Abstract

This report presents a first attempt at a model of a credit card system. A first version of this document, marked Uppsala Draft, was worked out for my Uppsala University lectures 18–20 April 2016. Remarks in this type-font refers to my paper: [Bjo16, Manifest Domains: Analysis & Description]. Appendix A presents a primer of RSL, the Raise Specification Language.
## Appendix A

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.3.1</td>
<td>Propositional Expressions</td>
<td>23</td>
</tr>
<tr>
<td>A.3.2</td>
<td>Simple Predicate Expressions</td>
<td>24</td>
</tr>
<tr>
<td>A.3.3</td>
<td>Quantified Expressions</td>
<td>24</td>
</tr>
<tr>
<td>A.4</td>
<td>Concrete RSL Types: Values and Operations</td>
<td>25</td>
</tr>
<tr>
<td>A.4.1</td>
<td>Arithmetic</td>
<td>25</td>
</tr>
<tr>
<td>A.4.2</td>
<td>Set Expressions</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Set Enumerations</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Set Comprehension</td>
<td>25</td>
</tr>
<tr>
<td>A.4.3</td>
<td>Cartesian Expressions</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Cartesian Enumerations</td>
<td>25</td>
</tr>
<tr>
<td>A.4.4</td>
<td>List Expressions</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>List Enumerations</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>List Comprehension</td>
<td>26</td>
</tr>
<tr>
<td>A.4.5</td>
<td>Map Expressions</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>Map Enumerations</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>Map Comprehension</td>
<td>27</td>
</tr>
<tr>
<td>A.4.6</td>
<td>Set Operations</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>Set Operator Signatures</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>Set Examples</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>Informal Explication</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>Set Operator Definitions</td>
<td>28</td>
</tr>
<tr>
<td>A.4.7</td>
<td>Cartesian Operations</td>
<td>29</td>
</tr>
<tr>
<td>A.4.8</td>
<td>List Operations</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>List Operator Signatures</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>List Operation Examples</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>Informal Explication</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>List Operator Definitions</td>
<td>30</td>
</tr>
<tr>
<td>A.4.9</td>
<td>Map Operations</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>Map Operator Signatures and Map Operation Examples</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>Map Operation Explication</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>Map Operation Redefinitions</td>
<td>32</td>
</tr>
<tr>
<td>A.5</td>
<td>$\lambda$-Calculus + Functions</td>
<td>33</td>
</tr>
<tr>
<td>A.5.1</td>
<td>The $\lambda$-Calculus Syntax</td>
<td>33</td>
</tr>
<tr>
<td>A.5.2</td>
<td>Free and Bound Variables</td>
<td>33</td>
</tr>
<tr>
<td>A.5.3</td>
<td>Substitution</td>
<td>33</td>
</tr>
<tr>
<td>A.5.4</td>
<td>$\alpha$-Renaming and $\beta$-Reduction</td>
<td>34</td>
</tr>
<tr>
<td>A.5.5</td>
<td>Function Signatures</td>
<td>34</td>
</tr>
<tr>
<td>A.5.6</td>
<td>Function Definitions</td>
<td>34</td>
</tr>
<tr>
<td>A.6</td>
<td>Other Applicative Expressions</td>
<td>35</td>
</tr>
<tr>
<td>A.6.1</td>
<td>Simple let Expressions</td>
<td>35</td>
</tr>
<tr>
<td>A.6.2</td>
<td>Recursive let Expressions</td>
<td>35</td>
</tr>
<tr>
<td>A.6.3</td>
<td>Predicative let Expressions</td>
<td>36</td>
</tr>
<tr>
<td>A.6.4</td>
<td>Pattern and “Wild Card” let Expressions</td>
<td>36</td>
</tr>
<tr>
<td>A.6.5</td>
<td>Conditionals</td>
<td>36</td>
</tr>
<tr>
<td>A.6.6</td>
<td>Operator/Operand Expressions</td>
<td>37</td>
</tr>
<tr>
<td>A.7</td>
<td>Imperative Constructs</td>
<td>37</td>
</tr>
<tr>
<td>A.7.1</td>
<td>Statements and State Changes</td>
<td>37</td>
</tr>
<tr>
<td>A.7.2</td>
<td>Variables and Assignment</td>
<td>37</td>
</tr>
<tr>
<td>A.7.3</td>
<td>Statement Sequences and skip</td>
<td>38</td>
</tr>
<tr>
<td>A.7.4</td>
<td>Imperative Conditionals</td>
<td>38</td>
</tr>
<tr>
<td>A.7.5</td>
<td>Iterative Conditionals</td>
<td>38</td>
</tr>
<tr>
<td>A.7.6</td>
<td>Iterative Sequencing</td>
<td>38</td>
</tr>
<tr>
<td>A.8</td>
<td>Process Constructs</td>
<td>38</td>
</tr>
<tr>
<td>A.8.1</td>
<td>Process Channels</td>
<td>38</td>
</tr>
<tr>
<td>A.8.2</td>
<td>Process Composition</td>
<td>38</td>
</tr>
<tr>
<td>A.8.3</td>
<td>Input/Output Events</td>
<td>39</td>
</tr>
<tr>
<td>A.8.4</td>
<td>Process Definitions</td>
<td>39</td>
</tr>
<tr>
<td>A.9</td>
<td>Simple RSL Specifications</td>
<td>39</td>
</tr>
</tbody>
</table>
1 Introduction

We present a domain description of an abstracted credit card system. The narrative part of the description is terse, perhaps a bit too terse. I might “repair” this shortness if told so. A reference is made to my paper: [Bjø16, Manifest Domains: Analysis & Description]. That paper can be found on the Internet: http://www2.compute.dtu.dk/~dibj/2015/faoc/faoc-bjorner.pdf.

Credit cards are moving from simple plastic cards to smart phones. Uses of credit cards move from their mechanical insertion in credit card terminals to being swiped. Authentication (hence not modelled) moves from keying in security codes to eye iris “prints”, and/or finger prints or voice prints or combinations thereof.

This document abstracts from all that in order to understand a bare, minimum essence of credit cards and their uses. Based on a model, such as presented here, the reader should be able to extend/refine the model into any future technology – for requirements purposes.

2 Endurants

2.1 Credit Card Systems

[Bjø16, Sect.3.1.6, pg.11:]: observe_part_sorts

1 Credit card systems consists of three kinds of parts:
   a. a part, cs:CS, of credit cards
   b. a part, bs:BS, of banks, and
   c. a part, ss:SS, of shops.

type
1. CCS
   1a.. CS, C, CSI
   1b.. BS, B, BSI
   1c.. SS, S, SSI

case
1a.. obs_CS: CCS → CS, uid_CS: CS → CSI,
1b.. obs_BS: CCS → BS, uid_BS: BS → BSI,
1c.. obs_SS: CCS → SS, uid_SS: SS → SSI

The composite part CS can be thought of as a credit card company, say VISA. The composite part BS can be thought of as a bank society, say BBA: British Banking Association. The composite part SS can be thought of as the association of retailers, say bira: British Independent Retailers Association.[Bjø16, Sect.3.1.7, pg.13]: observe_part_type

---

1 We “equate” credit cards with their holders.
2 Our simple model allows for only one credit card company. But that model can easily be extended to model a set of credit card companies, viz.: VISA, MasterCard, American Express, Diner’s Club, etc..
3 The model does not prevent “shops” from being airlines, or car rental agencies, or dentists, or consultancy firms. In this case SS would be some appropriate association.
2 The credit card part, $cs:CS$, abstracts a set, $socs:Cs$, of credit cards.

3 The bank part, $bs:BS$, abstracts a set, $sobs:Bs$, of banks.

4 The shop part, $ss:SS$, abstracts a set, $soss:Sc$, of shops.

**type**
2. $Cs = C$-set, $C$
3. $Bs = B$-set, $B$
4. $Ss = S$-set, $S$

**value**
2. $obs_{CS}: CS \rightarrow Cs$, $obs_{Cs}: CS \rightarrow Cs$
3. $obs_{BS}: BS \rightarrow Bs$, $obs_{Bs}: BS \rightarrow Bs$
4. $obs_{SS}: SS \rightarrow Ss$, $obs_{Ss}: SS \rightarrow Ss$

[Bjø16, Sect.3.2, pg.16]: observe unique identifier

5 Each credit card, bank and shop has a unique identifier, $ci:CI$, $bi:BI$, respectively $si:SI$.

6 One can define functions which extract all the
   a. unique credit card,
   b. bank and
   c. shop identifiers

   from a credit card system.

**type**
5. $Cl$, $Bl$, $Sl$

**value**
5. $uid_{C}: C \rightarrow Cl$
6. $uid_{B}: B \rightarrow Bl$
5. $uid_{S}: S \rightarrow Sl$

6a. $xtr_{Cls}: CCS \rightarrow Cl$-set
6b. $xtr_{Bls}(ccs) \equiv \{uid_{B}(b) | b: B \in obs_{Bs}(obs_{BS}(ccs))\}$
6c. $xtr_{Sls}(ccs) \equiv \{uid_{S}(s) | s: S \in obs_{Ss}(obs_{SS}(ccs))\}$

7 For all credit card systems it is the case that
   a. all credit card identifiers are distinct from bank identifiers,
   b. all credit card identifiers are distinct from shop identifiers,
   c. all shop identifiers are distinct from bank identifiers,
axiom
7. \( \forall \text{ccs: CCS} \cdot \)
7. let cis = xtr\_Cls(ccs), bis = xtr\_Bls(ccs), sis = xtr\_Sl(s(ccs)) in
7a.. \( \text{cis} \cap \text{bis} = \{\} \)
7b.. \( \land \text{cis} \cap \text{sis} = \{\} \)
7c.. \( \land \text{sis} \cap \text{bis} = \{\} \) end

2.2 Credit Cards

We “equate” credit cards with their holders.

8 A credit card (besides a unique identification) has

\[Bjø16, \text{Sect.3.1.2, pg.17}: \text{observe \_mereology} \]

a. a mereology which “connects” it to any of the shops of the system and to exactly one bank of the system,

b. and some attributes — which we shall disregard.

c. The wellformedness of a credit card system includes the wellformedness of credit card mereologies with respect to the system of banks and shops:

i The unique shop identifiers of a credit card mereology must be those of the shops of the credit card system; and

ii the unique bank identifier of a credit card mereology must be of one of the banks of the credit card system.

type
8. C
8a.. \( \text{CM} = \text{SI\_set } \times \text{BI} \)
value
8a.. \( \text{mereso\_CM}: \text{C } \rightarrow \text{CM} \)
8c.. \( \text{wf\_CM\_of\_C}: \text{CCS } \rightarrow \text{Bool} \)
8c.. \( \text{wf\_CM\_of\_C}(\text{ccs}) = \)
8a.\( \text{let bis = xtr\_Bls(ccs), sis = xtr\_Sl(s(ccs)) in} \)
8a.. \( \forall c; c \in \text{obs\_Cs(\text{obs\_CS}(ccs))) } \Rightarrow \)
8a.. \( \text{let (ccsis, bi) = mereso\_CM(c) in} \)
8(c.\).
8(c.\).
8a.. \( \text{ccsis } \subseteq \text{sis} \)
8(c.\).
8a.. \( \land b \in \text{bis} \)
8a.. end end

Constraint 8(c.\).
8(c.\).
8a.. limits a credit card to potentially be used only in a proper subset of all shops. To allow for all shops one must change the wording to ‘be all those of the shops ...’, and change \( \subseteq \) in formula line 8(c.\.)ii to ‘=’.
2.3 Banks

Our model of banks is very limited.

9 A bank has

a. a unique bank identifier,
b. a mereology which “connects” it to a subset of all credit cards and a subset of all shops,
c. and, as attributes:
   i a cash register, and
   ii a ledger.

9(c.)ii. The ledger records

a. for every card, by unique credit card identifier,
b. the current balance: how much money, credit or debit, i.e., plus or minus, that customer is
   owed, respectively has borrowed from the bank,
c. the dates-of-issue and -expiry of the credit card, and
d. the name, address, and other information about the credit card holder.

10 The wellformedness of the credit card system includes the wellformedness of the banks with
respect to the credit cards and shops:

a. the bank mereology’s
b. must list a subset of the credit card identifiers and a subset of the shop identifiers.

<table>
<thead>
<tr>
<th>type</th>
<th>9. B</th>
</tr>
</thead>
<tbody>
<tr>
<td>9b.. BM = Cl-set × S1-set</td>
<td></td>
</tr>
<tr>
<td>9(c.)i. CR = Bal</td>
<td></td>
</tr>
<tr>
<td>9(c.)ii. LG = Cl → (Bal × DoI × DoE × ...)</td>
<td></td>
</tr>
<tr>
<td>9b.. Bal = Int</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>value</th>
<th>9b.. mereo_B: B → BM</th>
</tr>
</thead>
<tbody>
<tr>
<td>9(c.)i. attr_CR: B → CR</td>
<td></td>
</tr>
<tr>
<td>9(c.)ii. attr_LG: B → LG</td>
<td></td>
</tr>
<tr>
<td>10. wf_BM_B: CCS → Bool</td>
<td></td>
</tr>
<tr>
<td>10. wf_BM_B(ccs) ≡</td>
<td></td>
</tr>
<tr>
<td>10a.. let allcis = xtr_CIs(ccs), allsis = xtr_SIs(ccs) in</td>
<td></td>
</tr>
<tr>
<td>10. ∀ b:B • b ∈ obs_Bs(obs_BS(ccs)) in</td>
<td></td>
</tr>
<tr>
<td>10b.. let (cis,sis) = mereo_B(b) in</td>
<td></td>
</tr>
<tr>
<td>10.. cis ⊆ ∀ cis ∧ sis ⊆ allsis</td>
<td></td>
</tr>
<tr>
<td>10. end end</td>
<td></td>
</tr>
</tbody>
</table>
2.4 Shops

11 A shop (besides a unique shop identifier) has a
   a. mereology and some attributes.

11a. The mereology of a shop is a pair:
   a. a unique bank identifiers, and
   b. a set of unique credit card identifiers.

11a.. We omit treatment of shop attributes.

11a.. The mereology of a shop
   a. must list a bank of the credit card system,
   b. and a subset (or all) of the unique credit identifiers.

\[
\text{type} \quad SM = \text{Cl-set} \times \text{Bl}
\]
\[
\text{value} \quad \text{mereo}_S : S \rightarrow SM
\]
\[
\text{value} \quad \text{wf}_S : \text{CCS} \rightarrow \text{Bool}
\]
\[
\text{value} \quad \text{wf}_S(c) \equiv
\]
\[
\text{let} \quad \text{allcis} = \text{xtr}_S(c), \text{allbis} = \text{xtr}_S(c) \quad \text{in}
\]
\[
\forall s:S \cdot s \in \text{obs}_S(\text{obs}_S(c)) \Rightarrow
\]
\[
\text{let} \quad (\text{cis}, \text{bi}) = \text{mereo}_S(s) \quad \text{in}
\]
\[
\text{cis} \subseteq \text{allcis}
\]
\[
\land \text{bi} \in \text{allbis}
\]
\[
\text{end end}
\]

3 Perdurants

3.1 Behaviours

3.1.1 System

[Bjø16, Sect.4.11.2, pg.35]: Process Schema I: Abstract is \textit{composite}(p) and Process Schema II: Concrete is \textit{concrete}(p).

12 We ignore the behaviours related to the CCS, CS, HS and SS parts.

13 We therefore only consider the behaviours related to the Cs, Hs and Ss parts.

14 And we therefore compile the credit card system into the parallel composition of the parallel compositions of all the credit card, \textit{crd}, all the bank, \textit{bnk}, and all the shop, \textit{shp}, behaviours.
value
12. ccs: CCS
12. cs: CS = obs CS(ccb), uics: CSI = uid CS(cs),
12. bs: BS = obs BS(ccb), uibs: BSI = uid BS(bs),
12. ss: SS = obs SS(ccb), uiss: SSI = uid SS(ss),
13. socs: Cs = obs Cs(cs),
13. sobs: Bs = obs Bs(bs),
13. sosss; Ss = obs Ss(ss),
14. sys: Unit → Unit,
12. sys() ≡
14. cards_{uics}(mereo CS(cs),...) ∥ ||{crd_{uid,C}(c)(mereo C(c))|c:C ∈ socs}
14. || banks_{uibs}(mereo BS(bs),...) ∥ ||{bnk_{uid,B}(b)(mereo B(b))|b:B ∈ sobs}
14. || shops_{uiss}(mereo SS(ss),...) ∥ ||{shp_{uid,S}(s)(mereo S(s))|s:S ∈ sosss},
12. cards_{uics}(...) ≡ skip,
12. banks_{uibs}(...) ≡ skip,
12. shops_{uiss}(...) ≡ skip
axiom skip || behaviour(...) ≡ behaviour(...)

3.1.2 Channels

[Bjø16, Sect.4.5.1, pg.30]: Channels and Communication

15 Credit card behaviours interact with
   a. many bank (each with one) and
   b. many shop behaviours.

16 Shop behaviours interact with
   a. many bank (each with one) and
   15b.. many credit card behaviours.

15a...16a.. Bank behaviours interact with many credit card and many shop behaviours.

17 The inter-behaviour interactions concern:
   a. between credit cards and banks: withdrawal requests as to a sufficient, mk Wdrw(am),
      balance on the credit card account for buying am:aM amounts of goods or services, with
      the bank response of either is_OK() or is_NOK(), or the revoke of a card;
   b. between credit cards and shops: the buying, for an amount, am:aM, of goods or services:
      mk Buy(am), or the refund of an amount; and
   c. between shops and banks: the deposit of an amount, am:aM, in the shops’ bank account:
      mk Dep(am).
channel
15a. \{ch_{cb}[ci,bi]|ci:Cl, bi:Bl|ci ∈ cis ∧ bi ∈ bis\}:CB_Msg
15b. \{ch_{cs}[ci,si]|ci:Cl, si:Sl|ci ∈ cis ∧ si ∈ sis\}:CS_Msg
16a. \{ch_{sb}[si,bi]|si:Sl, bi:Bl|si ∈ sis ∧ bi ∈ bis\}:SB_Msg

type
17a. CB_Msg == mk_Wdrw(am:aM) | is_OK() | is_NOK() | mk_Rev()
17b. CS_Msg == mk_Buy(am:aM) | mk_Ref(am:aM)
17c. SB_Msg == mk_Dep((ci:CI, si:SI), am:aM)

3.1.3 Behaviour Interactions

18 The credit card initiates

a. buy transactions

i. [1.Buy] by inquiring with its bank as to sufficient purchase funds;

ii. [2.Buy] if NOK then there are presently no further actions; if OK

iii. [3.Buy] the credit card behaviour requests the purchase from the shop – handing it an appropriate amount \(am:aM\);

iv. [4.Buy] finally the shop requests its bank to deposit the purchase amount in the shop’s bank account.

b. refund transactions

i. [1.Refund] by requesting such refunds, in the amount of \(am:aM\), from any shop;

ii. [2.Refund] whereupon the shop requests its bank to move the amount \(am:aM\) from the shop’s bank account to the credit card’s account.

Thus the three sets of behaviours, \(crd\), \(bnk\) and \(shp\) interact as sketched in Fig. 1 on the following page.

[1.Buy] Item 23 on the next page: ch_{cb}[ci,bi]!mk_Wdrw(am);

Item 33 on Page 12: let mk_Wdrw(ci,am) = [] \{ch_{cb}[bi,bi]?|ci:Cl\} in


Item 33 on Page 12: let mk_Buy(am) = [] \{ch_{cs}[ci,si]?|ci:Cl\} in

[3.Buy] Item 25 on the next page: ch_{cs}[ci,si]!mk_Buy(am)

Item 45 on Page 13: let mk_Dep(si,am) = [] \{ch_{cs}[ci,si]?|ci:Cl\} in

[4.Buy] Item 45 on Page 13: ch_{sb}[si,bi]!mk_Dep(si,am)

Item 39 on Page 12: let mk_Dep(si,am) = [] \{ch_{cs}[ci,si]?|si:Sl\} in

[1.Refund] Item 27 on Page 11: ch_{cs}[ci,si]!mk_Ref((ci,si),am)

Item 46 on Page 13: let (si,mk_Ref(ci,am)) = [] \{(si',ch_{sb}[si,bi]?)|si,si':Sl\} in

[2.Refund] Item 41 on Page 13: ch_{sb}[ci,bi]!mk_Ref(ci,am)

3.1.4 Credit Card

19 The credit card behaviour, \(crd\), takes the credit card unique identifier, the credit card mereology, and attribute arguments (omitted). The credit card behaviour, \(crd\), accepts inputs from and offers outputs to the bank, \(bi\), and any of the shops, \(si\).
The credit card behaviour, $\text{crd}$, non-deterministically, internally “cycles” between

a. buying,

b. getting refunds.

\begin{equation}
\text{crd}_{ci} \buildrel \text{CI} \over \longrightarrow \text{CM} \rightarrow \text{in, out} \ ch_{cb}[ci,bi],\{ch_{cs}[ci,si]|si:SI \in \text{sis}\} \quad \text{Unit}
\end{equation}

\begin{align}
\text{20a.} & \quad \text{buy}(ci,(bi,sis)) \\
\text{20b.} & \quad \text{rfu}(ci,(bi,sis))
\end{align}

By $am:AM$ we mean an amount of money, and by $si:SI$ we refer to a shop in which we have selected a number or goods or services (not detailed) costing $am:AM$.

The amount for which to buy and the shop from which to buy are selected (arbitrarily).

The credit card (holder) withdraws $am:AM$ from the bank, if sufficient funds are available$^4$.

The response from the bank

is either OK and the credit card [holder] completes the purchase by buying the goods or services offered by the selected shop.

$^4$First the credit card [holder] requests a withdrawal. If sufficient funds are available, then the withdrawal takes place, otherwise not – and the credit card holder is informed accordingly.
26 or the response is “not OK”, and the transaction is skipped.

```
type
21. AM = Int
value
20a.. buy: ci:CI × (bi,sis):CM → \text{in, out} ch\_cb[ci,bi] \text{ out} \{ch\_cs[ci,si]|si:SI\,si \in sis\} \text{ Unit}
20a.. buy(ci,(bi,sis)) ≡
22. let am:AM • am>0, si:SI • si \in sis \text{ in} See Discussion note 49a. on Page 15
23. let msg = (ch\_cb[ci,bi]!mk\_Wdrw(am);ch\_cb[ci,bi]?) \text{ in}
24. case msg of
25. is\_OK() → ch\_cs[ci,si]!mk\_Buy(am),
26. is\_NOK() → \text{skip}
20a.. \text{ end end end}
```

27 The credit card [handler] requests a refund \textit{am:AM} from shop \textit{si:SI}.

This request is handled by the shop behaviour’s sub-action \textit{ref}, see lines 43.–47. page 13.

```
value
20b.. rfu: ci:CI × (bi,sis):CM → \text{ out} \{ch\_cs[ci,si]|si:SI\,si \in sis\} \text{ Unit}
20b.. rfu(ci,(bi,sis)) ≡
27. let am:AM • am>0, si:SI • si \in sis \text{ in} See Discussion note 49b. on Page 15
27. ch\_cs[ci,si]!mk\_Ref((ci,si),am)
19. end
```

### 3.1.5 Banks

28 The bank behaviour, \textit{bnk}, takes the bank’s unique identifier, the bank mereology, and the programmable attribute arguments: the ledger and the cash register. The bank behaviour, \textit{bnk}, accepts inputs from and offers outputs to the any of the credit cards, \textit{ci\in cis}, and any of the shops, \textit{si\in sis}.

29 The bank behaviour non-deterministically internally chooses to accept

30 either withdrawal requests from credit cards

31 or deposit requests from shops or

32 or refund requests from credit cards.

```
value
28. bnk\_bi: (cis,sis):BM → (LG×CR) →
28. \text{ in, out} \{ch\_cb[ci,bi]|ci:CI\,ci \in cis\} \{ch\_sb[si,bi]|si:SI\,si \in sis\} \text{ Unit}
28. bnk\_bi((cis,sis))(lg:(bal,doi,doe,...),cr) ≡
30. withdraw(ci,(cis,sis))(lg,cr)
29. ⊤
```
31. deposit(bi,(cis,sis))(lg,cr)
29. ⌈⌉ ⌊⌋
32. refund(bi,(cis,sis))(lg,cr)

33 The withdraw request (an action) non-deterministically, externally offers to accept input from a credit card behaviour and marks the only possible form of input from credit cards, \( mk_{Wdrw}(am) \), with the identity of the credit card.

34 If the requested amount (to be withdrawn) is not within balance on the account
35 then we, at present, refrain from defining an outcome (chaos),
36 otherwise the bank behaviour informs the credit card behaviour that the amount can be withdrawn.
37 Whereupon the bank behaviour is resumed notifying a lower balance and “withdraws” the monies from the cash register.

value
30. withdraw: bi:BI \times (cis,sis):BM \rightarrow (LG \times CR) \rightarrow \text{in,out} \ (ch_{cb}[bi,ci]|ci:\textit{Cl}:ci \in cis) \quad \text{Unit}
30. withdraw(bi,(cis,sis))(lg,cr) \equiv 
33. let mk_{Wdrw}(ci,am) = \{ ch_{cb}[ci,bi]|ci:\textit{Cl}:ci \in cis \} \text{ in}
37. let (bal,doi,doe) = lg(ci) \text{ in}
34. if am>bal
35. then ch_{cb}[ci,bi]!is_NOK()
36. else ch_{cb}[ci,bi]!is_OK() \text{ end } ;
37. bnk_{bi}(cis,sis)(lg[ci \rightarrow (bal-\text{am},doi,doe)],cr-\text{am})
38. end end

38 The deposit action is invoked, buy a shop behaviour, when a credit card [holder] buy’s for a certain amount, \textit{am:am}, or requests a refund of that amount. The deposit is made by shop behaviours, either on behalf of themselves, hence \textit{am:am}, is to be inserted into the shops’ bank account, \textit{si:SI}, or on behalf of a credit card [i.e., a customer], hence \textit{am:am}, is to be inserted into the credit card holder’s account, \textit{ci:CI}.

39 The message, \textit{ch_cs}[ci,si]?, received from a credit card behaviour is either concerning a buy [in which case \textit{i} is a cit:CI, hence sale, or a refund order [in which case \textit{i} is a si:SI].

40 In either case, the respective bank account is “upped” by \textit{am:am} – and the bank behaviour is resumed.

value
31. deposit: bi:BI \times (cis,sis):BM \rightarrow (LG \times CR) \rightarrow \text{in,out} \ (ch_{sb}[bi,si]|si:\textit{Sl}:si \in sis) \quad \text{Unit}
31. deposit(bi,(cis,sis))(lg,cr) \equiv 
39. let mk_{Dep}(si,am) = \{ ch_{cs}[ci,si]|si:\textit{Sl}:si \in sis \} \text{ in}
37. let (bal,doi,doe) = lg(si) \text{ in}
40. bnk_{bi}(cis,sis)(lg[si \rightarrow (bal+\text{am},doi,doe)],cr+\text{am})
38. end end
The \textit{refund} action non-deterministically externally offers to accept a \texttt{mk\_Ref(ci,am)} request from a shop behaviour, \textit{si}.

The bank behaviour is then resumed with the credit card balance incremented by \textit{am} and the shop balance decremented by that same amount.

\begin{verbatim}
value
31. refund: bi:BI \times (cis,sis):BM \rightarrow (LG \times CR) \rightarrow \text{in, out} \{ ch\_sb[bi,si]|si:Sl\times si \in sis \} \text{ Unit}
31. refund(bi,(cis,sis))(lg,cr) ≡
41. \texttt{let (si,mk\_Ref(ci,am)) = } ⌈⌉ ⌊⌋ { } \text{ in }
32. \texttt{let (balc,doic,doec) = lg(ci), (sbal,sdoi,sdoe) = lg(si) in }
33. \texttt{bnk}(bi,(cis,si))(lg\[ci\rightarrow(bcal+am,cdoi,cdoe)]\[si\rightarrow(sbal−am,sdoi,sdoe)\],cr)
31. end end

3.1.6 Shops

The shop behaviour, \textit{shp}, takes the shop’s unique identifier, the shop mereology, and attribute arguments (omitted). The shop behaviour, \textit{shp}, accepts inputs from and offers outputs to the any of the credit cards, \textit{ci}∈cis, and any of the shops, \textit{si}∈sis.

The shop behaviour non-deterministically, externally

either offers to accept a Buy request from a credit card behaviour.

This input is (via the \textit{sale} action) of the form \texttt{mk\_Buy(am)},

or

offers to accept a refund request in this amount, \textit{am} from a credit card [holder].

This input is (via the \textit{refe} action) of the form \texttt{mk\_Ref(am)},

Whereupon the shop behaviour resumes being a shop behaviour.

\begin{verbatim}
value
43. shp_{si}: sm:(cis:Cl\_set\times bi:Bl) \times ... \rightarrow \text{in, out} : \{ cs[ci,si]|ci:Cl\times ci \in cis \},sb[si,bi] \text{ Unit}
43. shp_{si}((cis,bi),...) ≡
45. \texttt{let mk\_Buy(am) = } ⌈⌉ ⌊⌋ { } \text{ in }
46. \texttt{ch\_sb[si,bi]!mk\_Dep(si,am) end }
47. \texttt{shp}_{si}((cis,bi),...)

45. ref: sm:(cis:Cl\_set\times bi:Bl) \rightarrow \text{in, out} : \{ cs[ci,si]|ci:Cl\times ci \in cis \},sb[si,bi] \text{ Unit}
46. ref(si,(cis,bi)) ≡
\end{verbatim}
let \( \text{mk\_Ref}(ci,si,am) = \left\lfloor \{ \text{ch\_cs}[ci,si]\?|ci:\text{CI}\land ci \in cis\} \right\rfloor \in \text{ch\_sb}[ci,bi]\!\text{mk\_Ref}(ci,am) \) end

shp\_u((cis,bi),...)
4 Discussion

48 The credit card system narrated and formalised in this document is an abstraction. We claim that it portrays an essence of credit cards.

49 The reader may object to certain things:

a. We do not model how a credit card holder selects services from a service provider (here modelled as shops) or products in a shop. Nor do we model that the card holder actually obtains those services or products.

   All this is summarised in Item 3.1.4 on Page 11: let am:aM • am>0, si:Sl • si ∈ sis in.
   In other words: this is not considered an element of “an essence” of credit cards.

b. We, “similarly” do not model how the refund request is arrived at.

   All this is summarised in Item 3.1.4 on Page 11: let am:AM • am>0, si:Sl • si ∈ sis in.
   In other words: this is not considered an element of “an essence” of credit cards.

c. Also: we do not model whether the balance of the shop’s bank account is sufficient to refund a card holder.

d. Etcetera.

The present credit card system model can “easily” be extended to incorporate these and other matters.

50 Without showing explicit evidence we claim that present domain description can serve as a basis for both domain and requirements modelling standard as well as current and future credit/pay/etc. card systems.

51 Etcetera.

5 Bibliography

5.1 Some Remarks

We refer to texts on RSL and Software Engineering: [Bjø06a, Bjø06b, Bjø06c, Bjø08b, Bjø08c, Bjø08d, Bjø10a, Bjø10b, Bjø10c, Bjø09b]

5.2 Other Domain Descriptions

We list a number of reports all of which document descriptions of domains. These descriptions were carried out in order to research and develop the domain analysis and description concepts now summarised in the present paper. These reports ought now be revised, some slightly, others less so, so as to follow all of the prescriptions of the current paper. Except where a URL is given in full, please prefix the web reference with: http://www2.compute.dtu.dk/~dibj/.


5.2.1 Published Papers

- Web page www.imm.dtu.dk/~dibj/domains/ lists the published papers and reports mentioned below.

- I have thought about domain engineering for more than 25 years.

- But serious, focused writing only started to appear since [Bjø06c, Part IV] — with [Bjø03, Bjø97] being exceptions:
  - [Bjø07, 2007] suggests a number of domain science and engineering research topics;
  - [Bjø10d, 2008] covers the concept of domain facets;
  - [BE10, 2008] explores compositionality and Galois connections.
  - [Bjø08a, Bjø10f, 2008,2009] show how to systematically, but, of course, not automatically, “derive” requirements prescriptions from domain descriptions;
  - [Bjø11a, 2008] takes the triptych software development as a basis for outlining principles for believable software management;
  - [Bjø09a, Bjø14a, 2009,2013] presents a model for Stanislaw Leśniewski’s [?] concept of mereology;
  - [Bjø10e, Bjø11b] present an extensive example and is otherwise a precursor for the present paper;
  - [Bjø11c, 2010] presents, based on the TripTyCh view of software development as ideally proceeding from domain description via requirements prescription to software design, concepts such as software demos and simulators;
  - [Bjø13, 2012] analyses the TripTyCh, especially its domain engineering approach, with respect to Maslow’s and Peterson’s and Seligman’s notions of humanity: how can computing relate to notions of humanity;

---


6Character strengths and virtues: A handbook and classification. Oxford University Press, 2004
the first part of [Bjø14b, 2014] is a precursor for the present paper with its second part presenting a first formal model of the elicitation process of analysis and description based on the prompts more definitively presented in the current paper; and

[Bjø14c, 2014] focus on domain safety criticality.

The present paper basically replaces the domain analysis and description section of all of the above reference — including [Bjø06c, Part IV, 2006].

5.3 References


A RSL: The Raise Specification Language

A.1 Type Expressions

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

A.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”.

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

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"

A.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.

Concrete Composite Types 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:

7. A-set
8. A-infset
9. A × B × ... × C
10. A^*
The following are generic type expressions:

1. The Boolean type of truth values \texttt{false} and \texttt{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", ...
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 \(B \times C \times \ldots \times D\), 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., \((A \rightarrow^m B)\), or \((A^\ast\text{-set})\), or \((A\text{-set})\text{list}\), or \((A|B) \rightarrow^m (C|D|(E \rightarrow^m F))\), etc.
16. The postulated disjoint union of types A, B, ..., and C.
17 The record type of $\text{mk}\_\text{id}$-named record values $\text{mk}\_\text{id}(av, \ldots, bv)$, where $av, \ldots, bv$ are values of respective types. The distinct identifiers $\text{sel}\_a$, etc., designate selector functions.

18 The record type of unnamed record values $(av, \ldots, bv)$, where $av, \ldots, bv$ are values of respective types. The distinct identifiers $\text{sel}\_a$, etc., designate selector functions.

**Sorts and Observer Functions**

type

A, B, C, ..., D

value

$\text{obs}\_B$: $A \to B$, $\text{obs}\_C$: $A \to C$, ..., $\text{obs}\_D$: $A \to D$

- 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

$A = B \times C \times \ldots \times D$

### A.2 Type Definitions

#### A.2.1 Concrete Types

- Types can be concrete

- in which case the structure of the type is specified by type expressions:

type

$A = \text{Type}\_\text{expr}$

- Some schematic type definitions are:

  1. $\text{Type}\_\text{name} = \text{Type}\_\text{expr} /* \text{without } | \text{s or subtypes */}$
  2. $\text{Type}\_\text{name} = \text{Type}\_\text{expr}\_1 \mid \text{Type}\_\text{expr}\_2 \mid \ldots \mid \text{Type}\_\text{expr}\_n$
  3. $\text{Type}\_\text{name} = \mk\_\text{id}(s\_a1: \text{Type}\_\text{name}\_a1, \ldots, s\_ai: \text{Type}\_\text{name}\_ai) \mid$

   \ldots \mid

   $\mk\_\text{id}(s\_z1: \text{Type}\_\text{name}\_z1, \ldots, s\_zk: \text{Type}\_\text{name}\_zk)$

  4. $\text{Type}\_\text{name} = \{ \text{sel}\_a: \text{Type}\_\text{name}\_a \ldots \text{sel}\_z: \text{Type}\_\text{name}\_z$

  5. $\text{Type}\_\text{name} = \{ | v.\text{Type}\_\text{name}\{ \cdot \mathcal{P}(v) \} \}$
where a form of \([2-3]\) is provided by combining the types:

\[
\begin{align*}
\text{Type\_name} &= A \mid B \mid \ldots \mid Z \\
A &= \text{mk\_id\_1}(s_{a1:A_1}, \ldots, s_{ai:A_i}) \\
B &= \text{mk\_id\_2}(s_{b1:B_1}, \ldots, s_{bj:B_j}) \\
\ldots \\
Z &= \text{mk\_id\_n}(s_{z1:Z_1}, \ldots, s_{zk:Z_k})
\end{align*}
\]

Types \(A, B, \ldots, Z\) are disjoint, i.e., shares no values, provided all \(\text{mk\_id\_k}\) are distinct and due to the use of the disjoint record type constructor \(==\).

\textbf{axiom}
\[
\forall a1:A_1, a2:A_2, \ldots, ai:A_i \cdot \\
s_{a1}(\text{mk\_id\_1}(a1,a2,\ldots,ai))=a1 \land s_{a2}(\text{mk\_id\_1}(a1,a2,\ldots,ai))=a2 \land \\
\ldots \land s_{ai}(\text{mk\_id\_1}(a1,a2,\ldots,ai))=ai \land \\
\forall a:A \cdot \text{let } \text{mk\_id\_1}(a1',a2',\ldots,ai') = a \text{ in } \\
a1' = s_{a1}(a) \land a2' = s_{a2}(a) \land \ldots \land ai' = s_{ai}(a) \text{ end}
\]

\textbf{A.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 \(A:\)

\[
\text{type } A = \{ | b:B \cdot \mathcal{P}(b) | \}
\]

\textbf{A.2.3 Sorts — Abstract Types}

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

\[
\text{type } A, B, \ldots, C
\]

\textbf{A.3 The RSL Predicate Calculus}

\textbf{A.3.1 Propositional Expressions}

- Let identifiers (or propositional expressions) \(a, b, \ldots, c\) designate Boolean values (\textbf{true} or \textbf{false} [or \textbf{chaos}]).
Then:

false, true
\( a, b, ..., c \sim a, a \land b, a \lor b, a \Rightarrow b, a=b, a \neq b \)

are propositional expressions having Boolean values.

\( \sim, \land, \lor, \Rightarrow, = \) 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

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
\( a, b, ..., c \)
\( \sim a, a \land b, a \lor b, a \Rightarrow b, a=b, a \neq b \)
\( x=y, x \neq y \)
\( i<j, i \leq j, i \geq j, i \neq j, i \geq j, i > j \)

are simple predicate expressions.

A.3.3 Quantified Expressions

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

\( \forall x:X \cdot P(x) \)
\( \exists y:Y \cdot Q(y) \)
\( \exists! z:Z \cdot R(z) \)

are quantified expressions — also being predicate expressions.

They are “read” as: For all \( x \) (values in type \( X \)) the predicate \( P(x) \) holds; there exists (at least) one \( y \) (value in type \( Y \)) such that the predicate \( Q(y) \) holds; and there exists a unique \( z \) (value in type \( Z \)) such that the predicate \( R(z) \) holds.
A.4 Concrete RSL Types: Values and Operations

A.4.1 Arithmetic

type
Nat, Int, Real

class
value
+,-,∗: Nat × Nat → Nat | Int × Int → Int | Real × Real → Real
/: Nat × Nat ↦ Nat | Int × Int ↦ Int | Real × Real ↦ Real
<,≤,=,≠,≥,> (Nat|Int|Real) → (Nat|Int|Real)

A.4.2 Set Expressions

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\}, ...\} ∈ A-set
\{\{\}, \{a\}, \{e_1,e_2,...,e_n\}, ..., \{e_1,e_2,...\}\} ∈ A-infset

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 ↦ B

class
value
comprehend: A-infset × P × Q → B-infset
comprehend(s,P,Q) ≡ \{ Q(a) | a:A • a ∈ s ∧ P(a)\}

A.4.3 Cartesian Expressions

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

class
value
(e1,e2,...,en)
A.4.4 List Expressions

List Enumerations

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

\[
\{ \langle \rangle, \langle e \rangle, \ldots, \langle e_1,e_2,\ldots, en \rangle, \ldots \} \in A^*
\]
\[
\{ \langle \rangle, \langle e \rangle, \ldots, \langle e_1,e_2,\ldots, en \rangle, \ldots, \langle e_1,e_2,\ldots, en,\ldots \rangle, \ldots \} \in A^\omega
\]
\[
\langle a_i \ldots a_j \rangle
\]
- 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

- The last line below expresses list comprehension.

\[
\text{type } A, B, P = A \rightarrow \text{Bool}, Q = A \sim \rightarrow B
\]
\[
\text{value } \text{comprehend}: A^\omega \times P \times Q \sim \rightarrow B^\omega
\]
\[
\text{comprehend}(l,P,Q) \equiv \langle Q(l(i)) | i \in \langle 1..\text{len} l \rangle \cdot P(l(i)) \rangle
\]

A.4.5 Map Expressions

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:

\[
\text{type } T1, T2
\]
\[
M = T1 \rightarrow m T2
\]
\[
\text{value } u,u_1,u_2,\ldots, un:T1, v,v_1,v_2,\ldots, vn:T2
\]
\[
[\], [u \mapsto v], \ldots, [u_1 \mapsto v_1,u_2 \mapsto v_2,\ldots, un \mapsto vn] \forall \in M
\]
Map Comprehension

- The last line below expresses map comprehension:

\[
\begin{align*}
\text{type} & \quad U, V, X, Y \\
M & = U \rightarrow m \\
F & = U \sim \rightarrow X \\
G & = V \sim \rightarrow Y \\
P & = U \rightarrow \text{Bool} \\
\text{value} & \quad \text{comprehend}: M \times F \times G \times P \rightarrow (X \sim \rightarrow Y) \\
\text{comprehend}(m,F,G,P) & \equiv \\
& \quad [ F(u) \mapsto G(m(u)) \mid u:U \cdot u \in \text{dom} \ m \land P(u)] \\
\end{align*}
\]

A.4.6 Set Operations

Set Operator Signatures

\[
\begin{align*}
19 & \in: A \times A\text{-infset} \rightarrow \text{Bool} \\
20 & \notin: A \times A\text{-infset} \rightarrow \text{Bool} \\
21 & U: A\text{-infset} \times A\text{-infset} \rightarrow A\text{-infset} \\
22 & U: (A\text{-infset})\text{-infset} \rightarrow A\text{-infset} \\
23 & \cap: A\text{-infset} \times A\text{-infset} \rightarrow A\text{-infset} \\
24 & \cap: (A\text{-infset})\text{-infset} \rightarrow A\text{-infset} \\
25 & \setminus: A\text{-infset} \times A\text{-infset} \rightarrow A\text{-infset} \\
26 & \subset: A\text{-infset} \times A\text{-infset} \rightarrow \text{Bool} \\
27 & \subseteq: A\text{-infset} \times A\text{-infset} \rightarrow \text{Bool} \\
28 & =: A\text{-infset} \times A\text{-infset} \rightarrow \text{Bool} \\
29 & \neq: A\text{-infset} \times A\text{-infset} \rightarrow \text{Bool} \\
30 & \text{card}: A\text{-infset} \sim \rightarrow \text{Nat} \\
\end{align*}
\]

Set Examples

\[
\begin{align*}
\text{examples} & \quad a \in \{a,b,c\} \\
& \quad a \notin \{\}, \ a \notin \{b,c\} \\
& \quad \{a,b,c\} \cup \{a,b,d,e\} = \{a,b,c,d,e\} \\
& \quad \bigcup\{\{a\},\{a,b\},\{a,d\}\} = \{a,b,d\} \\
& \quad \{a,b,c\} \cap \{c,d,e\} = \{c\} \\
& \quad \cap\{\{a\},\{a,b\},\{a,d\}\} = \{a\} \\
& \quad \{a,b,c\} \setminus \{c,d\} = \{a,b\} \\
& \quad \{a,b\} \subset \{a,b,c\} \\
\end{align*}
\]
{a,b,c} ⊆ {a,b,c}
{a,b,c} = {a,b,c}
{a,b,c} \neq \{a,b\}
card \{\} = 0, card \{a,b,c\} = 3

Informal Explication

19 \in: The membership operator expresses that an element is a member of a set.
20 \notin: The nonmembership operator expresses that an element is not a member of a set.
21 \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.
22 \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.
23 \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.
24 \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.
25 \setminus: 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.
26 \subseteq: The proper subset operator expresses that all members of the left operand set are also in the right operand set.
27 \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.
28 =: The equal operator expresses that the two operand sets are identical.
29 \neq: The nonequal operator expresses that the two operand sets are not identical.
30 card: The cardinality operator gives the number of elements in a finite set.

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s' \cup s'' )</td>
<td>{ a</td>
</tr>
<tr>
<td>( s' \cap s'' )</td>
<td>{ a</td>
</tr>
<tr>
<td>( s' \setminus s'' )</td>
<td>{ a</td>
</tr>
<tr>
<td>( s' \subseteq s'' )</td>
<td>\forall a:A : a \in s' \Rightarrow a \in s''</td>
</tr>
<tr>
<td>( s' \subset s'' )</td>
<td>\exists a:A : a \in s' \land a \notin s''</td>
</tr>
<tr>
<td>( s' = s'' )</td>
<td>\forall a:A : a \in s' \Leftrightarrow a \in s''</td>
</tr>
<tr>
<td>( s' \subseteq s'' )</td>
<td>\exists s' \subseteq s''</td>
</tr>
</tbody>
</table>
\[ s' \neq s'' \equiv s' \cap s'' \neq \{\} \]
\[ \text{card } s \equiv \]
\[ \text{if } s = \{\} \text{ then } 0 \text{ else} \]
\[ \text{let } a : A \cdot a \in s \text{ in } 1 + \text{card } (s \setminus \{a\}) \text{ end end} \]
\[ \text{pre s */ is a finite set */} \]
\[ \text{card } s \equiv \text{chaos */ tests for infinity of s */} \]

A.4.7 Cartesian Operations

type
A, B, C
\[ g_0 : G_0 = A \times B \times C \]
\[ g_1 : G_1 = (A \times B \times C) \]
\[ g_2 : G_2 = (A \times B) \times C \]
\[ g_3 : G_3 = A \times (B \times C) \]

value
va : A, vb : B, vc : C, vd : D
\[(va, vb, vc) : G_0, \]
\[(va, vb, vc) : G_1, \]
\[((va, vb), vc) : G_2, \]
\[(va_3, (vb_3, vc_3)) : G_3 \]

decomposition expressions
\[ \text{let } (a_1, b_1, c_1) = g_0, \]
\[ (a_1', b_1', c_1') = g_1 \text{ in .. end} \]
\[ \text{let } ((a_2, b_2), c_2) = g_2 \text{ in .. end} \]
\[ \text{let } (a_3, (b_3, c_3)) = g_3 \text{ in .. end} \]

A.4.8 List Operations

List Operator Signatures

value
\[ \text{hd: } A^\omega \hookrightarrow A \]
\[ \text{tl: } A^\omega \hookrightarrow A^\omega \]
\[ \text{len: } A^\omega \hookrightarrow \text{Nat} \]
\[ \text{inds: } A^\omega \rightarrow A^\omega - \text{infset} \]
\[ \text{elems: } A^\omega \rightarrow A^\omega - \text{infset} \]
\[ .(\cdot): A^\omega \times \text{Nat} \hookrightarrow A \]
\[ \wedge: A^* \times A^\omega \rightarrow A^\omega \]
\[ =: A^\omega \times A^\omega \rightarrow \text{Bool} \]
\[ \neq: A^\omega \times A^\omega \rightarrow \text{Bool} \]
List Operation Examples

examples
\[ \begin{align*} 
\text{hd}(a_1,a_2,\ldots,a_m) &= a_1 \\
\text{tl}(a_1,a_2,\ldots,a_m) &= (a_2,\ldots,a_m) \\
\text{len}(a_1,a_2,\ldots,a_m) &= m \\
\text{inds}(a_1,a_2,\ldots,a_m) &= \{1,2,\ldots,m\} \\
\text{elems}(a_1,a_2,\ldots,a_m) &= \{a_1,a_2,\ldots,a_m\} \\
(a_1,a_2,\ldots,a_m)(i) &= a_i \\
(a,b,c) \hat{\ast} (a,b,d) &= (a,b,c,a,b,d) \\
(a,b,c) &= (a,b,c) \\
(a,b,c) &\neq (a,b,d) 
\end{align*} \]

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 \(\ell\) having a number of elements larger than or equal to \(i\), gives the \(i\)th element of the list.
- **\(\hat{\ast}\)**: 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:

**List Operator Definitions**

\[
\text{value} \\
\text{is\_finite\_list}: \mathbb{A}^\omega \to \mathbb{B}
\]

\[
\text{len} \ q \equiv \\
\text{case} \ \text{is\_finite\_list}(q) \ \text{of} \\
\quad \text{true} \to \text{if} \ q = {} \ \text{then} \ 0 \ \text{else} \ 1 + \ \text{len} \ \text{tl} \ q \ \text{end} \\
\quad \text{false} \to \text{chaos} \ \text{end}
\]
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)ˆq’ 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” ≡
  indq iq’ = indq iq” ∧ ∀ i:Nat • i ∈ indq iq’ ⇒ iq’(i) = iq”(i)

iq’ ≠ iq” ≡ ∼(iq’ = iq”)

A.4.9 Map Operations
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’”]

∪: M × M → M [merge ∪]
  [a→b,a’→b’,a”→b’”] ∪ [a”’→b’”] = [a→b,a’→b’,a”→b”’]
\[ \setminus : \mathcal{M} \times \text{A-infset} \to \mathcal{M} \text{ [restriction by]} \]
\[ [a \mapsto b, a' \mapsto b', a'' \mapsto b''] \setminus \{a\} = [a' \mapsto b', a'' \mapsto b''] \]

\[ / : \mathcal{M} \times \text{A-infset} \to \mathcal{M} \text{ [restriction to]} \]
\[ [a \mapsto b, a' \mapsto b', a'' \mapsto b''] / \{a', a''\} = [a' \mapsto b', a'' \mapsto b''] \]

\[ =, \neq : \mathcal{M} \times \mathcal{M} \to \text{Bool} \]
\[ \circ : (\mathcal{A} \to \mathcal{B}) \times (\mathcal{B} \to \mathcal{C}) \to (\mathcal{A} \to \mathcal{C}) \text{ [composition]} \]
\[ [a \mapsto b, a' \mapsto b', a'' \mapsto b''] \circ [b \mapsto c, b' \mapsto c', b'' \mapsto c''] = [a \mapsto c, a' \mapsto c'] \]

**Map Operation Explication**

- **\( m(a) \):** Application gives the element that \( a \) maps to in the map \( m \).
- **\( \text{dom} \):** Domain/Definition Set gives the set of values which maps to in a map.
- **\( \text{rng} \):** Range/Image Set gives the set of values which are mapped to in a map.
- \( \dagger \): 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.
- \( \setminus \): 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 nonequal 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 \( c \).

**Map Operation Redefinitions**

The map operations can also be defined as follows:

**value**

\[ \text{rng} \ m \equiv \{ m(a) \mid a : \mathcal{A} \cdot a \in \text{dom} \ m \} \]

\[ m_1 \dagger m_2 \equiv \]
\[ [ a \mapsto b \mid a : \mathcal{A}, b : \mathcal{B} \cdot \]
\[ a \in \text{dom} \ m_1 \setminus \text{dom} \ m_2 \wedge b = m_1(a) \vee a \in \text{dom} \ m_2 \wedge b = m_2(a) ] \]
\[ m_1 \cup m_2 \equiv [ \ a \mapsto b \mid a : A, b : B \ \cdot \ a \in \text{dom} \ m_1 \land b = m_1(a) \lor a \in \text{dom} \ m_2 \land b = m_2(a) ] \]

\[ m \setminus s \equiv [ \ a \mapsto m(a) \mid a : A \cdot a \in \text{dom} \ m \setminus s ] \]

\[ m / s \equiv [ \ a \mapsto m(a) \mid a : A \cdot a \in \text{dom} \ m \cap s ] \]

\[ m_1 = m_2 \equiv \text{dom} \ m_1 = \text{dom} \ m_2 \land \forall a : A \cdot a \in \text{dom} \ m_1 \Rightarrow m_1(a) = m_2(a) \]

\[ m_1 \neq m_2 \equiv \sim(m_1 = m_2) \]

\[ m^n \equiv [ \ a \mapsto c \mid a : A, c : C \cdot a \in \text{dom} \ m \land c = n(m(a)) ] \]

\[ \text{pre rng } m \subseteq \text{dom } n \]

### A.5 \(\lambda\)-Calculus + Functions

#### A.5.1 The \(\lambda\)-Calculus Syntax

**type** /* A BNF Syntax: */

\[ \langle L \rangle ::= \langle V \rangle \mid \langle F \rangle \mid \langle A \rangle \mid (\langle L \rangle) \]

\[ \langle V \rangle ::= */ \text{variables, i.e. identifiers } */ \]

\[ \langle F \rangle ::= \lambda \langle V \rangle \cdot \langle L \rangle \]

\[ \langle A \rangle ::= (\langle L \rangle)(\langle L \rangle) \]

**value** /* Examples */

\[ \langle L \rangle: e, f, a, ... \]

\[ \langle V \rangle: x, ... \]

\[ \langle F \rangle: \lambda x \cdot e, ... \]

\[ \langle A \rangle: f a, (f a), f(a), (f)(a), ... \]

#### A.5.2 Free and Bound Variables

Let \(x, y\) be variable names and \(e, f\) be \(\lambda\)-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).

#### A.5.3 Substitution

In RSL, the following rules for substitution apply:

- \(\text{subst}([N/x]x) \equiv N;\)
\[ \text{subst}(\{N/x\}a) \equiv a, \]

for all variables \( a \neq x \);

\[ \text{subst}(\{N/x\}(P\ Q)) \equiv (\text{subst}(\{N/x\}P) \ \text{subst}(\{N/x\}Q)); \]

\[ \text{subst}(\{N/x\}(\lambda x\cdot P)) \equiv \lambda y\cdot \text{subst}(\{N/x\}P), \]

if \( x \neq y \) and \( y \) is not free in \( N \) or \( x \) is not free in \( P \);

\[ \text{subst}(\{N/x\}(\lambda y\cdot P)) \equiv \lambda z\cdot \text{subst}(\{N/z\}\text{subst}(\{z/y\}P)), \]

if \( y \neq x \) and \( y \) is free in \( N \) and \( x \) is free in \( P \)

(where \( z \) is not free in \( (N\ P) \)).

A.5.4 \( \alpha \)-Renaming and \( \beta \)-Reduction

- \( \alpha \)-renaming: \( \lambda x\cdot M \)
  
  If \( x, y \) are distinct variables then replacing \( x \) by \( y \) in \( \lambda x\cdot M \) results in \( \lambda y\cdot \text{subst}(\{y/x\}M) \). We can rename the formal parameter of a \( \lambda \)-function expression provided that no free variables of its body \( M \) thereby become bound.

- \( \beta \)-reduction: \( (\lambda x\cdot M)(N) \)
  
  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. \( (\lambda x\cdot M)(N) \equiv \text{subst}(\{N/x\}M) \)

A.5.5 Function Signatures

For sorts we may want to postulate some functions:

\textbf{type}

- \( A, B, C \)

\textbf{value}

- \( \text{obs\_B}: A \rightarrow B \)
- \( \text{obs\_C}: A \rightarrow C \)
- \( \text{gen\_A}: B \times C \rightarrow A \)

A.5.6 Function Definitions

Functions can be defined explicitly:

\textbf{value}

- \( f: \text{Arguments} \rightarrow \text{Result} \)
- \( f(\text{args}) \equiv \text{DValueExpr} \)
g: Arguments $\rightsquigarrow$ Result
\[ g(args) \equiv \text{ValueAndStateChangeClause} \]
\[ \text{pre} \ P(args) \]

Or functions can be defined implicitly:

\[
\text{value}
\]
\[
f: \text{Arguments} \rightarrow \text{Result}
\]
\[
f(args) \ \text{as result}
\]
\[
\text{post} \ P1(args,\text{result})
\]
\[
g: \text{Arguments} \approx \rightarrow \text{Result}
\]
\[
g(args) \ \text{as result}
\]
\[
\text{pre} \ P2(args)
\]
\[
\text{post} \ P3(args,\text{result})
\]

The symbol $\approx \rightarrow$ 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.

### A.6 Other Applicative Expressions

#### A.6.1 Simple let Expressions

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

\[
\text{let } a = E \ \text{in} \ E_b(a) \ \text{end}
\]

is an “expanded” form of:

\[
(\lambda a.E_b(a))(E_d)
\]

#### A.6.2 Recursive let Expressions

Recursive let expressions are written as:

\[
\text{let } f = \lambda a:A \cdot E(f) \ \text{in} \ B(f,a) \ \text{end}
\]

is “the same” as:

\[
\text{let } f = YF \ \text{in} \ B(f,a) \ \text{end}
\]

where:

\[
F \equiv \lambda g:\lambda a:(E(g)) \text{ and } YF = F(YF)
\]
A.6.3 Predicative let Expressions

Predicative let expressions:

\[
\text{let } a: A \cdot P(a) \text{ in } B(a) \text{ end}
\]

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

A.6.4 Pattern and “Wild Card” let Expressions

Patterns and wild cards can be used:

\[
\begin{align*}
\text{let } \{a\} \cup s = \text{set in ... end} \\
\text{let } \{a, \_\} \cup s = \text{set in ... end} \\
\text{let } (a, b, \ldots , c) = \text{cart in ... end} \\
\text{let } (a, \_ , \ldots , c) = \text{cart in ... end} \\
\text{let } \langle a \rangle ^\ell = \text{list in ... end} \\
\text{let } \langle a, \_ \rangle ^\ell = \text{list in ... end} \\
\text{let } [a \mapsto b] \cup m = \text{map in ... end} \\
\text{let } [a \mapsto b, \_ ] \cup m = \text{map in ... end}
\end{align*}
\]

A.6.5 Conditionals

Various kinds of conditional expressions are offered by RSL:

\[
\begin{align*}
\text{if } b_{\text{expr}} \text{ then } c_{\text{expr}} \text{ else } a_{\text{expr}} \text{ end}
\end{align*}
\]

\[
\begin{align*}
\text{if } b_{\text{expr}} \text{ then } c_{\text{expr}} \text{ end} \equiv \text{ /* same as: */}
\text{if } b_{\text{expr}} \text{ then } c_{\text{expr}} \text{ else skip end}
\end{align*}
\]

\[
\begin{align*}
\text{if } b_{\text{expr}1} \text{ then } c_{\text{expr}1} \\
\text{elsif } b_{\text{expr}2} \text{ then } c_{\text{expr}2} \\
\text{elsif } b_{\text{expr}3} \text{ then } c_{\text{expr}3} \\
\ldots \\
\text{elsif } b_{\text{expr}n} \text{ then } c_{\text{expr}n} \text{ end}
\end{align*}
\]

\[
\begin{align*}
\text{case expr of}
\text{ choice_pattern}_1 \rightarrow \text{expr}_1, \\
\text{ choice_pattern}_2 \rightarrow \text{expr}_2.
\end{align*}
\]
... choice_pattern n_or_wild_card → expr_n
end

A.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> ::=!

A.7 Imperative Constructs

A.7.1 Statements and State Changes

Often, following the RAISE method, software development starts with highly abstract-applicative constructs which, through stages of refinements, are turned into concrete and imperative constructs. Imperative constructs are thus inevitable in RSL.

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.

A.7.2 Variables and Assignment

0. variable v:Type := expression
1. v := expr
A.7.3 Statement Sequences and skip

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

2. skip
3. \( \text{stm}_1;\text{stm}_2;\ldots;\text{stm}_n \)

A.7.4 Imperative Conditionals

4. \( \text{if } \text{expr} \text{ then } \text{stm}_c \text{ else } \text{stm}_a \text{ end} \)
5. \( \text{case } e \text{ of: } p_1 \rightarrow S_1(p_1), \ldots, p_n \rightarrow S_n(p_n) \text{ end} \)

A.7.5 Iterative Conditionals

6. \( \text{while } \text{expr} \text{ do } \text{stm} \text{ end} \)
7. \( \text{do } \text{stmt} \text{ until } \text{expr} \text{ end} \)

A.7.6 Iterative Sequencing

8. \( \text{for } e \text{ in list}\_\text{expr} \cdot P(b) \text{ do } S(b) \text{ end} \)

A.8 Process Constructs

A.8.1 Process Channels

Let A and B stand for two types of (channel) messages and \( i:K_{\text{ldx}} \) for channel array indexes, then:

channel \( c:A \)

channel \( \{ \ k[i]:B \cdot i:K_{\text{ldx}} \ \} \)

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

A.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 $\parallel$ Q \hspace{1em} \text{Parallel composition}

P \# Q \hspace{1em} \text{Nondeterministic external choice (either/or)}

P \@ Q \hspace{1em} \text{Nondeterministic internal choice (either/or)}

P \| Q \hspace{1em} \text{Interlock parallel composition}

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

A.8.3 Input/Output Events

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

\begin{align*}
    c &\leadsto k[i] \text{ ? Input} \\
    c &\rightarrow e, k[i] \rightarrow e \text{ Output}
\end{align*}

- expresses the willingness of a process to engage in an event that
  - $\diamond$ “reads” an input, respectively
  - $\lozenge$ “writes” an output.

A.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.

\begin{align*}
    \text{value} & \\
    P : \text{Unit} &\rightarrow \text{in} c \text{ out} k[i] \\
    \text{Unit} & \\
    Q : i:K\text{Idx} &\rightarrow \text{out} c \text{ in} k[i] \text{ Unit}
\end{align*}

\begin{align*}
    P() &\equiv ... c ? ... k[i] ! e ... \\
    Q(i) &\equiv ... k[i] ? ... c ! e ...
\end{align*}

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

A.9 Simple RSL Specifications

Often, we do not want to encapsulate small specifications in schemes, classes, and objects, as is often done in RSL. An RSL specification is simply a sequence of one or more types, values (including functions), variables, channels and axioms:

\begin{align*}
    \text{type} & \\
    \text{...} & \\
    \text{variable} & \\
    \text{...} & \\
\end{align*}
In practice a full specification repeats the above listings many times, once for each “module” (i.e., aspect, facet, view) of specification. Each of these modules may be “wrapped” into scheme, class or object definitions.