no groups [respectively networks]. The group [network] with the largest coverage in terms of number of hubs in which that group [network] is represented.

.

. . . . . **End of Example 7**

Set Comprehension 294

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.

Set Comprehension

```
type
  A, B
  P = A → Bool
  Q = A ~→ B
value
  comprehend: A-infset × P × Q → B-infset
  comprehend(s,P,Q) ≡ { Q(a) | a:A • a ∈ s ∧ P(a)}
```

295

Example 8 . . . . . **Set Comprehensions:**

Item 48 on page 22, the *wf_N(hs,ls,g)* wellformedness predicate definition, includes:

```
type
47a. PLAN = HI -m→ LHIM
47b. LHIM = LI -m→ HI-set
value
48c. no_junk: PLAN → Bool
48c. no_junk(plan) ≡ dom plan = ∪{rng(plan(hi))|hi:HI•hi ∈ dom plan}
```


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

It expresses the distributed union of sets (**rng** *(plan(li))* of hub identifiers (for each of the *hi* indexed maps from (definition set, **dom**) link identiers to (range set, **rng**) hub identifiers, where *hi:HI* ranges over **dom** *plan*).

. . . . . **End of Example 8**

A.2.3 Cartesian Expressions 296

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

Cartesian Enumerations

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

297

Example 9 . . . . . **Cartesian Net Types:**
So far we have abstracted hubs and links as sorts. That is, we have not defined their types concretely. Instead we have postulated some attributes such as: observable hub identifiers of hubs and sets of observable link identifiers of links connected to hubs. We now claim the
298 following further attributes of hubs and links.

- Concrete links have
  - link identifiers,
  - link names – where two or more connected links may have the same link name,
  - two (unordered) hub identifiers,
  - lenghts,
  - locations – where we do not presently defined what we main by locations,
  - etcetera
- Concrete hubs have
  - hub identifiers,
  - unique hub names,
  - a set of one or more observable link identifiers,
  - locations,
  - etcetera.

299

type
LN, HN, LEN, LOC
cL = LI × LN × (HI × HI) × LOC × ...
cH = HI × HN × LI-**set** × LOC × ...

. . . . . **End of Example 9**

A.2.4 List Expressions 300

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

List Enumerations

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

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

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.

List Comprehension 301

The last line below expresses list comprehension.

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))⟩

302

Example 10 . . . . . **Routes in Nets:**

- A phenomenological (i.e., a physical) route of a net is a sequence of one or more adjacent links of that net.
- A conceptual route is a sequence of one or more link identifiers.
- An abstract route is a conceptual route

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

– for which there is a phenomenological route of the net for which the link identifiers of the abstract route map one-to-one onto links of the phenomenological route.

303

```
type
  N, H, L, HI, LI
  PR′ = L*
  PR = {| pr:PR′ • ∃ n:N • wf_PR(pr)(n)|}
  CR = LI*
  AR′ = LI*
  AR = {| ar:AR′ • ∃ n:N • wf_AR(ar)(n) |}
value
  wf_PR: PR′ → N → Bool
  wf_PR(pr)(n) ≡
    ∀ i:Nat • {i,i+1}⊆inds pr ⇒
      obs_HIs(l(i)) ∩ obs_HIs(l(i+1)) ≠ {}
  wf_AR′: AR′ → N → Bool
  wf_AR(ar)(n) ≡
    ∃ pr:PR • pr ∈ routes(n) ∧ wf_PR(pr)(n) ∧ len pr=len ar ∧
      ∀ i:Nat • i ∈ inds ar ⇒ obs_LI(pr(i))=ar(i)
```

304

- A single link is a phenomenological route.
- If r and r' are phenomenological routes
  - such that the last link r
  - and the first link of r'
  - share observable hub identifiers,

  then the concatenation $r^\frown r'$ is a route.

  This inductive definition implies a recursive set comprehension.
- A circular phenomenological route is a phenomenological route whose first and last links are distinct but share hub identifiers.
- A looped phenomenological route is a phenomenological route where two distinctly positions (i.e., indexed) links share hub identifiers.

305

```
value
  routes: N → PR-infset
  routes(n) ≡
    let prs = {⟨l⟩|l:L•obs_Ls(n)} ∪
```

```
                    ∪ {pr⌢pr′|pr,pr′:PR•{pr,pr′}⊆prs∧obs_HIs(r(len pr))∩obs_HIs(pr′(1))≠{}}
  prs end

is_circular: PR → Bool
is_circular(pr) ≡ obs_HIs(pr(1))∩obs_HIs(pr(len pr))≠{}

is_looped: PR → Bool
is_looped(pr) ≡ ∃ i,j:Nat • i≠j∧{i,j}⊆index pr ⇒ obs_HIs(pr(i))∩obs_HIs(pr(j))≠{}
```

306

- Straight routes are Phenomenological routes without loops.
- Phenomenological routes with no loops can be constructed from phenomenological routes by removing suffix routes whose first link give rise to looping.

```
value
  straight_routes: N → PR-set
  straight_routes(n) ≡
    let prs = routes(n) in {straight_route(pr)|pr:PR•ps ∈ prs} end

  straight_route: PR → PR
  straight_route(pr) ≡
    ⟨pr(i)|i:Nat•i:[1..len pr] ∧ pr(i)∉ elems⟨pr(j)|j:Nat•j:[1..i]⟩⟩
```

. . . . . **End of Example 10**

A.2.5 Map Expressions 307

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:

Map Enumerations

```
type
  T1, T2
  M = T1 ~m~> T2
value
  u,u1,u2,...,un:T1, v,v1,v2,...,vn:T2
  {[], [u↦v], ..., [u1↦v1,u2↦v2,...,un↦vn],...} ⊆ M
```

Map Comprehension 308

The last line below expresses map comprehension:

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

Map Comprehension

```
type
  U, V, X, Y
  M = U ~m~> V
  F = U ~~> X
  G = V ~~> Y
  P = U → Bool
value
  comprehend: M×F×G×P → (X ~m~> Y)
  comprehend(m,F,G,P) ≡
    [ F(u) ↦ G(m(u)) | u:U • u ∈ dom m ∧ P(u) ]
```

309

Example 11 ..... **Concrete Net Type Construction:**

- We Define a function *con[struct]_*N_γ (of the **Cartesians + Maps + Wellformedness** part of Example 3.
  - The base of the construction is the fully abstract sort definition of N_α in the **Sorts + Observers + Axioms** part of Example 3 – where the sorts of hub and link identifiers are taken from earlier examples.
  - The target of the construction is the N_γ of the **Cartesians + Maps + Wellformedness** part of Example 3.
  - First we recall the ssential types of that N_γ.

310

```
type
  Nγ = HUBS × LINKS × GRAPH
  HUBS = HI ~m~> H
  LINKS = LI ~m~> L
  GRAPH = HI ~m~> (LI ~m~> HI)
value
  con_Nγ: Nα → Nγ
  con_Nγ(nα) ≡
    let hubs = [obs_HI(h) ↦ h | h:H • h ∈ obs_Hs(nα)],
        links = [obs_LI(h) ↦ l | l:L • l ∈ obs_Ls(nα)],
        graph = [obs_HI(h) ↦ [obs_LI(l) ↦ ι(obs_HIs(l)\{obs_HI(h)})
                                | l:L • l ∈ obs_Ls(nα)∧li ∈ obs_LIs(h)]
                 | H:h • h ∈ obs_Hs(nα)] in
        (hubs.links,graph) end

  ι: A-set ~~> A [A could be LI-set]
  ι(as) ≡ if card as=1 then let {a}=as in a else chaos end end
```

311

theorem:
n_α **satisfies axioms** 2.–3., 5.–6., and 8. (Page 13) $\Rightarrow$ wf_N$_\gamma$(con_N$_\gamma$(n_α))

. . . . . **End of Example 11**

A.2.6 Set Operations 312

Set Operator Signatures

Set Operations

value
21 $\in$: A $\times$ A-**infset** $\rightarrow$ **Bool**
22 $\notin$: A $\times$ A-**infset** $\rightarrow$ **Bool**
23 $\cup$: A-**infset** $\times$ A-**infset** $\rightarrow$ A-**infset**
24 $\cup$: (A-**infset**)-**infset** $\rightarrow$ A-**infset**
25 $\cap$: A-**infset** $\times$ A-**infset** $\rightarrow$ A-**infset**
26 $\cap$: (A-**infset**)-**infset** $\rightarrow$ A-**infset**
27 $\setminus$: A-**infset** $\times$ A-**infset** $\rightarrow$ A-**infset**
28 $\subset$: A-**infset** $\times$ A-**infset** $\rightarrow$ **Bool**
29 $\subseteq$: A-**infset** $\times$ A-**infset** $\rightarrow$ **Bool**
30 $=$: A-**infset** $\times$ A-**infset** $\rightarrow$ **Bool**
31 $\neq$: A-**infset** $\times$ A-**infset** $\rightarrow$ **Bool**
32 **card**: A-**infset** $\xrightarrow{\sim}$ **Nat**

Set Examples 313

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

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

Informal Explication 314

21. ∈: The membership operator expresses that an element is a member of a set.
22. ∉: The non-membership operator expresses that an element is not a member of a set.
23. ∪: 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.
24. ∪: 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.
25. ∩: The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
26. ∩: The prefix distributed intersection operator. When applied to a set of sets, the
315 operator gives the set whose members are in some of the operand sets.
27. \: 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.
28. ⊆: The proper subset operator expresses that all members of the left operand set are also in the right operand set.
29. ⊂: 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.
30. =: The equal operator expresses that the two operand sets are identical.
31. ≠: The non-equal operator expresses that the two operand sets are *not* identical.
32. **card**: The cardinality operator gives the number of elements in a finite set.

Set Operator Definitions 316

The operations can be defined as follows (≡ is the definition symbol):

Set Operation Definitions

value

$$s' \cup s'' \equiv \{ a \mid a{:}A \bullet a \in s' \vee a \in s'' \}$$
$$s' \cap s'' \equiv \{ a \mid a{:}A \bullet a \in s' \wedge a \in s'' \}$$
$$s' \setminus s'' \equiv \{ a \mid a{:}A \bullet a \in s' \wedge a \notin s'' \}$$
$$s' \subseteq s'' \equiv \forall\, a{:}A \bullet a \in s' \Rightarrow a \in s''$$
$$s' \subset s'' \equiv s' \subseteq s'' \wedge \exists\, a{:}A \bullet a \in s'' \wedge a \notin s'$$

```
  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 ∗/
```

A.2.7 Cartesian Operations 317

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
```

A.2.8 List Operations 318

List Operator Signatures

List Operations

```
value
  hd: A^ω ~→ A
  tl: A^ω ~→ A^ω
  len: A^ω ~→ Nat
  inds: A^ω → Nat-infset
  elems: A^ω → A-infset
  .(.): A^ω × Nat ~→ A
  ⌢: A* × A^ω → A^ω
  =: A^ω × A^ω → Bool
  ≠: A^ω × A^ω → Bool
```

List Operation Examples 319

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

List 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⟩

Informal Explication 320

- **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 321 of elements larger than or equal to i, gives the ith element of the list.
- ⌢: 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.
- ≠: The non-equal operator expresses that the two operand lists are *not* identical.

322

The operations can also be defined as follows:

List Operator "Definitions"

value
is_finite_list: $A^\omega \to$ **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 }
```

323

```
  q(i) ≡
    case (q,i) of
        (⟨⟩,1) → chaos,
        (_,1) → let a:A,q′:Q • q=⟨a⟩⌢q′ in a end
        _ → 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″)
```

A.2.9 Map Operations 324

Map Operator Signatures and Map Operation Examples

```
value
  m(a): M → A ~→ B, m(a) = 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′]
```


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

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

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.
- $\cup$: Merge. When applied to two operand maps, it gives a merge of these maps.
- $\backslash$: 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.
- $\neq$: The non-equal operator expresses that the two operand maps are *not* identical.
- $\circ$: 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.

Example 12 ..... **Miscellaneous Net Expressions: Maps:**

Example 3 on page 98 left out defining the well-formedness of the map types:

```
type
    GRAPH = HI ~m~> (LI ~m~> HI-set)
    HUBS = HI ~m~> H
    LINKS = LI ~m~> L
    N_γ = HUBS × LINKS × GRAPH
value
    wf_HUBS: H-set → Bool
    wf_HUBS(hubs) ≡ ∀ hi:HI • hi ∈ dom hubs ⇒ obs_HI(hubs(hi))=hi
    wf_LINKS: L-set → Bool
    wf_LINKS(links) ≡ ∀ li:LI • li ∈ dom links ⇒ obs_LI(links(li))=li
    wf_N_γ: N_γ → Bool
    wf_N_γ(hs,ls,g) ≡
        dom hs = dom g ∧
        ∪ {dom g(hi)|hi:HI • hi ∈ dom g} = dom links ∧
        ∪ {rng g(hi)|hi:HI • hi ∈ dom g} = dom g ∧
        ∀ hi:HI • hi ∈ dom g ⇒ ∀ li:LI • li ∈ dom g(hi) ⇒ (g(hi))(li)≠hi
        ∀ hi:HI • hi ∈ dom g ⇒ ∀ li:LI • li ∈ dom g(hi) ⇒
            ∃ hi′:HI • hi′ ∈ dom g ⇒ ∃ ! li:LI • li ∈ dom g(hi) ⇒ (g(hi))(li) = hi′ ∧ (g(hi′))(li) = hi
```

326

- *Hubs* record the same hubs as do the net corresponding *GRAPH* (**dom** *hs* = **dom** *g* ∧).
- *GRAPH* record the same links as do the net corresponding *LINKS* (∪ {**dom** *g(hi)*|*hi:HI* • *hi* ∈ **dom** *g*} = **dom** *links*).
- The target (or range) hub identifiers of graphs are the same as the domain of the graph (∪ {**rng** *g(hi)*|*hi:HI* • *hi* ∈ **dom** *g*} = **dom** *g*), that is none missing, no new ones !
- No links emanate from and are incident upon the same hub (∀ *hi:HI* • *hi* ∈ **dom** *g* ⇒ ∀ *li:LI* • *li* ∈ **dom** *g(hi)* ⇒ *(g(hi))(li)≠hi*).
- If there is a link from one hub to another in the *GRAPH*, then the same link also connects the other hub to the former (∀ *hi:HI* • *hi* ∈ **dom** *g* ⇒ ∀ *li:LI* • *li* ∈ **dom** *g(hi)* ⇒ ∃ *hi′:HI* • *hi′* ∈ **dom** *g* ⇒ ∃ ! *li:LI* • *li* ∈ **dom** *g(hi)* ⇒ *(g(hi))(li) = hi′* ∧ *(g(hi′))(li) = hi*).

..... **End of Example 12**

Map Operation "Redefinitions" 327

The map operations can also be defined as follows:

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

```
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) ]
```

328

```
  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
```

A.3 The RSL Predicate Calculus 329

A.3.1 Propositional Expressions

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

Propositional Expressions

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*.

A.3.2 Simple Predicate Expressions 330

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:

Simple Predicate Expressions

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.

A.3.3 Quantified Expressions 331

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:

Quantified Expressions

∀ x:X • $\mathcal{P}(x)$
∃ y:Y • $\mathcal{Q}(y)$
∃ ! z:Z • $\mathcal{R}(z)$

are quantified expressions — also being predicate expressions.

They are "read" as: For all x (values in type X) the predicate $\mathcal{P}(x)$ holds; there exists (at least) one y (value in type Y) such that the predicate $\mathcal{Q}(y)$ holds; and there exists a unique z (value in type Z) such that the predicate $\mathcal{R}(z)$ holds. 332

Example 13 ..... **Predicates Over Net Quantities:**

From earlier examples we show some predicates:

- Example 1: Right hand side of function definition *is_two_way_link(l):*
  ∃ *lσ:LΣ* • *lσ* ∈ *obs_HΣ(l)*∧**card** *lσ=2* 333
- Example 3:
  - The **Sorts + Observers + Axioms** part:
    - Right hand side of the wellformedness function *wf_N(n)* definition:
      ∀ *n:N* • **card** *obs_Hs(n)*≥*2* ∧ **card** *obs_Ls(n)*≥*1* ∧ **axioms** 2.–3., 5.–6., and 8., (Page 13)
    - Right hand side of the wellformedness function *wf_N(hs,ls)* definition:
      card *hs*≥*2* ∧ **card** *ls*≥*1* ... 334
  - The **Cartesians + Maps + Wellformedness** part:
    - Right hand side of the *wf_HUBS* wellformedness function definition:
      ∀ *hi:HI* • *hi* ∈ **dom** *hubs* ⇒ *obs_HIhubs(hi)*=*hi*

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

∗ Right hand side of the *wf_LINKS* wellformedness function definition:
∀ *li:LI* • *li* ∈ **dom** *links* ⇒ *obs_LI*links(*li*)=*li*

∗ Right hand side of the *wf_N(hs,ls,g)* wellformedness function definition:
[*c*] **dom** *hs* = **dom** *g* ∧
[*d*] ∪ {**dom** *g(hi)*|*hi:HI* • *hi* ∈ **dom** *g*} = **dom** *links* ∧
[*e*] ∪ {**rng** *g(hi)*|*hi:HI* • *hi* ∈ **dom** *g*} = **dom** *g* ∧
[*f*] ∀ *hi:HI* • *hi* ∈ **dom** *g* ⇒ ∀ *li:LI* • *li* ∈ **dom** *g(hi)* ⇒ *(g(hi))(li)*≠*hi*
[*g*] ∀ *hi:HI* • *hi* ∈ **dom** *g* ⇒ ∀ *li:LI* • *li* ∈ **dom** *g(hi)* ⇒
∃ *hi′:HI* • *hi′* ∈ **dom** *g* ⇒ ∃ ! *li:LI* • *li* ∈ **dom** *g(hi)* ⇒
(g(hi))(li) = *hi′* ∧ *(g(hi′))(li)* = *hi*

. . . . . **End of Example 13**

A.4 λ-Calculus + Functions 335

A.4.1 The λ-Calculus Syntax

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), ...

A.4.2 Free and Bound Variables 336

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

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

A.4.3 Substitution 337

In RSL, the following rules for substitution apply:
Substitution of an expression N for all free free x in M is expressed: **subst**([N/x]M).

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)) ≡ λy•P;
- **subst**([N/x](λy•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)).

A.4.4 α-Renaming and β-Reduction 338

α and β Conversions

- 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)

339

Example 14 ..... **Network Traffic:**
We model traffic by introducing a number of model concepts. We simplify, without loosing the essence of this example, namely to show the use of λ–functions, by omitting consideration of dynamically changing nets. These are introduced next:

- Let us assume a net, *n:N*.

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

- There is a dense set, *T*, of times – for which we omit giving an appropriate definition.
- There is a sort, *V*, of vehicles.
- *TS* is a dense subset of *T*.
- For each *ts:TS* we can define a minimum and a maximum time.
- The $\mathcal{MIN}$ and $\mathcal{MAX}$ functions are meta-linguistic, that is, are not defined in our formal specification language RSL, but can be given a satisfactory meaning.
- At any moment some vehicles, *v:V*, have a *pos:Pos*ition on the net and *VP* records those.
- A *Pos*ition is either on a link or at a hub.
- An *onL*ink position can be designated by the link identifier, the identifiers of the from and to hubs, and the fraction, *f:F*, of the distance down the link from the from hub to the to hub.
- An *atH*ub position just designates the hub (by its identifier).
- Traffic, *tf:TF*, is now a continuous function from *T*ime to *NP* ("recordings").
- Modelling traffic in this way, in fact, in whichever way, entails a ("serious") number of well-formedness conditions. These are defined in *wf_TF* (omitted: ...).

```
value
  n:N
type
  T, V
  TS = T-infset
axiom
  ∀ ts:TS • ∃ tmin,tmax:T: tmin ∈ ts ∧ tmax ∈ ts ∧ ∀ t:T • t ∈ ts ⇒ tmin ≤ t ≤ tmax
  [ that is: ts = {MIN(ts)..MAX(ts)} ]
type
  VP = V -m-> Pos
  TF' = T → VP,                          TF = {|tf:TF'•wf_TF(tf)(n)|}
  Pos = onL | atH
  onL == mkLPos(hi:HI,li:LI,f:F,hi:HI),   atH == mkHPos(hi:HI)
value
  wf_TF: TF→ N → Bool
  wf_TF(tf)(n) ≡ ...
  DOMAIN: TF → TS
  MIN, MAX: TS → T
```

342

We have defined the continuous, composite entity of traffic. Now let us define an operation of inserting a vehicle in a traffic.

- To insert a vehicle, v, in a traffic, tf, is prescribable as follows:
  - the vehicle, v, must be designated;
  - a time point, t, "inside" the traffic tf must be stated;
  - a traffic, vtf, from time t of vehicle v must be stated;
  - as well as traffic, tf, into which vtf is to be "merged".
- The resulting traffic is referred to as tf'.

value
insert_V: V × T × TF → TF → TF
insert_V(v,t,vtf)(tf) **as** tf′

343

- The function *insert_V* is here defined in terms of a pair of pre/post conditions.
- The pre-condition can be prescribed as follows:
  - The insertion time t must be within to open interval of time points in the traffic tf to which insertion applies.
  - The vehicle v must not be among the vehicle positions of tf.
  - The vehicle must be the only vehicle "contained" in the "inserted" traffic vtf.

pre: $\mathcal{MIN}(\mathcal{DOMAIN}(\text{tf}) \leq \text{t} \leq \mathcal{MAX}(\mathcal{DOMAIN}(\text{tf}))$ ∧
∀ t′:T • t′ ∈ $\mathcal{DOMAIN}$(tf) ⇒ v ∉ **dom** tf(t′) ∧
$\mathcal{MIN}(\mathcal{DOMAIN}(\text{vtf})) = \text{t}$ ∧
∀ t′:T•t′ ∈ $\mathcal{DOMAIN}$(vtf) ⇒ **dom** vtf(t′)={v}

344

- The post condition "defines" tf', the traffic resulting from merging vtf with tf:
  - Let ts be the time points of tf and vtf, a time interval.
  - The result traffic, tf', is defines as a λ-function.
  - For any t'' in the time interval
  - if t'' is less than t, the insertion time, then tf' is as tf;
  - if t'' is t or larger then tf' applied to t'', i.e., $tf'(t'')$
    - for any $v' : V$ different from v yields the same as $(tf(t))(v')$,
    - but for v it yields $(vtf(t))(v)$.

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

345

```
  post: tf′ = λt″•
            let ts = 𝒟𝒪𝓜𝒜𝐼𝒩(tf) ∪ 𝒟𝒪𝓜𝒜𝐼𝒩(vtf) in
            if 𝓜𝐼𝒩(ts) ≤ t″ ≤ 𝓜𝒜𝒳(ts)
              then
                ((t″<t) → tf(t″),
                (t″≥t) → [ v′↦ if v′≠v then (tf(t))(v′) else (vtf(t))(v) end
                           |v′:V•v′ ∈ vehicles(tf) ])
              else chaos end
            end
  assumption: wf_TF(vtf)∧wf_TF(tf)
  theorem: wf_TF(tf′)
value
  vehicles: TF → V-set
  vehicles(tf) ≡ {v|t:T,v:V•t ∈ 𝒟𝒪𝓜𝒜𝐼𝒩(tf)∧v ∈ dom tf(t)}
```

We leave it as an exercise for the reader to define functions for: removing a vehicle from a traffic, changing to course of a vehicle from an originally (or changed) vehicle traffic to another. etcetera.

. . . . . **End of Example 14**

A.4.5 Function Signatures

346

For sorts we may want to postulate some functions:

Sorts and Function Signatures

```
type
  A, B, ..., C
value
  obs_B: A → B
  ...
  obs_C: A → C
```

These functions cannot be defined. Once a domain is presented in which sort A and sorts
347 or types B, ... and C occurs these observer functions can be demonstrated.

Example 15 . . . . . **Hub and Link Observers:**

Let a net with several hubs and links be presented. Now observer functions obs_Hs and obs_Ls can be demonstrated: one simply "walks" along the net, pointing out this hub and that link,
348 one-by-one until all the net has been visited.

The observer functions obs_HI and obs_LI can be likewise demonstrated, for example: when a hub is "visited" its unique identification can be postulated (and "calculated") to be the unique

geographic position of the hub one which is not overlapped by any other hub (or link), and likewise for links.

. . . . . **End of Example 15**

A.4.6 Function Definitions 349

Functions can be defined explicitly:

```
type
  A, B
value
  f: A → B [ a total function ]
  f(a_expr) ≡ b_expr
```

```
g: A ~→ B [ a partial function ]
g(a_expr) ≡ b_expr
pre P(a_expr)
P: A → Bool
```

a_expr, b_expr are A, respectively B valued expressions of any of the kinds illustrated in earlier and later sections of this primer. 350

Or functions can be defined implicitly:

```
value
  f: A→B
  f(a_expr) as b
  post P(a_expr,b)
  P: A×B→Bool
```

```
g: A~→B
g(a_expr) as b
pre P′(a_expr)
post P(a_expr,b)
P′: A→Bool
```

where b is just an identifier.

The symbol $\xrightarrow{\sim}$ indicates that the function is partial and thus not defined for all arguments. Partial functions should be assisted by preconditions stating the criteria for arguments to be meaningful to the function. 351

Finally functions, f, g, ..., h, can be defined in terms of axioms over function identifiers, f, g, ..., h, and over identifiers of function arguments and results.

```
type
  A, B, ..., C, D
value
  f: A → B, g: B → C, ..., h: C → D
axiom
  ∀ a:A, b:B, ..., c:C, d:D
    P1(f,g,...,h,a,b,...,c,d) ∧ ... ∧ Pn(f,g,...,h,a,b,...,c,d)
```

where $\mathcal{P}_1, \ldots, \mathcal{P}_m$ and $\mathcal{Q}_1, \ldots, \mathcal{Q}_n$ designate suitable predicate expressions. 352

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

Example 16 ..... **Axioms over Hubs, Links and Their Observers:**
The axioms displayed in Items 2–3 and 5–8 on Page 13 of Sect. 2.1 demonstrates how a number of entities and observer functions are constrained (that is, partially defined) by function signatures.

..... **End of Example 16**

A.5 Other Applicative Expressions 353

A.5.1 Simple let Expressions

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

Let Expressions

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

is an "expanded" form of:

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

A.5.2 Recursive let Expressions 354

Recursive **let** expressions are written as:

Recursive let Expressions

let f = λa•E(f,a) **in** B(f,a) **end**
let f = (λg•λa•E(g,a))(f) **in** B(f.a) **end**
let f = F(f) **in** E(f,a) **end where** F ≡ λg•λa•E(g,a)
let f = **Y**F **in** B(f,a) **end where** **Y**F = F(**Y**F)

We read f = **Y**F as "*f is a fix point of F*".

A.5.3 Non-deterministic let Clause 355

The non-deterministic **let** clause:

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

expresses the non-deterministic selection of a value a of type A which satisfies a predicate $\mathcal{P}$(a) for evaluation in the body $\mathcal{B}$(a). If no a:A • P(a) the clause evaluates to **chaos**.

A.5.4 Pattern and "Wild Card" let Expressions 356

Patterns and *wild cards* can be used:

Patterns

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**

A.5.5 Conditionals 357

Various kinds of conditional expressions are offered by RSL:

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**

358

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

Example 17 ..... **Choice Pattern Case Expressions: Insert Links:**
We consider the meaning of the Insert operation designators.

33. The insert operation takes an Insert command and a net and yields either a new net or **chaos** for the case where the insertion command "is at odds" with, that is, is not semantically well-formed with respect to the net.
34. We characterise the "is not at odds", i.e., is semantically well-formed, that is:
    - pre_int_Insert(op)(hs,ls),

    as follows: it is a propositional function which applies to Insert actions, op, and nets, (hs.ls), and yields a truth value if the below relation between the command arguments and the net is satisfied. Let (hs,ls) be a value of type N.
35. If the command is of the form 2oldH(hi′,l,hi′) then
    - ⋆1 hi′ must be the identifier of a hub in hs,
    - ⋆s2 l must not be in ls and its identifier must (also) not be observable in ls, and
    - ⋆3 hi″ must be the identifier of a(nother) hub in hs.
36. If the command is of the form 1oldH1newH(hi,l,h) then
    - ⋆1 hi must be the identifier of a hub in hs,
    - ⋆2 l must not be in ls and its identifier must (also) not be observable in ls, and
    - ⋆3 h must not be in hs and its identifier must (also) not be observable in hs.
37. If the command is of the form 2newH(h′,l,h″) then
    - ⋆1 h′ — left to the reader as an exercise (see formalisation !),
    - ⋆2 l — left to the reader as an exercise (see formalisation !), and
    - ⋆3 h″ — left to the reader as an exercise (see formalisation !).

Conditions concerning the new link (second ⋆s, ⋆2, in the above three cases) can be expressed independent of the insert command category.

```
value
  33 int_Insert: Insert → N ~→ N
  34′ pre_int_Insert: Ins → N → Bool
  34″ pre_int_Insert(Ins(op))(hs,ls) ≡
⋆2        s_l(op)∉ ls ∧ obs_LI(s_l(op)) ∉ iols(ls) ∧
    case op of
  35)    2oldH(hi′,l,hi″) → {hi′,hi″}∈ iohs(hs),
```

36) 1oldH1newH(hi,l,h) →
    hi ∈ iohs(hs) ∧ h∉ hs ∧ obs_HI(h)∉ iohs(hs),
37) 2newH(h′,l,h″) →
    {h′,h″}∩ hs={} ∧ {obs_HI(h′),obs_HI(h″)}∩ iohs(hs)={}

end

RSL Explanation

- 33: The value clause **value** int_Insert: Insert → N $\xrightarrow{\sim}$ N names a value, int_Insert, and defines its type to be Insert → N $\xrightarrow{\sim}$ N, that is, a partial function ($\xrightarrow{\sim}$) from Insert commands and nets (N) to nets.
  (int_Insert is thus a function. What that function calculates will be defined later.)

- 34′: The predicate pre_int_Insert: Insert → N → **Bool** function (which is used in connection with int_Insert to assert semantic well-formedness) applies to Insert commands and nets and yield truth value **true** if the command can be meaningfully performed on the net state.

- 34″: The action pre_int_Insert(op)(hs,ls) (that is, the effect of performing the function pre_int_Insert on an Insert command and a net state is defined by a case distinction over the category of the Insert command. But first we test the common property:

- ⋆2: s_l(op)∉ls∧obs_LI(s_l(op))∉iols(ls), namely that the new link is not an existing net link and that its identifier is not already known.

  - 35): If the Insert command is of kind 2oldH(hi',l,hi") then {hi′,hi″}∈ iohs(hs), that is, then the two distinct argument hub identifiers must not be in the set of known hub identifiers, i.e., of the existing hubs hs.
  - 36): If the Insert command is of kind 1oldH1newH(hi,l,h) then ... exercise left as an exercises to the reader.
  - 37): If the Insert command is of kind 2newH(h',l,h") ... exercise left as an exercises to the reader. The set intersection operation is defined in Sect. A.2.6 on page 111 Item 25 on page 112.

End of RSL Explanation

362

38. Given a net, (hs,ls), and given a hub identifier, (hi), which can be observed from some hub in the net, xtr_H(hi)(hs,ls) extracts the hub with that identifier.

39. Given a net, (hs,ls), and given a link identifier, (li), which can be observed from some link in the net, xtr_L(li)(hs,ls) extracts the hub with that identifier.

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

```
value
38: xtr_H: HI → N ~→ H
38: xtr_H(hi)(hs,_) ≡ let h:H•h ∈ hs ∧ obs_HI(h)=hi in h end
                      pre hi ∈ iohs(hs)
39: xtr_L: HI → N ~→ H
39: xtr_L(li)(_,ls) ≡ let l:L•l ∈ ls ∧ obs_LI(l)=li in l end
                      pre li ∈ iols(ls)
```

RSL Explanation

- 38: Function application xtr_H(hi)(hs,_) yields the hub h, i.e. the value h of type H, such that (•) h is in hs and h has hub identifier hi.
- 38: The wild-card, _, expresses that the extraction (xtr_H) function does not need the L-set argument.
- 39: Left as an exercise for the reader.

End of RSL Explanation

363

40. When a new link is joined to an existing hub then the observable link identifiers of that hub must be updated to reflect the link identifier of the new link.
41. When an existing link is removed from a remaining hub then the observable link identifiers of that hub must be updated to reflect the removed link (identifier).

```
value
  aLI: H × LI → H, rLI: H × LI ~→ H
  40: aLI(h,li) as h′
        pre li ∉ obs_LIs(h)
        post obs_LIs(h′) = {li} ∪ obs_LIs(h) ∧ non_I_eq(h,h′)
  41: rLI(h′,li) as h
        pre li ∈ obs_LIs(h′) ∧ card obs_LIs(h′)≥2
        post obs_LIs(h) = obs_LIs(h′) \ {li} ∧ non_I_eq(h,h′)
```

RSL Explanation

- 40: The add link identifier function aLI:
  - The function definition clause aLI(h,li) as h′ defines the application of aLI to a pair (h,li) to yield an update, h′ of h.
  - The pre-condition pre li ∉ obs_LIs(h) expresses that the link identifier li must not be observable h.

  - The post-condition **post** obs_LIs(h) = obs_LIs(h′) \ {li} ∧ non_l_eq(h,h′) expresses that the link identifiers of the resulting hub are those of the argument hub except (\) that the argument link identifier is not in the resulting hub.

- 41: The remove link identifier function rLI:
  - The function definition clause rLI(h′,li) **as** h defines the application of rLI to a pair (h′,li) to yield an update, h of h′.
  - The pre-condition clause **pre** li ∈ obs_LIs(h′) ∧ **card** obs_LIs(h′)≥2 expresses that the link identifier li must not be observable h.
  - post-condition clause **post** obs_LIs(h) = obs_LIs(h′) \ {li} ∧ non_l_eq(h,h′) expresses that the link identifiers of the resulting hub are those of the argument hub except that the argument link identifier is not in the resulting hub.

End of RSL Explanation

364

42. If the Insert command is of kind 2newH(h',l,h'') then the updated net of hubs and links, has
    - the hubs hs joined, ∪, by the set {h′,h″} and
    - the links ls joined by the singleton set of {l}.
43. If the Insert command is of kind 1oldH1newH(hi,l,h) then the updated net of hubs and links, has
    - 43.1 : the hub identified by hi updated, hi′, to reflect the link connected to that hub.
    - 43.2 : The set of hubs has the hub identified by hi replaced by the updated hub hi′ and the new hub.
    - 43.2 : The set of links augmented by the new link.
44. If the Insert command is of kind 2oldH(hi',l,hi'') then
    - 44.1–.2 : the two connecting hubs are updated to reflect the new link,
    - 44.3 : and the resulting sets of hubs and links updated.

365

```
int_Insert(op)(hs,ls) ≡
⋆ᵢ case op of
42    2newH(h′,l,h″) → (hs ∪ {h′,h″},ls ∪ {l}),
43    1oldH1newH(hi,l,h) →
43.1    let h′ = aLI(xtr_H(hi,hs),obs_LI(l)) in
43.2    (hs\{xtr_H(hi,hs)}∪{h,h′},ls ∪{l}) end,
44    2oldH(hi′,l,hi″) →
```


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

```
44.1      let hsδ = {aLI(xtr_H(hi′,hs),obs_LI(l)),
44.2                 aLI(xtr_H(hi″,hs),obs_LI(l))} in
44.3      (hs\{xtr_H(hi′,hs),xtr_H(hi″,hs)}∪ hsδ,ls ∪{l}) end
⋆j  end
⋆k  pre pre_int_Insert(op)(hs,ls)
```

RSL Explanation

- $\star_i$–$\star_j$: The clause **case** op **of** $p_1 \rightarrow c_1$, $p_2 \rightarrow c_2$, ... $p_n \rightarrow c_n$ **end** is a conditional clause.
- $\star_k$: The pre-condition expresses that the insert command is semantically well-formed — which means that those reference identifiers that are used are known and that the new link and hubs are not known in the net.
- $\star_i$ + 42: If op is of the form 2newH(h′,l,h″ then — the narrative explains the rest;

  else
- $\star_i$ + 43: If op is of the form 1oldH1newH(hi,l,h) then
  - 43.1: h′ is the known hub (identified by hi) updated to reflect the new link being connected to that hub,
  - 43.2: and the pair [(updated hs,updated ls)] reflects the new net: the hubs have the hub originally known by hi replaced by h′, and the links have been simple extended (∪) by the singleton set of the new link;

  else
- $\star_i$ + 44: 44: If op is of the form 2oldH(hi′,l,hi″) then
  - 44.1: the first element of the set of two hubs (hsδ) reflect one of the updated hubs,
  - 44.2: the second element of the set of two hubs (hsδ) reflect the other of the updated hubs,
  - 44.3: the set of two original hubs known by the argument hub identifiers are removed and replaced by the set hsδ;

  else — well, there is no need for a further 'else' part as the operator can only be of either of the three mutually exclusive forms !

End of RSL Explanation

366

45. The remove command is of the form Rmv(li) for some li.
46. We now sketch the meaning of removing a link:

a) The link identifier, li, is, by the pre_int_Remove pre-condition, that of a link, l, in the net.
b) That link connects to two hubs, let us refer to them as h′ and h′.
c) For each of these two hubs, say h, the following holds wrt. removal of their connecting link:
   i. If l is the only link connected to h then hub h is removed. This may mean that
      - either one
      - or two hubs

      are also removed when the link is removed.
   ii. If l is not the only link connected to h then the hub h is modified to reflect that it is no longer connected to l.
d) The resulting net is that of the pair of adjusted set of hubs and links.

367

```
value
  45  int_Remove: Rmv → N ~→ N
  46  int_Remove(Rmv(li))(hs,ls) ≡
  46a)  let l = xtr_L(li)(ls), {hi′,hi″} = obs_HIs(l) in
  46b)  let {h′,h″} = {xtr_H(hi′,hs),xtr_H(hi″,hs)} in
  46c)  let hs′ = cond_rmv(h′,hs) ∪ cond_rmv_H(h″,hs) in
  46d)  (hs\{h′,h″} ∪ hs′,ls\{l}) end end end
  46a)  pre li ∈ iols(ls)

  cond_rmv: LI × H × H-set → H-set
  cond_rmv(li,h,hs) ≡
  46(c)i)   if obs_HIs(h)={li} then {}
  46(c)ii)  else {sLI(li,h)} end
  pre li ∈ obs_HIs(h)
```

RSL Explanation

- 45: The int_Remove operation applies to a remove command Rmv(li) and a net (hs,ls) and yields a net — provided the remove command is semantically well-formed.
- 46: To Remove a link identifier by li from the net (hs,ls) can be formalised as follows:
  - 46a): obtain the link l from its identifier li and the set of links ls, and
  - 46a): obtain the identifiers, {hi′,hi″}, of the two distinct hubs to which link l is connected;
  - 46b): then obtain the hubs {h′,h″} with these identifiers;
  - 46c): now examine cond_rmv each of these hubs (see Lines 46(c)i–46(c)ii)).

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

- The examination function cond_rmv either yields an empty set or the singleton set of one modified hub (a link identifier has been removed).
- 46c) The set, hs′, of zero, one or two modified hubs is yielded.
- That set is joined to the result of removing the hubs {h′,h″}
- and the set of links that result from removing l from ls.

The conditional hub remove function cond_rmv

– 46(c)i): either yields the empty set (of no hubs) if li is the only link identifier inh,
– 46(c)ii): or yields a modification of h in which the link identifier li is no longer observable.

End of RSL Explanation

. . . . . **End of Example 17**

A.5.6 Operator/Operand Expressions 368

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⟩ ::= !

A.6 Imperative Constructs 369

A.6.1 Statements and State Changes

Often, following the RAISE method, software development starts with highly abstract, sorts and applicative constructs which, through stages of refinements, are turned into concrete types and imperative constructs.

Imperative constructs are thus inevitable in RSL.

Unit
value
stmt: **Unit** → **Unit**
stmt()

- The **Unit** clause, in a sense, denotes "an underlying state"
  - which we, for simplicity, can consider as
  - a mapping from identifiers of declared variables into their values.
- Statements accept no arguments and, usually, operate on the state
  - through "reading" the value(s) of declared variables and
  - through "writing", i.e., assigning values to such declared variables.
- Statement execution thus changes the state (of declared variables).
- **Unit** → **Unit** designates a function from states to states.
- Statements, stmt, denote state-to-state changing functions.
- Affixing () as an "only" arguments to a function "means" that () is an argument of type **Unit**.

A.6.2 Variables and Assignment 370

Variables and Assignment

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

A.6.3 Statement Sequences and skip

Sequencing is expressed using the ';' operator. **skip** is the empty statement having no value or side-effect.

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

A.6.4 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**

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

A.6.5 Iterative Conditionals 371

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

A.6.6 Iterative Sequencing

8. **for** i **in** list • P(list(i)) **do** S(list(i)) **end**
9. **for** e **in** set • P(e) **do** S(e) **end**

A.7 Process Constructs 372

A.7.1 Process Channels

Let A, B and C stand for three types of (channel) messages and i:IIdx, j:JIdx for channel array indexes, then:

Process Channels

```
channel
  c:A
channel
  {k[ i ]|i:IIdx}:B
  {ch[ i,j ]i:IIdx,j:JIdx}:C
```

declare a channel, c, and a set (an array) of channels, k[i], capable of communicating values
373 of the designated types (A and B).

Example 18 ..... **Modelling Connected Links and Hubs:**
Examples (18–21) of this section, i.e., Sect. A.7 are building up a model of one form of meaning of a transport net. We model the movement of vehicles around hubs and links. We think of each
374 hub, each link and each vehicle to be a process. These processes communicate via channels.

- We assume a net, $n : N$, and a set, vs, of vehicles.
- Each vehicle can potentially interact
  - with each hub and
  - with each link.
- Array channel indices *(vi,hi):IVH* and *(vi,li):IVL* serve to effect these interactions.
- Each hub can interact with each of its connected links and indices *(hi,li):IHL* serves these interactions.

```
type
  N, V, VI
value
  n:N, vs:V-set
  obs_VI: V → VI
type
  H, L, HI, LI, M
  IVH = VI×HI, IVL = VI×LI, IHL = HI×LI
```

375

- We need some auxiliary quantities in order to be able to express subsequent channel declarations.
- Given that we assume a net, $n : N$ and a set of vehicles, $vs : VS$, we can now define the following (global) values:
  - the sets of hubs, hs, and links, ls of the net;
  - the set, $ivhs$, of indices between vehicles and hubs,
  - the set, $ivls$, of indices between vehicles and links, and
  - the set, $ihls$, of indices between hubs and links.

```
value
  hs:H-set = obs_Hs(n), ls:L-set = obs_Ls(n)
  his:HI-set = {obs_HI(h)|h:H•h ∈ hs}, lis:LI-set = {obs_LI(h)|l:L•l ∈ ls},
  ivhs:IVH-set = {(obs_VI(v),obs_HI(h))|v:V,h:H•v ∈ vs∧h ∈ hs}
  ivls:IVL-set = {(obs_VI(v),obs_LI(l))|v:V,l:L•v ∈ vs∧l ∈ ls}
  ihls:IHL-set = {(hi,li)|h:H,(hi,li):IHL• h ∈ hs∧hi=obs_HI(h)∧li ∈ obs_LIs(h)}
```

376

- We are now ready to declare the channels:
  - a set of channels, {vh[i]|i:IVH•i∈ivhs} between vehicles and all potentially traversable hubs;
  - a set of channels, {vh[i]|i:IVH•i∈ivhs} between vehicles and all potentially traversable links; and
  - a set of channels, {hl[i]|i:IHL•i∈ihls}, between hubs and connected links.

```
channel
  {vh[ i ] | i:IVH • i ∈ ivhs} : M
  {vl[ i ] | i:IVL • i ∈ ivls} : M
  {hl[ i ] | i:IHL • i ∈ ihls} : M
```

. . . . . **End of Example 18**

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

A.7.2 Process Definitions 377

A process definition is a function definition. 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.

Processes P and Q are to interact, and to do so "ad infinitum". Processes R and S are
378 to interact, and to do so "once", and then yielding B, respectively D values.

```
value
  P: Unit → in c out {k[i]||i:IIdx} Unit
  Q: i:KIdx →  out c in k[i] Unit

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

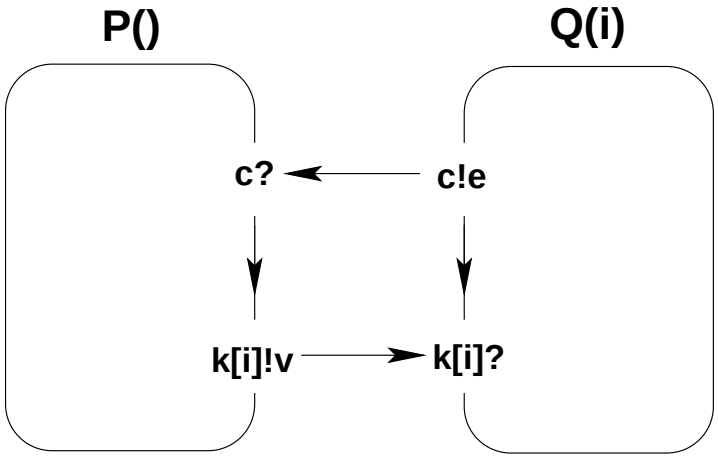

Figure 9: The P —— Q Process

379

Example 19 ..... **Communicating Hubs, Links and Vehicles:**

- Hubs interact with links and vehicles:
  - with all immediately adjacent links,
  - and with potentially all vehicles.
- Links interact with hubs and vehicles:
  - with both adjacent hubs,
  - and with potentially all vehicles.

- Vehicles interact with hubs and links:
  - with potentially all hubs.
  - and with potentially all links.

380

```
value
  hub: hi:HI × h:H → in,out {hl[(hi,li)|li:LI•li ∈ obs_LIs(h)]}
                       in,out {vh[(vi,hi)|vi:VI•vi ∈ vis]} Unit
  link: li:LI × l:L → in,out {hl[(hi,li)|hi:HI•hi ∈ obs_HIs(l)]}
                      in,out {vl[(vi,li)|vi:VI•vi ∈ vis]} Unit
  vehicle: vi:VI → (Pos × Net) → v:V → in,out {vh[(vi,hi)|hi:HI•hi ∈ his]} Unit
                       in,out {vl[(vi,li)|li:LI•li ∈ lis]} Unit
```

. . . . . **End of Example 19**

A.7.3 Process Composition

381

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 $\mathcal{P}$ and $\mathcal{Q}$ stand for process expressions, and let $\mathcal{P}_i$ stand for an indexed process expression, then:

$\mathcal{P} \parallel \mathcal{Q}$	Parallel composition
$\mathcal{P} \,[]\, \mathcal{Q}$	Nondeterministic external choice (either/or)
$\mathcal{P} \sqcap \mathcal{Q}$	Nondeterministic internal choice (either/or)
$\mathcal{P} \sharp \mathcal{Q}$	Interlock parallel composition
$\mathcal{O}\{\mathcal{P}_i \mid \text{i:Idx}\}$	Distributed composition, $\mathcal{O} = \parallel, [], \sqcap, \sharp$

express the parallel ($\parallel$) of two processes, or the nondeterministic choice between two processes: either external ($[]$) or internal ($\sqcap$). The interlock ($\sharp$) composition expresses that the two processes are forced to communicate only with one another, until one of them terminates.

382

Example 20 . . . . . **Modelling Transport Nets:**

- The net, with vehicles, potential or actual, is now considered a process.
- It is the parallel composition of
  - all hub processes,
  - all link processes and
  - all vehicle processes.

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

```
value
  net: N → V-set → Unit
  net(n)(vs) ≡
    ‖ {hub( obs_HI(h))(h)|h:H•h ∈ obs_Hs(n)} ‖
    ‖ {link( obs_LI(l))(l)|l:L•l ∈ obs_Ls(n)} ‖
    ‖ {vehicle(obs_VI(v))(obs_PN(v))(v)|v:V•v ∈ vs}

  obs_PN: V → (Pos×Net)
```

383

- We illustrate a schematic definition of simplified hub processes.
- The hub process alternates, internally non-deterministically, ⌈⌉, between three sub-processes
  - a sub-process which serves the link-hub connections,
  - a sub-process which serves thos vehicles which communicate that they somehow wish to enter or leave (or do something else with respect to) the hub, and
  - a sub-process which serves the hub itself — whatever that is !

```
hub(hi)(h) ≡
      □{let m = hl[(hi,li)] ? in hub(hi)(E_hℓ(li)(m)(h)) end|i:LI•li ∈ obs_LI(h)}
  ⌈⌉ □{let m = vh[(vi,hi)] ? in hub(vi)(E_hv(vi)(m)(h)) end|vi:VI•vi ∈ vis}
  ⌈⌉ hub(hi)(E_hown(h))
```

384

- The three auxiliary processes:
  - $\mathcal{E}_{h_\ell}$ update the hub with respect to (wrt.) connected link, *li*, information *m*,
  - $\mathcal{E}_{h_v}$ update the hub with wrt. vehicle, *vi*, information *m*,
  - $\mathcal{E}_{h_{own}}$ update the hub with wrt. whatever the hub so decides. An example could be signalling dependent on previous link-to-hub communicated information, say about traffic density.

$\mathcal{E}_{h_\ell}$: LI → M → H → H
$\mathcal{E}_{h_v}$: VI → M → H → H
$\mathcal{E}_{h_{own}}$: H → H

The reader is encouraged to sketch/define similarly schematic link and vehicle processes.

.

. . . . . **End of Example 20**

A.7.4 Input/Output Events 385

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

[1] c?, k[i]? input A value
[2] c!e, k[i]!e output A value

value

[3] P: ... → **out** c ..., P(...) ≡ ... c!e ... offer an A value,
[4] Q: ... → **in** c ..., Q(...) ≡ ... c? ... accept an A value
[5] S: ... → ..., S(...) = P(...)‖Q(...) synchronise and communicate

[5] expresses the willingness of a process to engage in an event that [1,3] "reads" an input, respectively [2,4] "writes" an output. If process P reaches the c!e "program point before" process Q 'reaches program point' c? then process P "waits" on Q — and vice versa. Once both processes have reached these respective program points they "synchronise while communicating the message vale e.

The process function definitions (i.e., their bodies) express possible [output/input] events. 386

Example 21 ..... **Modelling Vehicle Movements:**

- Whereas hubs and links are modelled as basically static, passive, that is, inert, processes we shall consider vehicles to be "highly" dynamic, active processes.
- We assume that a vehicle possesses knowledge about the road net.
  - The road net is here abstracted as an awareness of
  - which links, by their link identifiers,
  - are connected to any given hub, designated by its hub identifier,
  - the length of the link,
  - and the hub to which the link is connected "at the other end", also by its hub identifier

  387
- A vehicle is further modelled by its current position on the net in terms of either hub or link positions
  - designated by appropriate identifiers
  - and, when "on a link" "how far down the link", by a measure of a fraction of the total length of the link, the vehicle has progressed.

type

Net = HI $\xrightarrow{m}$ (LI $\xrightarrow{m}$ HI)
Pos = atH | onL
atH == mk_atH(hi:HI)
onL == mk_onL(fhi:HI,li:LI,f:F,thi:HI)
F = {|f:**Real**•0≤f≤1|}

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

388

- We first assume that the vehicle is at a hub.
- There are now two possibilities (1–2] versus [4–8]).
  - Either the vehicle remains at that hub
    - [1] which is expressed by some non-deterministic *wait*
    - [2] followed by a resumption of being that vehicle at that location.
  - [3] Or the vehicle (driver) decides to "move on":
    - [5] Onto a link, *li*,
    - [4] among the links, *lis*, emanating from the hub,
    - [6] and towards a next hub, *hi′*.
  - [4,6] The *lis* and *hi′* quantities are obtained from the vehicles own knowledge of the net.
  - [7] The hub and the chosen link are notified by the vehicle of its leaving the hub and entering the link,
  - [8] whereupon the vehicle resumes its being a vehicle at the initial location on the chosen link.

389

- The vehicle chooses between these two possibilities by an internal non-deterministic choice ([3]).

```
type
  M == mk_L_H(li:LI,hi:HI) | mk_H_L(hi:HI,li:LI)
value
  vehicle: VI → (Pos × Net) → V → Unit
  vehicle(vi)(mk_atH(hi),net)(v) ≡
  [1] (wait ;
  [2]  vehicle(vi)(mk_atH(hi),net)(v))
  [3] ⌈⌉
  [4] (let lis=dom net(hi) in
  [5]  let li:LI•li ∈ lis in
  [6]  let hi′=(net(hi))(li) in
  [7]  (vh[(vi,hi)]!mk_H_L(hi,li)‖vl[(vi,li)]!mk_H_L(hi,li));
  [8]  vehicle(vi)(mk_onL(hi,li,0,hi′),net)(v)
  [9]  end end end)
```

390

- We then assume that the vehicle is on a link and at a certain distance "down", *f*, that link.

- There are now two possibilities ([1–2] versus [4–7]).
  - Either the vehicle remains at that hub
    - [1′] which is expressed by some non-deterministic *wait*
    - [2′] followed by a resumption of being that vehicle at that location.
  - [3′] Or the vehicle (driver) decides to "move on".
  - [4′] Either
    - [5′] The vehicle is at the very end of the link and signals the link and the hub of its leaving the link and entering the hub,
    - [6′] whereupon the vehicle resumes its being a vehicle at hub h'.
  - [7′] or the vehicle moves further down, some non-zero fraction down the link.
- The vehicle chooses between these two possibilities by an internal non-deterministic choice ([3]).

391

```
type
  M == mk_L_H(li:LI,hi:HI) | mk_H_L(hi:HI,li:LI)
value
  δ:Real = move(h,f) axiom 0<δ≪1
  vehicle(vi)( mk_onL(hi,li,f,hi′),net)(v) ≡
  [1′] (wait ;
  [2′]  vehicle(vi)(mk_onL(hi,li,f,hi′),net)(v))
  [3′] ⌈⌉
  [4′] (case f of
  [5′]    1 → ((vl[vi,hi′]!mk_L_H(li,hi′)‖vh[vi,li]!mk_L_H(li,hi′));
  [6′]          vehicle(vi)(mk_atH(hi′),net)(v)),
  [7′]    _ → vehicle(vi)(mk_onL(hi,li,f+δ,hi′),net)(v)
  [8′]  end)
  move: H × F → F
```

. . . . . **End of Example 21**

A.8 Simple RSL Specifications 392

Besides the above constructs RSL also possesses module-oriented scheme, class and object constructs. We shall not cover these here. An RSL specification is then simply a sequence of one or more clusters of zero, one or more sort and/or type definitions, zero, one or more variable declarations, zero, one or more channel declarations, zero, one or more value definitions (including functions) and zero, one or more and axioms. We can illustrate these specification components schematically: 393

Simple RSL Specifications

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

```
type
  A, B, C, D, E, F, G
  Hf = A-set, Hi = A-infset
  J = B×C×...×D
  Kf = E*, Ki = E^ω
  L = F ~m~> G
  Mt = J → Kf, Mp = J ~→ Ki
  N == alpha | beta | ... | omega
  O == mk_Hf(as:Hf)
       | mk_Kf(el:Kf) | ...
  P = Hf | Kf | L | ...
variable
  vhf:Hf := ⟨⟩
channel
  chf:F, chg:G, {chb[i]|i:A}:B
value
  va:A, vb:B, ..., ve:E
  f1: A → B, f2: C ~→ D
  f1(a) ≡ E_f1(a)
  f2: E → in|out chf F
  f2(e) ≡ E_f2(e)
  f3: Unit → in chf out chg Unit
  ...
axiom
  P_i(f1,va),
  P_j(f2,vb),
  ...
  P_k(f3,ve)
```

The ordering of these clauses is immaterial. Intuitively the meaning of these definitions and declarations are the following.

The **type** clause introduces a number of user-defined type names; the type names are visible anywhere in the specification; and either denote sorts or concrete types.

The **variable** clause declares some variable names; a variable name denote some value of decalred type; the variable names are visible anywhere in the specification: assigned to ('written') or values 'read'.

The **channel** clause declares some channel names; either simple channels or arrays of channels of some type; the channel names are visible anywhere in the specification.

The **value** clause bind (constant) values to value names. These value names are visible anywhere in the specification. The specification

```
type
  A
value
  a:A
```

non-deterministically binds a to a value of type A. Thuis includes, for example

```
type
  A, B
value
  f: A → B
```

which non-deterministically binds f to a function value of type A→B.

The **axiom** clause is usually expressed as several "comma (,) separated" predicates:

$$\mathcal{P}_i(\overline{A_i}, \overline{f_i}, \overline{v_i}), \mathcal{P}_j(\overline{A_j}, \overline{f_j}, \overline{v_j}), \ldots, \mathcal{P}_k(\overline{A_k}, \overline{f_k}, \overline{v_k})$$

where $(\overline{A_k}, \overline{f_\ell}, \overline{v\ell})$ is an abbreviation for $A_{\ell_1}, A_{\ell_2}, \ldots, A_t, f_{\ell_1}, f_{\ell_2}, \ldots, f_{\ell_f}, v_{\ell_1}, v_{\ell_2}, \ldots, v_{\ell_v}$. The indexed sort or type names, A and the indexed function names, d, are defined elsewhere in the specification. The index value names, v are usually names of bound 'variables' of universally or existentially quantified predicates of the indexed ("comma"-separated) $\mathcal{P}$.

Example 22 ..... **A Neat Little "System":**

We present a self-contained specification of a simple system: The system models vehicles moving along a net, *vehicle*, the recording of vehicles entering links, *enter_sensor*, the recording of vehicles leaving links, *leave_sensor*, and the *road_pricing payment* of a vehicle having traversed (*entered* and *left*) a link. Note that vehicles only pay when completing a link traversal; that 'road pricing' only commences once a vehicle enters the first link after possibly having left an earlier link (and hub); and that no *road_pricing payment* is imposed on vehicles entering, staying-in (or at) and leaving hubs. 398

We assume the following: that each *link* is somehow associated with two pairs of *sensors:* a pair of *enter* and *leave sensors* at one end, and a pair of *enter* and *leave sensors* at the other end; and a *road pricing* process which records pairs of link enterings and leavings, first one, then, after any time interval, the other, with leavings leading to debiting of traversal fees; Our first specification define types, assume a net value, declares channels and state signatures of all processes. 399

- *ves* stand for vehicle entering (link) sensor channels,
- *vls* stand for vehicle leaving (link) sensor channels,
- *rp* stand for 'road pricing' channel
- *enter_sensor(hi,li)* stand for vehicle entering [sensor] process from hub *hi* to link *(*li).
- *leave_sensor(li,hi)* stand for vehicle leaving [sensor] process from link *li* to hub *(*hi).
- *road_pricing()* stand for the unique 'road pricing' process.
- *vehicle(vi)(...)* stand for the vehicle *vi* process.

400

```
type
  N, H, HI, LI, VI
  RPM == mk_Enter_L(vi:VI,li:LI) | mk_Leave_L(vi:VI,li:LI)
value
  n:N
channel
  {ves[ obs_HI(h),li ]|h:H•h ∈ obs_Hs(n)∧li ∈ obs_LIs(h)}:VI
  {vls[ li,obs_HI(h) ]|h:H•h ∈ obs_Hs(n)∧li ∈ obs_LIs(h)}:VI
  rp:RPM
type
  Fee, Bal
  LVS = LI ~m~> VI-set,  FEE = LI ~m~> Fee,  ACC = VI ~m~> Bal
value
  link: (li:LI × L) → Unit
  enter_sensor: (hi:HI × li:LI) → in ves[ hi,li ],out rp Unit
```


© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark

```
leave_sensor: (li:LI × hi:HI) → in vls[ li,hi ],out rp  Unit
road_pricing: (LVS×FEE×ACC) → in rp  Unit
```

To understand the sensor behaviours let us review the vehicle behaviour. In the *vehicle* behaviour defined in Example 21, in two parts, Page 142 and Page 143 we focus on the events [7] where the vehicle enters a link, respectively [5′] where the vehicle leaves a link. These are summarised in the schematic reproduction of the vehicle behaviour description. We redirect the interactions between vehicles and links to become interactions between vehicles and enter and leave sensors.

```
value
  δ:Real = move(h,f) axiom 0<δ≪1
  move: H × F → F
  vehicle: VI → (Pos × Net) → V → Unit
  vehicle(vi)(pos,net)(v) ≡
  [ 1 ] (wait ;
  [ 2 ]  vehicle(vi)(pos,net)(v))
  [ 3 ]  ⌈⌉
      case pos of
          mk_atH(hi) →
  [ 4−6 ]   (let lis=dom net(hi) in let li:LI•li ∈ lis in let hi′=(net(hi))(li) in
  [ 7 ]       ves[ hi,li ]!vi;
  [ 8 ]       vehicle(vi)(mk_onL(hi,li,0,hi′),net)(v)
  [ 9 ]       end end end)
          mk_onL(hi,li,f,hi′) →
  [ 4′ ]     (case f of
  [ 5′−6′ ]     1 → (vls[ li,hi ]!vi; vehicle(vi)(mk_atH(hi′),net)(v)),
  [ 7′ ]        _ → vehicle(vi)(mk_onL(hi,li,f+δ,hi′),net)(v)
  [ 8′ ]      end)
      end
```

- As mentioned on Page 145 *link* behaviours are associated with two pairs of sensors:
  - a pair of *enter* and *leave sensors* at one end, and
  - a pair of *enter* and *leave sensors* at the other end;

```
value
  link(li)(l) ≡
     let {hi,hi′} = obs_HIs(l) in
     enter_sensor(hi,li) ‖ leave_sensor(li,hi) ‖
     enter_sensor(hi′,li) ‖ leave_sensor(li,hi′) end
  enter_sensor(hi,li) ≡
     let vi = ves[ hi,li ]? in rp!mk_Enter_LI(vi,li); enter_sensor(hi,li) end
  leave_sensor(li,hi) ≡
     let vi = ves[ li,hi ]? in rp!mk_Leave_LI(vi,li); enter_sensor(li,hi) end
```

404

- The *LVS* component of the *road_pricing* behaviour serves,
  - among other purposes that are not mentioned here,
  - to record whether the movement of a vehicles "originates" along a link or not.
- Otherwise we leave it to the reader to carefully read the formulas.

```
value
  payment: VI × LI → (ACC × FEE) → ACC
  payment(vi,li)(fee,acc) ≡
    let bal′ = if vi ∈ dom acc then add(acc(vi),fee(li)) else fee(li) end
    in acc † [ vi ↦ bal′ ] end
  add: Fee × Bal → Bal [ add fee to balance ]
  road_pricing(lvs,fee,acc) ≡ in rp
    let m = rp? in
    case m of
      mk_Enter_LI(vi,li) →
          road_pricing(lvs†[ li↦lvs(li)∪{vi} ],fee,acc),
      mk_Leave_LI(vi,li) →
          let lvs′ = if vi ∈ lvs(li) then lvs†[ li↦lvs(li)\{vi} ] else lvs end,
              acc′ = payment(vi,li)(fee,acc) in
          road_pricing(lvs′,fee,acc′)
    end end end
```

405

. . . . . **End of Example 22**

© Dines Bjørner 2010, Fredsvej 11, DK–2840 Holte, Denmark