Data Modelling & Model-oriented Specification

Contents

Summary of RSL Type Definitions

You have learned about two main kinds:

- sort definitions (**type** id)
  which declare names of *abstract* types
- abbreviation definitions (**type** id = type_expr)
  which (usually) give names to *concrete* types

Later in this semester you will learn about:

- short record definitions
  (a la records in programming languages)
- union and variant type definitions
  (a la ML's datatype declarations)
  includes definition of
  - enumeration types
    type id == v1 | ... | vn
  - option types
    type optT == none | some(valof : T)

Summary of RSL concrete data types

- basic types (**Bool**, **Nat**, **Int**, **Real**, **Char**, **Text**, **Unit**)
- composite types
  - products (T1 × T2)
  - functions (T1 → T2, T1 $\xrightarrow{\sim}$ T2)
  - sets (T-**set**, T-**infset**)
  - lists (T*, T$^\omega$)
  - maps (T1 $\xrightarrow[m]{}$ T2, T1 $\xrightarrow[m]{\sim}$ T2)
- subtypes of concrete types

Short Records

Example

Declaration of a record type named Book:

type
Book ::
title : Title
author : Author
publisher : Publisher

Construction of values of type Book:

mk_Book(t, a, p), where t : Title, a : Author, p : Publisher

Field extraction of a Book value b:

title(b)
author(b)
publisher(b)

Short Records
General form

Declaration of a record type named id:

type
id ::
field_1 : type_expr_1
⋮
field_n : type_expr_n

(where $n \geq 1$)

Values of type id:

$\text{mk\_id}(v_1, ..., v_n)$, where v_i : type_expr_i

Field extraction of a value v : id:

$\text{field}_i(v)$

Short Records versus Products

1. **type** id :: field_1 : type_expr_1 ... field_n : type_expr_n
2. **type** id = $\text{type\_expr}_1 \times ... \times \text{type\_expr}_n$

Construction of values:

1. $\text{mk\_id}(v_1, ..., v_n)$, where v_i : type_expr_i
2. $(v_1, ..., v_n)$, where v_i : type_expr_i

Extraction of field values of v : id:

1. $\text{field}_i(v)$
2. **let** $(v_1, ..., v_n) = v$ **in** v_i **end**

Specification styles
Model-oriented versus Property-oriented

Characterization:

- **algebraic/property-oriented**:
  (main) types are *abstract*,
  typically declared as *sorts* and implicitly specified by declared values and axioms.
  Example: type Database
- **model-oriented**:
  (main) types are *concrete* types that are constructed explicitly, typically from basic types and type constructors in *abbreviation* definitions.
  Example: type Database = Person**-set**

Pragmatics:

- **algebraic/property-oriented**:
  normally used in early development phases
- **model-oriented**:
  normally used in later development phases

In this lecture we will give examples of model-oriented specifications.

Example: Bank System (Exercise 9.3)
Requirements

1. Each customer may have 0, 1 or more accounts.
2. Two distinct customers can't share an account.
3. An account has a balance.
4. Each customer has an overdraft.
5. Accounts must not be overdrawn (more than overdraft).

Question 1

Define appropriate types!

Bank System Basic Types

type
CuNo,
AcNo,
Balance = **Int**,
Overdraft = **Nat**

Bank Model 1

type
  Bank′1 =
    (CuNo $\xrightarrow[m]{}$ Overdraft) ×
    (CuNo $\xrightarrow[m]{}$ (AcNo $\xrightarrow[m]{}$ Balance)),

  Bank1 = {| b : Bank′1 • is_wff(b) |}

value
  is_wff : Bank′1 → **Bool**
  is_wff(om, km) ≡
    dom km ⊆ **dom** om ∧
    (∀ cu1, cu2 : CuNo •
      {cu1, cu2} ⊆ **dom** km ∧ cu1 ≠ cu2 ⇒
        dom km(cu1) ∩ **dom** km(cu2) = {}
)
    ∧
    (∀ cu : CuNo • cu ∈ **dom** km ⇒
      (∀ ac : AcNo • ac ∈ **dom** km(cu) ⇒
        km(cu)(ac) + om(cu) ≥ 0)
)

Bank Model 2

type
  Bank′2 = CuNo $\xrightarrow[m]{}$ (Overdraft × (AcNo $\xrightarrow[m]{}$ Balance)),
  Bank2 = {| b : Bank′2 • is_wff(b) |}

value
  is_wff : Bank′2 → **Bool**
  is_wff(b) ≡
    (∀ cu1, cu2 : CuNo •
      {cu1, cu2} ⊆ **dom** b ∧ cu1 ≠ cu2 ⇒
      let (o1, m1) = b(cu1), (o2, m2) = b(cu2) **in**
        dom m1 ∩ **dom** m2 = {}
      end
)
    ∧
    (∀ cu : CuNo • cu ∈ **dom** b ⇒
      let (o, m) = b(cu) **in**
        ∀ ac : AcNo • ac ∈ **dom** m ⇒ m(ac) + o ≥ 0
      end
)

Bank Model 3

type
  Bank′3 =
    (CuNo $\xrightarrow[m]{}$ Overdraft) ×
    (AcNo $\xrightarrow[m]{}$ (CuNo × Balance)),

  Bank3 = {| b : Bank′3 • is_wff(b) |}

value
  is_wff : Bank′3 → **Bool**
  is_wff(om, m) ≡
    (
     ∀ cu : CuNo, i : Balance • (cu, i) ∈ **rng** m ⇒
       cu ∈ **dom** om ∧ i + om(cu) ≥ 0
)

A lesson

The more redundancy in the data types –

the more conditions are found in the is_wff functions

Question 2

Define the following functions:

new_customer,
new_account,
balance,
delete_customer,
delete_account

for bank model 3.

Banking Functions

value

new_customer : CuNo × Overdraft × Bank3 $\xrightarrow{\sim}$ Bank3
new_customer(cu, o, (om, m)) ≡
(om ∪ [cu ↦ o], m)
pre cu ∉ **dom** om,

new_account : CuNo × AcNo × Bank3 $\xrightarrow{\sim}$ Bank3
new_account(cu, ac, (om, m)) ≡
(om, m ∪ [ac ↦ (cu, 0)])
pre cu ∈ **dom** om ∧ ac ∉ **dom** m,

balance : AcNo × Bank3 $\xrightarrow{\sim}$ Balance
balance(ac, (om, m)) ≡
let (cu, i) = m(ac) **in** i **end**
pre ac ∈ **dom** m,

delete_customer : CuNo × Bank3 $\xrightarrow{\sim}$ Bank3
delete_customer(cu, (om, m)) ≡
(om \ {cu}, m)
pre cu ∈ **dom** om ∧ (∀ i : Balance • (cu, i) ∉ **rng** m),

delete_account : AcNo × Bank3 $\xrightarrow{\sim}$ Bank3
delete_account(ac, (om, m)) ≡
(om, m \ {ac})
pre ac ∈ **dom** m

Example: Bill of Materials

Exercise 9.4

- Revise the original specification: For each product in a bop, the system should keep track of the number of occurrences of subproducts.
- Define a function making parts explosion.

Original specification

scheme BILL_OF_PRODUCTS =
class
type
Product,
Bop = Product $\xrightarrow[m]{}$ Product-**set**
value
is_wf_Bop : Bop → **Bool**
is_wf_Bop(bop) ≡
(∀ ps : Product-**set** •
ps ∈ **rng** bop ⇒ ps ⊆ **dom** bop) ∧
(∀ p : Product •
p ∈ **dom** bop ⇒ p ∉ sub_products(p,bop)),

sub_products : Product × Bop $\xrightarrow{\sim}$ Product-**infset**
sub_products(p,bop) ≡
{p_sub | p_sub : Product • depends_on(p,p_sub,bop)}
pre p ∈ **dom** bop,

depends_on : Product × Product × Bop $\xrightarrow{\sim}$ **Bool**,
depends_on(p1,p2,bop) ≡
p2 ∈ bop(p1) ∨
(∃ p : Product •
(p ∈ bop(p1) ∧ depends_on(p,p2,bop)))
pre p1 ∈ **dom** bop

Original Specification — Continued

empty : Bop = [ ],

enter : Product × Product-**set** × Bop $\xrightarrow{\sim}$ Bop
enter(p,ps,bop) ≡ bop ∪ [p ↦ ps]
pre p ∉ **dom** bop ∧ ps ⊆ **dom** bop,

delete : Product × Bop $\xrightarrow{\sim}$ Bop
delete(p,bop) ≡ bop\{p}
pre p ∈ **dom** bop ∧
  ∼(∃ ps : Product-**set** • ps ∈ **rng** bop ∧ p ∈ ps),

add : Product × Product × Bop $\xrightarrow{\sim}$ Bop
add(p1,p2,bop) ≡ bop † [p1 ↦ bop(p1) ∪ {p2}]
pre
  {p1,p2} ⊆ **dom** bop ∧ p1 ≠ p2 ∧ p2 ∉ bop(p1) ∧
  p1 ∉ sub_products(p2,bop),

erase : Product × Product × Bop $\xrightarrow{\sim}$ Bop
erase(p1,p2,bop) ≡ bop † [p1 ↦ bop(p1)\{p2}]
pre p1 ∈ **dom** bop ∧ p2 ∈ bop(p1)
end

Revised specification

scheme BILL_OF_PRODUCTS =
  class
    type
      Product,
      Bop = {| bop : Bop′ • is_wf_Bop(bop) |},
      Bop′ = Product $\xrightarrow[m]{}$ Table,
      Table = Product $\xrightarrow[m]{}$ N1, N1 = {| n : **Nat** • n ≥ 1 |}

    value
      is_wf_Bop : Bop′ → **Bool**
      is_wf_Bop(bop′) ≡
        (∀ t : Table • t ∈ **rng** bop′ ⇒ **dom** t ⊆ **dom** bop′) ∧
        (∀ p : Product •
          p ∈ **dom** bop′ ⇒ p ∉ sub_products(p, bop′)),
      sub_products : Product × Bop′ $\xrightarrow{\sim}$ Product-**infset**
      sub_products(p, bop′) ≡
        { p_sub | p_sub : Product • depends_on(p, p_sub, bop′) }
        pre p ∈ **dom** bop′,
      depends_on : Product × Product × Bop′ $\xrightarrow{\sim}$ **Bool**
      depends_on(p1, p2, bop′) ≡
        p2 ∈ **dom** bop′(p1) ∨
        (∃ p : Product •
          (p ∈ **dom** bop′(p1) ∧ depends_on(p, p2, bop′)))
        pre p1 ∈ **dom** bop′,

Revised specification – continued

empty : Bop = [ ],

enter : Product × Table × Bop $\xrightarrow{\sim}$ Bop
enter(p, t, bop) ≡ bop ∪ [p ↦ t]
pre p ∉ **dom** bop ∧ **dom** t ⊆ **dom** bop,

delete : Product × Bop $\xrightarrow{\sim}$ Bop
delete(p, bop) ≡ bop \ {p}
pre p ∈ **dom** bop ∧
  ∼ (∃ t : Table • t ∈ **rng** bop ∧ p ∈ **dom** t),

add : Product × Product × N1 × Bop $\xrightarrow{\sim}$ Bop
add(p1, p2, n, bop) ≡ bop † [p1 ↦ bop(p1) ∪ [p2 ↦ n]]
pre
  {p1, p2} ⊆ **dom** bop ∧ p1 ≠ p2 ∧ p2 ∉ **dom** bop(p1) ∧
  p1 ∉ sub_products(p2, bop),

erase : Product × Product × Bop $\xrightarrow{\sim}$ Bop
erase(p1, p2, bop) ≡ bop † [p1 ↦ bop(p1) \ {p2}]
pre p1 ∈ **dom** bop ∧ p2 ∈ **dom** bop(p1)
end

Parts-Explosion: aux. functions

scheme
  TABLE_OPERATIONS =
    extend BILL_OF_PRODUCTS **with**
      class
        value
          + : Table × Table → Table
          t1 + t2 ≡
            [p ↦ count(p, t1) + count(p, t2) |
              p : Product • p ∈ **dom** t1 ∪ **dom** t2
            ],

          ∗ : N1 × Table → Table
          n1 ∗ t ≡
            [p ↦ n1 ∗ t(p) | p : Product • p ∈ **dom** t],

          count : Product × Table → **Nat**
          count(p, t) ≡
            if p ∈ **dom** t **then** t(p) **else** 0 **end**,
      end

Technical University of Denmark
Informatics and Mathematical Modelling
Computer Science and Technology

Parts-explosion

```
scheme PARTS_EXPLOSION =
  extend TABLE_OPERATIONS with class
      value
        parts : Product × Bop ~→ Table
        parts(p, bop) ≡ parts_of_product(p, bop) \ {p}
          pre p ∈ dom bop,
        parts_of_product : Product × Bop ~→ Table
        parts_of_product(p, bop) ≡
          let t = bop(p) in
            if t = [ ] then [ p ↦ 1 ]
              else parts_of_table(bop(p), bop)
          end end
          pre p ∈ dom bop,
        parts_of_table : Table × Bop ~→ Table
        parts_of_table(t, bop) ≡
          if t = [ ] then [ ]
          else let p : Product • p ∈ dom t in
                 t(p) ∗ parts_of_product(p, bop) +
                 parts_of_table(t \ {p}, bop)
          end end
          pre dom t ⊆ dom bop
  end
```