An RSL Tutorial

Part No.: RAISE/CRI/DOC/1/V1

Date: April 6, 1990

Original Author: Klaus Havelund

Note

This document is a pre-release produced for the VDM '90 RAISE tutorial.

Contents

1 Introduction

1.1 Purpose

The purpose of this document is to describe the RAISE Specification Language, RSL, in a pedagogical manner. The description is supposed to be suited for sequential reading.

1.2 Target Group

The target group of this document is users of RSL.

1.3 Relations to Other Documents

A more formal and complete description of RSL than given here can be found in [1].

1.4 Structure of Document

The document is divided into four parts corresponding to the following four facets of RSL:

- Applicative Specifications.
- State-based Specifications.
- Concurrency-based Specifications.
- Composing Systems from Modules.

Finally, an appendix contains a syntax summary.

1.5 Acknowledgements

I would like to thank Jan Storbank Pedersen for reading and commenting on the document during its production.

1.6 Shortcomings

Some facets of RSL are not described yet. The most important are the following

- Part four: ‘Composing Systems from Modules’.
- Overloading.

Part I

Applicative Specifications

2 Some Basic Concepts

This section introduces some basic concepts. This is done mainly through an example RSL specification of a database for registering voters at an election.

First some informal requirements are given for the election database. Then the formal specification follows, annotated with explanatory comments. The annotations will introduce the basic concepts as they occur in the specification.

2.1 Requirements for an Election Database

Consider the following requirements for an election database.

The database is supposed to support the administration of an election such that the database at any point in time holds all the persons having given their votes until that point of time.

The database must provide the following "functions"

1. *Register_Person:* Registers a person in the database when he or she has voted.
2. *Check_Person:* Checks whether a person has been registered in the database.
3. *Number_Registered:* Returns the number of persons currently registered in the database.

2.2 Formal Specification

Parts of the informal requirements specification (except for *Number_Registered*) can be modelled by the following RSL module.

Example 2.1

```
DATABASE =
  class
    type
      Person,
      Database = Person-set
    value
      empty : Database,
      register : Person × Database → Database,
      check : Person × Database → Bool
    axiom
      empty = { },
      ∀ p : Person, db : Database •
        register(p,db) = {p} ∪ db,
      ∀ p : Person, db : Database •
        check(p,db) = p ∈ db
  end
```

□

A module definition generally has the form

```
  id =
    class
      declaration_1
      ⋮
      declaration_n
    end
```

where a declaration begins with a keyword (**type**, **value**, **axiom**) indicating the kind of declaration to come and then follows one or more definitions of that kind, separated by commas.

The module definition contains three declarations,

1. A type declaration defining the types *Person* and *Database*.
2. A value declaration defining the values *empty*, *register* and *check*.
3. An axiom declaration expressing properties of the values.

2.2.1 Type Declarations

A *type* is a set of logically related values together with a number of operations for generating and manipulating these values. Types are well-known from most programming languages.

Types can be named in type declarations. A type declaration has the form

type
  type_definition$_1$,
  ⋮
  type_definition$_n$

In our specification there are two such definitions.

The first type definition which is of the form

  id

introduces the type *Person* as an *abstract type.* That is, a type with no predefined operations for generating and manipulating its values, except for '=' which compares two values of the type to check whether they are equal.

In general, each type is associated with an *equal* operator '=' as well as a *not equal* operator '≠'.

The fact that *Person* is defined as an abstract type reflects the requirements where no information is given about how persons are identified in terms of their name and the like. We simply abstract away from such details.

An abstract type is also referred to as a *sort* and a definition of such a type is referred to as a *sort definition.*

The next type definition which is of the form

  id = type_expr

is an *abbreviation definition* where the name *id* is specified to be an abbreviation for the *type expression* occurring on the right-hand side of =.

A database is specified to be a set of persons. The *type operator* **-set** when applied to the type *Person* yields a new type containing as values all (finite) subsets of *Person.*

A type obtained by applying a type operator to one or more other types is referred to as a *compound type*. Abstract types (like *Person*) are thus not compound.

We could have chosen another representation for the database, but modelling it as a set seems natural for our purpose. To see this, note that a set of elements is characterised by the property that one cannot detect in which order the elements have been inserted into the set. By observing the functions required in the informal requirements specification, one sees that none of these need knowledge about insertion order.

2.2.2 Value Declarations

Values can be named in value declarations. A value declaration has the form

value
  value_definition$_1$,
  ⋮
  value_definition$_n$

In our specification there are three such definitions.

A value definition is of the form

id : type_expr

That is, the identifier *id* is defined to represent a value within the type represented by the type expression.

The first value definition introduces the constant *empty* of the type *Database*. This value simply represents the empty database.

The actual value that the name *empty* represents is not described in the value definition, but instead in one of the axioms. Likewise for the other value names.

The second value definition defines the function *register* that adds a person to the database when that person has voted. Suppose the "current" database is *db* and that we want to register the person *hamid*, then

register(hamid,db)

represents the database after having made the registration.

The type of *register* is represented by the type expression

$$\text{Person} \times \text{Database} \to \text{Database}$$

This type expression is built up by applying two type operators, just like the type expression defining the *Database* using **-set**. To better illustrate how the type operators associate, it helps to note that the above type expression is equivalent to the following

$$(\text{Person} \times \text{Database}) \to \text{Database}$$

The type operator $\times$ (cartesian product) is thus applied to the pair *Person* and *Database*, and the type operator $\to$ (function space) is applied to the pair consisting of the resulting cartesian product and *Database*.

The cartesian product of *Person* and *Database* is the type containing as values all pairs (p, db) where $p \in Person$ and $db \in Database$.

The third value definition defines the function *check*, that when applied to a person and a database returns a boolean value **true** or **false** depending on whether the person is registered in the database or not.

The type **Bool** is a *built-in type.* That is, it is predefined within RSL.

It contains two values represented by the literals **true** and **false** and with it comes a number of operators which can be applied to its values. Examples of such operators are $\wedge$ (and) and $\vee$ (or), for example

$$\mathbf{true} \wedge \mathbf{false} = \mathbf{false}$$
$$\mathbf{true} \vee \mathbf{false} = \mathbf{true}$$

Just like abstract types, built-in types are not compound types, which are types obtained by applying type operators.

Until now we have only explained how values are introduced by giving their name and type. In the next paragraph, we shall see how the actual values that value names represent can be characterised by axioms.

To summarise, in the simple case which we consider here, a module provides zero or more named types together with zero or more named values.

2.2.3 Axiom Declarations

Axioms express properties of value names. In our example there are three axioms. The first axiom defines the name *empty* to represent the empty set (of persons). Remember that the type of *empty* is *Person*−**set**.

The axiom equates two *expressions*, namely *empty* and {}. An expression evaluates to a value. The expression *empty* evaluates to a set s_1 and the expression {} evaluates to a set s_2 (the empty set). The axiom then requires s_1 to be equal to s_2.

In fact the whole axiom

```
empty = {}
```

is itself an expression of type **Bool**. In general, all axioms are boolean expressions.

An axiom declaration thus has the form

axiom
  expr_1,
  ⋮
  expr_n

The second axiom expresses that the function *register* adds a person *p* to a database *db* by making the set union of the database, which is a set, and the singleton set containing the person.

The axiom is a quantified expression reading as follows: for all persons *p* and for all databases *db*, *register* applied to the pair (p, db) yields $\{p\} \cup db$.

The third axiom defines the function *check*. A person has voted if he or she belongs to the set representing the database.

2.2.4 Module Extension

The specification does not reflect all our requirements. We still need to specify a function for returning the number of persons registered in the database. We can do that by extending our first module with a value definition and an axiom

Example 2.2

```
ELECTION_DATABASE =
   **extend** DATABASE **with**
      **value**
         number : Database → **Nat**
      **axiom**
         ∀ db : Database •
            number(db) = **card** db
   **end**
```

□

The general form of an extending module is

```
id =
   **extend** $id_1$,...,$id_m$ **with**
      $declaration_1$
      ⋮
      $declaration_n$
   **end**
```

where each id_i is the name of some module. The module concept in its full power will be explained in part IV of this document.

The function *number*, when applied to a database returns a natural number, another built-in type, being the number of persons registered in the database.

The axiom defining *number* makes use of the cardinality (**card**) operator generally applicable on any finite set.

3 Built-in Types

3.1 Booleans

The boolean type literal

Bool

represents the type containing the two truth values

true

false

3.1.1 If Expressions

A boolean valued expression *b_expr* can be used to choose between the evaluation of two alternative expressions $expr_1$ and $expr_2$ in an if-expression of the form

if b_expr **then** expr_1 **else** expr_2 **end**

If *b_expr* evaluates to **true** the first expression, $expr_1$, is evaluated, otherwise the second expression, $expr_2$, is evaluated.

As an example consider the following expression returning the non-negative difference between two (non-negative) natural numbers

if $\text{x} > \text{y}$ **then** $\text{x} - \text{y}$ **else** $\text{y} - \text{x}$ **end**

The two expressions $expr_1$ and $expr_2$ must have the same type which is also the type of the if-expression.

3.1.2 Prefix and Infix Combinators

A boolean valued expression *b_expr* can be negated

$\sim$ b_expr

reading "not *b_expr*" and which is short for

if b_expr **then false else true end**

Two boolean valued expressions *b_expr*$_1$ and *b_expr*$_2$ can be combined with any of the binary operators "and", "or" and "implies" as shown below where the equivalent if-expressions are listed.

b_expr$_1$ $\wedge$ b_expr$_2$ $\equiv$ **if** b_expr$_1$ **then** b_expr$_2$ **else false end**

b_expr$_1$ $\vee$ b_expr$_2$ $\equiv$ **if** b_expr$_1$ **then true else** b_expr$_2$ **end**

b_expr$_1$ $\Rightarrow$ b_expr$_2$ $\equiv$ **if** b_expr$_1$ **then** b_expr$_2$ **else true end**

Note the use of '$\equiv$' (equivalence) instead of '$=$' (equality). The difference between the two concepts will be explained when variables and channels are introduced. That is, the two concepts are identical in applicative contexts.

As examples consider the following boolean valued expressions which are tautologies (evaluating to true)

$(x \leq 0) \vee (x > 0)$

$\sim ((x < 0) \wedge (x > 0))$

$(x > 0) \Rightarrow (x \geq 1)$

The explanation of the boolean combinators in terms of if-expressions requires that one considers evaluation order when using the infix combinators. Consider for example the following expression

$(x \neq 0) \wedge (1/x < \text{epsilon})$

and suppose that $x = 0$. The evaluation of the sub-expression $1/x$ will not yield a well-defined result. Fortunately, with the if-expression interpretation this sub-expression will never be evaluated.

This can be seen by the following reduction where the original expression together with equivalent expressions are listed

$(x \neq 0) \wedge (1/x < \text{epsilon})$
$\equiv$ **if** $(x \neq 0)$ **then** $(1/x < \text{epsilon})$ **else false end**
$\equiv$ **if** $(0 \neq 0)$ **then** $(1/0 < \text{epsilon})$ **else false end**
$\equiv$ **if false then** $(1/0 < \text{epsilon})$ **else false end**
$\equiv$ **false**

The logic obtained follows a so-called conditional logic where in general the second sub-expression is evaluated only if the value of the first sub-expression is not enough to determine the value of the composite expression. In this way any partiallity of the second sub-expression (here caused by 1/0) causes no evaluation error since evaluation is avoided.

3.1.3 Quantifiers

The following expression is an example of a quantified expression

$\forall\, x : \textbf{Nat} \bullet (x = 0) \vee (x > 0)$

and it reads: “for all natural numbers *x*, either *x* is equal to 0 or *x* is greather than 0”.

The quantifier ∀ binds the identifier *x* and we say that *x* is bound within the quantified expression expression. On the other hand, *x* is free within the expression

$(x = 0) \vee (x > 0)$

A quantified expression in general has the form

quantifier $\text{typing}_1,\ldots,\text{typing}_n$ • *boolean*_expr

where a quantifier is one of the following

∀, ∃, ∃!

reading “for all”, “there exists” and “there exists exactly one”, respectively

and where a typing in the simple case has the form

$\text{id}_1,\ldots,\text{id}_m$: type_expr

Some more examples of quantified expressions

```
∃ x : Nat •
   x > 99

∀ x,y : Nat •
   ∃! z : Nat •
      x + y = z

∃ x,y : Nat, b : Bool •
   b = (x = y)
```

3.1.4 Axiom Quantifications

Sometimes many of the axioms within an axiom declaration are quantified over a common set of value names as is the case in our election *DATABASE* module (section 2)

```
axiom
   empty = { },
   ∀ p : Person, db : Database •
      register(p,db) = {p} ∪ db,
   ∀ p : Person, db : Database •
      check(p,db) = p ∈ db
```

The axioms get somewhat "big" due to the repeated quantifications. One could instead make a global quantification as follows

```
axiom
   ∀ p : Person, db : Database •
      empty = { } ∧
      register(p,db) = {p} ∪ db ∧
      check(p,db) = p ∈ db
```

Note how the three axioms have been converted into one axiom by replacing commas with '∧'.

Unfortunately this solution makes the RAISE tools unparse (print) the single axiom "wrongly" by not making appropriate linebreaks. In addition, the solution does not work if axioms are named. Therefore the following form of axiom quantification has been introduced

```
axiom forall p : Person, db : Database •
```

empty = { },
register(p,db) = {p} ∪ db,
check(p,db) = p ∈ db

Now there are three axioms again, separated by commas. In general,

axiom forall typing_list •
opt_axiom_naming$_1$ expr$_1$,
⋮
opt_axiom_naming$_n$ expr$_n$

is short for

axiom
opt_axiom_naming$_1$ ∀ typing_list • expr$_1$,
⋮
opt_axiom_naming$_n$ ∀ typing_list • expr$_n$

3.2 Integers

The integer type literal

Int

represents the type containing the negative as well as non-negative whole numbers

...,-2,-1,0,1,2,...

3.2.1 Prefix Operators

There is one prefix operator for taking the "absolute value" of an integer

abs : **Int** → **Nat**

That is, if the argument is negative, the negated value is returned. The operator is the identity on non-negative numbers. The result is a natural number (section 3.3).

Some examples are

abs -5 = 5

abs 5 = 5

3.2.2 Infix Operators

A collection of binary infix operators are defined on integers.

There are the relational operators “greather than”, “less than”, “greather than or equal” and “less than or equal”

$$
\begin{aligned}
&> : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Bool}\\
&< : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Bool}\\
&\geq : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Bool}\\
&\leq : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Bool}
\end{aligned}
$$

Some examples are

$5 > 2$

$1 \leq 1$

There are the four arithmetic operators for “addition”, “subtraction”, “multiplication” and “division”

$$
\begin{aligned}
&+ : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}\\
&- : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}\\
&* : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}\\
&/ : \mathbf{Int} \times \mathbf{Int} \overset{\sim}{\to} \mathbf{Int}
\end{aligned}
$$

Note that the integer division operator returns an integer, the absolute value of which is the number of times that the absolute value of the second argument can be within the absolute value of the first. The sign of the result is the traditional product of the signs of the arguments.

The integer division operator is partial in that its result is undefined if the second argument is zero.

Some examples are

$2 * (5 + 7 - 2) = 20$

5 / 2 = 2

5 / -2 = -2

-5 / 2 = -2

-5 / -2 = 2

Associated with integer division is the "integer remainder" operator

$\backslash$: $\mathbf{Int} \times \mathbf{Int} \overset{\sim}{\rightarrow} \mathbf{Int}$

which returns an integer, the absolute value of which is the remainder after having divided the absolute value of the second argument into the absolute value of the first argument. The sign of the result is the sign of the first argument.

This implies the following relation between integer division and integer remainder. Let *a* and *b* be integers, then

a = (a/b)∗b + (a\b)

Some examples are

5 \ 2 = 1

5 \ -2 = 1

-5 \ 2 = -1

-5 \ -2 = -1

There is finally the "exponentation" operator

$\uparrow$: $\mathbf{Int} \times \mathbf{Int} \overset{\sim}{\rightarrow} \mathbf{Real}$

which raises the first integer to the power of the second integer. The result is a real number (section 3.4).

The exponentiation operator is partial in that its result is undefined if the first argument is zero while the second argument is negative.

An example is

$2 \uparrow 2 = 4.0$

3.3 Natural Numbers

The natural number type literal

Nat

represents the type containing the non-negative integers

0,1,2,...

3.3.1 Infix Operators

The natural number type is a subtype (section 9) of the integer type. Consequently, all the integer infix operators are defined for the natural numbers.

3.4 Real Numbers

The real number type literal

Real

represents the type containing the real numbers

...,-4.3,...,1.0,...,12.23,...

Note that all real number literals must be written with a decimal point.

3.4.1 Conversion Operators

The integer type is not considered a subtype (section 9) of the real number type, in contrast to the natural number type which is a subtype of the integer type. One set of operators is thus defined for the integers (section 3.2) and another set of operators is defined for the reals (see below). There is for example an “integer addition” operator and a “real addition” operator.

Since the two worlds are thus separated and since there will typically in calculations be a need to switch from one world to another, two operators “integer to real” and “real to integer” for doing that are defined

it : **Real** → **Int**
rl : **Int** → **Real**

The **it** operator returns the nearest integer towards zero.

Some examples are

it 4.6 = 4

it -4.6 = -4

rl 5 = 5.0

rl((**it** 5.2)/2) = 2.0

3.4.2 Other Prefix Operators

As for integers there is one prefix operator for taking the “absolute value” of a real number

abs : **Real** → **Real**

3.4.3 Infix Operators

A collection of binary infix operators are defined on real numbers corresponding to the similarily named infix integer operators.

> : **Real** × **Real** → **Bool**
< : **Real** × **Real** → **Bool**

$\geq$: **Real** $\times$ **Real** $\to$ **Bool**
$\leq$: **Real** $\times$ **Real** $\to$ **Bool**
$+$: **Real** $\times$ **Real** $\to$ **Real**
$-$: **Real** $\times$ **Real** $\to$ **Real**
$*$: **Real** $\times$ **Real** $\to$ **Real**
$/$: **Real** $\times$ **Real** $\overset{\sim}{\to}$ **Real**
$\uparrow$: **Real** $\times$ **Real** $\overset{\sim}{\to}$ **Real**

Note that the "real division" performs the traditional arithmetic division without truncating as does "integer division".

As for integers, the exponentiation operator is partial in that its result is undefined if the first argument is zero while the second argument is negative. In addition the result is undefined if the first argument is negative and the second argument is not a whole number.

3.5 Characters

The character type literal

Char

represents the type containing the characters

′A′,′B′,...,′a′,′b′,...

Note that a character begins and ends with single quotes.

3.6 Texts

The text type literal

Text

represents the type containing strings of characters. A text begins and ends with double quotes and has the general form

$''c_1 \ldots\ c_n''$

where for each c_i, $'c_i'$is a value of type **Char**.

Some examples are

```
″this is a text″

″Formal Methods″

″″
```

In section 7 more will be said about texts.

3.7 The Unit Value

The unit type literal

Unit

represents the type containing the single value

()

It might appear strange to have a type with only one value. It is, however, quite useful when dealing with imperative and concurrent specifications as will be illustrated later in this document.

4 Products

The cartesian product type expression

$\text{type\_expr}_1 \times \ldots \times \text{type\_expr}_n$

represents the type containing products of length n

$(\text{v}_1,\ldots,\text{v}_n)$

where each v_i is a value of type $\textit{type\_expr}_i$.

As an example consider the type expression

Bool × **Bool**

The type represented by that is finite and contains the following four products of length 2

(**true**,**true**) (**true**,**false**) (**false**,**true**) (**false**,**false**)

As another example, the type expression

Nat × **Nat** × **Bool**

represents an infinite type containing the following products

(0,0,**true**) (0,0,**false**)
(0,1,**true**) (0,1,**false**)
(1,0,**true**) (1,0,**false**)
(2,0,**true**) (2,0,**false**)
$\vdots$

4.1 Representing Products

An expression of the form

$(\mathrm{expr}_1,...,\mathrm{expr}_n)$

evaluates to a product

$(\mathrm{v}_1,...,\mathrm{v}_n)$

where v_i is the value of $expr_i$.

Some examples of expressions together with their types are

(**true**,p ⇒ q) : **Bool** × **Bool**

(x + 1,0,″`this is a text`″) : **Nat** × **Nat** × **Text**

4.2 Example

Example 4.1

A system of coordinates provides a set of positions

(x,y)

where x and y are real numbers. The center of a system of coordinates is $(0.0, 0.0)$ and is referred to as *origo*.

The distance between to positions is obtained by the well-known Pythagorean theorem.

```
SYSTEM_OF_COORDINATES =
  class
    type
      Position = Real × Real
    value
      origo : Position,
      distance : Position × Position → Real
    axiom
      origo = (0.0,0.0),
      ∀ x1,y1,x2,y2 : Real •
        distance((x1,y1),(x2,y2)) =
          ((x2−x1)↑2.0 + (y2−y1)↑2.0)↑0.5
  end
```

The type *Position* contains all possible positions being pairs of real numbers.

In the axiom for *distance* the following product expressions occur

```
(x1,y1)
(x2,y2)
((x1,y1),(x2,y2))
```

The function *distance* is thus applied to a pair of positions, each being a pair of coordinates.

□

5 Functions

5.1 Total Functions

A type expression of the form

$\text{type\_expr}_1 \to \text{type\_expr}_2$

represents a type containing all total functions from the type represented by *type_expr*$_1$ to the type represented by *type_expr*$_2$.

A total function

$\text{f} : \text{type\_expr}_1 \to \text{type\_expr}_2$

has the following property

```
∀ x : type_expr₁ •
   ∃ y : type_expr₂ •
      f(x) = y
```

We have already seen some examples of functions. In the election database (example 2.1 and example 2.2) we defined

```
value
   register : Person × Database → Database,
   check : Person × Database → Bool,
   number : Database → Nat
```

and in the system of coordinates (example 4.1) we defined

```
value
   distance : Position → Real
```

We have also seen how functions are applied

```
register(p,db)
check(p,db)
number(db)
distance((x1,y1),(x2,y2))
```

In general, a function is applied via an application expression of the form

$$\mathrm{expr}(\mathrm{expr}_1,...,\mathrm{expr}_n)$$

where *expr* represents a function of the type

$$\mathrm{T}_1 \times ... \times \mathrm{T}_n \to \mathrm{T}$$

and where each $expr_i$ is of type T_i. The result is thus of type T.

In the following is illustrated how functions are defined.

5.2 Definitions by Axioms

For the purpose of illustration we shall choose an example, the factorial function $n!$, which we shall specify in several ways in order to show different possibilities. The function has the signature

```
value
  fac : Nat → Nat
```

In mathematical notation

$$\mathrm{fac(n)} = \mathrm{n} * (\mathrm{n} - 1) * ... * 2 * 1$$

However, since the factorial operator '!' is not built into RSL we must specify *fac* ourselves. A first solution is

```
axiom
  ∀ n : Nat •
    fac(n) ≡ if n = 0 then 1 else n ∗ fac(n − 1) end
```

One could alternatively define the function through two axioms, one for the zero case and one for the non-zero case

```
axiom
   fac(0) ≡ 1,
   ∀ n : Nat • n > 0 ⇒
      fac(n) ≡ n ∗ fac(n − 1)
```

5.3 Explicit Definition of Total Functions

A shorter way of writing

```
value
   fac : Nat → Nat
axiom
   ∀ n : Nat •
      fac(n) ≡ if n = 0 then 1 else n ∗ fac(n − 1) end
```

is

```
value
   fac : Nat → Nat
   fac(n) ≡
      if n = 0 then 1 else n ∗ fac(n − 1) end
```

Thus the signature and the axiom have been merged into one definition called an explicit function definition. This saves writing the keyword **axiom** and the quantification over the formal parameter.

The merging of signature and axiom also makes the axiom more “local” to the signature.

The explicit function definition is an instance of the form

```
value
   id : type_expr$_1$ × ... × type_expr$_n$ → type_expr
   id(id$_1$,...,id$_n$) ≡ expr
```

which is short for

value
  id : type_expr$_1$ × ... × type_expr$_n$ → type_expr
axiom
  ∀ (id$_1$,...,id$_n$) : type_expr$_1$ × ... × type_expr$_n$ •
    id(id$_1$,...,id$_n$) ≡ expr

5.4 Partial Functions

A type expression of the form

type_expr$_1$ $\overset{\sim}{\rightarrow}$ type_expr$_2$

represents a type containing all partial functions from the type represented by *type_expr*$_1$ to the type represented by *type_expr*$_2$. That is, for some function

f : type_expr$_1$ $\overset{\sim}{\rightarrow}$ type_expr$_2$

there might exist a value v : *type_expr*$_1$ such that $f(v)$ is not well-defined.

Note that the type of partial functions contains all total functions as well.

As an example consider the following function

value
  fraction : **Real** $\overset{\sim}{\rightarrow}$ **Real**
axiom
  ∀ x : **Real** • x ≠ 0.0 ⇒
    fraction(x) ≡ 77.0/x

The axiom only defines the function for arguments different from zero. This is done by preceeding the the defining equivalence with a pre-condition.

5.5 Explicit Definition of Partial Functions

A shorter way of writing the above is

value
  fraction : **Real** $\overset{\sim}{\rightarrow}$ **Real**
  fraction(x) ≡ 77.0/x
    pre x ≠ 0

It is thus possible to write an equivalent explicit function definition which includes the pre-condition.

In the general case, however, this explicit definition is really short for

value
  fraction : **Real** $\overset{\sim}{\to}$ **Real**
axiom
  $\forall$ x : **Real** •
    fraction(x) $\equiv$ 77.0/x **pre** x $\neq$ 0

where the expression following the • is of the form

$\mathrm{expr}_1 \equiv \mathrm{expr}_2$ **pre** expr_3

This is the general form of an equivalence expression, and in the simple case it is equivalent to

$\mathrm{expr}_3 \Rightarrow (\mathrm{expr}_1 = \mathrm{expr}_2)$

This is, however, in general only true in case all of the expressions $expr_1$, $expr_2$ and $expr_3$ are applicative. When describing the non-applicative language constructs in parts two and three we will explain the general equivalence expression in more detail.

The above explicit definition of *fraction* is an instance of the form

value
  id : $\mathrm{type\_expr}_1 \times ... \times \mathrm{type\_expr}_n \overset{\sim}{\to} \mathrm{type\_expr}$
  $\mathrm{id}(\mathrm{id}_1,...,\mathrm{id}_n) \equiv \mathrm{expr}_1$ **pre** expr_2

5.6 Lambda Abstraction

The following axiom defines *fac* as the function represented by the lambda expression occurring on the right-hand side of '$\equiv$'

axiom
  fac $\equiv$
    λ n : **Nat** •
      if n = 0 **then** 1 **else** n $*$ fac(n $-$ 1) **end**

The lambda expression evaluates to a function. The general form of a lambda expression consists of a single typing (a binding and a type expression) and an expression

λ binding : type_expr • expr

representing a function of type

type_expr $\overset{\sim}{\rightarrow}$ T

where T is the type of *expr*.

Some other examples are

```
value
   incr : Int → Int,
   add : Int × Int → Int,
   cond : Bool × (Nat × Nat) → Nat
axiom
   incr ≡ λ x : Nat • x + 1,
   add ≡ λ (x,y) : Nat × Nat • x + y
   cond ≡
      λ (b,(x,y)) : Bool × (Nat × Nat) • if b then x else y end
```

It is possible to write the lambda expressions defining *add* and *cond* in a slightly different way, though the meaning is unchanged

```
axiom
   add ≡ λ (x : Nat, y : Nat) • x + y,
   cond ≡
      λ (b : Bool, x,y : Nat) • if b then x else y end
```

In general a lambda expression can in addition to the previous form also have the following form

λ (typing$_1$,...,typing$_n$) • expr

where $n \geq 0$. This second form can be rewritten into the first form as indicated by the above axioms.

The case where $n = 0$ in the last mentioned form represents expressions of the form

λ () • expr

representing a function of type

Unit $\overset{\sim}{\rightarrow}$ T

where T is the type of *expr*. Such an expression can of course instead be written as

λ dummy : **Unit** • expr

where *dummy* is then not referred to within *expr*. The '()' version, however, saves one from inventing a parameter name.

Functions with parameter type **Unit** are, however, primarily interesting when *expr* has side-effects as will be described in parts two and three of this document.

5.7 Higher Order Functions

Since function types are just like other types, a function can in particular take a function as parameter and return a function as result. Consider for example the definition

```
value
  twice : (Nat → Nat) → Nat → Nat
  twice(f) ≡
    λ n : Nat • f(f(n))
```

The function arrow '→' associates to the right, so the type of *twice* could instead have been written in the following way

twice : (**Nat** → **Nat**) → (**Nat** → **Nat**)

The function *twice* when applied to a function f yields a function (represented by the lambda expression) that when applied to a natural number n applies f twice.

Some examples of *twice* applications are twice(fac) = λ n : **Nat** • fac(fac(n))

twice(fac)(3) = 720

twice(λ n : **Nat** • n + 1)(1) = 3

Note that *twice* can be partially applied as in the first expression.

A function that returns a function upon application is called a curried function.

We don't need to use a lambda expression to define *twice*. We can also write an axiom like

```
axiom
  ∀ f : Nat → Nat, n : Nat •
    twice(f)(n) ≡ f(f(n))
```

As a a third possibility we could use the built-in operator '∘' for function composition which for arbitrary types T_1, T_2 and T_3 has the type

$$\circ : (\mathrm{T}_2 \overset{\sim}{\rightarrow} \mathrm{T}_3) \times (\mathrm{T}_1 \overset{\sim}{\rightarrow} \mathrm{T}_2) \rightarrow \mathrm{T}_1 \overset{\sim}{\rightarrow} \mathrm{T}_3$$

and which is defined as follows

$$(\mathrm{expr}_1 \circ \mathrm{expr}_2)(\mathrm{expr}) \equiv \mathrm{expr}_1(\mathrm{expr}_2(\mathrm{expr}))$$

Our axiom for *twice* would then be

```
axiom
  ∀ f : Nat → Nat •
    twice(f) ≡ f ∘ f
```

5.8 Explicit Definition of Curried Functions

A shorter way of writing

```
value
  twice : (Nat → Nat) → Nat → Nat
axiom
  ∀ f : Nat → Nat, n : Nat •
    twice(f)(n) ≡ f(f(n))
```

is

> **value**
> twice : (**Nat** → **Nat**) → **Nat** → **Nat**
> twice(f)(n) ≡ f(f(n))

This explicit function definition is an instance of the form

> **value**
> id : type_expr$_1$ → ... → type_expr$_n$ → type_expr
> id(id$_1$)...(id$_n$) ≡ expr

5.9 Currying and Un-currying

The function *twice* above is curried. We can "un-curry" it by redefining it as

> **value**
> twice : (**Nat** → **Nat**) × **Nat** → **Nat**
> twice(f,n) ≡ f(f(n))

We have turned an arrow '→' into a cartesian product '×'. Previous applications of the form

> twice(f)(x)

now have to be written

> twice(f,x)

However, previous partial applications of the form

> twice(f)

must now be written

> λ n : **Nat** • twice(f,n)

which is longer. This is one reason for choosing the curried version.

5.10 Predicative Definition of Functions

The function definitions given so far have all been algorithmic in the sense that they suggest a strategy for constructing an answer.

Function definitions can also be more predicative in the sense of just saying what properties the result must have based on the arguments.

Consider the following specification of the square-root function.

Example 5.1

```
SQUARE_ROOT =
  class
    value
      square_root : Real $\xrightarrow{\sim}$ Real
    axiom
      $\forall$ x : Real $\bullet$ x $\geq$ 0.0 $\Rightarrow$
        $\exists$ s : Real $\bullet$
          square_root(x) = s
            $\wedge$
          s $*$ s = x
            $\wedge$
          s $\geq$ 0.0
  end
```

□

If the predicate did not include the *s*-property that $s \geq 0.0$ then the *square_root* function would be underspecified yielding either a possitive or negative result on application – one would not know.

5.11 Implicit Definition of Functions

A shorter way of writing the above is given below.

Example 5.2

```
SQUARE_ROOT =
  class
    value
      square_root : Real $\xrightarrow{\sim}$ Real
      square_root(x) as s
        post
          s ∗ s = x
            ∧
          s ≥ 0.0
        pre
          x ≥ 0.0
  end
```

□

This is an implicit function definition reading as follows.

The function *square_root* is only well-defined for non-negative real numbers as expressed by **Real** and the pre-condition following **pre**.

When applied to an x it returns a value, call it s, that satisfies the post-condition following **post**.

The implicit value definition is an instance of the form

```
  value
    id : type_expr$_1$ × ... × type_expr$_n$ $\xrightarrow{\sim}$ type_expr
    id(id$_1$,...,id$_n$) as id$_r$
      post expr$_1$
      pre expr$_2$
```

where $expr_2$ can refer to the arguments $id_1, \ldots, id_n$ and where $expr_1$ in addition can refer to id_r.

5.12 Algebraic Definition of Functions

Most of the function definitions we have seen until now are of the form

```
  id(id$_1$,...,id$_n$) ≡ expr
```

the key issue here being that between the brackets '(' and ')' is a list of identifiers. This corresponds to the traditional way of defining functions also known from many programming languages.

RSL, however, also allows for more algebraic function definitions. A well-known example from arithmetic of this specification style is the set of properties satisfied by the '+' operator (commutativity, associativity, etc.)

$$a + b = b + a$$

$$a + (b + c) = (a + b) + c$$

Using this style function definitions will generally have the form

$$\mathrm{id}(\mathrm{expr}_1,\ldots,\mathrm{expr}_n) \equiv \mathrm{expr}$$

where the expressions between '(' and ')' typically themselves contain calls of functions other than *id*. In this way functions are defined by relating them to each other.

Consider the specification of integer lists. A list is an ordered sequence of elements. One can construct a new list from another list by adding an element to it. The added element is referred to as the head of the new list while the old list contained in the new list is referred to as the tail.

Example 5.3

```
LIST =
  class
    type
      List
    value
      empty : List,
      add : Int × List → List,
      head : List ~→ Int,
      tail : List ~→ List
    axiom
      forall i : Int, l : List •
      [head_add]
        head(add(i,l)) ≡ i,
      [tail_add]
        tail(add(i,l)) ≡ l
  end
```

□

The *List* type is given as an abstract type since we will not explicitely say how lists are represented.

If the *empty* constant were not there, we would not be able to write any list expressions. The list of numbers from 1 to 3 is for example expressed as

add(1,add(2,add(3,empty)))

The *head* and *tail* functions are partial in that they are not defined for the empty list. This is reflected in the axioms where nothing is said about *head*(*empty*) and *tail*(*empty*).

The *head* axiom says that adding an element i to a list and then taking the head yields the element just added.

The *tail* axiom says that adding an element to a list l and then taking the tail yields the original list.

So we have

head(add(1,add(2,add(3,empty)))) ≡ 1

tail(add(1,add(2,add(3,empty)))) ≡ add(2,add(3,empty))

5.13 Examples

Example 5.4

Consider the specification of a database. The database associates unique keys with data. That is, one key is associated with at most one data element in the database. The database should provide the following functions

- *Insert* which associates a key with a data element in the database. If the key already is associated with a data element the new association overrides the old.
- *Remove* which removes an association between a key and a data element.
- *Defined* which checks whether a key is associated with a data element.

- *Lookup* which returns the data element associated with a particular key.

The specification of this can be given in terms of algebraic function definitions.

```
DATABASE =
 class
   type
     Database,
     Key, Data
   value
     empty : Database,
     insert : Key × Data × Database → Database,
     remove : Key × Database → Database,
     defined : Key × Database → Bool,
     lookup : Key × Database ~→ Data
   axiom
     forall k,k1 : Key, d : Data, db : Database •
     [remove_empty]
       remove(k,empty) ≡ empty,
     [remove_insert]
       remove(k,insert(k1,d,db)) ≡
         if k = k1 then remove(k,db) else insert(k1,d,remove(k,db)) end,
     [defined_empty]
       defined(k,empty) ≡ false,
     [defined_insert]
       defined(k,insert(k1,d,db)) ≡
         k = k1 ∨ defined(k,db),
     [lookup_insert]
       lookup(k,insert(k1,d,db)) ≡
         if k = k1 then d else lookup(k,db) end
 end
```

The *Database* type is given as an abstract type since we don't want to say anything about (bother with) how databases are represented. Likewise, nothing is said about keys and data.

The *lookup* function is partial since it is un-defined when applied to a key and a database not associating that key with a data element. This is also reflected in that there is only one axiom for *lookup*, namely *lookup_insert*.

The *remove_insert* axiom is the most elaborate of the axioms, so here follows a short explanation. The right-hand side is an if-expression with two arms

- If the key k to be removed equals the inserted key $k1$, then the association of k with d is ignored (removed) and the *remove* function is applied recursively to the rest. This recursive call may seem strange since one could argue that a key is at most associated with one data element and therefore only needs to be removed once. A simpler axiom would thus be

```
remove(k,insert(k1,d,db)) ≡
   **if** k = k1 **then** db **else** ... **end**
```

This is, however, wrong and the reason is the following. We have quantified *db* over *Database* and therefore *db* can be any database, especially one associating *k* with some data element.

- If the key *k* to be removed does not equal the inserted key *k*1, then *k* must be removed from the remaining database. The succeeding association of *k*1 with *d* is necessary to keep that association.

The database example illustrates a useful technique for "inventing" axioms. The technique can be characterised as follows

1. Identify the constructors by which any database can be constructed. These are the constant *empty* and the function *insert*. Any database can thus be represented by an expression of the form

   $\mathrm{insert}(\mathrm{k}_1,\mathrm{d}_1,\mathrm{insert}(\mathrm{k}_2,\mathrm{d}_2,...\mathrm{insert}(\mathrm{k}_n,\mathrm{d}_n,\mathrm{empty})...\ ))$

2. Define the remaining functions "by case" over the constructors called with identifiers as parameters. In the above axioms, *remove*, *defined* and *lookup* are thus defined over the two constructor-expressions

   ```
   empty

   insert(k1,d,db)
   ```

   We thus get "for free" all the left-hand sides of the axioms we must write. That is

   ```
   remove(k,empty)
   remove(k,insert(k1,d,db))

   defined(k,empty)
   defined(k,insert(k1,d,db))

   lookup(k,empty)
   lookup(k,insert(k1,d,db))
   ```

   Note, however, that due the the partiality of *lookup* we don't bother with giving the right-hand side corresponding to *lookup*(*k*, *empty*).

The list-axioms (example 5.3) actually has the same form.

The technique is useful in many applications, but there are of course applications where one must be more imaginative when writing axioms.

□

Example 5.5

Consider the specification of natural numbers. This specification is not really needed from a pragmatic viewpoint since RSL provides the built-in type **Nat**. The example is thus given for illustration purposes. Functions will be defined algebraically.

```
PEANO =
  class
    type
      N
    value
      zero : N,
      succ : N → N,
    axiom
      forall n,n1,n2 : N •
      [first_is_zero]
        ∼ (succ(n) ≡ zero),
      [linear_order]
        succ(n1) ≡ succ(n2) ⇒ n1 ≡ n2,
      [induction]
        ∀ p : N → Bool •
          (p(zero) ∧ ∀ n : N • (p(n) ⇒ p(succ(n)))) ⇒
            ∀ n : N • p(n)
  end
```

The axioms are Peano's axioms for natural numbers. There is a zero value and a successor function (adding one to its argument). The *first_is_zero* axiom says that *zero* is not the successor of any number. The *linear_order* axiom says that for any natural number there is at most one predecessor, the successor of which is the natural number.

The *induction* axiom makes it possible to make proofs about natural numbers based on mathematical induction.

The axiom says: "for any predicate p, if $p(zero)$ holds and if $p(n)$ implies $p(succ(n))$ then p holds for all n".

What may be difficult to see is that the *induction* axiom implies that *N* only contains numbers that can be represented by RSL expressions of finite size. That is, for any number *n* in *N*, *n* is represented by the expression

```
  succ(succ(...(succ(zero))..))
```

with *n* applications of *succ*.

Note that a similar induction property should have been stated in example 5.3 and example 5.4. In section 10 we shall see a shorthand for such induction axioms.

We could now extend our *PEANO* module with functions for performing addition and multiplication

```
NATURAL_NUMBERS =
  extend PEANO with
    value
      plus : N × N → N,
      mult : N × N → N
    axiom
      forall n,n1,n2 : N •
      [plus_zero]
        plus(n,zero) ≡ n,
      [plus_succ]
        plus(n1,succ(n2)) ≡ succ(plus(n1,n2)),
      [mult_zero]
        mult(n,zero) ≡ zero,
      [mult_succ]
        mult(n1,succ(n2)) ≡ plus(mult(n1,n2),n1)
  end
```

It may be a mystery that exactly these axioms define *plus* and *mult*, but it should be easy to see that they are true – there is a difference.

□

6 Sets

A set is an unordered collection of distinct values of the same type. Examples of sets are

{1,3,5}

{″`John`″,″`Peter`″,″`Ann`″}

The first set is an integer set and the second set is a text set.

A type expression of the form

type_expr**-set**

represents a type of finite sets. Each set is a subset of the type represented by *type_expr*.

Consider for example the type expression

Bool-set

which represents the type containing the four sets

{}
{**true**}
{**false**}
{**true**,**false**}

Note that the empty set {} is included.

The type expression

Nat-set

represents the infinite type containing all finite subsets of the natural numbers

{}
{0} {1} {0,1}
{2} {0,2} {1,2} {1,2,3}
⋮

A type expression of the form

> type_expr**-infset**

represents the type of infinite as well as finite sets. Each set is a subset of the type represented by *type_expr*.

The type expression

> **Bool-infset**

represents the same type as the finite set type above since there are no infinite subsets of a finite type like **Bool**.

The type

> **Nat-infset**

however, contains infinite sets in addition to the finite ones

```
{}
{0} {1} {0,1}
{2} {0,2} {1,2} {1,2,3}
⋮
{0,1,2,3,4,...}
{1,2,3,5,7,...}
```

The dots '...' indicate the infinity (note that this is not RSL).

An example of an infinite set is **Nat** itself as indicated by the first infinite set above. Another example of an infinite set is the set of all prime numbers as indicated by the second infinite set above.

In general, for any type T, $T-$**set** is a subtype of $T-$**infset**. So all the sets belonging to **Nat-set** belong to **Nat-infset** as well.

6.1 Representing Sets

A set may be written by explicitly enumerating its members. We have already seen examples of such expressions

{1,2,3}

{"`John`","`Peter`","`Ann`"}

The general form of an enumerated set expression is

{expr_1,...,expr_n}

Each expression is evaluated to a value which is included in the resulting set. The order of the expressions does not matter. As an example consider the two set expressions which represent the same set

{1,2,3} = {3,2,1}

A set contains distinct values, so the following set expressions represent the same sets

{1,2,3} = {1,2,3,3}

A special set is that with no members

{}

A set can be defined implicitly by giving a predicate which defines the members. An example of such a so-called comprehended set expression is

{2∗n | n : **Nat** • n ≤ 3} = {0,2,4,6}

The comprehended set expression reads: "the set of values $2 * n$ where n is a natural number such that n is less than or equal to 3".

Other examples are

{n | n : **Nat** • is_a_prime(n)} = {1,2,3,5,7,...}

{(x,y) | x,y : **Nat** • y = x + 1} = {(0,1),(1,2),(2,3),...}

The first set contains all the prime numbers. The function *is_a_prime* must have the signature

value
  is_a_prime : **Nat** → **Bool**

The second set contains pairs (x, y) where y is x plus one.

The general form of a comprehended set expression is

{expr$_1$ | typing$_1$,...,typing$_n$ • expr$_2$}

where *expr*$_2$ must be a boolean expression.

A ranged set expression gives a set of integers in a range delimited by a lower bound and an upper bound

{3 .. 7} = {3,4,5,6,7}

{3 .. 3} = {3}

{3 .. 2} = {}

The general form of a ranged set expression is

{expr$_1$ .. expr$_2$}

where *expr*$_1$ and *expr*$_2$ are integer valued expressions. The expression represents the set of integers between and including the two bounds.

6.2 Infix Operators

A basic operator on sets is the "test for membership" and its negated version. Let T be an arbitrary type, then the signatures of these two operators are

∈ : T × T-**infset** → **Bool**
∉ : T × T-**infset** → **Bool**

An expression

e ∈ s

is **true** if and only if e is a member of the set s. For the negated version we have

$$\mathrm{e} \notin \mathrm{s} \equiv \sim(\mathrm{e} \in \mathrm{s})$$

Some examples are

$$3 \in \{1,3\} \equiv \textbf{true}$$

$$2 \notin \{1,3\} \equiv \textbf{true}$$

$$2 \in \{1,3\} \equiv \textbf{false}$$

A new set can be composed from two other sets by taking their "union" or their "intersection"

$$\cup : \mathrm{T}\textbf{-infset} \times \mathrm{T}\textbf{-infset} \to \mathrm{T}\textbf{-infset}$$
$$\cap : \mathrm{T}\textbf{-infset} \times \mathrm{T}\textbf{-infset} \to \mathrm{T}\textbf{-infset}$$

These operators can be defined in terms of test for membership

$$\mathrm{s}_1 \cup \mathrm{s}_2 \equiv \{\mathrm{e} \mid \mathrm{e} : \mathrm{T} \bullet \mathrm{e} \in \mathrm{s}_1 \vee \mathrm{e} \in \mathrm{s}_2\}$$

$$\mathrm{s}_1 \cap \mathrm{s}_2 \equiv \{\mathrm{e} \mid \mathrm{e} : \mathrm{T} \bullet \mathrm{e} \in \mathrm{s}_1 \wedge \mathrm{e} \in \mathrm{s}_2\}$$

Some examples are

$$\{1,3,5\} \cup \{5,7\} \equiv \{1,3,5,7\}$$

$$\{1,3,5\} \cap \{5,7\} \equiv \{5\}$$

$$\{1,3,5\} \cap \{7,8\} \equiv \{\}$$

A new set can be obtained from two other sets by a "set difference"

$$\backslash : \mathrm{T}\textbf{-infset} \times \mathrm{T}\textbf{-infset} \to \mathrm{T}\textbf{-infset}$$

Its definition is

$s_1 \setminus s_2 \equiv \{e \mid e : T \bullet e \in s_1 \wedge e \notin s_2\}$

Some examples are

$\{1,3,5\} \setminus \{1\} \equiv \{3,5\}$

$\{1,3,5\} \setminus \{7\} \equiv \{1,3,5\}$

$\{1,3,5\} \setminus \{n \mid n : \mathbf{Nat} \bullet \text{is\_a\_prime}(n)\} \equiv \{\}$

There are two operators for comparing sets, namely "subset" and "proper subset"

$\subseteq : T\textbf{-infset} \times T\textbf{-infset} \to \mathbf{Bool}$
$\subset : T\textbf{-infset} \times T\textbf{-infset} \to \mathbf{Bool}$

Their definitions are

$s_1 \subseteq s_2 \equiv \forall\, e : T \bullet e \in s_1 \Rightarrow e \in s_2$

$s_1 \subset s_2 \equiv s_1 \subseteq s_2 \wedge s_1 \neq s_2$

Some examples are

$\{1,3,5\} \subseteq \{1,3,5\} \equiv \mathbf{true}$

$\{1,3\} \subset \{1,3,5\} \equiv \mathbf{true}$

$\{1,3,5\} \subset \{1,3,5\} \equiv \mathbf{false}$

$\{1,3\} \subseteq \{3,5\} \equiv \mathbf{false}$

For convenience there are reversed versions of the comparison operators

$\supset : T\textbf{-infset} \times T\textbf{-infset} \to \mathbf{Bool}$
$\supseteq : T\textbf{-infset} \times T\textbf{-infset} \to \mathbf{Bool}$

6.3 Prefix Operators

The cardinality operator yields the "size" of a finite set, that is: the number of elements contained in the set

card : T-**set** → **Nat**

Some examples are

card {1,4,67} ≡ 3
card {} ≡ 0

6.4 Examples

Example 6.1

Consider the specification of a resource manager. A number of resources are to be shared between a number of users. A resource manager controls the resources by maintaining a pool (a set) of free resources.

When a user wants a resource, the resource manager *obtains* an arbitrary one from the pool. When the user no longer needs the resource, the manager *releases* it by putting it back into the pool.

```
RESOURCE_MANAGER =
  class
    type
      Resource,
      Pool = Resource-set
    value
      initial : Pool,
      obtain : Pool ~→ Pool × Resource,
      release : Resource × Pool ~→ Pool
    axiom forall r : Resource, p : Pool •
      obtain(p) as (p1,r)
        post r ∈ p ∧ p1 = p\{r}
        pre p ≠ {},
      release(r,p) ≡
        p ∪ {r}
        pre r ∈ initial\p
  end
```

The *Resource* type is defined as an abstract type since we don't consider here what resources are and how they are identified.

A *Pool* is defined as a set of resources. The *initial* pool is un-specified (there is no axiom for *initial*).

The definition of *obtain* reads as follows. When applied to a pool p that is non-empty, a pair $(p1, r)$ is returned. The resource r must be a member of the old pool p. The new pool $p1$ is equal to the old p except for r which has been removed.

Note that it is un-specified which resource is obtained from a pool containing more than one resource. The function is, however, deterministic in that if applied twice to the same pool it will return the same resource.

The *release* function just returns a resource to the pool. The resource must, however, not already be free.

Different styles have been used for defining *obtain* and *release*. An implicit style has been used to define *obtain* since there is no "algorithmic" strategy for selecting a member from a set. We only say that the returned resource must belong to the argument pool.

An explicit style has been used for defining *release* since RSL provides the union operator ∪ which perfectly does the job.

□

Example 6.2

Consider a set version of the database from example 5.4.

```
SET_DATABASE =
  class
    type
      Record = Key × Data,
      Database = Record-set,
      Key, Data
    value
      is_wf_Database : Database → Bool,
      empty : Database,
      insert : Key × Data × Database → Database,
      remove : Key × Database → Database,
      defined : Key × Database → Bool,
      lookup : Key × Database ~→ Data
    axiom forall k : Key, d : Data, db : Database •
      is_wf_Database(db) ≡
        ∀ k : Key, d1,d2 : Data •
          ((k,d1) ∈ db ∧ (k,d2) ∈ db) ⇒ d1 = d2,
      empty ≡
        {},
      insert(k,d,db) ≡
        remove(k,db) ∪ {(k,d)},
      remove(k,db) ≡
        db \ {(k,d) | d : Data},
      defined(k,db) ≡
        ∃ d : Data • (k,d) ∈ db,
      lookup(k,db) as d
        post (k,d) ∈ db
        pre defined(k,db)
  end
```

A database is modelled as a set of records, where a record consists of a key and a data element.

Not all databases are "wellformed". Some databases we are not interested in, namely those holding more than one record with the same key. The function *is_wf_Database* defines when a database is wellformed.

The *empty* database is represented by the empty set.

In order to *insert* a record into the database, one must first remove any existing record with the same key. This is necessary in order to keep the database wellformed.

To *remove* a key corresponds to removing all records containing that key – note that there will be at most one such record.

A key is *defined* if the database contains a record containing that key.

Finally, to *lookup* a key corresponds to finding a data element such that a record containing the key and that data element is in the database.

The set database actually implements the database from example 5.4. We shall not go into a detailed definition of the implementation relation here, but just outline a strategy for showing implementation.

SET_DATABASE implements *DATABASE* because

1. *SET_DATABASE* defines all the types that *DATABASE* defines with the only change that some sorts (*Database*) have been replaced by concrete definitions (*Database* = *Record*−**set**).

2. *SET_DATABASE* defines all the constants and functions that *DATABASE* defines with the same signatures.
3. All the axioms of *DATABASE* are true in *SET_DATABASE*. As an example consider the *DATABASE* axiom *defined_empty* (ignoring quantification)

   defined(k,empty) ≡ **false**

   To prove that this axiom holds in *SET_DATABASE* one can "unfold" the calls of *defined* and *empty*. That is, we replace the calls with the definitions that these functions have in *SET_DATABASE*. We unfold the functions one by one starting with *empty*

   defined(k,{}) ≡ **false**

   Then we unfold *defined* putting brackets around the unfolded text

   (∃ d : Data • (k,d) ∈ {}) ≡ **false**

   which reduces to

   false ≡ **false**

   which reduces to

   true

   So the *DATABASE* axiom *defined_empty* is true in *SET_DATABASE*.

□

Example 6.3

Consider a specification of equivalence relations. A set consisting of disjoint sets of elements is said to define an equivalence relation. We call the member sets for equivalence classes. All the elements of an equivalence class are equivalent.

An essential function on equivalence relations is *make_equivalent* for making two elements equivalent. Basically this function will join the equivalence classes of the two elements.

Another essential function *are_equivalent* tests whether two elements are equivalent. That is, whether they belong to the same equivalence class.

```
EQUIVALENCE_RELATION =
  class
    type
      Element,
      Class = Element-set,
      Relation = Class-infset
    value
      is_wf_Relation : Relation → Bool,
      initial : Relation,
      make_equivalent : Element × Element × Relation → Relation,
      are_equivalent : Element × Element × Relation → Bool
    axiom forall e,e1,e2 : Element, r : Relation •
      is_wf_Relation(r) ≡
        {} ∉ r
          ∧
        ∀ e : Element •
          ∃ c : Class •
            c ∈ r ∧ e ∈ c
          ∧
        ∀ c1,c2 : Class •
          c1 ∈ r ∧ c2 ∈ r ∧ c1 ≠ c2 ⇒
            c1 ∩ c2 = {},
      initial ≡
        {{e} | e : Element},
      make_equivalent(e1,e2,r) ≡
        {c | c : Class • c ∈ r ∧ {e1,e2} ∩ c = {}}
          ∪
        {c1 ∪ c2 | c1,c2 : Class •
          c1 ∈ r ∧ c2 ∈ r ∧ e1 ∈ c1 ∧ e2 ∈ c2},
      are_equivalent(e1,e2,r) ≡
        ∃ c : Class • c ∈ r ∧ e1 ∈ c ∧ e2 ∈ c
  end
```

Note that an equivalence class *Class* is a finite set of elements. This reflects the intuition that elements can only be made equivalent by the function *make_equivalent* and one can only apply a function finitely many times (thinking of a user).

A *Relation* on the other hand may be an infinite set of equivalence classes. This may happen if the type *Element* itself is infinite.

A relation is wellformed *is_wf_Relation* if

1. it does not contain the empty equivalence class,
2. every element in *Element* is represented in some equivalence class,
3. any two different equivalence classes are disjoint.

The *initial* relation makes no elements equivalent. This corresponds to a class for each element.

Two elements are made equivalent *make_equivalent* by collapsing into one class the two classes to which the two elements belong. The right-hand side of the axiom defining *make_equivalent* is the union of two sets. The first set contains those classes that do not contain any of the two elements. Such classes thus remain unchanged. The second set performs the collapse (union) of those sets containing the respective elements. Note that they might already belong to the same class in which case the relation remains completely unchanged.

Two elements are equivalent *are_equivalent* if there exists a class to which both belong.

□

7 Lists

A list is an ordered sequence of values of the same type, possibly including duplicates. Examples of lists are

$\langle 1,3,3,1,5 \rangle$

$\langle$**true**,**false**,**true**$\rangle$

The first list is an integer list and the second is a boolean list.

A type expression of the form

type_expr*

represents a type of finite lists. Each list contains elements from the type represented by *type_expr*.

Consider for example the type expression

Bool*

This type contains infinitely many finite lists of booleans

$\langle\rangle$
$\langle$**true**$\rangle$
$\langle$**false**$\rangle$
$\langle$**true**,**false**$\rangle$
$\langle$**false**,**true**$\rangle$
$\langle$**true**,**true**$\rangle$
$\langle$**false**,**false**$\rangle$
$\langle$**true**,**false**,**true**$\rangle$
$\vdots$

Note that the empty list $\langle\rangle$ is included. The reader should compare the above boolean lists with the boolean sets contained in **Bool-set** (section 6).

A type expression of the form

$\text{type\_expr}^{\omega}$

represents the type of infinite as well as finite lists. The type

$\mathbf{Bool}^{\omega}$

thus contains infinite boolean lists in addition to the finite ones

$\langle\rangle$
$\langle\mathbf{true}\rangle$
$\langle\mathbf{false}\rangle$
$\langle\mathbf{true},\mathbf{false}\rangle$
$\langle\mathbf{false},\mathbf{true}\rangle$
$\langle\mathbf{true},\mathbf{true}\rangle$
$\langle\mathbf{false},\mathbf{false}\rangle$
$\langle\mathbf{true},\mathbf{false},\mathbf{true}\rangle$
$\vdots$
$\langle\mathbf{false},\mathbf{true},\mathbf{true},\mathbf{true},\mathbf{false},...\ \rangle$

An example of an infinite list is the one containing all the prime numbers in increasing order.

In general, for any type T, T^* is a subtype of T^{ω}. So, for example, all the lists belonging to $\mathbf{Bool}^*$ belong to $\mathbf{Bool}^{\omega}$ as well.

7.1 Representing Lists

A list may be written by explicitly enumerating its elements. We have already seen examples of such expressions

$\langle 1,3,3,1,5\rangle$

$\langle\mathbf{true},\mathbf{false},\mathbf{true}\rangle$

The general form of an enumerated list expression is

$\langle \text{expr}_1,...,\text{expr}_n\rangle$

Each expression is evaluated to a value which is included in the resulting list at the appropriate position. Note that the order of the expressions matters. As an example consider the two list expressions which represent different lists

$\langle 1,2,3 \rangle \neq \langle 3,2,1 \rangle$

A list may contain duplicates, so the following list expressions represent different lists

$\langle 1,2,3 \rangle \neq \langle 1,2,3,3 \rangle$

A special list is that with no members, the empty list

$\langle\rangle$

A ranged list expression represents a list of integers in a range delimited by a lower bound and an upper bound

$\langle 3 \mathrel{..} 7 \rangle = \langle 3,4,5,6,7 \rangle$

$\langle 3 \mathrel{..} 3 \rangle = \langle 3 \rangle$

$\langle 3 \mathrel{..} 2 \rangle = \langle\rangle$

The general form of a ranged list expression is

$\langle \mathrm{expr}_1 \mathrel{..} \mathrm{expr}_2 \rangle$

where $expr_1$ and $expr_2$ are integer valued expressions. The expression represents the list of increasingly ordered integers between and including the two bounds, $expr_1$ being the lower bound.

A new list can be generated from an old list by applying a function to each member of the old list. An example of such a so-called comprehended list expression is

$\langle 2{*}\mathrm{n} \mid \mathrm{n}\ \mathbf{in}\ \langle 0 \mathrel{..} 3 \rangle \rangle = \langle 0,2,4,6 \rangle$

The comprehended list expression reads: “the list of values $2 * n$ where n ranges over the list ⟨0..3⟩”. Note that the ordering of the old list is preserved in the new list.

It is possible via a predicate to limit the selection of elements from the old list. Consider for example the list consisting of all the prime numbers between 1 and 100, ordered increasingly

```
⟨n | n in ⟨1 .. 100⟩ • is_a_prime(n)⟩ = ⟨1,2,3,5,7,...,97⟩
```

This comprehended list expression reads as follows: “the list of values n where n ranges over the list ⟨1..100⟩, considering only the prime numbers”.

As a third example consider a database which is a list of records

```
type
   Record = Key × Data,
   Database = Record*
```

Suppose we want to extract a report from the database, only involving those records that are *interesting* as defined by some boolean-valued function on keys. For each interesting record, the report will contain an entry consisting of the key and a *transformation* of the corresponding data element. So the following functions are assumed

```
value
   is_interesting : Key → Bool,
   transformation : Data → Report_Data
```

The report can then be represented by the following comprehended list expression, assuming the existence of a database *db*

```
⟨(k,transformation(d)) | (k,d) in db • is_interesting(k)⟩
```

The general form of a comprehended list expression is

$$\langle \mathrm{expr}_1 \mid \mathrm{binding}\ \mathbf{in}\ \mathrm{expr}_2 \bullet \mathrm{expr}_3 \rangle$$

where $expr_2$ is a list expression and $expr_3$ is boolean expression. The *binding* must match the elements of the list represented by $expr_2$.

7.2 List Indexing

A particular element of a list may be extracted by indexing, where the index must be a natural number between one and the length of the list. As an example consider the list *l* defined by

> **value**
> l : $\mathbf{Nat}^*$
> **axiom**
> l = $\langle$10,20,30$\rangle$

Then indexing *l* with index 2 yields the second element in the list

> l(2) = 20

The general form of an indexing expression is

> $\mathrm{expr}_1(\mathrm{expr}_2)$

where $expr_1$ is a list expression and $expr_2$ is an integer expression evaluating to a value between one and the length of the list.

7.3 Defining Infinite Lists

An infinite list can be defined through a value definition and an axiom specifying it to be infinite.

Consider for example the list containing all natural numbers in increasing order

```
value
   all_natural_numbers : Nat^ω
axiom
   all_natural_numbers(1) = 0,
   ∀ idx : Nat •
      idx ≥ 2 ⇒
         all_natural_numbers(idx) = all_natural_numbers(idx − 1) + 1
```

From the infinite list of natural numbers we can define the list of all prime numbers by a comprehended list expression

> $\langle$n | n **in** all_natural_numbers • is_a_prime(n)$\rangle$ = $\langle$1,2,3,5,7,... $\rangle$

7.4 Infix Operators

The "concatenation" operator concatenates two lists

$\frown : \mathrm{T}^{*} \times \mathrm{T}^{\omega} \rightarrow \mathrm{T}^{\omega}$

It produces the list containing all the elements from the first argument followed by all the elements from second

$\langle \mathrm{e}_1,...,\mathrm{e}_n\rangle \frown \langle \mathrm{e}_{n+1},...\ \rangle = \langle \mathrm{e}_1,...,\mathrm{e}_n,\mathrm{e}_{n+1},...\ \rangle$

Some examples are

$\langle 1,2,3\rangle \frown \langle 4,5\rangle = \langle 1,2,3,4,5\rangle$

$\langle 1,2,3\rangle \frown \langle\rangle = \langle 1,2,3\rangle$

Note that the first argument to the concatenation operator must be a finite list (one cannot append anything to the end of an infinite list since it has no end).

The second argument can, however, very well be infinite as in

$\langle 0\rangle \frown \mathrm{all\_natural\_numbers} = \langle 0,0,1,2,3,4,5,...\ \rangle$

where *all_natural_numbers* is defined above.

7.5 Prefix Operators

Two basic operators on lists are "head" and "tail"

$\mathbf{hd} : \mathrm{T}^{\omega} \overset{\sim}{\rightarrow} \mathrm{T}$
$\mathbf{tl} : \mathrm{T}^{\omega} \rightarrow \mathrm{T}^{\omega}$

The head of a list is the first element in the list "from the left"

$\mathbf{hd}\ \langle \mathrm{e}_1,\mathrm{e}_2,...\ \rangle = \mathrm{e}_1$

The tail of a list is that list which remains after having removed the head element (if any)

tl $\langle$e$_1$,e$_2$,... $\rangle$ = $\langle$e$_2$,... $\rangle$

Some examples are

hd $\langle$1,2,3$\rangle$ = 1

tl $\langle$1,2,3$\rangle$ = $\langle$2,3$\rangle$

tl $\langle\rangle$ = $\langle\rangle$

hd all_natural_numbers = 0

tl all_natural_numbers = $\langle$1,2,3,4,... $\rangle$

Note that the head operator is only well-defined for non-empty list arguments.

The operators "last" and "front" are the reverse of the head and tail operators

last : $T^* \overset{\sim}{\rightarrow} T$
front : $T^* \rightarrow T^*$

The last of a list is the last element in the list "from the left"

last $\langle$e$_1$,...,e$_{n-1}$,e$_n\rangle$ = e$_n$

The front of a list is that list which remains after having removed the last element (if any)

front $\langle$e$_1$,...,e$_{n-1}$,e$_n\rangle$ = $\langle$e$_1$,...,e$_{n-1}\rangle$

Some examples are

last $\langle$1,2,3$\rangle$ = 3

front $\langle$1,2,3$\rangle$ = $\langle$1,2$\rangle$

The "length" operator yields the length of a finite list

len : T^* → **Nat**

Some examples are

len ⟨2,4,2⟩ = 3

len ⟨⟩ = 0

Finally there are two operators for extracting the "indices" and "elements" of a list

inds : T^ω → **Nat-infset**
elems : T^ω → T-**infset**

The indices operator is defined as follows. Let *fl* be a finite list and let *il* be an infinite list

inds fl = {1 .. **len** fl}
inds il = {idx | idx : **Nat** • idx ≥ 1}

The elements operator is defined as follows

elems l = {l(idx) | idx : **Nat** • idx ∈ **inds** l}

Some examples are

inds ⟨2,4,2⟩ = {1,2,3}
elems ⟨2,4,2⟩ = {2,4}

inds ⟨⟩ = {}
elems ⟨⟩ = {}

inds all_natural_numbers = {i | i : **Nat** • i ≥ 1}
elems all_natural_numbers = {n | n : **Nat**}

7.6 Examples

Example 7.1

Consider the specification of a queue. Elements can be put into the queue, one by one. Elements can leave the queue, "first in first out", thereby reducing the queue.

```
QUEUE =
  class
    type
      Element,
      Queue = Element*
    value
      empty : Queue,
      put : Element × Queue → Queue,
      get : Queue ~→ Queue × Element
    axiom forall e : Element, q : Queue •
      empty ≡
        ⟨⟩,
      put(e,q) ≡
        q ⁀ ⟨e⟩,
      get(q) ≡
        (tl q,hd q)
        pre q ≠ empty
  end
```

A *Queue* is conveniently modelled as a list. Note that a queue is characterised by having an ordering on its members, just like lists. Only finite lists will be considered since infinite queues make no sense.

The *empty* queue is represented by the empty list.

To *put* an element into the queue corresponds to adding the element to the end of the list.

To *get* an element from the queue corresponds to take the head of the list.

□

Example 7.2

Consider the specification of a sorting function that sorts an integer list to yield an increasingly ordered list. We will not design an algorithm, but rather specify it implicitly in terms of the two functions *is_permutation* and *is_sorted*

```
LIST_PROPERTIES =
  class
    value
      is_permutation : Int* × Int* → Bool,
      is_sorted : Int* → Bool
    axiom forall l,l1,l2 : Int* •
      is_permutation(l1,l2) ≡
        ∀ i : Int •
          card {idx | idx : Nat • idx ∈ inds l1 ∧ l1(idx) = i} =
          card {idx | idx : Nat • idx ∈ inds l2 ∧ l2(idx) = i},
      is_sorted(l) ≡
        ∀ idx1,idx2 : Nat •
          {idx1,idx2} ⊆ inds l ∧ idx1 < idx2 ⇒
            l(idx1) ≤ l(idx2)
  end
```

The function *is_permutation* takes two lists and determines whether they are permutations of each other: they have the same length, contain the same elements and each element occurs the same number of times. In the definition this is expressed as follows: "for every integer i, the number of indices in the one list which denote i must be equal the the number of indices in the other list which denote i".

The function *is_sorted* takes a list and determines whether it is increasingly ordered: for any two different indices, the element denoted by the smallest must be less than or equal to the element denoted by the biggest.

We can now extend the *LIST_PROPERTIES* module with the definition of a sorting function

```
SORTING =
  extend LIST_PROPERTIES with
    value
      sort : Int* → Int*
    axiom forall l : Int* •
      sort(l) as l1
        post is_permutation(l1,l) ∧ is_sorted(l1)
  end
```

The *sort* function takes a list and returns a new list which is a permutation of the old one and which is sorted.

□

Example 7.3

Consider a list version of the database from example 5.4. The database will now be a list of records, corresponding to the traditional notion of a "sequential file".

To illustrate how a specification can be implementation oriented, we shall in addition require the database to be sorted on keys. For that purpose we must assume a function *less_than* defined on pairs of keys.

The sortedness property can now be utilized when searching for a record with a particular key *k*: the search is terminated as soon as a key greather than or equal to *k* is found. If the key found is greather than *k*, the search has failed. This algorithm saves time (in average) in case the key is not defined in the database.

We will also make the function *lookup* total such that when applied to a key and a database not defining that key, an error data element will be returned. We thus introduce such an error value named *not_found*. The types *Key* and *Data* together with the function *less_than* and the constant *not_found* are now defined in a separate module. The decomposition into sub-modules reduces the size, and thereby the readability, of each module.

```
KEY_AND_DATA =
  class
    type
      Key, Data
    value
      not_found : Data,
      less_than : Key × Key → Bool
    axiom forall k,k1,k2,k3 : Key •
      [not_reflexive]
        ∼less_than(k,k),
      [transitive]
        less_than(k1,k2) ∧ less_than(k2,k3) ⇒ less_than(k1,k3),
      [anti_symmetric]
        less_than(k1,k2) ⇒ ∼less_than(k2,k1),
      [total_order]
        less_than(k1,k2) ∨ less_than(k2,k1)
  end
```

The error element *not_found* is un-specified – we don't care at this point.

The function *less_than* is supposed to define an ordering on keys. If the keys were integers, the ordering could be '<'. The function is specified through a number of axioms. The reader should check that these axioms actually hold for '<'.

Considering records, we will make an abstraction, "hiding" the fact that they are pairs of key and data. For that purpose we will define functions for generating new records *new_record*, and for decomposing records: *key_of* and *data_of* for extracting the key field and the data field of a record, respectively.

A new module which is an extension of *KEY_AND_DATA* defines just these functions

```
RECORD =
  extend KEY_AND_DATA with
    type
      Record = Key × Data
    value
      new_record : Key × Data → Record,
      key_of : Record → Key,
      data_of : Record → Data
    axiom forall k : Key, d : Data •
      new_record(k,d) ≡
        (k,d)
      key_of(k,d) ≡
        k,
      data_of(k,d) ≡
        d
  end
```

The definition of *new_record* may look a bit strange since it is the identify function, taking a pair and yielding a pair. We have, however, obtained that we don't need to bother anymore with how records are represented. From now on records are only created and decomposed by these three functions.

It is now time to define the database as a sorted list of records

```
LIST_DATABASE =
  extend RECORD with
    type
      Database = Record*
    value
      is_wf_Database : Database → Bool,
      empty : Database,
      insert : Key × Data × Database → Database,
      remove : Key × Database → Database,
      defined : Key × Database → Bool,
      lookup : Key × Database → Data
    axiom forall k : Key, d : Data, db : Database •
      is_wf_Database(db) ≡
        ∀ r1,r2 : Record, db_left,db_right : Database •
          db = db_left ⌢ ⟨r1,r2⟩ ⌢ db_right ⇒
            less_than(key_of(r1),key_of(r2)),
      empty ≡
        ⟨⟩,
      insert(k,d,db) as db1
        post
          elems db1 = (elems remove(k,db)) ∪ {new_record(k,d)}
            ∧
          is_wf_Database(db1),
      remove(k,db) ≡
        ⟨r | r **in** db • key_of(r) ≠ k⟩,
      defined(k,db) ≡
        **if** db = ⟨⟩ ∨ less_than(k,key_of(**hd** db)) **then**
          **false**
        **else**
          key_of(**hd** db) = k ∨ defined(k,**tl** db)
        **end**,
      lookup(k,db) ≡
        **if** db = ⟨⟩ ∨ less_than(k,key_of(**hd** db)) **then**
          not_found
        **else**
          **if** key_of(**hd** db) = k **then**
            data_of(**hd** db)
          **else**
            lookup(k,**tl** db)
          **end**
        **end**
**end**
```

A database is well-formed, *is_wf_Database*, if for any two successive records, the key of the "leftmost" record is less than the key of the "rightmost" record.

Note that this wellformedness condition also prevents duplicate keys, i.e. two records having the same key. This is actally a consequence of the *not_reflexive* axiom in the module *KEY_AND_DATA*.

The function *insert* is defined implicitly by saying that the result of an insertion must contain the correct set of records and that these in addition must be sorted, without duplicates (*is_wf_Database*).

Although we are trying to be implementation oriented, the implicit style has been used here, since our focus at this point will be to optimize especially the function *lookup*.

The funtion *remove* is defined by a list comprehension expression that removes all the records having the specified key (there is at most one).

The functions *defined* and *lookup* are defined by nearly the same algorithm. They search the database sequentially for a key until either the end is reached or a greater key is found or the key is found. Note that due to the conditional interpretation of '∨', the function *defined* will not call itself recursively if the key is found.

In the case of *lookup*, note how the error value *not_found* is returned in case of failure to find the specified key.

An interesting point to note here is that *LIST_DATABASE* formally implements *DATABASE* from example 5.4.

We have actually strived to obtain that relation, at the cost of some problems, however.

The first "problem" is that the function *defined* is really not needed any more. Instead of calling *defined* one can call *lookup* and see whether the result differs from *not_found*.

The other more severe problem is introduced by the constant *not_found* which is a value of *Data* just like any other value of *Data*. It is thus possible to insert it into the database by *insert*. This is not the intension and users of *LIST_DATABASE* should not do so.

We could have made the function *insert* partial with the pre-condition that the inserted data element should be different from *not_found*. This would, however, destroy the implementation relation: one cannot implement a total function with a partial function and still obtain implementation.

The above list database specification is rather implementation oriented. We could have chosen to give a more abstract specification, still in terms of lists, but without the "sorting". That is to say, one can also use lists for abstract high-level specifications.

□

8 Maps

A map is a table-like structure that maps values of one type into values of another type. Examples of maps are

$[3 \mapsto \textbf{true}, 5 \mapsto \textbf{false}]$

$[''\texttt{Ib}'' \mapsto 7, ''\texttt{John}'' \mapsto 2, ''\texttt{Mary}'' \mapsto 7]$

The first is a map from integers to booleans. The value 3 is mapped to **true** while the value 5 is mapped to **false**. The second is a map from texts to integers.

The values for which a map is defined is referred to as the domain of the map. The second map above thus has the domain

$\{''\texttt{Ib}'',''\texttt{John}'',''\texttt{Mary}''\}$

The range of a map is the set of values mapped to. The second map above thus has the range

$\{2,7\}$

Maps are very similar to functions in that a map can be applied to a domain value to yield the associated range value.

The pragmatic difference between functions and maps is primarily a question of updating. Once a function has been created, it will typically remain "unchanged". A map, on the other hand, will typically be subjected to dynamic updatings.

Essential operators on maps are therefore

1. Update a map with a new association between a domain value and a range value.
2. Delete an association from a map.
3. Check whether some value belongs to the domain of a map.

As a real-world example of a map, consider a file directory mapping file identifiers into files. Such a map is typically subjected to the following operations

1. List all names of existing files.

2. Add a file.
3. Change a file.
4. Delete a file.

A type expression of the form

$$\text{type\_expr}_1 \ \underset{m}{\rightarrow} \ \text{type\_expr}_2$$

represents a type of maps, each mapping *type_expr*$_1$ values into *type_expr*$_2$ values. A map can be partial in having a domain which is only a subset of the type represented by *type_expr*$_1$.

Consider for example the type expression

$$\textbf{Text} \ \underset{m}{\rightarrow} \ \textbf{Nat}$$

This type contains infinitely many maps

$$\begin{array}{l} [\,] \\ [''\texttt{3}'' \mapsto 3] \\ [''\texttt{Ib}'' \mapsto 7, ''\texttt{John}'' \mapsto 2, ''\texttt{Mary}'' \mapsto 7] \\ \vdots \end{array}$$

Note that the empty map [ ] is included.

Maps may be infinite in having an infinite domain. The above map type thus also contains infinite maps.

8.1 Representing Maps

A map may be written by explicitly enumerating its associations. We have already seen examples of such expressions

$$[3 \mapsto \textbf{true}, 5 \mapsto \textbf{false}]$$

$$[''\texttt{Ib}'' \mapsto 7, ''\texttt{John}'' \mapsto 2, ''\texttt{Mary}'' \mapsto 7]$$

The general form of an enumerated map expression is

$[\mathrm{expr}_1 \mapsto \mathrm{expr}_1{}',...,\mathrm{expr}_n \mapsto \mathrm{expr}_n{}']$

Each expression pair $(expr_i,expr_i{}')$ is evaluated to values v_i and $v_i{}'$ and the resulting map then maps v_i to $v_i{}'$.

Note that the order of the associations does not matter. As an example consider the two map expressions which represent the same map

$[3 \mapsto \textbf{true}, 5 \mapsto \textbf{false}] = [5 \mapsto \textbf{false}, 3 \mapsto \textbf{true}]$

A special map is that with no associations

[ ]

A map can be defined implicitly by giving a predicate which defines the associations. An example of such a so-called comprehended map expression is

$[\mathrm{n} \mapsto 2{*}\mathrm{n} \mid \mathrm{n} : \textbf{Nat} \bullet \mathrm{n} \leq 2] = [0 \mapsto 0, 1 \mapsto 2, 2 \mapsto 4]$

The comprehended map expression reads: "the map from n to $2*n$ where n is a natural number such that n is less than or equal to 2".

It is possible via a comprehended map expression to create an infinite map. Consider for example

$[\mathrm{n} \mapsto 2{*}\mathrm{n} \mid \mathrm{n} : \textbf{Nat} \bullet \mathrm{is\_a\_prime(n)}] =$
$\quad [1 \mapsto 2, 2 \mapsto 4, 3 \mapsto 6, 5 \mapsto 10, 7 \mapsto 14, ...\ ]$

Cases like this are the reason for having infinite maps.

It is also possible via a comprehended map expression to create a so-called non-deterministic map. Consider for example

$[\mathrm{x} \mapsto \mathrm{y} \mid \mathrm{x,y} : \textbf{Nat} \bullet \{\mathrm{x,y}\} \subseteq \{1,2\}] =$
$\quad [1 \mapsto 1, 1 \mapsto 2, 2 \mapsto 1, 2 \mapsto 2]$

Such maps should be avoided in specifications and in the following we shall ignore their existence. It is, however, possible to create them.

The general form of a comprehended map expression is

$[\mathrm{expr}_1 \mapsto \mathrm{expr}_2 \mid \mathrm{typing}_1,...,\mathrm{typing}_n \bullet \mathrm{expr}_3]$

where $expr_3$ is a boolean expression.

8.2 Looking Up a Value

A value can be looked up in a map if it belongs to the domain of the map. As an example consider the map m defined by

value
  m : **Text** $\overrightarrow{m}$ **Nat**
axiom
  m = ["Ib" ↦ 7, "John" ↦ 2, "Mary" ↦ 7]

Then looking up "John" in m yields the value 2

m("John") = 2

The general form of a map-lookup expression is

$\mathrm{expr}_1(\mathrm{expr}_2)$

where $expr_1$ is a map expression and $expr_2$ must yield a value within the domain of the map.

8.3 Prefix Operators

A basic operator on maps is the "domain" operator which for a particular map yields its domain

dom : $(\mathrm{T}_1 \overrightarrow{m} \mathrm{T}_2) \to \mathrm{T}_1$**-infset**

Some examples are

dom ["Ib" ↦ 7, "John" ↦ 2] = {"Ib", "John"}

dom [n ↦ 2∗n | n : **Nat** • is_a_prime(n)] = {n | n : **Nat** • is_a_prime(n)}

dom [ ] = {}

A related operator is the "range" operator which yields the range of a map

$\textbf{rng} : (T_1 \underset{m}{\rightarrow} T_2) \rightarrow T_2\textbf{-infset}$

It is defined as follows

$\textbf{rng}\ m = \{m(d) \mid d : T_1 \bullet d \in \textbf{dom}\ m\}$

Some examples are

$\textbf{rng}\ [''\texttt{Ib}'' \mapsto 7, ''\texttt{John}'' \mapsto 2] = \{7,2\}$

$\textbf{rng}\ [n \mapsto 2*n \mid n : \textbf{Nat} \bullet \text{is\_a\_prime}(n)] = \{2*n \mid n : \textbf{Nat} \bullet \text{is\_a\_prime}(n)\}$

$\textbf{rng}\ [\,] = \{\}$

8.4 Infix Operators

The "override" operator overrides one map with another

$\dagger : (T_1 \underset{m}{\rightarrow} T_2) \times (T_1 \underset{m}{\rightarrow} T_2) \rightarrow (T_1 \underset{m}{\rightarrow} T_2)$

Priority is given to the associations in the second argument in cases where the domain values match. Some examples are

$[3 \mapsto \textbf{true}, 5 \mapsto \textbf{false}] \dagger [5 \mapsto \textbf{true}] = [3 \mapsto \textbf{true}, 5 \mapsto \textbf{true}]$

$[3 \mapsto \textbf{true}] \dagger [5 \mapsto \textbf{false}] = [3 \mapsto \textbf{true}, 5 \mapsto \textbf{false}]$

$[3 \mapsto \textbf{true}] \dagger [\,] = [3 \mapsto \textbf{true}]$

The "union" of two maps combines the two maps just like the override operator. The two maps should, however, have disjoint domains

$\cup : (T_1 \underset{m}{\rightarrow} T_2) \times (T_1 \underset{m}{\rightarrow} T_2) \rightarrow (T_1 \underset{m}{\rightarrow} T_2)$

An example is

$[3 \mapsto \textbf{true}] \cup [5 \mapsto \textbf{false}] = [3 \mapsto \textbf{true}, 5 \mapsto \textbf{false}]$

The union operator is typically used when one wants to “signal” that the two arguments are known to have disjoint domains.

There are two operators for restricting the domain of a map, namely “restrict with” and “restrict to”

$\backslash : (T_1 \underset{m}{\rightarrow} T_2) \times T_1\textbf{-infset} \rightarrow (T_1 \underset{m}{\rightarrow} T_2)$
$/ : (T_1 \underset{m}{\rightarrow} T_2) \times T_1\textbf{-infset} \rightarrow (T_1 \underset{m}{\rightarrow} T_2)$

They are defined as follows

$m \backslash s = [d \mapsto m(d) \mid d : T_1 \bullet d \in \textbf{dom}\ m \wedge d \notin s]$
$m / s = [d \mapsto m(d) \mid d : T_1 \bullet d \in \textbf{dom}\ m \wedge d \in s]$

Some examples are

$[3 \mapsto \textbf{true}, 5 \mapsto \textbf{false}] \backslash \{3\} = [5 \mapsto \textbf{false}]$

$[3 \mapsto \textbf{true}, 5 \mapsto \textbf{false}] / \{3\} = [3 \mapsto \textbf{true}]$

The “map composition” operator makes it possible to compose two maps

$\circ : (T_2 \underset{m}{\rightarrow} T_3) \times (T_1 \underset{m}{\rightarrow} T_2) \rightarrow (T_1 \underset{m}{\rightarrow} T_3)$

It is defined as follows

$(\text{expr}_1 \circ \text{expr}_2)(\text{expr}) \equiv \text{expr}_1(\text{expr}_2(\text{expr}))$

Some examples are

$[3 \mapsto \textbf{true}] \circ [''\texttt{Ib}'' \mapsto 3] = [''\texttt{Ib}'' \mapsto \textbf{true}]$

$[3 \mapsto \textbf{true}, 5 \mapsto \textbf{false}] \circ [''\texttt{Ib}'' \mapsto 3, ''\texttt{John}'' \mapsto 7] = [''\texttt{Ib}'' \mapsto \textbf{true}]$

$[3 \mapsto \textbf{true}] \circ [''\texttt{Ib}'' \mapsto 5] = [\,]$

The second and third map compositions show what happens when the range of the second argument include values that are not in the domain of the first argument: associations for which no match exists are just removed.

8.5 Examples

Example 8.1

Consider a map version of the database from example 5.4. The map datatype is very well suited for modelling the database since the database operations correspond closely to map operators.

```
MAP_DATABASE =
  class
    type
      Database = Key $\overrightarrow{m}$ Data,
      Key, Data
    value
      empty : Database,
      insert : Key × Data × Database → Database,
      remove : Key × Database → Database,
      defined : Key × Database → Bool,
      lookup : Key × Database $\xrightarrow{\sim}$ Data
    axiom forall k : Key, d : Data, db : Database •
      empty ≡
        [ ],
      insert(k,d,db) ≡
        db † [k ↦ d],
      remove(k,db) ≡
        db \ {k},
      defined(k,db) ≡
        k ∈ dom db,
      lookup(k,db) ≡
        db(k)
        pre defined(k,db)
  end
```

The *Database* is a mapping from keys to data.

The *empty* database is the empty mapping.

To *insert* an association between a key and a data element corresponds to overriding the original database with the new association. Any old association between the key and some data element will be overridden.

To *remove* a key corresponds to removing it from the domain.

To check whether a key is *defined* corresponds to finding out whether it belongs to the domain.

To *lookup* a key corresponds to applying the map to the key.

□

Example 8.2

Consider a map version of the equivalence relation from example 6.3. It should be remembered that elements of some type *Element* are separated into partitions. All the elements in the same partition are said to be equivalent.

Partitions can be merged by a function *make_equivalent*. Another function, *are_equivalent*, makes it possible to test whether two elements belong to the same partition (are equivalent).

A relation will now be modelled as a map from elements to partition identifiers. All elements in the same partition are mapped to the same partition identifier.

```
EQUIVALENCE_RELATION =
  class
    type
      Element,
      Partition_Id,
      Relation = Element $\overrightarrow{m}$ Partition_Id
    value
      is_wf_Relation : Relation → Bool,
      initial : Relation,
      make_equivalent : Element × Element × Relation $\xrightarrow{\sim}$ Relation,
      are_equivalent : Element × Element × Relation $\xrightarrow{\sim}$ Bool
    axiom forall e1,e2 : Element, r : Relation •
      is_wf_Relation(r) ≡
        ∀ e : Element •
          e ∈ dom r,
      is_wf_Relation(initial),
      e1 ≠ e2 ⇒ initial(e1) ≠ initial(e2),
      make_equivalent(e1,e2,r) ≡
        r † [e ↦ r(e2) | e : Element • r(e) = r(e1)]
        pre {e1,e2} ⊆ dom r
      are_equivalent(e1,e2,r) ≡
        r(e1) = r(e2)
        pre {e1,e2} ⊆ dom r
  end
```

A relation is wellformed, *is_wf_Relation*, if it maps every element in *Element* into some partition identifier. Every element thus belongs to a partition.

The *initial* relation must be wellformed (the second axiom).

The *initial* relation must in addition map different elements to different partition identifiers (the third axiom). No elements are thus equivalent from the beginning.

To make two elements e_1 and e_2 equivalent, *make_equivalent*, all the elements e that belong to the same partition as e_1 are moved to the partition that e_2 belongs to.

Two elements are equivalent, *are_equivalent*, if they are mapped to the same partition identifier.

Note the pre-conditions to the functions *make_equivalent* and *are_equivalent*. We could have replaced these with

```
  pre is_wf_Relation(r)
```

In section 9 we shall see how a wellformedness predicate can be used to restrict a type via a so-called subtype expression. This will make it possible to avoid pre-conditions as the one above. Indeed the second axiom defining *initial* to be wellformed can also be avoided.

□

Example 8.3

Consider the specification of a bill of products. We are dealing with products, each of which is either basic or compound. A compound product is built from one or more immediate sub-products, each of which is either basic or again compound. A basic product is not built from (immediate) sub-products.

The sub-products of a product are all the immediate ones plus their sub-products. Each product thus defines a hierarchy with itself as the top-node.

Compound products cannot be recursively composed. That is, a product cannot have itself as a sub-product.

Our system must keep track of which products are basic and which are compound, and in the latter case what the sub-products are.

A function must thus be provided that for any product returns the set of its sub-products.

Functions must be provided for entering new products into the system and for deleting products from the system.

Finally, functions must be provided for adding and erasing sub-product relations between existing products.

```
BILL_OF_PRODUCTS =
  class
    type
      Product,
      Bop = Product →m Product-set
    value
      empty : Bop,
      is_wf_Bop : Bop → Bool,
      sub_products : Product × Bop →̃ Product-set,
      enter : Product × Product-set × Bop →̃ Bop,
      delete : Product × Bop →̃ Bop,
      add : Product × Product × Bop →̃ Bop,
      erase : Product × Product × Bop →̃ Bop
    axiom forall p,p1,p2 : Product, ps : Product-set, bop : Bop •
      is_wf_Bop(bop) ≡
        ∀ ps : Product-set • ps ∈ rng bop ⇒ ps ⊆ dom bop
          ∧
        ∀ p : Product • p ∈ dom bop ⇒ p ∉ sub_products(p,bop),
      empty ≡
        [ ],
      sub_products(p,bop) as ps
        post ps =
          {p1 | p1 : Product • p1 ∈ dom bop ∧
            (p1 ∈ bop(p)
              ∨
            ∃ p2 : Product • p2 ∈ ps ∧ p1 ∈ bop(p2))}
        pre p ∈ dom bop,
      enter(p,ps,bop) ≡
        bop ∪ [p ↦ ps]
        pre p ∉ dom bop ∧ ps ⊆ dom bop,
      delete(p,bop) ≡
        bop\{p}
        pre p ∈ dom bop ∧ ∼∃ ps : Product-set • ps ∈ rng bop ∧ p ∈ ps,
      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(p1,p2,bop) ≡
        bop † [p1 ↦ bop(p1)\{p2}]
        pre p1 ∈ dom bop ∧ p2 ∈ bop(p1)
  end
```

The *Product* type is abstractly given since we don't care about how to identify products.

A bill of products, *Bop*, is a map from products to sets of products. A compound product p is mapped to the set $\{p_1, \ldots, p_n\}$ if it consists of the immediate sub-products $p_1 \ldots p_n$. A basic product is mapped to the empty set.

A bill of products is wellformed, *is_wf_Bop*, if

1. every sub-product is in the domain of the map. That is, every product mentioned must occur in the domain,
2. no product is a sub-product of itself. The call of the function *sub_products*(p, bop) yields all the sub-products of p.

The *empty* bill of products is the empty mapping.

The sub-products, *sub_products*, of a product is the smallest set ps that satisfies the following

1. it must include a product $p1$ if $p1$ is an immediate sub-product,
2. it must include a product $p1$ if $p1$ is an immediate sub-product of a product $p2$ which is in ps.

The RSL formulation of this condition may appear a bit tricky because of the occurrence of ps on both sides of the equation.

The pre-condition of *sub_products* says that the product examined must be an existing one.

To *enter* a new product together with an identification of all its immediate sub-products corresponds to directly appending that association to the map. The pre-condition says that the product must not already exist but that all the immediate sub-products must.

To *delete* a product corresponds to removing it from the domain of the map. The pre-condition says that the product must be an existing one and that it must not be a sub-product of some other product.

To *add* an immediate sub-product to a product corresponds to adding it to the set mapped to by the product. The pre-condition says that

1. the product as well as the sub-product must exist,
2. they must be different,
3. the sub-product must not already be an immediate sub-product of the product,
4. the product must not be a sub-product of the sub-product. This prevents the violation of the wellformedness condition that a bill of products must be acyclic.

To *erase* an immediate sub-product from a product corresponds to removing it from the set mapped to by the product. Note that the sub-product is not deleted from the domain of the map. The pre-condition says that the product must be an existing one and that the sub-product really is an immediate sub-product.

□

9 Subtypes

A type can be constrained by a predicate, resulting in a subtype (subset) of the original type. Consider for example the type expression

$\{|\ \mathrm{t} : \mathbf{Text} \bullet \mathbf{len}\ \mathrm{t} > 0\ |\}$

The type **Text** is here constrained to the subtype containing "those *t* of type **Text** where the length of *t* is greater than zero" (remember that a text is a list of characters). We thus get a type containing non-empty texts.

Another example is

$\{|\ (\mathrm{x},\mathrm{y}) : \mathbf{Int} \times \mathbf{Int} \bullet \mathrm{x} < \mathrm{y}\ |\}$

That is, "those pairs (x, y) of type **Int** × **Int** where x is less than y".

The general form of a subtype expression is

$\{|\ \mathrm{binding} : \mathrm{type\_expr} \bullet \mathrm{expr}\ |\}$

where *expr* must be a boolean expression. The *binding* must match the values of the type represented by *type_expr*.

Generally, a type T_1 is a subtype of a type T_2 if there exists a predicate $p : T_2 \rightarrow$ **Bool** such that

$\mathrm{T}_1 = \{|\ \mathrm{x} : \mathrm{T}_2 \bullet \mathrm{p(x)}\ |\}$

We have a meta-notation for this, namely

$\mathrm{T}_1 \preceq \mathrm{T}_2$

As a special case, any type is a subtype of itself

$\mathrm{T} \preceq \mathrm{T}$

It should be no surprise that the subtype relation is transitive

$$\begin{array}{l} T_1 \preceq T_2 \wedge T_2 \preceq T_3 \Rightarrow \\ \quad T_1 \preceq T_3 \end{array}$$

9.1 Maximal Types

We shall now define the concept of a maximal type. This concept is central for the automated type-checker implemented as part of the RAISE-tools (see the next section). The concept of maximal type, however, also helps us to see which types are generally subtypes of other ones.

A type is maximal if it is not a subtype of any other type than itself.

The maximal type of a type T is the largest type of which T is a subtype. We shall use the meta-notation $[T]$ for the maximal type of T. Below is defined what the maximal types are for the different possible type expressions introduced until now.

All the built-in types except **Nat** and **Text** have themselves as maximal types

$[\textbf{Bool}] = \textbf{Bool}$

$[\textbf{Int}] = \textbf{Int}$

$[\textbf{Real}] = \textbf{Real}$

$[\textbf{Char}] = \textbf{Char}$

$[\textbf{Unit}] = \textbf{Unit}$

For natural numbers we have

$[\textbf{Nat}] = \textbf{Int}$

The natural number type contains those integers from the maximal type that are greater than or equal to zero. We can in fact write **Nat** as

$\{|\ n : \textbf{Int} \bullet n \geq 0\ |\}$

The maximal type of **Text** will be given in connection with list type expresssions below.

For products we have

$[\mathrm{A}_1 \times ... \times \mathrm{A}_n] = [\mathrm{A}_1] \times ... \times [\mathrm{A}_n]$

The product type contains those products of the maximal type that consist of components from A_1 to A_n only.

An example is

$[\mathbf{Nat} \times \mathbf{Nat}] = \mathbf{Int} \times \mathbf{Int}$

For finite sets we have

$[\mathrm{A}\textbf{-set}] = [\mathrm{A}]\textbf{-infset}$

The finite set type contains those sets from the maximal type that consist of A elements only and have finite size.

An example is

$[\textbf{Nat-set}] = \textbf{Int-infset}$

For infinite sets we have

$[\mathrm{A}\textbf{-infset}] = [\mathrm{A}]\textbf{-infset}$

The infinite set type contains those sets from the maximal type that consist of A elements only.

An example is

$[\textbf{Nat-infset}] = \textbf{Int-infset}$

For finite lists we have

$[\mathrm{A}^*] = [\mathrm{A}]^\omega$

The finite list type contains those lists from the maximal type that consist of A elements only and have finite size.

An example is

$[\mathbf{Nat}^*] = \mathbf{Int}^\omega$

For texts we have

$[\mathbf{Text}] = \mathbf{Char}^\omega$

Recall that *Text* is short for $\mathbf{Char}^*$. The text type contains those character lists from the maximal type that have finite size.

For infinite lists we have

$[\mathrm{A}^\omega] = [\mathrm{A}]^\omega$

The infinite list type contains those lists from the maximal type that consist of A elements only.

An example is

$[\mathbf{Nat}^\omega] = \mathbf{Int}^\omega$

For maps we have

$[\mathrm{A} \underset{m}{\overrightarrow{\ }} \mathrm{B}] = [\mathrm{A}] \underset{m}{\overrightarrow{\ }} [\mathrm{B}]$

The map type contains those maps from the maximal type that have a domain within A and a range within B.

An example is

$[\mathbf{Nat} \underset{m}{\overrightarrow{\ }} \mathbf{Nat}] = \mathbf{Int} \underset{m}{\overrightarrow{\ }} \mathbf{Int}$

For partial functions we have

$[\mathrm{A} \overset{\sim}{\rightarrow} \mathrm{B}] = [\mathrm{A}] \overset{\sim}{\rightarrow} [\mathrm{B}]$

The partial function type contains those functions from the maximal type that for any A value are either undefined or return a B value.

An example is

$[\mathbf{Nat} \overset{\sim}{\rightarrow} \mathbf{Nat}] = \mathbf{Int} \overset{\sim}{\rightarrow} \mathbf{Int}$

As another example, consider the definition

value
  f : $\mathbf{Nat} \overset{\sim}{\rightarrow} \mathbf{Nat}$

The function *f* will when applied to a natural number either be undefined for that number or yield a natural number.

When applied to a negative number, *f* will either be undefined or yield an integer. To see this one can observe that *f* really is among the functions in the maximal type

$\mathbf{Int} \overset{\sim}{\rightarrow} \mathbf{Int}$

which for natural numbers return natural numbers.

For total functions we have

$[\mathrm{A} \rightarrow \mathrm{B}] = [\mathrm{A}] \overset{\sim}{\rightarrow} [\mathrm{B}]$

The total function type contains those functions from the maximal type that for any *A* value are defined and return a *B* value.

An example is

$[\mathbf{Nat} \rightarrow \mathbf{Nat}] = \mathbf{Int} \overset{\sim}{\rightarrow} \mathbf{Int}$

As another example, consider the definition

value
  f : $\mathbf{Nat} \rightarrow \mathbf{Nat}$

The function *f* will when applied to a natural number be defined for that number and yield a natural number. When applied to a negative number, *f* will either be undefined or yield an integer.

9.2 Maximal Type Correctness

In this section we shall outline what it means for a RSL-specification to be maximal type correct. The automated type-checker implemented as part of the RAISE-tools checks for exactly maximal type correctness.

Loosely formulated, an expression is type correct with respect to the context in which it occurs, if the type of the expression matches the type required by the context. All sub-expressions must of course also be type correct.

To strengthen this definition we must specify what we mean by "type", what we mean by "context" and what we mean by "matches". Consider the following example

Example 9.1

```
EXAMPLE1 =
  class
    value
      is_a_prime : Nat → Bool,
      four : Int
    axiom
      four ≡ 4,
      is_a_prime(four) ≡ false
  end
```

□

In the second axiom, the expression *four* occurs in the context *is_a_prime*(*four*). The declared type of *four* is **Int** and the context requires the type **Nat** due to the declared type of *is_a_prime*.

The question is what it means for **Int** to match **Nat**. A reasonable answer could be to say that the type of the expression must be a subtype of the type required by the context. That is, **Int** must be a subtype of **Nat**. This is obviously not true, and the expression *four* is thus not type correct with respect to its context *is_a_prime*(*four*).

This result is problematic since we from the axiom can deduce that *four* actually belongs to **Nat**. From a pragmatic viewpoint we would thus like the specification to be type correct.

A second problem with the suggested solution is more severe: it is in general not possible to design an algorithm that decides whether one type is a subtype of another. This is due to subtype expressions that generally involve predicates of arbitrary complexity.

In order to avoid these two problems, only maximal types are considered when deciding type correctness.

An expression is maximal type correct with respect to the context in which it occurs, if the maximal type of the expression equals the maximal type required by the context.

In our example above the maximal type of *four* is **Int** and this type obviously equals the maximal type required by the context.

Consider another more complex example

Example 9.2

```
EXAMPLE2 =
  class
    type
      Positive = {| n : Nat • n > 0 |}
    value
      apply : (Positive → Int) × Positive-infset → Int-infset,
      square : Int → Nat,
      one_to_five : Positive-set,
      squares : Positive-set
    axiom forall f : Positive → Int, s : Positive-infset, i : Int •
      apply(f,s) ≡
        {f(e) | e : Positive • e ∈ s},
      square(i) ≡
        i ↑ 2,
      one_to_five ≡
        {1 .. 5},
      squares ≡
        apply(square,one_to_five)
  end
```

□

The interesting axiom is the last one defining *squares*. First of all the reader should convince himself or herself that it would be pragmaticly usefull if it is maximal type correct and has a proper meaning. Note especially that

1. the function *square* will be defined for all positive numbers and for these yield integers as required by the function *apply*.

2. the result of *apply*(*square*, *one_to_five*) will be a finite set of positive numbers.

It may, however, not be obvious that all the constituent sub-expressions of the axiom are maximal type correct. Below follows the same specification transformed such that all types are replaced by their corresponding maximal types. It should now be obvious that the axiom is maximal type correct.

```
type
  Positive = Int
value
  apply : (Int $\overset{\sim}{\to}$ Int) × Int-infset $\overset{\sim}{\to}$ Int-infset,
  square : Int $\overset{\sim}{\to}$ Int,
  one_to_five : Int-infset,
  squares : Int-infset
axiom forall f : Int $\overset{\sim}{\to}$ Int, s : Int-infset, i : Int •
  apply(f,s) ≡
    {f(e) | e : Int • e ∈ s},
  square(i) ≡
    i ↑ 2,
  one_to_five ≡
    {1 .. 5},
  squares ≡
    apply(square,one_to_five)
```

9.3 Proving Subtype Relations

Three ways of introducing subtypes have been introduced until now:

1. Through subtype expressions of the form

   {| binding : type_expr • expr |}

2. Through the built-in types of natural numbers and texts

   Nat, **Text**

3. Through the type operators for finite sets and lists and the type operator for total functions

   A-**set**, A^* and $A \to B$

That is to say, one can only define a subtype by using at least one of these constructs. The remaining type operators for products, infinite sets, infinite lists, maps and partial functions only generate subtypes if their arguments are already subtypes.

Let us see how we in general decide whether one type is a subtype of another. Let there be given two types T_1 and T_2 for which we want to prove that

$$\mathrm{T}_1 \preceq \mathrm{T}_2$$

First we recognize that the two types are subtypes of their maximal types

$$\mathrm{T}_1 = \{|\ \mathrm{x} : [\mathrm{T}_1] \bullet \mathrm{p}_1(\mathrm{x})\ |\}$$

$$\mathrm{T}_2 = \{|\ \mathrm{x} : [\mathrm{T}_2] \bullet \mathrm{p}_2(\mathrm{x})\ |\}$$

One then proves the subtype relation by ensuring that the maximal types are equal and by proving that the subtype predicate p_1 implies the subtype predicate p_2. That is

$$\begin{array}{l} \mathrm{T}_1 \preceq \mathrm{T}_2 \Leftrightarrow \\ \quad ([\mathrm{T}_1] = [\mathrm{T}_2]) \\ \quad\quad \wedge \\ \quad (\mathrm{p}_1 \Rightarrow \mathrm{p}_2) \end{array}$$

As an example, suppose we want to prove

$$\mathbf{Int} \rightarrow \mathbf{Nat} \preceq \mathbf{Nat} \overset{\sim}{\rightarrow} \mathbf{Int}$$

The two types can be written as subtypes of their maximal types

$$\mathbf{Int} \rightarrow \mathbf{Nat} = \{|\ \mathrm{f} : \mathbf{Int} \overset{\sim}{\rightarrow} \mathbf{Int} \bullet \mathrm{p}_1(\mathrm{f})\ |\}$$

$$\mathbf{Nat} \overset{\sim}{\rightarrow} \mathbf{Int} = \{|\ \mathrm{f} : \mathbf{Int} \overset{\sim}{\rightarrow} \mathbf{Int} \bullet \mathrm{p}_2(\mathrm{f})\ |\}$$

The maximal types are equal so it remains to be proven that p_1 implies p_2

The predicate $p_1(f)$ says that f must be defined for all integers and further must yield natural numbers for these.

The predicate $p_2(f)$ says that f must yield integers for natural numbers, if defined. This is always true since the partial functions from integers to integers in particular map natural numbers to integers, if defined.

Since p_2 is always true p_1 implies p_2 and the subtype relation then holds.

9.4 Subtype Correctness

Maximal type correctness of an RSL specification is checked by the automated type-checker. In addition the specifier should prove that the RSL specification is also subtype correct.

An expression is subtype correct with respect to the context in which it occurs, if the value of the expression belongs to the subtype required by the context.

Consider again example 9.1. The expression *four* is subtype correct with respect to the context *is_a_prime*(*four*) since the value of *four*, namely 4, belongs to the subtype, **Nat**, required by the context.

Then consider again example 9.2. For the axiom defining *squares* we have to check that

1. for the application of *apply*, the actual parameter value belongs to the formal parameter subtype

   (square,one_to_five) ∈ (Positive → **Int**) × Positive-**infset**

   For the *one_to_five*-parameter this is obviously true. The signature of *one_to_five* is namely

   one_to_five : Positive-**set**

   and

   Positive-**set** ⪯ Positive-**infset**

   For the *square*-parameter we have the following signature of *square*

   square : **Int** → **Nat**

   We must then prove that

   Int → **Nat** ⪯ Positive → **Int**

   The proof is similar to the one given in the previous section.

2. the result of *apply* belongs to the subtype of *squares*. That is

   apply(square,one_to_five) $\in$ Positive-**set**

   Note that the result type of *apply* (**Int-infset**) does not help us, since *Positive*$-$**set** is a subtype thereof. We can see that the required subtype constraint is satisfied by unfolding the application of *apply*

   {square(e) | e : Positive • e $\in$ one_to_five} $\in$ Positive-**set**

   It is obvious that the result is a finite set of positive numbers.

9.5 Subtypes Versus Axioms

When defining a value to have some type and to have some properties, one has a choice between specifying the properties as part of the type via a subtype predicate or to specify them in axioms.

Assume the type definition

```
type
   Prime = {| n : Nat • is_a_prime(n) |}
```

Consider then the definition

```
value
   p : Prime
```

This could also have been written as

```
value
   p : Nat
axiom
   is_a_prime(p)
```

or even as

```
value
   p : Int
axiom
   p ≥ 0 ∧ is_a_prime(p)
```

As another example, consider the following function definition

```
value
   f : Nat ~→ Prime
   f(n) as p
      post p > 1/n
      pre n ≠ 0
```

This could almost equivalently have been written as

```
value
   f : Int ~→ Int
   f(n) as p
      post p > 1/n ∧ is_a_prime(n)
      pre n > 0
```

Thus, for function definitions, predicates from subtypes in the domain will appear in the pre-condition, and predicates from subtypes in the range will appear in the post-condition.

For arguments different from zero (0) the two specifications of *f*'s behaviour are the same. The specifications, however, differ when it comes to the zero argument. The first *f* is specified to yield a prime number for zero, if defined at all. The second *f* may yield any integer number for zero, if defined.

From a pragmatic viewpoint, the two specifications seem equally good since we are not interested in *f*'s behaviour for the zero argument. When it comes to considering the implementation relation between specifications, the difference, however, becomes important: the first *f* implements the second, but not the other way round. That is, if we assume that *f* maps zero to a prime number, if defined for zero, then an implementation cannot violate that property.

9.6 Examples

Example 9.3

We have seen a number of examples where a wellformedness function expresses which values of a particular type that are wellformed (example 6.2, 6.3, 7.3, 8.2 and 8.3). The general form has been

```
type
   T = ...
```

```
  value
    is_wf_T : T → Bool
```

The function *is_wf_T* has, however, not been used to actually eliminate the undesired values from *T*. This can now be done through a subtype expression.

Consider the map version of the equivalence relation from example 8.2. We recall that a relation is a mapping from elements to partition identifiers. A relation is wellformed if it maps every element into some partition identifier.

The *Relation* type can now be defined by a subtype expression restricting the values to those relations that satisfy the wellformedness condition.

```
EQUIVALENCE_RELATION =
  class
    type
      Element,
      Partition_Id,
      Relation =
        {| r : Element →m Partition_Id • is_wf_Relation(r) |}
    value
      is_wf_Relation : (Element →m Partition_Id) → Bool,
      initial : Relation,
      make_equivalent : Element × Element × Relation → Relation,
      are_equivalent : Element × Element × Relation → Bool
    axiom forall e1,e2 : Element, r : Relation, m : Element →m Partition_Id •
      is_wf_Relation(m) ≡
        ∀ e : Element •
          e ∈ dom m,
      e1 ≠ e2 ⇒ initial(e1) ≠ initial(e2),
      make_equivalent(e1,e2,r) ≡
        r † [e ↦ r(e2) | e : Element • r(e) = r(e1)]
      are_equivalent(e1,e2,r) ≡
        r(e1) = r(e2)
  end
```

The following changes have been performed compared to the specification in example 8.2

1. The type *Relation* has been defined by a subtype expression which uses the function *is_wf_Relation*.
2. The function *is_wf_Relation* has got a different signature in that the argument type has become

   (Element $\underset{m}{\rightarrow}$ Partition_Id) instead of *Relation*. This has been necessary in order to avoid a recursion between *Relation* and the type of *is_wf_Relation*.

The functions *make_equivalent* and *are_equivalent* have become total. Their pre-conditions are no longer needed due to the wellformedness of their arguments.

3. The axiom saying that *initial* must be wellformed has been removed. It will be satisfied due to the type of *initial*.

We could have written the definition of type *Relation* in at least two other ways. The body of the function *is_wf_Relation* could have been unfolded into the subtype expression, thus removing the need to define the function explicitly

```
type
   Relation =
      {| r : Element ⇸ Partition_Id • ∀ e : Element • e ∈ dom r |}
```

The two solutions shown so far have the drawback that the "primary information", which is the map type expression, is textually hidden by the "secondary information", which is the wellformedness predicate.

An alternative solution could be to first give the primary information in one type definition, and then to give the secondary in another one

```
type
   Loose_Relation = Element ⇸ Partition_Id,
   Relation = {| r : Loose_Relation • ∀ e : Element • e ∈ dom r |}
```

□

Example 9.4

Consider a bounded version of the queue from example 7.1. Elements can be put into the queue and elements can be removed from the queue, in a "first in first out" manner. The queue is bounded in that there is a maximum size, *max*, which is a natural number greater than zero, such that no queue can have more than *max* elements.

The boundedness is expressed via a subtype expression. In addition, subtypes will be introduced for extendable queues (with a size less than *max*) and for reducable queues (different from *empty*). The last two subtypes illustrate how partial functions can be replaced by total functions, using subtypes.

```
QUEUE =
  class
    type
      Element,
      Queue = {| q : Element* • len q ≤ max |},
      Extendable_Queue = {| q : Queue • len q < max |},
      Reducable_Queue = {| q : Queue • q ≠ empty |}
    value
      max : Nat,
      empty : Extendable_Queue,
      put : Element × Extendable_Queue → Reducable_Queue,
      get : Reducable_Queue → Extendable_Queue × Element
    axiom forall e : Element, eq : Extendable_Queue, rq : Reducable_Queue •
      max > 0,
      empty ≡
        ⟨⟩,
      put(e,eq) ≡
        eq ^ ⟨e⟩,
      get(rq) ≡
        (tl rq,hd rq)
  end
```

□

10 Variant Definitions

Through a variant definition, one can in a short way define a sort together with a number of functions and constants over that sort.

10.1 Constant Constructors

As a very simple example, consider the following variant definition, defining a type by enumerating its values

```
**type**
  Colour == black | white
```

The type *Colour* contains two values represented by the so-called constant constructors *black* and *white*. In the scope of this definition one can then for example define a function for colour-inversion

```
value
  invert : Colour → Colour
axiom
  invert(black) ≡ white,
  invert(white) ≡ black
```

The definition of *Colour* is actually short for a sort definition, two value definitions and two axioms. The above type definition is short for

```
type
  Colour
value
  black : Colour,
  white : Colour
axiom
  [disjoint]
    black ≠ white
```

and an additional axiom, which is described below. The axiom says that *black* is different from *white*. This implies that the type *Colour* contains at least two values. Note, however, that no axiom prevents *Colour* from containing more than two values. An extra axiom is thus needed which expresses this limit on the size of *Colour*. The axiom should thus state that *Colour* contains only the values represented by *black* and *white*.

This is stated in a slightly different way: a so-called induction-axiom is generated

```
axiom
    [induction]
        ∀ p : Colour → Bool •
            (p(black) ∧ p(white)) ⇒ (∀ c : Colour • p(c))
```

The axiom says: "for all predicates *p*, if *p* holds for *black* and *p* holds for *white*, then *p* holds for all colours".

It may be a bit mysterious that this axiom implies that *Colour* only contains the values *black* and *white*.

10.2 Record Constructors

The individual alternatives separated by bars (|) can be composite instead of just constant-constructors. Consider the following variant definition

```
type
    Set == empty | add(Elem,Set)
```

The type *Set*, which is recursively defined, contains two kinds of values

1. the value *empty*
2. values of the form $add(e, s)$ where $e \in Elem$ and $s \in Set$. *add* is a so-called "record constructor".

The definition is short for a sort definition, two value definitions and two axioms. Ignoring again the induction-axiom, the above definition is short for

```
type
    Set
value
    empty : Set,
    add : Elem × Set → Set
axiom
    [disjoint]
        ∀ e : Elem, s : Set •
            empty ≠ add(e,s)
```

The *add* constructor is a function generating values different from *empty*. Note that in general, the bar implies disjointness.

The generated induction-axiom is a bit more complex than in the *Colour*-case

```
axiom
  [induction]
    ∀ p : Set → Bool •
      (p(empty) ∧ (∀ e : Elem, s : Set • p(s) ⇒ p(add(e,s)))) ⇒
        ∀ s : Set • p(s)
```

The axiom says: "for all predicates p, if p holds for *empty* and p holding for a set s implies p holding for $add(e, s)$ for any element e, then p holds for all sets".

Note that the above variant definition is equivalent to the following

```
type
  Set == empty | add(Elem × Set)
```

This can be seen by transforming the latter into its canonical form.

The *disjoint*-axiom says that *empty* differs from $add(e, s)$ for any element e and set s. Note, however, that there are no axioms stating that different applications of *add* yield different sets in *Set*. In fact, nothing is said about *add* beyond the disjointness from *empty* and the generatedness of *Set* by *empty* and *add*.

Given two different elements $e1$ and $e2$, the following property is thus not a consequence although we would like it to be

```
add(e1,empty) ≠ add(e2,empty)
```

To obtain this, we can introduce an observer function that tests whether a particular element is in a set

```
value
  is_in : Elem × Set → Bool
axiom forall e,e1 : Elem, s : Set •
  is_in(e,empty) ≡
    false
  is_in(e,add(e1,s)) ≡
    e = e1 ∨ is_in(e,s)
end
```

This function will now distinguish the two sets above. That is

```
is_in(e1,add(e1,empty)) = **true**

is_in(e1,add(e2,empty)) = **false**
```

and as a consequence of this we get the desired property

$$\text{add(e1,empty)} \neq \text{add(e2,empty)}$$

Note that we can deduce that due to the rule that

$$\text{f(x)} \neq \text{f(y)} \Rightarrow \text{x} \neq \text{y}$$

The introduction of the function *is_in* thus implies that those sets are destinguishable that we want to be destinguishable. The definition of the function *is_in* in general implies that sets show the expected behaviour: one can decide whether a particular element has been added or not.

Normally we think of a set as an unordered collection of distinct elements. The two important words here being “unordered” and “distinct”.

None of the above axioms, however, prevents sets from being ordered or to contain duplicates. That is, none of the following two axioms are consequences

```
axiom
  forall e,e1,e2 : Elem, s : Set •
  [unordered]
    add(e1,add(e2,s)) ≡ add(e2,add(e1,s)),
  [no_duplicates]
    add(e,add(e,s)) ≡ add(e,s)
```

Since they do not generally hold we can for example not compare two sets by ‘=’. Instead we must define a function for comparing sets that makes two sets equal if and only if they can be observed as equal through *is_in*.

```
value
  equal : Set × Set → Bool
axiom forall s1,s2 : Set •
  equal(s1,s2) ≡
    ∀ e : Elem •
      is_in(e,s1) = is_in(e,s2)
```

By adding the axioms *unordered* and *no_duplicates* we don't need to define the function *equal*, but can instead use '=' to test for equality between sets.

Irrespective of whether we add the axioms *unordered* and *no_duplicates* we can define a function for removing an element from a set and a function for returning an arbitrary element from a set

```
value
  remove : Elem × Set → Set
  choose : Set ~→ Elem
axiom
  forall e,e1 : Elem, s : Set •
  [remove_empty]
    remove(e,empty) ≡ empty,
  [remove_add]
    remove(e,add(e1,s)) ≡
      if e = e1 then
        remove(e,s)
      else
        add(e1,remove(e,s))
      end,
  [choose]
    choose(s) as e
      post is_in(e,s)
      pre s ≠ empty
```

Note in particular the definition of *choose*. It returns some arbitrary element which is just required to be in the the set. Consider then the following alternative axiom for *choose*

```
axiom forall e : Elem, s : Set •
  [choose_add]
    choose(add(e,s)) ≡ e
```

This axiom may seem harmless although it is not. It says that the *choose* function returns the last inserted element. For two different elements $e1$ and $e2$ we have

```
choose(add(e1,add(e2,s))) = e1

choose(add(e2,add(e1,s))) = e2
```

implying that

add(e1,add(e2,s)) $\neq$ add(e2,add(e1,s))

We thus get an inconsistency with the *unordered*-axiom. The *choose_add* axiom implies that a set must contain information about which element has been added as the last one.

Note, however, that we were not forced to add the *unordered*-axiom. Suppose we did not. Then we have got rid of the inconsistency. There is, however, still a problem concerned with implementation.

The problem arises when implementing the above set specification with another more concrete one. The implementing set specification must now implement *choose* to return the last added element. This means that the following type definition will not be satisfactory as an implementation, although it seems an obvious choice

```
type
  Set = Elem-set
```

Sets within this *Set* are mathematical sets and do thus not contain information about which element is the last one added.

10.3 Destructors

Consider the following variant definition

```
type
  List == empty | add(head : Elem, tail : List)
```

The difference between this definition and the previous *Set*-definition is, besides the new name *List* instead of *Set*, the introduction of the destructors *head* and *tail*.

Like *Set*, the type *List* contains two kinds of values

1. the value *empty*
2. values of the form $add(e, l)$ where $e \in Elem$ and $l \in List$.

The definition is short for a sort definition, four value definitions and four axioms. Ignoring the destructors, we get a sort definition, two value definitions and two axioms exactly as for *Set*

```
**type**
  List
**value**
  empty : List,
  add : Elem × List → List,
**axiom**
  [disjoint]
    ∀ e : Elem, l : List •
      empty ≠ add(e,l),
  [induction]
    ∀ p : List → **Bool** •
      (p(empty) ∧ (∀ e : Elem, l : List • p(l) ⇒ p(add(e,l)))) ⇒
        ∀ l : List • p(l)
```

Beyond these definitions, the destructors give rise to the following ones

```
**value**
  head : List $\xrightarrow{\sim}$ Elem,
  tail : List $\xrightarrow{\sim}$ List
**axiom**
  **forall** e : Elem, l : List •
  [head_add]
    head(add(e,l)) ≡ e,
  [tail_add]
    tail(add(e,l)) ≡ l
```

The destructors are partial in that their behaviour is un-specified for the *empty* list.

The destructors can be used to deconstruct *List* values generated by the *add* constructor. As an example consider the following specification

```
**value**
  replace_head : Elem × List $\xrightarrow{\sim}$ List
**axiom forall** e : Elem, l : List •
  replace_head(e,l) ≡
    add(e,tail(l))
    **pre** l ≠ empty
```

The destructor *tail* has here been used to remove the old head element before adding the new one.

The axiom defining *replace_head* could of course also have been written without the use of destructors

```
axiom forall e,e1 : Elem, l : List •
   replace_head(e1,add(e,l)) ≡ add(e1,l)
```

Destructors are thus not needed from a notational viewpoint in order to destruct values, but they provide a convenient way of doing it.

The occurrence of destructors in a variant definition is, however, not just a matter of convenience. The destructor-axioms, in this case *head_add* and *tail_add*, have an important effect on the properties of the constructor, in this case *add*: the constructor must be information-preserving. That is, the list value, say l_1, generated by $add(e, l)$ must be such that e can be recovered by $head(l_1)$ and such that l can be recovered by $tail(l_1)$.

Stated more formally, the axioms *unordered* and *no_duplicates*

```
axiom
   forall e,e1,e2 : Elem, l : List •
   [unordered]
      add(e1,add(e2,l)) = add(e2,add(e1,l)),
   [no_duplicates]
      add(e,add(e,l)) = add(e,l)
```

are inconsistent with the destructor-axioms *head_add* and *tail_add*. To see this for the *unordered*-axiom consider the following deduction. Given two different elements

```
e1 ≠ e2
```

By using the *head_add*-axiom, then the *unordered*-axiom and then again the *head_add*-axiom we can deduce the following

```
e1
=
head(add(e1,add(e2,l)))
=
head(add(e2,add(e1,l)))
=
e2
```

This is obviously inconsistent with the assumption that $e1$ and $e2$ are different.

The destructors thus really make *List*-values into ordered collections of elements, with the possibility of duplicates.

10.4 Reconstructors

The function *replace_head* introduced in the previous section can be introduced in a slightly more convenient way as a reconstructor. Below we repeat the definition of *List* with the addition of the reconstructor

```
type
   List == empty | add(head : Elem ↔ replace_head, tail : List)
```

The occurrence of the reconstructor is short for the following definitions, to be added to the previous ones

```
value
   replace_head : Elem × List ~→ List
axiom
   forall e : Elem, l : List •
   [head_replace_head]
      head(replace_head(e,l)) ≡ e,
   [tail_replace_head]
      tail(replace_head(e,l)) ≡ tail(l)
```

The two axioms relate the reconstructor *replace_head* to the destructors *head* and *tail*. The *head_replace_head*-axiom says that the *head*-destructor recovers the new head. The *tail_replace_head*-axiom says that the tail is unaffected.

If there are no destructors, no reconstructor-axioms are generated.

10.5 Subtype Namings

Through subtype namings, it is possible to name subtypes of the type generated by a variant definition. In the *List*-case, we can give the name *Empty_List* to the subtype of *List* containing the single value *empty*. Similarily we can give the name *Non_Empty_List* to the subtype of *List* containing all the other lists, i.e. those generated by *add*.

A subtype naming is introduced by appending '@ subtype_id' to the variant alternative in question. We thus get

```
type
   List ==
      empty @ Empty_List |
      add(head : Elem ↔ replace_head, tail : List) @ Non_Empty_List
```

The two subtype namings are short for the following type definitions

```
type
   Empty_List =
      {| l : List • l = empty |},
   Non_Empty_List =
      {| l : List • ∃ e : Elem, l1 : List • l = add(e,l1) |}
```

Note that subtype namings do not change the properties of lists.

The subtype namings can be utilized when specifying functions over the variant type. Consider for example a function for testing whether the heads of two non-empty lists are equal

```
value
   equal_heads : Non_Empty_List × Non_Empty_List → Bool
axiom forall l1,l2 : Non_Empty_List •
   equal_heads(l1,l2) ≡
      head(l1) = head(l2)
```

10.6 Forming Disjoint Unions of Types

The types *Set* and *List* represent so-called “containers”, where a container is a collection of elements. Variant definitions can also be used to form a type as a disjoint “union” of other types, without necessarily defining a container. Consider the following definition of a type of two-dimensional figures which are either boxes or circles

```
type
   Figure == box(length : Real, width : Real) | circle(radius : Real)
```

10.7 Wildcard Constructors

We have mentioned that a variant definition is short for a number of definitions including an induction-axiom. The induction-axiom restricts the variant type to contain only values generated by the constructors mentioned in the variant definition.

Sometimes one, however, wants to be loose about what the constructors are. As an example consider the above definition of the type *Figure*. Suppose that we are not sure whether there are more figures than boxes and circles. The definition of *Figure* can then alternatively be stated as follows

```
type
  Figure == box(length : Real, width : Real) | circle(radius : Real) | _
```

The difference is the last added variant which is a wildcard variant '_'. Formally, its occurrence means that no induction-axiom is generated. As a consequence, a later implementation may have more variants. An implementation may thus be

```
type
  Figure ==
    box(length : Real, width : Real) |
    circle(radius : Real) |
    triangle(base_line : Real, left_angle : Real, right_angle : Real)
```

It is not only the number of constructors which can be left open in a variant definition. It is also the components of a single constructor. Suppose for example that we are unsure of how to represent a triangle. An alternative to the above is to represent it by a base-line, a left-angle and a left-line length. This uncertancy can be stated as follows

```
type
  Figure ==
    box(length : Real, width : Real) |
    circle(radius : Real) |
    _(base_line : Real)
```

The difference from the previous definition of *Figure* is that the constructor *triangle* has been replaced by a wildcard and that the *left_angle* and *right_angle* components have been left out (we only know that a base-line component is needed).

Formally, the occurrence of the wildcard instead of the triangle-constructor means that there will be generated no value definition of a (triangle) constructor. As a consequence, there will be generated no axioms for the destructors, in this case *base_line* (recall that axioms for destructors relate these to corresponding constructors). As a third consequence, no induction-axiom will be generated as was also the case in the former use of wildcard.

An implementation of the latter definition of *Figure* is the former one.

10.8 Overloading

Due to overloading, constructors, destructors and reconstructors may be operators. Consider for example the following version of type *List*

type
  List == empty | ⁀(**hd** : Elem ↔ † , **tl** : List)

In the scope of this definition, the following expressions are valid for any $e \in \mathit{Elem}$ and non-empty $l \in \mathit{List}$

e ⁀ l

hd l

tl l

e † l

10.9 The General Form of a Variant Definition

In general, a variant definition has the form

type
  id == variant_1 | ... | variant_n

Each variant is either a constant variant of the form

id_or_op_or_wildcard @ subtype_id

or a record variant of the form

id_or_op_or_wildcard(
  destr_id_or_op_1 : type_expr_1 ↔ recon_id_or_op_1,
  ⋮
  destr_id_or_op_n : type_expr_n ↔ recon_id_or_op_n) @ subtype_id

Subtype namings, destructors and reconstructors are all optional.

10.10 Examples

Example 10.1

Consider the specification of an ordered binary tree of elements. A binary tree is either

1. empty
2. or composed of an element and two sub-trees: a left tree and a right tree.

   The ordering means that any of the elements in the left tree are less than the top element, which again is less than any of the elements in the right tree.

A function *less_than* represents the ordering on elements.

```
ORDERED_TREE =
  class
    type
      Elem,
      Tree ==
        empty |
        node(
          left : Tree,
          elem : Elem,
          right : Tree),
      Ordered_Tree =
        {| t : Tree • is_ordered(t) |}
    value
      is_ordered : Tree → Bool,
      extract_elems : Tree → Elem-set,
      less_than : Elem × Elem → Bool
    axiom
      forall e : Elem, t1,t2 : Tree •
      [is_ordered_empty]
        is_ordered(empty) ≡
          true,
      [is_ordered_node]
        is_ordered(node(t1,e,t2)) ≡
          ∀ e1 : Elem • e1 ∈ extract_elems(t1) ⇒ less_than(e1,e)
            ∧
          ∀ e2 : Elem • e2 ∈ extract_elems(t2) ⇒ less_than(e,e2)
            ∧
          is_ordered(t1)
            ∧
          is_ordered(t2),
      [extract_elems_empty]
        extract_elems(empty) ≡
          {},
      [extract_elems_node]
        extract_elems(node(t1,e,t2)) ≡
```

```
          extract_elems(t1) ∪ {e} ∪ extract_elems(t2)
  end
```

The type *Tree* is the type of binary trees, including the un-ordered ones. The type *Ordered_Tree* of ordered trees is defined as a subtype of *Tree*. Note that this two-step approach is necessary when defining subtypes of variant types.

The function *is_ordered* examines whether a tree is ordered.

The function *extract_elems* yields all the elements contained in a tree.

When a goal is execution-time efficiency, ordered trees are well-suited for modelling large sets of elements. The execution-time used for testing whether an element "belongs" to an ordered tree can be kept relatively low since sub-trees can be ignored which only contain elements smaller than or bigger than the element in question.

Consider an extension of the *ORDERED_TREE*-module with set-like functions for adding an element to a tree, *add*, and for testing whether an element belongs to a tree, *is_in*

```
SET_OPERATIONS =
  extend ORDERED_TREE with
    value
      add : Elem × Ordered_Tree → Ordered_Tree,
      is_in : Elem × Ordered_Tree → Bool
    axiom
      forall e,e0 : Elem, t1,t2 : Ordered_Tree •
      [add_empty]
        add(e,empty) ≡
          node(empty,e,empty),
      [add_node]
        add(e,node(t1,e0,t2)) ≡
          if e = e0 then
            node(t1,e0,t2)
          elsif less_than(e,e0) then
            node(add(e,t1),e0,t2)
          else
            node(t1,e0,add(e,t2))
          end,
      [is_in_empty]
        is_in(e,empty) ≡
          false,
      [is_in_node]
        is_in(e,node(t1,e0,t2)) ≡
          if e = e0 then
            true
          elsif less_than(e,e0) then
            is_in(e,t1)
          else
            is_in(e,t2)
          end
  end
```

The definition of the function *is_in* utilizes the fact that trees are ordered. That is, a sub-tree is ignored in the search of an element if all the elements in that sub-tree are either less than or greater than the searched element. This improves execution-time efficiency of a search.

The effeciency could further be improved, if trees were always balanced. A tree is balanced, if its two sub-trees have depths that at most differ by a choosen fixed maximum. The *add* function should then make sure that the resulting tree is balanced.

Choosing the maximum to be one (1) we can obtain balanced trees as follows

```
BALANCED_SET_OPERATIONS =
  extend SET_OPERATIONS with
    value
      is_balanced : Ordered_Tree → Bool,
      depth : Ordered_Tree → Nat,
      add_balanced : Elem × Ordered_Tree → Ordered_Tree
    axiom forall e : Elem, t,t1,t2 : Ordered_Tree •
      [is_balanced_empty]
        is_balanced(empty) ≡
          true,
      [is_balanced_node]
        is_balanced(node(t1,e,t2)) ≡
          abs(depth(t1) − depth(t2)) ≤ 1
            ∧
          is_balanced(t1)
            ∧
          is_balanced(t2),
      [depth_empty]
        depth(empty) ≡
          0,
      [depth_node]
        depth(node(t1,e,t2)) ≡
          1 + if depth(t1) > depth(t2) then depth(t1) else depth(t2) end,
      [add_balanced]
        add_balanced(e,t) as rt
          post
            extract_elems(rt) = extract_elems(t) ∪ {e}
              ∧
            is_balanced(rt)
  end
```

The function *is_balanced* examines whether a tree is balanced.

The function *depth* calculates the depth of a tree, which is the length of the longest path in the tree.

The function *add_balanced* adds an element to a tree, ensuring that the resulting tree is balanced. Note that since the type *Ordered_Tree* only includes ordered trees, the result tree will be ordered due to the result type of *add_balanced*.

The post-condition style used for specifying *add_balanced* is an appropriate initial specification of that function, since a concrete specification is rather more complicated.

We don't have to re-specify the function *is_in* since the one coming from *SET_OPERATIONS* is still sufficient.

□

Example 10.2

Consider a version of the map database from example 8.1. In that example, the database-operations *empty*, *insert*, *remove* and *lookup* were modelled as functions, one function for each. This style has in fact been applied in all examples until now.

An alternative is to only define a single function, say *evaluate*, which among its arguments takes an input-command, being either an empty-command, an insert-command, a remove-command or a lookup-command.

The *evaluate*-function further takes a database as argument. As result it yields a possibly changed database and an output-result.

The type of input-commands is defined as the “union” of the different kinds of input-commands. Likewise, the type of output-results is defined as the “union” of the different kinds of output-results. Both types can be given as variant-types.

```
VARIANT_DATABASE =
  class
    type
      Database = Key →m Data,
      Key, Data,
      Input == mk_empty | mk_insert(Insert) | mk_remove(Remove) | mk_lookup(Lookup),
      Insert = Key × Data,
      Remove = Key,
      Lookup = Key,
      Output == lookup_failed | lookup_succeeded(Data) | change_done
    value
      evaluate : Input × Database → Database × Output
    axiom forall k : Key, d : Data, db : Database •
      evaluate(mk_empty,db) ≡
        ([ ], change_done),
      evaluate(mk_insert(k,d),db) ≡
        (db † [k ↦ d], change_done),
      evaluate(mk_remove(k),db) ≡
        (db\{k}, change_done),
      evaluate(mk_lookup(k),db) ≡
        if k ∈ dom db then
          (db, lookup_succeeded(db(k)))
        else
          (db, lookup_failed)
        end
  end
```

The type *Output* contains three variants of values. Values of the form *lookup_failed* and *lookup_succeeded*(d), where $d \in Data$, are results of evaluating a *mk_lookup*(k)-command, where $k \in Key$. The result *lookup_failed* is returned if the key k is not in the domain of the database.

The *Output*-value *change_done* is the result of evaluating any of the commands *mk_empty*, *mk_insert*(k, d) and *mk_remove*(k), where $k \in Key$ and $d \in Data$. Note that this result is not likely to be used for anything. In these cases it is only the changed database that is of interest.

The definition of type *Input* contains no destructors. We could instead have written

```
  type
    Input ==
      mk_empty |
      mk_insert(sel_insert : Insert) |
      mk_remove(sel_remove : Remove) |
      mk_lookup(sel_lookup : Lookup)
```

As can be seen from the above specification, the destructors are not needed, and writing them just increases the size of the specification and forces us to invent new names for the destructors. There are thus good reasons for not writing them.

As stated earlier in connection with *List*, the occurrence of destructors has the effect of making the constructors (in this case *mk_insert*, *mk_remove* and *mk_lookup*) information-preserving. This is a consequence of the axioms defining the destructors. It would thus be more correct to write them, although one can argue that it is just a matter of under-specification when leaving them out.

The under-specification consists of not writing the axioms requiring the constructors to be information-preserving, thus allowing unintended implementations. Destructors should only be left out when it from the context is obvious what implementations are unintended.

Likewise for the type *Output*, which instead could be written as follows

```
  type
    Output ==
      lookup_failed | lookup_succeeded(sel_data : Data) | change_done
```

Another observation is the two-step definition of type *Input* with the introduction of the types *Insert*, *Remove* and *Lookup*. In the first step, the variant definition, we are thus only concerned with identifying the different kinds of commands. In the second step, the definition of the types *Insert*, *Remove* and *Lookup*, we further consider what these commands consist of.

We could alternatively have done all this in one step

```
  Command ==
    mk_empty |
    mk_insert(sel_key : Key, sel_data : Data) |
    mk_remove(sel_key : Key) |
    mk_lookup(sel_key : Key),
```

□

Example 10.3

A number of examples have been given of recursive variant definitions (*Set*, *List* and *Tree*). Variant definitions can also define mutually recursive types. That is, several types that are recursively defined in terms of each other.

Consider the specification of a hierarchical file directory. Such a directory is a mapping from identifiers to entries. An entry is either a file or again a directory.

```
FILE_DIRECTORY =
  class
    type
```

```
      Id, File,
      Directory = Id $\overrightarrow{m}$ Entry,
      Entry == mk_file(sel_file : File) | mk_dir(sel_dir : Directory)
  end
```

□

11 Case Expressions

The case expression allows for the selection of one of several alternative expressions, depending on the value of some expression.

As an example of a case expression, consider the following definition of the function *error_message*, the "body" of which is a case expression

```
value
   error_message : Nat → Text
axiom forall error_code : Nat •
   error_message(error_code) ≡
      case error_code of
         1 → "buffer is exceeded",
         2 → "buffer is empty",
         _ → "error"
      end
```

The evaluation of the case expression is done by first evaluating the expression *error_code*. Depending of the obtained value, one of the texts "`buffer is exceeded`"(if 1), "`buffer is empty`"(if 2) or "`error`"(otherwise) is returned.

The general form of a case expression is

```
case expr of
   pattern₁ → expr₁,
   ⋮
   patternₙ → exprₙ
end
```

The literals '1', '2' and the wildcard '_' are all patterns. A number of different kinds of patterns are allowed as will be described below.

The value of *expr* is matched against the patterns from top to bottom until a successful match is obtained whereupon the corresponding expression is evaluated. If none of the matches are successful, the result is undefined.

11.1 Literal Patterns

A pattern may be a value literal. That is, a literal of type **Unit**, **Bool**, **Int**, **Real**, **Text** or **Char**. We have already seen an example above with the literal patterns 1 and 2. Let us re-capture what the literals are for each built-in type

Unit : ()
Bool : **true**, **false**
Int : 0,1,2,...
Real : 0.0,...,6.17,...
Text : "`this is a text`","",...
Char : '`A`','`a`',...

A value matches a literal pattern successfully if the value equals the literal.

11.2 Wildcard Patterns

A pattern may be a wildcard pattern '_' as already illustrated in the introductory example. Any value matches a wildcard pattern succesfully. Wildcard patterns occurring at the outermost level in a case expression should occur last, if at all, to catch values not successfully matching previous patterns.

11.3 Name Patterns

A pattern may be a value name. A value matches a name pattern if the value equals the value represented by the name. The most typical situation is where the name is a constant-constructor introduced in a variant definition.

As an example recall the definition of type *Colour* in section 10 and the definition of the function *invert*. Now (with a repetition of the type definition) *invert* can be written in terms of a case expression

```
type
  Colour == black | white
value
  invert : Colour → Colour
axiom forall c : Colour •
  invert(c) ≡
    case c of
      black → white,
      white → black
    end
```

11.4 Record Patterns

Consider the following example which essentially is a reformulation of the *Set*-function *is_in* defined in section 10

```
type
   Set == empty | add(Elem,Set)
axiom forall e,e1,e2 : Elem, s : Set •
   [no_duplicates]
      add(e,add(e,s)) ≡ add(e,s),
   [unordered]
      add(e1,add(e2,s)) ≡ add(e2,add(e1,s))
value
   is_in : Elem × Set → Bool
axiom forall e : Elem, s : Set •
   is_in(e,s) ≡
      case s of
         empty → false,
         add(e1,s1) → e = e1 ∨ is_in(e,s1)
      end
```

The pattern $add(e1, s1)$ is a record-pattern. The value s matches this pattern successfully if s is a non-empty *Set*-value generated by *add*. The names $e1$ and $s1$ are bound to the sub-components of s and the succeeding expression is then evaluated within the scope of these two bindings.

This can be re-formulated in a slightly more correct way, observing that *add* is a function: the value s matches this pattern successfully if there exist values $x \in Elem$ and $y \in Set$ such that $s = add(x, y)$. If this is the case, $e1$ is bound to x while $s1$ is bound to y, and the succeeding expression is then evaluated within the scope of these two bindings.

Note that there may exist several pairs of values (x, y) for which $s = add(x, y)$. Assume for example that s equals $add(a, add(b, empty))$. Due to the *unordered*-axiom, there are at least two pairs (x, y) such that $s = add(x, y)$

```
(x,y) = (a,add(b,empty))

(x,y) = (b,add(a,empty))
```

In such a case, a non-deterministic choice is made between the pairs, whereupon $e1$ and $t1$ are bound to the chosen x and y.

The non-determinism of the $add(e1, s1)$-pattern in the above example does, however, not make the case expression non-deterministic since the order of selection of pairs $(e1, t1)$ does not influence its result.

An example where the non-determinism matters is the following. Assuming the same definition of type *Set*, we define a function for selecting an arbitrary element from a set

```
type
```

```
    Choose_Result == set_is_empty | element(sel_element : Elem)
  value
    choose : Set $\overset{\sim}{\rightarrow}$ Coose_Result
  axiom forall s : Set •
    choose(s) ≡
      case s of
        empty → set_is_empty,
        add(e,s1) → element(e)
      end
```

Note that since the function *choose* is non-deterministic, its type must involve a partial arrow (a total arrow implies determinism). Due to its definition it will, however, be defined for all values in the *Set* domain.

It may be worth noting that the above definition of *choose* is not equivalent to the following

```
  value
    choose : Set $\overset{\sim}{\rightarrow}$ Choose_Result
  axiom forall e : Elem, s : Set •
    [choose_empty]
      choose(empty) ≡ set_is_empty,
    [choose_add]
      choose(add(e,s)) ≡ e
```

The *choose_add*-axiom says that *choose* selects the last added element. This axiom is, however, inconsistent with the *unordered*-axiom which says that the order in which elements are added is of no importance.

In general, a record-pattern has the form

name(inner_pattern$_1$,...,inner_pattern$_n$), $n \geq 1$

where *name* represents some function of the type

$T_1 \times ... \times T_n \overset{\sim}{\rightarrow} T$

T is the type of the expression the value of which is matched against the pattern. Each T_i represents values that can be matched against *inner_pattern$_i$*.

An inner pattern is either a binding or a wildcard pattern '_'. Note that the identifiers occurring in an inner pattern (that is, in the binding) are defining occurrences in that they are bound as part of the pattern matching. This is in contrast to the *name* which must have been defined somewhere else. such a *name* is called a “referring occurrence”. The *name* constituting a name pattern described in the previous section is also a referring occurrence.

As an example utilizing the possibilities of inner patterns consider the following

```
type
  List == empty | add(head : Int × Int, tail : List)
value
  sum_of_head : List → Int
axiom forall l : List •
  sum_of_head(l) ≡
    case l of
      empty → 0,
      add((i,j),_) → i + j
    end
```

In contrast to the previous examples, all patterns are here deterministic. That is, the pattern $add((i,j),\_)$ is deterministic in that there for any list l exists at most one pair (x,y) such that $l = add(x,y)$. This is due to the occurrence of destructors *head* and *tail*.

11.5 List Patterns

Consider the following example of a function that calculates the sum of all the integers in a list

```
value
  sum : Int* → Int
axiom forall l : Int* •
  sum(l) ≡
    case l of
      ⟨⟩ → 0,
      ⟨i⟩ ^ l1 → i + sum(l1)
    end
```

The list l matches the pattern $\langle\rangle$ successfully if l equals the empty list. If l is not empty, l matches successfuly the next pattern $\langle i\rangle\hat{}l1$ if there exist an $x \in$ **Int** and a $y \in$ **Int*** such that $l = \langle x\rangle\hat{}y$. If such values x and y exist (and they do exist here since a list is either empty or it contains at least one element) i is bound to x and $l1$ is bound to y.

In general, a list pattern has one of four forms. That is, it has either the form of a so-called constructed list pattern

$\langle$inner_pattern$_1$,...,inner_pattern$_n\rangle$

or the form of a so-called left-list pattern

$\langle$inner_pattern$_1$,...,inner_pattern$_n\rangle$ $^\frown$ id_or_wildcard

or the form of a so-called right-list pattern

id_or_wildcard $^\frown$ $\langle$inner_pattern$_1$,...,inner_pattern$_n\rangle$

or the form of a so-called left-right-list pattern

$\langle$inner_pattern$_{1,1}$,...,inner_pattern$_{1,m}\rangle$ $^\frown$
id_or_wildcard $^\frown$
$\langle$inner_pattern$_{2,1}$,...,inner_pattern$_{2,n}\rangle$

where each $m, n \geq 0$.

Examples of list patterns are

$\langle$x,y$\rangle$

$\langle$(x,y)$\rangle$ $^\frown$ _

$\langle$x$\rangle$ $^\frown$ l $^\frown$ $\langle$y,_,z$\rangle$

l $^\frown$ $\langle$x$\rangle$

All identifiers occurring in list patterns are defining occurrences. That is, they are bound as part of the pattern matching. Note that an infinite list can only successfully match a left-list pattern.

An example using constructed list patterns as well as a left-right-list pattern is the definition of a function that reverses the ends of a text (recall that a text is a list of characters).

value
  reverse_ends : **Text** → **Text**
axiom forall t : **Text** •

```
  reverse_ends(t) ≡
    case t of
      ⟨⟩ → t,
      ⟨_⟩ → t,
      ⟨f⟩ ⌢ mid ⌢ ⟨l⟩ → ⟨l⟩ ⌢ mid ⌢ ⟨f⟩
    end
```

11.6 Product Patterns

Consider the definition of a function that calculates the "exclusive or" of two booleans (exactly one must be true)

```
value
  exclusive_or : Bool × Bool → Bool
axiom forall b1,b2 : Bool •
  exclusive_or(b1,b2) ≡
    case (b1,b2) of
      (true,false) → true,
      (false,true) → true,
      _ → false
    end
```

In general, a product pattern has the form

(pattern$_1$,...,pattern$_n$), $n \geq 2$

As another example consider the definition of a function that determines whether two lists match by examining pairs of corresponding elements. A function is assumed which determines whether two elements match. Two lists then match if they have the same length and if elements in corresponding positions match.

```
type
  List == empty | add(left : Elem, head : List)
value
  elements_match : Elem × Elem → Bool,
  lists_match : List × List → Bool
axiom forall l1,l2 : List •
  lists_match(l1,l2) ≡
    case (l1,l2) of
      (⟨⟩,⟨⟩) →
        true,
            (⟨e1⟩ ⌢ l1_rest,⟨e2⟩ ⌢ l2_rest) →
               elements_match(e1,e2) ∧ lists_match(l1_rest,l2_rest),
            _ → false
         end
```

11.7 Examples

Example 11.1

Consider a version of the ordered trees from example 10.1. The functions *is_ordered* and *extract_elems* were defined by two axioms each, an axiom for each kind of argument. In the example below, the two functions are instead defined in terms of case expressions

```
ORDERED_TREE =
   class
      type
         Elem,
         Tree ==
            empty |
            node(
               left : Tree,
               elem : Elem,
               right : Tree),
         Ordered_Tree =
            {| t : Tree • is_ordered(t) |}
      value
         is_ordered : Tree → Bool,
         extract_elems : Tree → Elem-set,
         less_than : Elem × Elem → Bool
      axiom forall t : Tree •
         is_ordered(t) ≡
            case t of
               empty →
                  true,
               node(t1,e,t2) →
                  ∀ e1 : Elem • e1 ∈ extract_elems(t1) ⇒ less_than(e1,e)
                     ∧
                  ∀ e2 : Elem • e2 ∈ extract_elems(t2) ⇒ less_than(e,e2)
                     ∧
                  is_ordered(t1)
                     ∧
                  is_ordered(t2)
            end,
         extract_elems(t) ≡
```

```
        case t of
          empty →
            {},
          node(t1,e,t2) →
            extract_elems(t1) ∪ {e} ∪ extract_elems(t2)
        end
  end
```

□

Example 11.2

Consider a version of the database from example 10.2. In that example a function called *evaluate* was defined by four axioms, one for each kind of argument. In the example below, the function is instead defined in terms of a case expression.

```
VARIANT_DATABASE =
  class
    type
      Database = Key →m Data,
      Key, Data,
      Input ==
        mk_empty |
        mk_insert(sel_insert : Insert) |
        mk_remove(sel_remove : Remove) |
        mk_lookup(sel_lookup : Lookup),
      Insert = Key × Data,
      Remove = Key,
      Lookup = Key,
      Output == lookup_failed | lookup_succeeded(sel_data : Data) | change_done
    value
      evaluate : Input × Database → Database × Output
    axiom forall input : Input, db : Database •
      evaluate(input,db) ≡
        case input of
          mk_empty →
            ([ ], change_done),
          mk_insert(k,d) →
            (db † [k ↦ d], change_done),
          mk_remove(k) →
            (db\{k}, change_done),
          mk_lookup(k) →
            if k ∈ dom db then
```

```
                    (db, lookup_succeeded(db(k)))
                 **else**
                    (db, lookup_failed)
                 **end**
           **end**
  **end**
```

□

12 Let Expressions

By a let expression one can introduce local names for particular values. There are two kinds of let expressions, namely explicit and implicit let expressions.

12.1 Explicit Let Expressions

Consider the following definition of a function that replaces the head of a non-empty list by its square

```
value
  square_head : Int* ~→ Int*
axiom forall l : Int* •
  square_head(l) ≡
    let h = hd l in
      ⟨h∗h⟩ ^ tl l
    end
    pre l ≠ ⟨⟩
```

The "body" of the function *square_head* is a let expression. The expression **hd** *l* is evaluated to an integer which is then bound to the value name *h*. The expression between **in** and **end** is then evaluated within the scope of this binding.

An explicit let expression has one of three forms

let binding = expr_1 **in** expr_2 **end**

let record_pattern = expr_1 **in** expr_2 **end**

let list_pattern = expr_1 **in** expr_2 **end**

The reader is referred to section 11 for the description of record patterns and list patterns.

The above example is an instantiation of the first form. In general, the expression $expr_1$ is evaluated to yield a value which is then matched against the *binding*, *record_pattern* or *list_pattern*. If the match is successfull, the expression $expr_2$ is then evaluated within the scope of the bindings that occurred as part of the match. If the match is not successfull, the value of the whole let expression is undefined.

The following equivalences hold between let expressions and case expressions

```
let record_pattern = $expr_1$ in $expr_2$ end
≡ case $expr_1$ of record_pattern → $expr_2$ end

let list_pattern = $expr_1$ in $expr_2$ end
≡ case $expr_1$ of list_pattern → $expr_2$ end
```

Another way of defining the *square_head*-function using a let expression with a (product) binding is

```
value
  square_head : $\mathbf{Int}^* \overset{\sim}{\to} \mathbf{Int}^*$
axiom forall l : $\mathbf{Int}^*$ •
  square_head(l) ≡
    let (h,t) = (hd l,tl l) in
      ⟨h∗h⟩ ⌢ t
    end
    pre l ≠ ⟨⟩
```

An example of a let expression using a record pattern is

```
type
  Set == empty | add(Elem,Set)
value
  choose : Set $\overset{\sim}{\to}$ Elem
axiom forall s : Set •
  choose(s) ≡
    let add(e,_) = s in
      e
    end
    pre s ≠ empty
```

Note that the choose function is non-deterministically choosing some member e from s. This can be seen by observing the equivalent case expression formulation

```
axiom forall s : Set •
  choose(s) ≡
    case s of
      add(e,_) → e
    end
    pre s ≠ empty
```

The reader may consult section 11 on case expressions for a discussion of this non-determinism.

An example of a let expression using a list pattern is

```
value
  square_head : Int* ~→ Int*
axiom forall l : Int* •
  square_head(l) ≡
    let ⟨h⟩ ⌢ t = l in
      ⟨h∗h⟩ ⌢ t
    end
    pre l ≠ ⟨⟩
```

12.2 Implicit Let Expressions

Another kind of let expression is the implicit let expression. Consider the following definition of a function that returns an arbitrary element from a set

```
value
  choose : Elem-set ~→ Elem
axiom forall s : Elem-set •
  choose(s) ≡
    let e : Elem • e ∈ s in
      e
    end
    pre s ≠ {}
```

The "body" of the function *choose* is an implicit let expression. The expression e between **in** and **end** is evaluated in the scope of a binding of e to a value within *Elem* such that $e \in s$.

An implicit let expression has one of two forms

```
let binding : type_expr • expr₁ in expr₂ end

let typing in expr end
```

The above example is an instantiation of the first form. An implicit let expression of the first form is evaluated by evaluating $expr_2$ in the scope of the *binding* where the identifiers in *binding* are non-deterministically bound to values that make the boolean expression $expr_1$ hold.

An implicit let expression of the second form is evaluated by evaluating the expression *expr* within the scope of the identifiers introduced in the typing. These identifiers are only specified via their type and are thus non-deterministically bound to values within their respective types.

An example of an implicit let expression using a typing is

```
value
  some_number : Unit ~→ Nat
axiom
  some_number() ≡
    let n : Nat in
      n
    end
```

The function non-deterministically returns some arbitrary natural number. *some_number* has been defined as a function such that a new number is obtained on each application. We could thus not have defined *some_number* as a constant, see below, and still obtain this non-determinism

```
value
  some_number : Nat
```

12.3 Nested Let Expressions

Often one may want to nest let expressions as in the following example. Suppose that a list is represented by as a map from natural numbers (list positions) into elements together with a pointer (a natural number) to the head. The function *add* can then be defined as follows

```
type
  List = Nat × (Nat →m Elem)
value
  add : Elem × List → List
axiom forall e : Elem, l : List •
  add(e,l) ≡
    let (top,map) = l in
      let new_top = top + 1 in
        let new_map = map † [new_top ↦ e] in
          (new_top,new_map)
        end
      end
    end
```

A shorthand syntax allows us to avoid the nesting and instead to write the axiom for *add* as follows

```
axiom forall e : Elem, l : List •
   add(e,l) ≡
      let
         (top,map) = l,
         new_top = top + 1,
         new_map = map † [new_top ↦ e]
      in
         (new_top,new_map)
      end
```

In general, a multiple let expression of the form

```
let let_def_1,...,let_def_n in expr end
```

is short for

```
let let_def_1 in
   ⋮
   let let_def_n in
      expr
   end
   ⋮
end
```

where each *let_def* has one of the forms

```
binding = expr

record_pattern = expr

list_pattern = expr

binding : type_expr • expr

typing
```

12.4 Example

Example 12.1

Consider a version of the resource manager from example 6.1. Recall that the resource manager maintains a pool, which is a set of free resources. A function *obtain* selects an arbitrary resource from the pool while the function *release* returns a resource to the pool.

In example 6.1 the function *obtain* was defined in terms of a post-condition. This implied determinism: applied twice to the same pool, the function *obtain* would return the same resource.

We can now make the function non-deterministic by defining it in terms of an implicit let expression. We repeat the entire *RESOURCE_MANAGER*-module, although it is only the *obtain*-function that has been changed.

```
RESOURCE_MANAGER =
  class
    type
      Resource,
      Pool = Resource-set
    value
      initial : Pool,
      obtain : Pool $\overset{\sim}{\to}$ Pool × Resource,
      release : Resource × Pool $\overset{\sim}{\to}$ Pool
    axiom forall r : Resource, p : Pool •
      obtain(p) ≡
        let r : Resource • r ∈ p in
          (p\{r},r)
        end
        pre p ≠ {},
      release(r,p) ≡
        p ∪ {r}
        pre r ∈ initial\p
  end
```

□

13 Union and Record Definitions

There are situations where a type can be seen as a hierarchy of types. Consider for example the following requirements specification for airport events

1. An *airport event* is either an *airplane event* or a *passenger event.*
2. An *airplane event* is either a *landing* or a *take off.* A *landing* is characterised by a flight identification and a landing time. A *take off* is characterised by a flight identification.
3. A *passenger event* is either a *reservation*, a *check in* or a *cancel.* A *reservation* is characterised by a passenger identification and a flight identification. A *check in* is characterised by a passenger identification, a flight identification and a seat number. A *cancel* is characterised by a passenger identification and a flight identification.

13.1 Using a Layered Variant Definition

Using variant definitions the above requirements specification can be expressed as follows. Assume the following basic types.

```
BASIC_TYPES =
  class
    type
      Flight,
      Passenger,
      Time,
      Seat
  end
```

The airport types are then defined as an extension as follows.

Example 13.1

```
AIRPORT_TYPES =
  extend BASIC_TYPES with
    type
      Airport_Event ==
        mk_airplane_event(sel_airplane_event : Airplane_Event) |
        mk_passenger_event(sel_passenger_event : Passenger_Event),
      Airplane_Event ==
        mk_landing(sel_landing : Landing) |
        mk_take_off(sel_take_off : Take_Off),

      Passenger_Event ==
        mk_reservation(sel_reservation : Reservation) |
        mk_check_in(sel_check_in : Check_In) |
        mk_cancel(sel_cancel : Cancel),

      Landing = Flight × Time,
      Take_Off = Flight,

      Reservation = Passenger × Flight,
      Check_In = Passenger × Flight × Seat,
      Cancel = Passenger × Flight
  end
```

□

An immediate observation is that two layers of constructors are introduced. That is, given a $f \in \mathit{Flight}$ and a $t \in \mathit{Time}$, we must apply two constructors in order to obtain the airport event "landing flight"

```
  mk_airplane_event(mk_landing(f,t))
```

This may appear tedious when writing functions over the *Airport_Event*-type. Consider for example a function for extracting the flight identification of an event. This function can be defined in terms of a nested case expression.

Example 13.2

```
FLIGHT_IDENTIFICATION =
  extend AIRPORT_TYPES with
    value
      flight_identification : Airport_Event → Flight
    axiom forall airport_event : Airport_Event •
      flight_identification(airport_event) ≡
        case airport_event of
          mk_airplane_event(airplane_event) →
            case airplane_event of
```

```
              mk_landing(flight,_) → flight,
              mk_take_of(flight) → flight
            end,
          mk_passenger_event(passenger_event) →
            case passenger_event of
              mk_reservation(_,flight) → flight,
              mk_check_in(_,flight,_) → flight,
              mk_cancel(_,flight) → flight
            end
        end
  end
```

□

The above example has two layers of constructors. One can imagine examples with three or more layers, which thus become even more tedious.

13.2 Union Definitions

RSL provides a way of avoiding the constructors from layered variant definitions. Assume that the identifiers $id_1 \ldots id_n$ are names for types, then a so-called "union definition" of the form

```
  type
    id = id₁ | ... | idₙ
```

with $n \geq 2$ is short for

```
  type
    id ==
      id₁_to_id(id₁_from_id : id₁) | ... | idₙ_to_id(idₙ_from_id : idₙ)
```

That is, the shorthand allows one to omit the constructors and destructors in the type definition. What is more important is that one may omit the constructors when writing functions over the type *id*, or more formally: they can be omitted when writing expressions. Omitting them is short for writing them. Let us respecify the above example using union definitions.

Example 13.3

```
AIRPORT_TYPES =
  extend BASIC_TYPES with
    type
      Airport_Event = Airplane_Event | Passenger_Event,
      Airplane_Event = Landing | Take_Off,
      Passenger_Event = Reservation | Check_In | Cancel,

      Landing ==
        mk_landing(sel_flight : Flight, sel_time : Time),
      Take_Off ==
        mk_take_off(sel_flight : Flight),

      Reservation ==
        mk_reservation(sel_passenger : Passenger, sel_flight : Flight),
      Check_In ==
        mk_check_in(sel_passenger : Passenger, sel_flight : Flight, sel_seat : Seat),
      Cancel ==
        mk_cancel(sel_passenger : Passenger, sel_flight : Flight)
  end
```

□

Note the definition of the types *Landing*, *Take_Off*, *Reservation*, *Check_In* and *Cancel*. They are all defined by variant definitions, each with only a single alternative. We need to define these types as constructed and thus not as abbreviations of cartesian products for the following reason.

Our intension is to avoid referring to the implicit constructors introduced by the definition of *Airport_Event*, *Airplane_Event* and *Passenger_Event*. That is, no references will be made to the constructors

```
  Airplane_Event_to_Airport_Event
  Passenger_Event_to_Airport_Event
  Landing_to_Airplane_Event
  Take_Off_to_Airplane_Event
  Reservation_to_Passenger_Event
  Check_In_to_Passenger_Event
  Cancel_to_Passenger_Event
```

So in order to be able to distinguish values of the type *Airport_Event*, constructors must be introduced "at the lowest level".

Note also that we have to introduce destructors in order to make the constructors *mk_landing*, *mk_take_off*, *mk_reservation*, *mk_check_in* and *mk_cancel* information-preserving.

With these definitions one can now define the function *flight_identification* as follows, recalling that implicit constructors can be left out in patterns.

Example 13.4

```
FLIGHT_IDENTIFICATION =
  extend AIRPORT_TYPES with
    value
      flight_identification : Airport_Event → Flight
    axiom forall airport_event : Airport_Event •
      flight_identification(airport_event) ≡
        case airport_event of
          mk_landing(flight,_) → flight,
          mk_take_of(flight) → flight
          mk_reservation(_,flight) → flight,
          mk_check_in(_,flight,_) → flight,
          mk_cancel(_,flight) → flight
        end
  end
```

□

13.3 Record Definitions

Referring back to the definitions of types *Landing*, *Take_Off*, *Reservation*, *Check_In* and *Cancel* we recall that they are defined as variants, each with only a single alternative. Thus, for example *Landing* was defined as follows

```
  type
    Landing ==
      mk_landing(sel_flight : Flight, sel_time : Time)
```

Such a definition appears somewhat odd since there is only one alternative. A slightly shorter form allows us to omit the constructor in the type definition. We can thus write a so-called “record definition”

```
  type
    Landing ::
      sel_flight : Flight
      sel_time : Time
```

which is then short for

```
type
  Landing ==
    mk_Landing(sel_flight : Flight, sel_time : Time)
```

In general, a type definition of the form

```
type
  id ::
    destr_id$_1$ : type_expr$_1$ ↔ recon_id$_1$
    ⋮
    destr_id$_n$ : type_expr$_n$ ↔ recon_id$_n$
```

is standing for

```
type
  id ==
    mk_id(
      destr_id$_1$ : type_expr$_1$ ↔ recon_id$_1$,
      ⋮
      destr_id$_n$ : type_expr$_n$ ↔ recon_id$_n$)
```

Note that the constructor *mk_id* cannot be omitted when writing functions over the type *id*. As for traditional variant definitions, destructors and reconstructors are optional.

We can thus finally write the airport types as follows.

Example 13.5

```
AIRPORT_TYPES =
  extend BASIC_TYPES with
    type
      Airport_Event = Airplane_Event | Passenger_Event,
      Airplane_Event = Landing | Take_Off,
      Passenger_Event = Reservation | Check_In | Cancel,

      Landing ::
        sel_flight : Flight
        sel_time : Time,
      Take_Off ::
        sel_flight : Flight,

      Reservation ::
        sel_passenger : Passenger
        sel_flight : Flight,
      Check_In ::
        sel_passenger : Passenger
        sel_flight : Flight
        sel_seat : Seat,
      Cancel ::
        sel_passenger : Passenger
        sel_flight : Flight
  end
```

□

The function *flight_identification* can now be defined as follows.

Example 13.6

```
FLIGHT_IDENTIFICATION =
  extend AIRPORT_TYPES with
    value
      flight_identification : Airport_Event → Flight
    axiom forall airport_event : Airport_Event •
      flight_identification(airport_event) ≡
        case airport_event of
          mk_Landing(flight,_) → flight,
          mk_Take_Of(flight) → flight
          mk_Reservation(_,flight) → flight,
          mk_Check_In(_,flight,_) → flight,
          mk_Cancel(_,flight) → flight
        end
  end
```

□

13.4 Using a Flat Variant Definition

An alternative is of course to define the *Airport_Event*-type by a flat variant definition and then ignore the concepts of *airplane event* and *passenger event* in the formal specification. This is done below.

Example 13.7

```
AIRPORT_TYPES =
  extend BASIC_TYPES with
    type
      Airport_Event ==
        mk_landing(flight : Flight, time : Time) |
        mk_take_off(flight : Flight) |
        mk_reservation(passenger : Passenger, flight : Flight) |
        mk_check_in(passenger : Passenger, flight : Flight, seat : Seat) |
        mk_cancel(passenger : Passenger, flight : Flight)
  end
```

□

Alternatively, if the concepts of *airplane event* and *passenger event* are important, one can define the types *Airplane_Event* and *Passenger_Event* as subtypes of *Airport_Event*.

13.5 Example

Example 13.8

Consider a rewriting of the database from example 11.2 using union definitions instead of variant definitions of the types *Input* and *Output*.

```
VARIANT_DATABASE =
  class
    type
      Database = Key →m Data,
      Key, Data,
      Input = Empty | Insert | Remove | Lookup,
      Empty == mk_empty
      Insert :: sel_key : Key sel_data : Data,
      Remove :: sel_key : Key,
      Lookup :: sel_key : Key,
      Output = Lookup_Output | Change_Output,
      Lookup_Output = Lookup_Failed | Lookup_Succeeded,
      Lookup_Failed == lookup_failed,
      Lookup_Succeeded :: sel_data : Data,
      Change_Output == change_done
    value
      evaluate : Input × Database → Database × Output
    axiom forall input : Input, db : Database •
      evaluate(input,db) ≡
        case input of
          mk_empty →
            ([ ], change_done),
          mk_Insert(k,d) →
            (db † [k ↦ d], change_done),
          mk_Remove(k) →
            (db\{k}, change_done),
          mk_Lookup(k) →
            if k ∈ dom db then
              (db, mk_Lookup_Succeeded(db(k)))
            else
              (db, lookup_failed)
            end
        end
  end
```

Although the type *Input* only "consists of one layer" it has been defined by a union definition anyway. This is to illustrate that union definitions generally can be used as an alternative style of specification instead of variant definitions.

□

Part II

State-based Specifications

14 Some Basic Concepts

RSL allows for the declaration of variables as known from most programming languages like Ada, Pascal and C. A variable is a container capable of holding values of a particular type. The contents of a variable can be changed by assigning a new value to the variable. A variable can thus change contents within its lifetime.

The following module defines a variable *counter* and a function *increase* that increases the counter by one for each call. The function additionally returns the value of the counter after incrementation.

Example 14.1

```
COUNTER =
  class
    variable
      counter : Nat := 0
    value
      increase : Unit → write counter Nat,
    axiom
      increase() ≡
        counter := counter + 1 ; counter
  end
```

□

We shall in the following explain the individual declarations of the module.

14.1 Variable Declarations

A variable declaration has the form

```
  variable
    variable_definition₁,
    ⋮
    variable_definitionₙ
```

In our specification there is one such definition.

A variable definition has the form

id : type_expr := expr

That is, the variable *id* is defined to contain values of the type represented by *type_expr*. The initial value of the variable is set to the value obtained by evaluating *expr*. The initialisation is optional and if not given explicitly, the initial value will be some arbitray value within the specified type.

The variable *counter* in the example is defined to contain values of type **Nat**, with the initial value being zero (0).

When several variables have the same type, a multiple variable definition of the following form can be used

id_1,...,id_n : type_expr

which is short for

id_1 : type_expr,
$\vdots$
id_n : type_expr

A particular association of values with all declared variables is called a state. As will be seen, assignment is a state changing operation.

14.2 Functions with Variable Access

The function *increase* from the example has the type

Unit → **write** counter **Nat**

That is, it is a function that when applied to a value of type **Unit** returns a value of type **Nat**. As a side-effect it writes to the variable *counter*. A function with variable access, like *increase*, is also called an operation.

The example illustrates a typical use of the type **Unit**: as parameter type for operations that only depend on the state and not on any additional parameters. The parameter type of an operation can of course be any type. We shall later see examples of operations with result type **Unit**. That is, where the only interesting effect of an operation is the way it changes the state.

In general, a type expression for total operations has the form

  type_expr$_1$ $\rightarrow$ access_desc$_1$ ... access_desc$_n$ type_expr$_2$

A function of this type takes arguments from the type represented by *type_expr*$_1$ and returns results within the type represented by *type_expr*$_2$.

Each of the access descriptors *access_desc*$_i$ is either of the form

  write id$_1$,...,id$_n$

expressing which variables may be written to (as well as read from), or of the form

  read id$_1$,...,id$_n$

expressing which variables may only be read from. As an example illustrating the occurrence of a **read** access description, consider the definition of an operation that just returns the current value of the counter.

Example 14.2

```
READ_COUNTER =
  extend COUNTER with
    value
      return_counter : Unit → read counter Nat
    axiom
      return_counter() ≡
        counter
  end
```

□

A type expression for partial operations has the form

  type_expr$_1$ $\overset{\sim}{\rightarrow}$ access_desc$_1$ ... access_desc$_n$ type_expr$_2$

14.3 Assignment Expressions

A variable *id* can be assigned to by an assignment expression of the form

id := expr

The effect of such an expression is to assign the value of the expression to the variable represented by *id*.

Our example contains one assignment expression, namely

counter := counter + 1

There is an important point to note here. Assignment is an expression. In general there is no destinction between expressions and statements as often seen in traditional programming languages such as Ada and Pascal. In RSL there are only expressions.

Since assignment is an expression, it must in addition to its side-effect also yield a value of a certain type. The value returned by an assignment expression is the value '()' of type **Unit**.

14.4 Sequencing Expressions

Two expressions can be combined with the sequencing combinator yielding a new composite expression

expr_1 ; expr_2

The composite expression is evaluated by first evaluating expr_1 for the purpose of its possible side-effect on variables, and then by evaluating expr_2 in the changed state. The value returned by the composite expression is the value returned by expr_2. The type of expr_1 must be **Unit**.

Our example contains the following sequencing expression

counter := counter + 1 ; counter

14.5 Pure and Read-only Expressions

Expressions can occur in contexts where they are not allowed to refer to variables at all. There are other contexts where they must not write to variables, although reading from variables is allowed. We shall thus in the following often refer to the terms pure expression and read-only expression, defined as follows.

A pure expression is an expression that does not access variables. That is, a pure expression neither reads from or writes to variables. Examples of pure expressions are

5

{n | n : **Nat** • n > 0}

A read-only expression is an expression that does not write to variables, but it may read from variables. As example, assume the variable definition

variable
  x : **Int**

then the following are read-only expressions

5

x + 1

An expression that is neither pure nor read-only is

x := x + 1

An occurrence of an expression which is required to be pure is the initialisation expression in a variable definition.

14.6 Quantification over States

How do we interprete axioms in the context of variables? The most natural thing is to say that an axiom is **true** if it is **true** in any possible state satisfying the variable definitions. A state satisfies a variable definition if it associates the defined variable with a value within the specified type.

The always-combinator '□' performs this universal quantification over states. An always-expression has the form

$\Box$ expr

and has the type **Bool**. The value of the always-expression is **true** if and only if for all states satisfying the variable definitions, the expression *expr* evaluates deterministically to **true**. Otherwise, the always-expression evaluates to **false**.

The expression *expr* must not change the state, but it may depend on the state by reading from variables. That is, *expr* must be read-only. The always-expression itself is pure: it does not write to variables and it does not depend on the current value of variables (due to the quantification).

Axioms are interpreted with a universal quantification over all states. An axiom of the form

axiom
  expr_1

is thus short for

axiom
  $\Box$ expr

Since '$\Box$' requires the expression to be read-only (see above), axioms must in general be read-only.

In the general case, an axiom declaration of the form

axiom forall typing_1,...,typing_m •
  opt_axiom_naming$_1$ expr_1,
  ⋮
  opt_axiom_naming$_n$ expr_n

is short for

axiom
  opt_axiom_naming$_1$ $\Box\ \forall$ typing_1,...,typing_m • expr_1,
  ⋮
  opt_axiom_naming$_n$ $\Box\ \forall$ typing_1,...,typing_m • expr_n

14.7 Equivalence Expressions

Our example contains a single axiom

$$\begin{aligned}&\mathrm{increase}() \equiv\\ &\quad \mathrm{counter} := \mathrm{counter} + 1 \;;\; \mathrm{counter}\end{aligned}$$

which is an equivalence expression of the form

$$\mathrm{expr}_1 \equiv \mathrm{expr}_2$$

Since this equivalence expression occurs as an axiom, it is short for

$$\Box\ \mathrm{expr}_1 \equiv \mathrm{expr}_2$$

The left-hand side of the equivalence ($expr_1$) is the expression

$$\mathrm{increase}()$$

which represents the application of the operation *increase* to the unit value '()'. Note here that an application expression of the form

$$\mathrm{expr}()$$

is short for

$$\mathrm{expr}(())$$

The right-hand side of the equivalence ($expr_2$) is the sequencing expression

$$\mathrm{counter} := \mathrm{counter} + 1 \;;\; \mathrm{counter}$$

We will now explain in more detail what equivalence '≡' means.

The expression

$\text{expr}_1 \equiv \text{expr}_2$

is a boolean expression which is evaluated in the current state. It evaluates to **true** if and only if the effect of $\textit{expr}_1$ evaluated in the current state is exactly the same as the effect of $\textit{expr}_2$ evaluated in the same state. That is, the two expressions must have the same side-effects on variables as well as yield the same result value. If this is not the case, the equivalence expression evaluates to **false**.

The equivalence also requires equivalent effects concerning undefinedness. That is, if one of the expressions is undefined, the other one must also be. Note that an equivalence expression always evaluates to either **true** or **false**, it will never be undefined itself.

Finally, if one of the expressions is non-deterministic, the other one must show exactly the same degree of non-determinism in order for the equivalence to hold.

The equivalence expression itself has no side-effects since the side-effects obtained by evaluating the two sub-expressions are only utilized in the comparison of effects, and are thus ignored thereafter. The value of the equivalence expression may, however, depend on the state if variables are accessed. An equivalence expression is thus defined to be read-only.

When an equivalence expression occurs as an axiom, it says that for all states satisfying the visible variable definitions, the effects of the two expressions must be the same. This implies that any occurrence of $\textit{expr}_1$ within the scope of the variable definitions can be replaced by $\textit{expr}_2$ and vice versa.

The axiom from our example above says that the *increment* operation for any possible state must have the same effect as the right-hand side of the equivalence, just as one would expect from reading the axiom.

In later sections we shall see more "advanced" uses of equivalence where the left-hand side is not just a single function application, but a general expression. One can thus specify operations in an algebraic style similar to that decsribed in section 5 for applicative functions.

14.8 Conditional Equivalence Expressions

An equivalence may be conditional. Such an equivalence contains a pre-condition

$\text{expr}_1 \equiv \text{expr}_2 \ \textbf{pre} \ \text{expr}_3$

where $\textit{expr}_3$ must be a read-only boolean expression. This is short for

$(\text{expr}_3 \equiv \textbf{true}) \Rightarrow (\text{expr}_1 \equiv \text{expr}_2)$

As an example, suppose we want to specify also a *decrease* operation, but only for states where the *counter* is greater than zero. This could be done as follows.

Example 14.3

```
DECREASE =
  extend COUNTER with
    value
      decrease : Unit $\overset{\sim}{\rightarrow}$ write counter Nat,
    axiom
      decrease() $\equiv$
        counter := counter − 1 ; counter
        pre counter > 0
  end
```

□

14.9 Equivalence and Equality

Consider two expressions $expr_1$ and $expr_2$. If these have no side-effects on variables, are both defined and are both deterministic, equality '=' and equivalence '≡' mean the same. That is, the expression

$\mathrm{expr}_1 = \mathrm{expr}_2$

has the same meaning as

$\mathrm{expr}_1 \equiv \mathrm{expr}_2$

If one of the expressions has side-effects, is undefined or is non-deterministic, the meaning of equivalence is different from the meaning of equality. The expression

$\mathrm{expr}_1 = \mathrm{expr}_2$

is a boolean expression evaluated as follows.

If one of the expressions is undefined, the equality expression becomes undefined. That is to say, equality is a so-called "strict" operator. If both expressions are defined, they each (possibly non-deterministically) yield a side-effect and a value. The value of the equality expression is then **true** if the two values are equal, otherwise it is **false**. The side-effect of the equality expression is the sum of the side-effects of the sub-expressions.

The two sub-expressions of an equality must be independent in the following sense: if one expression writes to a variable, the other expression must not access that variable (read from it or write to it). This independency-requirement has the desired consequence that the order of evaluation of the two sub-expressions does not influence the result.

14.10 Operation Calls and the Result-type Unit

The operation *increase* can be called via an application expression, just like any other function. Consider the following definition of an operation that increments the counter and returns a boolean value depending on a comparison of the value of the resulting counter and the parameter.

Example 14.4

```
TEST_COUNTER =
  extend COUNTER with
    value
      increase_and_test : Nat → write counter Bool
    axiom forall n : Nat •
      increase_and_test(n) ≡
        increase() ≤ n
  end
```

□

Suppose now that we want to specify an operation for incrementing the counter twice and that we want to specify it in terms of two calls of the *increment* operation. We observe that the following expression is not allowed

```
  increment() ; increment()
```

due to the rule that the expression before the semicolon must have the type **Unit**. Instead we can specify the *increment_twice* function as follows.

Example 14.5

```
INCREMENT_TWICE =
  extend COUNTER with
    value
      increment_twice : Unit → write counter Nat
    axiom
      increment_twice() ≡
        let dummy = increment() in
          increment()
        end
  end
```

□

We thus have to introduce a *dummy* name for the result returned by the first application of *increment*. In general one should be careful when letting an operation have a result type different from **Unit**. It means that such an operation cannot be called immediately in front of semicolon.

In our example one could easily separate the operations for incrementing the counter and for reading the counter. This is done below.

Example 14.6

```
COUNTER =
  class
    variable
      counter : Nat := 0
    value
      increment : Unit → write counter Unit,
      return_counter : Unit → read counter Nat
    axiom
      increment() ≡
        counter := counter + 1,
      return_counter() ≡
        counter
  end
```

□

The operation *increment_twice* (with unchanged signature) will then become as follows.

Example 14.7

```
INCREMENT_TWICE =
  extend COUNTER with
    value
      increment_twice : Unit → write counter Nat
    axiom
      increment_twice() ≡
        increment() ; increment() ; return_counter()
  end
```

□

14.11 Example

Example 14.8

Consider a state-based version of the database from example 8.1. A variable containing the database is introduced, and all operations then read from and write to this variable.

```
DATABASE =
  class
    type
      Key, Data
    variable
      database : Key ⇀m Data
    value
      empty : Unit → write database Unit,
      insert : Key × Data → write database Unit,
      remove : Key → write database Unit,
      defined : Key → read database Bool,
      lookup : Key ~→ read database Data
    axiom forall k : Key, d : Data •
      empty() ≡
        database := [ ],
      insert(k,d) ≡
        database := database † [k ↦ d],
      remove(k) ≡
         database := database \ {k},
      defined(k) ≡
         k ∈ **dom** database,
      lookup(k) ≡
         database(k)
         **pre** defined(k)
   **end**
```

There may be several reasons for writing state-based specifications instead of applicative specifications. Some typical reasons are:

1. Programs written in traditional programming languages are typically state-based. Thus, at some point in the development of a program from a specification, one may want to switch to a state-based specification style, if the starting point was an applicative one.
2. The state-based style of specification reduces the number of parameters to functions. Thus, a call of *insert* has the form

   insert(k,d)

   for some $k \in Key$ and $d \in Data$. A call of the applicative *insert* from example 8.1 has an extra parameter, namely the database

   insert(k,d,db)

   for some $k \in Key$, $d \in Data$ and $db \in Database$. Recall that the applicative version of *insert* had the type

   ```
   value
      insert : Key × Data × Database → Database
   ```

   This argument for state-based specification can be reversed to an argument against the style: one cannot from the call of an operation see what variables are accessed, one has to look into the type of the operation.
3. Certain problems can be said to be of a state-based nature, like the database example. One may then prefer to model them as such.

□

15 Expressions Revisited

In general, all expressions are evaluated in a state. This also holds for the expressions introduced in part one on applicative specifications. In this section we shall shortly revisit these expressions in the light of their evaluation in a state.

15.1 Pure and Read-only Expressions

That an expression is introduced in part one does not mean that it cannot access variables. We have already seen examples of both if-expressions and let-expressions accessing variables.

There are though generally restrictions on how variables can be accessed as already indicated by the introduction of pure an read-only expressions in section 14. We shall not revisit all expressions here but just give some examples. The concrete syntax in appendix A describes the occurrences of expressions that must either be pure or read-only.

An example of an expression occurrence that is required to be pure is the predicate within a subtype expression (section 9)

{| binding : type_expr • expr |}

That is, *expr* must be pure. Examples of expression occurrences that are required to be read-only are the sub-expressions of a comprehended set expression (section 6)

{expr$_1$ | typing$_1$,...,typing$_n$ • expr$_2$}

That is, *expr*$_1$ and *expr*$_2$ must be read-only.

15.2 Independent Expressions

Recall from section 14 that the two sub-expressions of an equality expression

expr$_1$ = expr$_2$

must be independent: if one sub-expression writes to a variable, the other must not access that variable. The two expressions can thus be evaluated in any order without changing the meaning. In general, the sub-expressions of an infix expression of the form expr$_1$ op expr$_2$

are required to be independent, where 'op' is one of the operators

$= \neq$
$+ - / * \backslash \uparrow$
$> < \geq \leq$
$\in \notin$
$\cup \cap$
$\subset \supset \subseteq \supseteq$
$\widehat{\ }$
$\dagger$
$\circ$

There are two other places where sub-expressions are required to be independent. The sub-expressions of a product expression (section 4)

(expr$_1$,...,expr$_n$)

are required to be independent.

The sub-expressions of an application expression (section 5)

expr(expr$_1$,...,expr$_n$)

are required to be independent.

15.3 If Expressions

Recall that an if-expression has been described as having the form

if expr$_1$ **then** expr$_2$ **else** expr$_3$ **end**

It contains thus both a then-branch and an else-branch. In state-based specifications, a form without else-branch is often useful

if expr$_1$ **then** expr$_2$ **end**

This is short for

if expr_1 **then** expr_2 **else skip end**

where **skip** is a predefined side-effect free expression of type **Unit**. In fact

skip $\equiv$ ()

The reason for introducing **skip** when '()' is available is for reasons of readability.

Note that since both branches of an if-expression must have the same type, the type of expr_2 must also be **Unit**.

As an example illustrating an if-expression without else-branch, consider an operation for decreasing a counter. The counter is only decreased if it is greater than zero

```
variable
  counter : Nat
value
  decrease : Unit → write counter Unit
axiom
  decrease() ≡
    if counter > 0 then counter := counter − 1 end
```

16 Repetitive Expressions

A repetitive expression specifies that a certain expression shall be repeatedly evaluated for the purpose of its side-effect. There are three forms, all known from most traditional programming languages: 'while' expressions, 'until' expressions and 'for' expressions.

The three kinds of repetitive expressions all have result-type **Unit** since they are only evaluated for the purpose of their side-effects.

16.1 While Expressions

A while expression evaluates an expression as long as some predicate is satisfied. A while expression has the form

while expr_1 **do** expr_2 **end**

The expression $\textit{expr}_1$ is the controlling expression which must be of type **Bool**. The expression $\textit{expr}_2$ is the expression to be repeatedly evaluated for the purpose of its side-effect, and must be of type **Unit**.

For each iteration, $\textit{expr}_1$ is evaluated. If it evaluates to **true**, $\textit{expr}_2$ is evaluated, and a new iteration is begun. If $\textit{expr}_1$ on the other hand evaluates to **false**, the while expression terminates.

Note that due to the introduction of repetetive expressions, it becomes relevant to talk about termination and non-termination.

A while expression of the above form is equivalent to

```
if expr₁ then
   expr₂ ; while expr₁ do expr₂ end
else
   skip
end
```

Example 16.1

Consider an operation, *fraction_sum*, for calculating the number

1 + 1/2 + ... + 1/n

for some natural number n. The operation delivers the result in the variable *result*. An auxiliary variable, *counter*, is used to control the calculation.

```
FRACTION_SUM =
  class
    variable
      counter : Nat,
      result : Real
    value
      fraction_sum : Nat ~→ write counter, result Unit
    axiom
      fraction_sum(n) ≡
        counter := n;
        result := 0.0;
        while counter > 0 do
          result := result + 1.0/(rl counter);
          counter := counter − 1
        end
        pre n > 0
  end
```

Note that the *counter* variable must be converted to a real number before a real number fraction can be calculated.

□

16.2 Until Expressions

An until expression evaluates an expression until some predicate is satisfied. An until expression has the form

do expr_1 **until** expr_2 **end**

The expression $expr_2$ is the controlling expression which must be of type **Bool**. The expression $expr_1$ is the expression to be repeatedly evaluated for the purpose of its side-effect, and must be of type **Unit**. It is evaluated repeatedly until $expr_2$ evaluates to **true**, and is thus evaluated at least once.

An until expression of the above form is equivalent to

expr_1 ; **while** $\sim\text{expr}_2$ **do** expr_1 **end**

Example 16.2

Consider a reformulation of the *fraction_sum* operation in terms of an until expression.

```
FRACTION_SUM =
  class
    variable
      counter : Nat,
      result : Real
    value
      fraction_sum : Nat ~→ write counter, result Unit
    axiom
      fraction_sum(n) ≡
        counter := n;
        result := 0.0;
        do
          result := result + 1.0/(rl counter);
          counter := counter − 1
        until counter = 0
        pre n > 0
  end
```

□

16.3 For Expressions

A for expression "runs through a list" and evaluates an expression for each list member. A for expression in the simplest case has the form

for binding **in** expr_1 **do** expr_2 **end**

The expression $expr_1$ must be of a list type, T^*, for some type T. The expression $expr_2$ is the one to be repeatedly evaluated and must have type **Unit**.

The for expression is evaluated as follows

1. $expr_1$ is evaluated to yield a (possibly empty) list

   $\langle \mathrm{e}_1,\ldots,\mathrm{e}_n \rangle$

2. for each value, e_i, in the list, processed from left to right, $expr_2$ is evaluated in the scope of the bindings obtained by matching e_i against the *binding*.

Example 16.3

Consider a reformulation of the *fraction_sum* operation in terms of a for expression. Since the for expression itself scans all the numbers from 1 to n, there is no need for an auxiliary *counter* variable.

```
FRACTION_SUM =
  class
    variable
      result : Real
    value
      fraction_sum : Nat ~→ write result Unit
    axiom forall n : Nat •
      fraction_sum(n) ≡
        result := 0.0;
        for i in ⟨1 .. n⟩ do
          result := result + 1.0/(rl i)
        end
        pre n > 0
  end
```

□

In an extended form of the for expression, one can state a predicate, $expr_p$ of type **Bool**, that specifies which elements from the list

$\langle e_1, ..., e_n \rangle$

returned by $expr_1$ that shall lead to an evaluation of $expr_2$. The extended version has the form

for binding **in** expr_1 • expr_p **do** expr_2 **end**

An element e_i from the list returned by $expr_1$ only leads to an evaluation of $expr_2$ if the predicate $expr_p$ deterministically evaluates to **true** (in the scope of the bindings obtained by matching e_i against the *binding*).

The expressions $expr_1$ and $expr_p$ must be read-only.

Example 16.4

Consider the specification of a database being a list of records, each consisting of a key and some data.

```
DATABASE =
  class
    type
      Key, Data,
      Record = Key × Data,
      Database = Record*,
    variable
      database : Database
  end
```

The database is stored in a variable.

Suppose we want to generate reports based on the database. A report should only involve those records that are "interesting" as defined by some boolean-valued function, *is_interesting*, on keys. For each interesting record, the report will contain an entry consisting of the key and a *transformation* of the corresponding data element.

An operation, *make_report*, is defined that reads the *database* and delivers a report in the variable *report*.

```
REPORT =
  extend DATABASE with
    type
      Report_Data,
      Report_Record = Key × Report_Data,
      Report = Report_Record*
    variable
      report : Report
    value
      is_interesting : Key → Bool,
      transformation : Data → Report_Data,
      make_report : Unit → read database write report Unit
    axiom
      make_report() ≡
        report := ⟨⟩;
        for (key,data) in database • is_interesting(key) do
          report := report ⌢ ⟨(key,transformation(data))⟩
        end
  end
```

□

17 Algebraic Definition of Operations

In section 5 it was described how applicative functions can be defined abstractly in terms of algebraic equivalences. Recall in particular the algebraic specification of the *LIST*-module from example 5.3, which is repeated below. Constants and functions have been subscripted with an *a* to indicate that they are applicative.

```
LIST =
  class
    type
      List
    value
      $\text{empty}_a$ : List,
      $\text{add}_a$ : Int × List → List,
      $\text{head}_a$ : List $\overset{\sim}{\to}$ Int,
      $\text{tail}_a$ : List $\overset{\sim}{\to}$ List
    axiom
      forall i : Int, l : List •
      [head_add]
        $\text{head}_a(\text{add}_a(\text{i,l})) \equiv \text{i}$,
      [tail_add]
        $\text{tail}_a(\text{add}_a(\text{i,l})) \equiv \text{l}$
  end
```

The important point to note here is that nothing has been said about how lists are represented. The type *List* is a sort and the functions are defined without assuming any particular representation of lists.

The question now arises whether a state-based specification of lists can be given, that ignores representation details in a similar way. There are at least three ways of doing this and we shall treat each of them below.

17.1 Extending an Applicative Module

The first approach is to use the entities from the applicative *LIST*-module in defining the state-based module. The state-based module thus becomes an extension of the *LIST*-module.

Example 17.1

```
STATE_BASED_LIST =
**extend** LIST **with**
   **variable**
      list : List
   **value**
      empty : **Unit** → **write** list **Unit**,
      is_empty : **Unit** → **read** list **Bool**,
      add : **Int** → **write** list **Unit**,
      head : **Unit** $\overset{\sim}{\to}$ **read** list **Int**,
      tail : **Unit** $\overset{\sim}{\to}$ **write** list **Unit**
   **axiom**
      empty() ≡
         list := empty$_a$,
      is_empty() ≡
         list = empty$_a$,
      add(i) ≡
         list := add$_a$(i,list),
      head() ≡
         head$_a$(list)
         **pre** ∼is_empty(),
      tail() ≡
         list := tail$_a$(list)
         **pre** ∼is_empty()
**end**
```

□

A variable of type *List* is defined. This type comes from the *LIST*-module and is a sort. Nothing has thus been said about representation of its values.

The operations working on the *list* variable are defined by simple calls of the corresponding applicative functions. Since these are defined without assuming any particular representation, the operations share that property.

The operation *is_empty* has been introduced in order to make it possible to test whether the list is empty. In the applicative case, we could just compare a list *l* with $empty_a$ as follows

$$\mathrm{l} = \mathrm{empty}_a$$

if we wanted to test whether *l* was empty. In the state-based case, *empty* has been turned into an operation that resets the variable to contain the empty list. If we want all accesses to the variable to be done through operation calls (a reasonable requirement), we must introduce *is_empty*.

The approach of using an applicative specification in defining a state-based one may seem tedious, especially if the applicative one does not exist already.

17.2 Algebraic Equivalences

The second approach to abstractly specifying the state-based list module is to give algebraic equivalences between operation calls in a way very similar to the equivalences in the applicative *LIST*-module.

As an example, consider the applicative axiom *head_add* from *LIST*

```
axiom
  forall i : Int, l : List •
  [head_add]
    head_a(add_a(i,l)) ≡ i
```

The axiom says that adding an element i to a list and then taking the head yields the element just added. The corresponding “state-based” axiom is

```
axiom
  forall i : Int •
  [head_add]
    add(i) ; head() ≡ add(i) ; i
```

The extra occurrence of $add(i)$ on the right-hand side of the equivalence is necessary in order to make the equivalence **true**. Recall that in order for an equivalence to be **true**, the left-hand side and the right-hand side must have exactly the same side-effects.

The complete state-based specification of lists is as follows.

Example 17.2

```
LIST =
  class
    type
      List
    variable
      list : List
    value
```

```
      empty : **Unit** → **write** list **Unit**,
      is_empty : **Unit** → **read** list **Bool**,
      add : **Int** → **write** list **Unit**,
      head : **Unit** $\xrightarrow{\sim}$ **read** list **Int**,
      tail : **Unit** $\xrightarrow{\sim}$ **write** list **Unit**
    **axiom**
      **forall** i : **Int** •
      [is_empty_empty]
        empty() ; is_empty() ≡ empty() ; **true**,
      [is_empty_add]
        add(i) ; is_empty() ≡ add(i) ; **false**,
      [head_add]
        add(i) ; head() ≡ add(i) ; i,
      [tail_add]
        add(i) ; tail() ≡ **skip**
  **end**
```

□

The variable *list* is defined to have type *List* which is a sort. Nothing has thus been said about representation. The operations are defined without assuming any particular representation of lists.

Note in particular the *tail_add* axiom. It says that adding an element ($add(i)$) followed by removing the head ($tail()$) is equivalent to doing nothing (**skip**).

17.3 Being Implicit about Variables

An interesting observation concerning example 17.2 is that the variable *list* is not referred to in the axioms. It is only mentioned in the operation types where its role is to state what variables are accessed from the operations and how they are accessed.

It thus appears that we have said as little as possible about the variable: it is not mentioned in the axioms and its type is a sort. There is, however, a possibility of saying even less than that. In the third approach we shall be totally implicit about what the variables are, by simply not defining any. We can thus modify example 17.2 by removing the following definitions, observing that the type *List* is only used to give a type to the variable

```
  **type**
    List
  **variable**
    list : List
```

The operation types must now be modified such that they do not mention the variable *list*. As an example, the type of the operation *is_empty* was defined as follows

```
is_empty : Unit → read list Bool,
```

That is, its type contains the access description

```
read list
```

Instead of *list* one can write **any** in the access description to indicate that "any" variable defined my be read from. An access description can thus have the form

```
read any
```

The definition of the type of *is_empty* becomes

```
is_empty : Unit → read any Bool
```

A write-access description can similarily have the form

```
write any
```

indicating that the operation may write to any variable. Note that since a variable being written to is also regarded as being read from, only one of these two "any" forms should occur in a single operation type.

After having performed these changes, and leaving the axioms unchanged, the state-based list module becomes as follows.

Example 17.3

```
LIST =
  class
    value
      empty : Unit → write any Unit,
      is_empty : Unit → read any Bool,
      add : Int → write any Unit,
      head : Unit ~→ read any Int,
      tail : Unit ~→ write any Unit
    axiom
      forall i : Int •
      [is_empty_empty]
        empty() ; is_empty() ≡ empty() ; true,
      [is_empty_add]
        add(i) ; is_empty() ≡ add(i) ; false,
      [head_add]
        add(i) ; head() ≡ add(i) ; i,
      [tail_add]
        add(i) ; tail() ≡ skip
  end
```

□

The following has been gained by being implicit about variables

- We have avoided deciding what the variables shall be and what their types shall be.
- Suppose we later develop an implementation of the *LIST*-module from example 17.3. Our specification then places no restriction on what the variables of an implementation shall be.
- The specification places no restrictions on what variables the operations are allowed to access.

Note that **any**-accesses can also be used in operation types even if variables have been defined in the context. It then allows the operations to access any of the defined variables. Again, one can see this as giving freedom to an implementation.

A natural question is when to be implicit about variables and when to be explicit. It is difficult to give exact rules. Very roughly, one may be implicit in the following situations.

- One will not be bothered with what the variables are.
- One wants to leave freedom to a later development that is expected to be an implementation in the formal sense.

Being explicit, however, has its benefits.

- One can from the type of an operation see exactly what variables may be accessed and how they may be accessed. This makes state-based specifications a lot easier to read and prove properties about.
- One may simply prefer the explicit style since it perhaps more clearly expresses "what goes on".

A more detailed description of **any**-accesses will be given in part four on modules.

17.4 Example

Example 17.4

Consider an algebraic specification of the state-based database from example 14.8. We will be implicit about variables by not defining any. As a consequence, all access descriptions will use **any**.

```
DATABASE =
  class
    type
      Key, Data
    value
      empty : Unit → write any Unit,
      insert : Key × Data → write any Unit,
      remove : Key → write any Unit,
      defined : Key → read any Bool,
      lookup : Key ~→ read any Data
    axiom
      forall k,k1 : Key, d : Data •
      [remove_empty]
        empty() ; remove(k) ≡ empty(),
      [remove_insert]
        insert(k1,d) ; remove(k) ≡
          if k = k1 then
            remove(k)
          else
            remove(k) ; insert(k1,d)
          end,
      [defined_empty]
        empty() ; defined(k) ≡ empty() ; false,
      [defined_insert]
        insert(k1,d) ; defined(k) ≡
          if k = k1 then
```

```
            insert(k1,d) ; **true**
          **else**
            **let** result = defined(k) **in** insert(k1,d) ; result **end**
          **end**,
      [lookup_insert]
        insert(k1,d) ; lookup(k) ≡
          **if** k = k1 **then**
            insert(k1,d) ; d
          **else**
            **let** result = lookup(k) **in** insert(k1,d) ; result **end**
          **end**
  **end**
```

The reader should compare this specification with the algebraic specification of the corresponding applicative module from example 5.4.

The state-based database example illustrates the "constructor" technique for inventing axioms, which we also saw in example 5.4. The technique used in the state-based case can be characterised as follows

1. Identify the "constructor operations" by which any database can be constructed. These are the operations *empty* and *insert*. Any database can thus be generated as the side-effect of an expression of the form

   empty() ; insert(k_1,d_1) ; ... ; insert(k_n,d_n)

2. Define the remaining operations "by case" over the constructor operations called with identifiers as parameters. In the above axioms, *remove*, *defined* and *lookup* are thus defined over the two constructor-expressions

   empty()

   insert(k1,d)

   We thus get "for free" all the left-hand sides of the axioms we must write. That is

   empty() ; remove(k)
   insert(k1,d) ; remove(k)

   empty() ; defined(k)
   insert(k1,d) ; defined(k)

   empty() ; lookup(k)
   insert(k1,d) ; lookup(k)

> Note, however, that due the the partiality of *lookup* we don't bother with giving the right-hand side corresponding to $empty()\ ;\ lookup(k)$.

The list-axioms (example 17.2 and example 17.3) actually have the same form.

The technique is useful in many applications, but there are of course applications where one must be more imaginative when writing axioms.

The right-hand sides of the axioms *defined_insert* and *defined_lookup* are somewhat different from the corresponding applicative ones. This is due to the requirement that the side-effect of the left-hand side of an equivalence must be the same as the side-effect of the right-hand side.

More specifically, the call $insert(k1, d)$ must occur on the right-hand side since it occurs on the left-hand side and since it has side-effects.

Note also the use of let expressions in the two axioms. These are necessary in order to get $defined(k)$, respectively $lookup(k)$, evaluated before $insert(k1, d)$.

□

18 Post Expressions

We have just seen how operations can be defined in a very abstract way in terms of algebraic equivalences. Another way of being abstract about operations is to use post expressions. We have in fact already seen several examples of this style in the applicative case. See for instance example 5.2.

Consider the following specification of a *choose* operation that returns an arbitrary element from a set that is contained in a variable. The returned element is at the same time removed from the set, thus changing the contents of the variable.

Example 18.1

```
CHOOSE =
  class
    variable
      set : Int-set
    value
      choose : Unit $\overset{\sim}{\to}$ write set Int
    axiom
      choose() as i
        post i $\in$ set` $\wedge$ set = set`$\backslash$$\{$i$\}$
        pre set $\neq$ $\{\}$
  end
```

□

The pre-condition says that the operation is only specified for states where the contents of *set* is a non-empty set.

The post-condition is a conjunction of two boolean expressions. The first one

```
  i $\in$ set`
```

says that the returned *i* must be a member of *set* as this was before the call. In general, a hooked variable in a post-condition refers to the contents of that variable before calling the operation. Conversely, a normal non-hooked variable refers to the contents of the variable after having called the operation. Such a non-hooked variable occurs in the second part of the post-condition set = set` \ {i}

This says that the new *set* after a call must be equal to the *set* before the call, except for the chosen element which has been removed.

Let us examine the meaning of a post expression in more detail. The general form of a post expression without a pre-condition is

expr_1 **as** binding **post** expr_2

with the result naming '**as** *binding*' being optional.

The post-condition $expr_2$ must be of type **Bool**, which is also the type of the post expression itself.

The post expression is evaluated in the current state as follows. The expression $expr_1$ is evaluated in the current state, the pre-state, thus yielding a result, named by the *binding*, and a possibly changed state, the post-state. The value of the post expression is then **true** if and only if

1. $expr_1$ is defined and deterministic,
2. $expr_2$ ≡ **true** when evaluated in the post-state and in the scope of the *binding*. Hooked variables of the form *id*` though refer to the pre-state.

The post-condition $expr_2$ must be read-only. Concerning the post expression itself, the side-effect obtained by evaluating $expr_1$ is only used for evaluating the post-condition and is ignored thereafter. The post expression is thus read-only. A post expression is always defined and deterministic (both $expr_1$ and $expr_2$ are required to be deterministic).

Condition 1 above says that $expr_1$ must be defined and deterministic. In the above *CHOOSE*-module, $expr_1$ corresponds to *choose*(). The *choose* operation thus is defined where the pre-condition holds and, moreover, it is deterministic. That is, two applications of *choose* in the same state yield the same result state and result value.

A post expression may include a pre-condition, which is a read-only expression of type **Bool**

expr_1 **as** binding **post** expr_2 **pre** expr_3

This is short for

```
(expr$_3$ ≡ **true**) ⇒
   expr$_1$ **as** binding **post** expr$_2$
```

As said before, a post expression is evaluated in the current state. Recall, however, that when it occurs as an axiom, it is implicitly preceeded by the always-combinator '□' implying a universal quantification over all states.

Example 18.2

Consider the specification of an *insert* operation that inserts an integer into a list contained in a variable. The contents of the variable after insertion must be a sorted list without duplicates. Think of the variable as containing an effecient representation of a set.

```
INSERT_SORTED =
   class
      variable
         list : Int* := ⟨⟩
      value
         is_sorted : Unit → read list Bool,
         insert : Int → write list Unit
      axiom forall i : Int •
         is_sorted() ≡
            ∀ idx1,idx2 : Nat •
               ({idx1,idx2} ⊆ inds list ∧ idx1 < idx2) ⇒
                  list(idx1) < list(idx2),
         insert(i)
            post
               elems list = elems list` ∪ {i}
                  ∧
               is_sorted()
   end
```

The operation *is_sorted* examines the list contained in the variable and yields **true** if the list is sorted in increasing order.

The post-condition for the operation *insert* consists of two parts. The first part says that the elements of the new list must be those of the old list with the addition of the new element. The example thus illustrates how the parameters of an operation are referred to in the post-condition.

The second part of the post-condition says that the new list must be sorted. Note that one can generally call read-only operations in post-conditions. Such operation calls will be evaluated in the post-state.

Note finally that the post expression contains no result naming or pre-condition. The result naming is omitted since the result type is **Unit**. One is of course allowed to write a result naming, but in the **Unit**-case this makes little sense.

□

Part III

Concurrency-based Specifications

19 Some Basic Concepts

RSL provides means for specifying concurrent systems. More precisely, operators are provided for specifying the parallel evaluation of expressions. Moreover, communication primitives are provided such that parallel evaluating expressions can communicate with each other through "channels".

Concurrency becomes relevant in basically two situations. The first situation is where the system to be modelled is inherently concurrent. An example is a system where a number of airport check-in counters have access to the same passenger-flight database. This kind of concurrency could be called "conceptual concurrency".

The second situation is where an inherently sequential system due to efficiency reasons is made concurrent. An example is some number-calculation function which is specified to perform some of its calculations in parallel to save time. This kind of concurrency could be called "efficiency concurrency".

One can possibly from some philosophical viewpoint discuss this differentiation, but from a pragmatic viewpoint it appears useful.

The following module defines a 'one place buffer', *opb*, that communicates with the surrounding world through the two channels *add* and *get*. Values of type *Elem* are input from the *add* channel and are then output to the *get* channel.

Example 19.1

```
ONE_PLACE_BUFFER =
  class
    type
      Elem
    channel
      add : Elem,
      get : Elem
    value
      opb : Unit → in add out get Unit
    axiom
      opb() ≡
        let v = add? in get!v end ; opb()
  end
```

□

We shall in the following explain the individual declarations of the module.

19.1 Channel Declarations

A channel declaration has the form

```
channel
   channel_definition₁,
   ⋮
   channel_definitionₙ
```

In our specification there are two such definitions.

A channel definition has the form

```
id : type_expr
```

That is, the channel *id* is defined to “transport” values of the type represented by *type_expr*.

The channels *add* and *get* in the example are both defined to have the type *Elem*.

When several channels have the same type, a multiple channel definition of the following form can be used

```
id₁,...,idₙ : type_expr
```

which is short for

```
id₁ : type_expr,
⋮
idₙ : type_expr
```

19.2 Functions with Channel Access

The function *opb* from the example has the type

Unit → **in** add **out** get **Unit**

That is, it is a function that, when called, communicates with the surroundings through the channels *add* and *get*. More specifically, it receives values from the surroundings through the *add* channel and it sends values to the surroundings through the *get* channel. A function with channel access, like *opb*, is also called a process.

The function will only be called for the purpose of its ability to communicate through *add* and *get*, and therefore its parameter type and result type are **Unit**. We shall later see examples of more interesting parameter and result types.

The process *opb* can be pictured as follows

$$\text{add} \rightarrow \boxed{\text{opb}} \rightarrow \text{get}$$

In general, a type expression for total processes has the form

$$\text{type\_expr}_1 \rightarrow \text{access\_desc}_1 \ \ldots \ \text{access\_desc}_n \ \text{type\_expr}_2$$

A function of this type takes arguments from the type represented by *type_expr*$_1$ and returns results within the type represented by *type_expr*$_2$. In addition, the function accesses the channels mentioned in the access descriptors.

Each of the access descriptors *access_desc*$_i$ can be of the form

$$\textbf{in} \ \text{id}_1,\ldots,\text{id}_n$$

expressing which channels processes of the type may be input from, or it can be of the form

$$\textbf{out} \ \text{id}_1,\ldots,\text{id}_n$$

expressing which channels may be output to. In addition, since processes can also access variables, access descriptors can describe access to variables as explained in section 14 and in section 17.

A type expression for partial processes has the form

$$\text{type\_expr}_1 \overset{\sim}{\rightarrow} \text{access\_desc}_1 \ \ldots \ \text{access\_desc}_n \ \text{type\_expr}_2$$

19.3 Communication Expressions

RSL provides two communication primitives: one for inputting a value from a channel and one for outputting a value to a channel. An expression may input a value from a channel by an input expression of the form

```
id?
```

where *id* is the channel input from. Upon input from *id*, the received value is returned as result of the input expression. That is, the type of the input expression is the type T, where the channel has been defined to have type T:

```
channel
  id : T
```

An expression may output a value to a channel by an output expression of the form

```
id!expr
```

where *id* is the channel output to. The expression, *expr*, is evaluated to return a value, which is then output to the channel *id*. The type of the expression must be the same as the type of the channel. The type of the output expression itself is the type **Unit**.

Our example contains an input expression

```
add?
```

as well as an output expression

```
get!v
```

So the process *opb* repeatedly inputs a value from the *add* channel and then outputs the same value to the *get* channel. Note how the value input from the *add* channel is temporarily named in a let expression. The process calls itself recursively to obtain the repetition.

Note that input and output are just expressions. As stated earlier in connection with assignment: "there are only expressions". No special syntax category is thus introduced for expressing communication, just like no special syntax category was introduced for expressing assignment.

19.4 Putting Expressions in Parallel

Communication through channels is the means by which parallel evaluating expressions interact. Two expressions are put in parallel as follows

$\mathrm{expr}_1 \parallel \mathrm{expr}_2$

The two expressions expr_1 and expr_2 must both have type **Unit**, which is also the type of the composite expression itself.

As an example consider the following definitions

```
channel
  c : Int
variable
  x : Int
```

In the scope of these definitions, the two expressions $x := c?$ and $c!5$ can be put in parallel as follows

```
x:=c? ∥ c!5
```

The evaluation of the composite expression may lead to an interaction between the two expressions in that the rightmost expression outputs the value 5 on the channel c, which is then input from by the leftmost expression. If the communication takes place, the effect of the above parallel expression will be the following

```
x:=5
```

Communication is synchronized: the outputting expression only outputs to the channel if the inputting expression simultaniously inputs from the channel.

Parallel attempts to input from a channel and to output to the channel does, however, not necessarily lead to a communication. Whether it does, depends on an internal choice. The two expressions can thus communicate with a third expression which is put in parallel with the two. One can for example put the expression $c!7$ in parallel with the two expressions as follows

```
(x:=c? ∥ c!5) ∥ c!7
```

and then as one possible effect obtain

```
x:=7 ; c!5
```

That is, the rightmost expression outputs the value 7 to the channel *c*. The leftmost expression inputs the value and stores it in *x*. After the communication, the communication *c*!5 still remains to be performed.

Note, however, that the effect may also be

```
x:=5 ; c!7
```

or the effect may even be that no communication takes place at all.

The parallel operator is commutative as well as associative. That is

$$\begin{aligned}&\mathrm{expr}_1 \parallel \mathrm{expr}_2 \equiv\\&\quad \mathrm{expr}_2 \parallel \mathrm{expr}_1\end{aligned}$$

$$\begin{aligned}&\mathrm{expr}_1 \parallel (\mathrm{expr}_2 \parallel \mathrm{expr}_3) \equiv\\&\quad (\mathrm{expr}_1 \parallel \mathrm{expr}_2) \parallel \mathrm{expr}_3\end{aligned}$$

Two expressions running in parallel should be state-independent: if the one expression writes to a variable, the other should not access that variable (neither read from it or write to it). The RSL type checker does not force state-independency, but it is highly recommended.

As an example, suppose we want to use the one place buffer as a connection between two processes called *reader* and *writer*. The following figure illustrates the parallel processes and the channels that connect them.

$$\begin{tikzcd}\text{input} \arrow[r] & \boxed{\text{reader}} \arrow[r] & \text{add} \arrow[r] & \boxed{\text{opb}} \arrow[r] & \text{get} \arrow[r] & \boxed{\text{writer}} \arrow[r] & \text{output}\end{tikzcd}$$

The *reader* process inputs values from the *input* channel and the *writer* process outputs values to the *output* channel. Values moves from the *reader* process to the *writer* process via the one place buffer *opb*.

The *reader* and *writer* processes can be specified as follows.

Example 19.2

```
READER_WRITER =
  extend ONE_PLACE_BUFFER with
    type
      Input,
      Output
    channel
      input : Input,
      output : Output
    value
      transform1 : Input → Elem,
      transform2 : Elem → Output
      reader : Unit → in input out add Unit,
      writer : Unit → in get out output Unit,
    axiom
      reader() ≡
        let v = input? in add!(transform1(v)) end ; reader(),
      writer() ≡
        let v = get? in output!(transform2(v)) end ; writer(),
  end
```

□

The abstract types *Input* and *Output* are the types of the *input* channel respectively the *output* channel. We are abstract about the types since we want to illustrate the parallelism and not the particular kinds of values communicated.

The *reader* process repeatedly inputs a value *v* from the *input* channel and outputs the value *transform*1(*v*) to the *add* channel. The *writer* process repeatedly inputs a value *v* from the *get* channel and outputs the value *transform*2(*v*) to the *output* channel. The functions *transform*1 and *transform*2 are un-specified.

We can now put the processes *reader*, *opb* and *writer* together in parallel, calling the composed process for *system*.

Example 19.3

```
SYSTEM =
  extend READER_WRITER with
    value
      system : Unit → in input,add,get out output,add,get Unit
    axiom
      system() ≡
```

```
            reader() ‖ opb() ‖ writer()
    end
```

□

The type of the process *system* states that the process has **in** access as well as **out** access to the channels *add* and *get*. That is, the *system* process may unfortunately input from and output to both these channels as well as input from *input* and output to *output*. To better illustate this, we can unfold the calls of *reader*(), *opb*() and *writer*() in the axiom defining *system*

```
    axiom
       system() ≡
          let v = input? in add!(transform1(v)) end ; reader()
          ‖
          let v = add? in get!v end ; opb()
          ‖
          let v = get? in output!(transform2(v)) end ; writer()
```

We see that the *system* process is ready to input from any of the three channels *input*, *add* and *get*. Suppose for example that *system* is put in parallel as follows

```
    system() ‖ add!e
```

The effect of this expression may be

```
    let v = input? in add!(transform1(v)) end ; reader()
    ‖
    get!e ; opb()
    ‖
    let v = get? in output!(transform2(v)) end ; writer()
```

That is, the value *e* has been communicated over the *add* channel and the resulting expression is ready to either output *e* to the *get* channel or input from either of the channels *input* and *get*.

The expression may though perform an “internal” communication by communicating the value *e* over the *get* channel. In that case, the effect of the expression becomes

```
let v = input? in add!(transform1(v)) end ; reader()
‖
opb()
‖
output!(transform2(e)) ; writer()
```

19.5 Hiding Channels

We have just seen how the channels *add* and *get* are part of the interface of the *system* process. This is unfortunate since these channels together with the one place buffer should really be "internal stuff". The following figure illustrates how we really would like to regard the *system* process from the outside

$\text{input} \rightarrow \boxed{\text{system}} \rightarrow \text{output}$

That is, we want to hide the channels *add* and *get*. The only way channel hiding can be done in RSL is in terms of a local expression. Consider for example the following expression in the scope of the integer variable x

```
local
  channel
    c : Int
in
  x:=c? ‖ c!5
end
```

The scope of the definition of channel c is the expression

```
x:=c? ‖ c!5
```

The channel c is thus hidden outside the local expression. The effect of leaving the scope (moving beyound the **end** in the local expression) is that all internal communication via local channels is forced through. In the above expression, the communication of the value 5 over the channel c is forced through such that the effect becomes

```
x:=5
```

In fact, the following equivalence holds

```
  local
    channel
      c : Int
  in
    x:=c? ‖ c!5
  end
  ≡
  x:=5
```

We can now specify our *system* process such that the channels *add* and *get* are hidden. What we must do is to define all the processes to be put in parallel and their internal channels in a local expression. We thus get the following module.

Example 19.4

```
SYSTEM =
  class
    type
      Input,
      Output
    channel
      input : Input,
      output : Output
    value
      system : Unit → in input out output Unit
    axiom
      system() ≡
        local
          type
            Elem
          channel
            add : Elem,
            get : Elem
          value
            opb : Unit → in add out get Unit
          axiom
            opb() ≡
              let v = add? in get!v end ; opb()
          value
            transform1 : Input → Elem,
            transform2 : Elem → Output,
            reader : Unit → in input out add Unit,
            writer : Unit → in get out output Unit,
          axiom
```

```
            reader() ≡
              let v = input? in add!(transform1(v)) end ; reader(),
            writer() ≡
              let v = get? in output!(transform2(v)) end ; writer(),
        in
          reader() ∥ opb() ∥ writer()
        end
  end
```

□

The types *Input* and *Output* and the channels *input* and *output* are still defined at the outermost level since all these items are part of the interface of the *system* process. The rest is locally defined since it is internal stuff.

It may seem tedious to be forced to define all sub-processes of a process within a local expression. Especially when a system consists of many sub-processes and these perhaps again are composed. Part four of this document describes how the module concept can be used in combination with the local expression to model a hierarchy of processes.

19.6 External Choice

Reconsider the axiom defining the one place buffer

```
  axiom
    opb() ≡
      let v = add? in get!v end ; opb()
```

An application, *opb*(), of the buffer process offers a single kind of communication to the surroundings: an input from the *add* channel. After an input, still a single kind of communication is offered: an output to the *get* channel.

There are, however, situations where we want a process to offer several different kinds of communications at the same time. The *system* process from example 19.3 did in fact, though unintended, offer several communications since the "internal" channels *add* and *get* were not hidden. The call *system*() thus offered to input from any of the channels *input*, *add* and *get*.

The external choice combinator '⫿' serves to explicitely specify a choice between different kinds of communications. As an example, assume the following definitions

```
  channel
```

```
    c,d : Int
variable
    x : Int
```

Then consider the external choice expression

```
x:=c? ⌷ d!5
```

This expression offers two communications: either an input from the *c* channel or an output to the *d* channel. The choice is called external since it will be up the surroundings to choose between the two. Suppose we put this expression in parallel with the expression *c*!1 as follows

```
(x:=c? ⌷ d!5) ∥ c!1
```

A possible effect of this expression is that the value 1 is communicated over the channel *c* thus resulting in

```
x:=1
```

Recall, however, that parallel composition does only force communication to happen when channels are hidden in a local expression.

The external choice combinator in general puts expressions together as follows

$\text{expr}_1 \ \square\ \text{expr}_2$

The two expressions must have the same type. Typically, $expr_1$ and $expr_2$ each begins with some kind of communication. Only one of the expressions will be evaluated, depending on which kind of communication the surroundings want to do. The external choice combinator is commutative and associative.

As an example illustrating the use of external choice, consider a specification of a many place buffer capable of holding several elements at one time. There is no limit on the size of the buffer, except that it at any time can contain only finitely many elements.

The many place buffer process, *mpb*, holds all buffered elements in a list. The list is a parameter to *mpb* in the sense that any recursive call of *mpb* takes a possible modified list as actual parameter.

Example 19.5

```
MANY_PLACE_BUFFER =
  class
    type
      Elem,
      Buffer = Elem*
    channel
      empty : Unit,
      add : Elem,
      get : Elem
    value
      mpb : Buffer → in empty,add out get Unit
    axiom forall b : Buffer •
      mpb(b) ≡
        empty? ; mpb(⟨⟩)
        ⌷
        let v = add? in mpb(b ^ ⟨v⟩) end
        ⌷
        if b ≠ ⟨⟩ then
          get!(hd b) ; mpb(tl b)
        else
          stop
        end
  end
```

□

The buffer is connected with the surroundings by three channels. Values are added to the buffer via the *add* channel and leave the buffer again via the *get* channel. The *empty* channel makes it possible to empty the buffer. This is done by sending a signal (the unit value '()' of type **Unit**) on the *empty* channel.

The axiom defining *mpb* reads as follows. Assuming the buffer *b*, three kinds of communications may be offered:

- A value (the unit value) may be input from the *empty* channel. Upon input, the buffer process continues with the empty list as parameter, representing the empty buffer.
- A value, *v*, may be input from the *add* channel. Upon input, the buffer process continues with an extended list as parameter.

- If the list *b* is non-empty, the process may output the head of the list to the *get* channel and then continue with the tail of the list as parameter.

  The else-branch of the if expression is entered if the list *b* is empty. That is, the else-branch is entered if the buffer contains no elements to be output to the *get* channel. The predefined expression **stop** represents the "do nothing" effect. **stop** has the property that for any expression *expr*, the following equivalence holds

  ```
  expr ⌷ stop ≡ expr
  ```

  From this property we can deduce the following

  ```
  mpb(⟨⟩)

  ≡

  empty? ; mpb(⟨⟩)
  ⌷
  let v = add? in mpb(⟨⟩ ⁀ ⟨v⟩) end
  ⌷
  if ⟨⟩ ≠ ⟨⟩ then
     get!(hd ⟨⟩) ; mpb(tl ⟨⟩)
  else
     stop
  end

  ≡

  empty? ; mpb(⟨⟩)
  ⌷
  let v = add? in mpb(⟨v⟩) end
  ⌷
  stop

  ≡

  empty? ; mpb(⟨⟩)
  ⌷
  let v = add? in mpb(⟨v⟩) end
  ```

The many place buffer is put in parallel with an expression *expr* as follows, assuming the buffer to be initially empty

```
mpb(⟨⟩) ∥ expr
```

19.7 Internal Choice

The external choice combinator expresses a choice between two expressions. The term 'external' says that it is the surroundings that decide which expression to be selected. As an example, consider the expression

(x:=c? ▯ d!5) ∥ c!1

If a communication takes place, it will be the communication of the value 1 over the channel c, thus resulting in

x:=1

In addition to the external choice combinator, RSL provides an internal choice combinator '⊓' that specifies an internal choice between two expressions

expr$_1$ ⊓ expr$_2$

Whether $expr_1$ is evaluated or whether $expr_2$ is evaluated depends on an internal choice, which the surroundings cannot influence. The two expressions must have the same type. The internal choice combinator is commutative and assocative. As an example, consider the expression

(x:=c? ⊓ d!5) ∥ c!1

The expression $c!1$ has no influence on which of the two expressions $x := c?$ and $d!5$ are evaluated. If the internal choice falls on $x := c?$, the expression becomes equivalent to

x:=c? ∥ c!1

thus potentially leading to a communication over c. If the internal choice on the other hand falls on $d!5$, the expression becomes equivalent to

d!5 ∥ c!1

thus preventing any communication to take place.

The internal choice combinator is typically used in proofs about concurrent RSL specifications. One can, however, also use the combinator when writing specifications. Consider for example the specificaton of a "die-thrower".

```
  type
    Face_Of_Die == one|two|three|four|five|six
  value
    throw_die : Unit $\overset{\sim}{\rightarrow}$ Face_Of_Die
  axiom
    throw_die() ≡
      one ⌈⌉ two ⌈⌉ three ⌈⌉ four ⌈⌉ five ⌈⌉ six
```

The function *throw_die* will non-deterministically return a face of die. The axiom could also have been written as follows

```
  axiom
    throw_die() ≡
      let face_of_die : Face_Of_Die in
        face_of_die
      end
```

19.8 Examples

Example 19.6

Consider a concurrent version of the database from example 8.1. A database process, *database*, is introduced together with channels for communicating with it.

```
DATABASE =
  class
    type
      Key, Data,
      Database = Key $\overrightarrow{m}$ Data
    channel
      empty : Unit,
      insert : Key × Data,
      remove : Key,
      defined : Key,
      defined_res : Bool,
      lookup : Key,
      lookup_res : Data
    value
      database : Database →
        in empty,insert,remove,defined,lookup
        out defined_res,lookup_res
        Unit
    axiom forall db : Database •
      database(db) ≡
        empty? ; database([ ])
        ⌷
        let (k,d) = insert? in
          database(db † [k ↦ d])
        end
        ⌷
        let k = remove? in
          database(db\{k})
        end
        ⌷
        let k = defined? in
          defined_res!(k ∈ dom db) ; database(db)
        end
        ⌷
        let k = lookup? in
          if k ∈ dom db then
            lookup_res!(db(k)) ; database(db)
          else
            chaos
          end
        end
  end
```

Note how the "partialness" of a lookup communication is modelled by the use of **chaos**. The *database* process itself is not partial since it has to participate in at least one communication (an input from the *lookup* channel) before eventually diverging.

An essential task when specifying a process is to decide what the channels are and what the protocol is for their use. The above specification illustrates for example how certain channels may be connected: an ingoing communication on the *defined* channel is always followed by an outgoing communication on the *defined_res* channel. Likewise for the channels *lookup* and *lookup_res*.

It is quite illustrative to compare the channel definitions from the example above with the function and constant types from the applicative database in example 8.1. This is done below by listing the channels and the corresponding applicative functions and constants.

```
  channel
    empty : Unit,
  value
    empty : Database


  channel
```

```
   insert : Key × Data,
**value**
   insert : Key × Data × Database → Database


**channel**
   remove : Key,
**value**
   remove : Key × Database → Database


**channel**
   defined : Key,
   defined_res : **Bool**,
**value**
   defined : Key × Database → **Bool**


**channel**
   lookup : Key,
   lookup_res : Data
**value**
   lookup : Key × Database $\xrightarrow{\sim}$ Data
```

□

Example 19.7

Suppose we want to lookup a key *k* in the concurrent database defined in the previous example. We will have to write two communications

```
lookup!k ; ... lookup_res? ...
```

This may seem slightly tedious. The "problem" is that the interface to the *database* process is a set of channels. One can instead define a set of interaction processes that do all the channel communication and then recommend users to call these instead.

The module below is an extension of the concurrent *DATABASE* module with the definition of such interaction processes.

INTERFACED_DATABASE =

```
extend DATABASE with
   value
      Empty : Unit → out empty Unit,
      Insert : Key × Data → out insert Unit,
      Remove : Key → out remove Unit,
      Defined : Key → out defined in defined_res Bool,
      Lookup : Key → out lookup in lookup_res Data
   axiom forall k : Key, d : Data •
      Empty() ≡
         empty!(),
      Insert(k,d) ≡
         insert!(k,d),
      Remove(k) ≡
         remove!k,
      Defined(k) ≡
         defined!k ; defined_res?,
      Lookup(k) ≡
         lookup!k ; lookup_res?
end
```

Note that the two interaction processes *Defined* and *Lookup* both have result types different from **Unit**.

An interaction that before (with a channel interface) had the following form, assuming a $k \in Key$ and a $db \in Database$

```
(lookup!k ; x := lookup_res?) ∥ database(db)
```

is written as follows when using the interaction process *Lookup*

```
(x := Lookup(k)) ∥ database(db)
```

The use of interaction processes shorten specifications. In addition, they represent information hiding in that the specifier is not required to know about the details of channel communication.

□

20 Expressions Revisited

In this section we shall shortly revisit some of the expressions introduced in part one and part two of the document in the light of concurrency.

20.1 Pure and Read-only Expressions

In section 14 the concepts of pure expressions and read-only expressions were introduced. These concepts need a redefinition.

A pure expression is an expression that does not access variables and that does not communicate on channels.

A read-only expression is an expression that does not write to variables and that does not communicate on channels. It may though read from variables.

The concrete syntax in appendix A describes the occurrences of expressions that must either be pure or read-only.

20.2 Independent Expressions

In section 14 the concept of independent expressions was introduced. Recall for example that the two sub-expressions of an equality

$$\mathrm{expr}_1 = \mathrm{expr}_2$$

must be independent: if one sub-expression writes to a variable, the other must not access that variable. The two expressions can thus be evaluated in any order without changing the meaning.

With the possibility of channel communication we must additionally require that at most one of the expressions communicate. Otherwise the evaluation order would influence the order of communications.

The reader is referred to section 14 for a description of where expressions are required independent.

20.3 Equivalence Expressions

En equivalence expression (section 14) of the form

$\text{expr}_1 \equiv \text{expr}_2$

requires the two sub-expressions to represent the same communication behaviour in order to hold. The equivalence expression itself does not communicate since the communications of the two sub-expressions are only utilized in the comparison, and are thus kept “invisible” for the surroundings. An equivalence expression is thus still read-only.

21 State-based Processes

The many place buffer process, *mpb*, in example 19.5 is applicative in the sense that it is parameterised with respect to the "current" buffer contents. That is, it has the type

```
value
   mpb : Buffer → in empty,add out get Unit
```

An alternative is to keep the current buffer contents in a variable which the process then continuously modifies by means of assignments. There are essentially two approaches:

1. Introducing a variable global to the process which the process then has **write** access to.
2. Introducing a variable local to the process.

We shall treat each of the two approaches below.

21.1 Introducing a Global Variable

An alternative is to keep the current buffer contents in a global variable which the process then has **write** access to:

```
variable
   buffer : Buffer
value
   mpb : Unit → in empty,add out get write buffer Unit
```

Note that the parameter type of *mpb* now becomes **Unit**. The modified specification is as follows.

Example 21.1

```
MANY_PLACE_BUFFER =
   class
      type
         Elem,
         Buffer = Elem*
```

```
  variable
    buffer : Buffer := ⟨⟩
  channel
    empty : Unit,
    add : Elem,
    get : Elem
  value
    mpb : Unit → in empty,add out get write buffer Unit
  axiom
    mpb() ≡
      empty? ; buffer := ⟨⟩ ; mpb()
      ⌷
      let v = add? in
        buffer := buffer ⌢ ⟨v⟩
      end;
      mpb()
      ⌷
      if buffer ≠ ⟨⟩ then
        get!(hd buffer);
        buffer := tl buffer;
        mpb()
      else
        stop
      end
end
```

□

The imperative version differs from the applicative one in the following ways

- It defines the variable *buffer* with the initial contents being the empty buffer.
- The buffer process, *mpb*, has **write** access to the *buffer* variable. The parameter type has consequently become **Unit**.
- The axiom defining *mpb* specifies how the *buffer* variable is modified in terms of assignments.

21.2 Introducing a Local Variable

Another alternative is to keep the current buffer contents in a local variable:

```
  value
    mpb : Unit → in empty,add out get Unit
  axiom
    mpb() ≡
      local
        type
          Buffer = Elem*
        variable
          buffer : Buffer
      in
        ⋮
      end
```

In this way the process remains applicative by not accessing global variables.

The modified specification is as follows.

Example 21.2

```
MANY_PLACE_BUFFER =
  class
    type
      Elem
    channel
      empty : Unit,
      add : Elem,
      get : Elem
    value
      mpb : Unit → in empty,add out get Unit
    axiom
      mpb() ≡
        local
          type
            Buffer = Elem*
          variable
            buffer : Buffer := ⟨⟩
        in
          while true do
            empty? ; buffer := ⟨⟩
            ⌷
            let v = add? in
              buffer := buffer ⌢ ⟨v⟩
            end
            ⌷
            if buffer $\neq \langle\rangle$ then
              get!(hd buffer);
              buffer := tl buffer;
            else
              stop
            end
          end
        end
  end
```

□

Beyound making the variable local, the iterative evaluation of *mpb* has now been modelled by means of a while expression of the form

```
while true do
  ⋮
end
```

This formulation of infinite iteration is the most natural in connection with local variables, since the alternative of recursive process calls leads to re-initialisation of local variables for each call.

22 Algebraic Definition of Processes

Processes can be defined abstractly in terms of algebraic equivalences. We have already seen how this can be done for applicative functions (section 5) and for operations (section 17).

For the purpose of being able to compare with the applicative and imperative case, we shall abstractly specify a concurrent list module. Recall the algebraic specification of the *LIST*-module from example 5.3, which is repeated below. Constants and functions have been subscripted with an a to indicate that they are applicative.

```
LIST =
  class
    type
      List
    value
      $\mathrm{empty}_a$ : List,
      $\mathrm{add}_a$ : Int × List → List,
      $\mathrm{head}_a$ : List $\xrightarrow{\sim}$ Int,
      $\mathrm{tail}_a$ : List $\xrightarrow{\sim}$ List
    axiom
      forall i : Int, l : List •
      [head_add]
        $\mathrm{head}_a(\mathrm{add}_a(i,l)) \equiv i$,
      [tail_add]
        $\mathrm{tail}_a(\mathrm{add}_a(i,l)) \equiv l$
  end
```

There are at least three ways of abstractly specifying a concurrent list module and we shall treat each of them below. The three ways correspond closely to the three ways outlined for the state-based case in section 17.

22.1 Extending an Applicative Module

The first approach is to use the entities from the applicative *LIST*-module in defining the concurrent module. The concurrent module thus becomes an extension of the *LIST*-module.

Example 22.1

```
CONCURRENT_LIST =
  extend LIST with
```

```
    channel
      empty : Unit,
      is_empty : Bool,
      add : Int,
      head : Int,
      tail : Unit
    value
      list : List → in empty,add,tail out is_empty,head Unit
    axiom forall l : List •
      list(l) ≡
        empty? ; list(empty_a)
        ⌷
        is_empty!(l = empty_a) ; list(l)
        ⌷
        let i = add? in
          list(add_a(i,l))
        end
        ⌷
        if ∼(l = empty_a) then
          head!(head_a(l)) ; list(l)
        else
          stop
        end
        ⌷
        if ∼(l = empty_a) then
          tail? ; list(tail_a(l))
        else
          stop
        end
  end
```

□

A list process, *list*, is defined which communicates with its surroundings via the channels *empty*, *is_empty*, *add*, *head* and *tail*.

The process is parameterised with its "state" of the type *List*. This type comes from the *LIST*-module and is a sort. Nothing has thus been said about representation of its values.

The process behaviour following communication on the channels is defined by simple calls of the corresponding applicative functions. Since these are defined without assuming any particular representation, the process share that property.

The approach of using an applicative specification in defining a concurrent one may seem tedious, especially if the applicative one does not exist already.

22.2 Algebraic Equivalences

The second approach to abstractly specifying the concurrent list module is to give algebraic equivalences between process communications in a way very similar to the equivalences in the applicative *LIST*-module.

As an example, consider the applicative axiom *head_add* from *LIST*

```
axiom
   forall i : Int, l : List •
   [head_add]
      head_a(add_a(i,l)) ≡ i
```

The axiom says that adding an element *i* to a list and then taking the head yields the element just added.

In the concurrent case, we have to write a bit more. First of all, we shall need a variable to hold the value returned from the *head* channel

```
variable
   head_res : Int
```

The axiom could then be stated as follows

```
axiom
   forall i : Int, l : List •
   [head_add]
      list(l) ‖ (add!i ; head_res := head?) ≡
         list(l) ‖ (add!i ; head_res := i)
```

The axiom is supposed to make the following two interactions equivalent:

- Left-hand side: send a value *i* to the process on the *add* channel and then store in *head_res* the value received from the *head* channel.
- Right-hand side: send the value *i* to the process on the *add* channel and then store *i* in *head_res*.

In other words: one will from the *head* channel always get the element last added on the *add* channel. In adition, a communication on the *head* channel does not effect the state of the process.

The variable *head_res* has been necessary to introduce since the parallel operator needs both ingoing sub-expressions to have the type **Unit**.

The axiom is, however, not sufficient. The reason is that the parallel operator does not force communication to happen and thus allows other parallel evaluating expressions to interfeer. Such an interfeering expression could thus perform an $add!i1$ in between $add!i$ and $head\_res :=$ $head?$.

In other words, if the above axiom is to hold, then the following property must also hold (equivalence implies substitutability)

```
∀ i,i1 : Int, l : List •
   add!i1 ‖ (list(l) ‖ (add!i ; head_res := head?)) ≡
      add!i1 ‖ (list(l) ‖ (add!i ; head_res := i))
```

Let us first observe the left-hand side of this derived property. A possible evaluation will be the following

1. $list(l)$ accepts the communication $add!i$, thus resulting in

   add!i1 ‖ list(add$_a$(i,l)) ‖ head_res := head?

2. $list(add_a(i, l))$ accepts the communication $add!i1$, thus resulting in

   list(add$_a$(i1,add$_a$(i,l))) ‖ head_res := head?

3. $list(add_a(i1, add_a(i, l)))$ accepts the communication $head?$, thus resulting in

   head_res := i1 ; list(add$_a$(i,l))

This final expression is obviously not equivalent to the right-hand side of the derived property. The conclusion must thus be that the original axiom does not hold.

The solution is to introduce a new parallel combinator that is more “aggressive” in forcing communication between the two expressions to happen. The interlocking combinator does exactly that. An expression of the form

expr$_1$ ⫲ expr$_2$

is evaluated by evaluating the two sub-expressions (both having type **Unit**) in interlocked parallel: the two expressions are evaluated in parallel until one of them comes to an end, whereupon evaluation continues with the other. During the parallel evaluation, any external communication is disregarded. In our example above, *add*!*i*1 was the external communication that should have been disregarded.

The interlocking combinator can maybe best be explained by stating some equivalences between expressions using it.

Assume the following channel definitions and variable definition

```
channel
  c,c1,c2 : T
variable
  x : T
```

Then the following equivalence holds

```
x := c? ╫ c!e ≡
  x := e
```

That is: since the two expressions $x := c?$ and $c!e$ can communicate, they will communicate.

The corresponding equivalence for the normal parallel combinator is somewhat more complicated

```
x := c? ∥ c!e ≡
  (x := e) ⊓ ((x := c? ; c!e) [] (c!e ; x := c?) [] (x := e))
```

That is: the two expressions may communicate, leading to

```
x := e
```

Whether they do depends on an internal choice. Alternatively, it will be up the surroundings to make an external choice between communications. Note that in this case, the surroundings can choose to let the two expressions communicate.

Another example involving the external choice combinator is the following

```
(x := c1 [] c2!e2) ╫ c1!e1 ≡
  x := e1
```

That is: the interlocking combinator forces through the external choice of the expression $x := c_1$.

These equivalences show how the interlocking combinator leaves no possible communications outstanding. In the reverse case, where both of the interlocked expressions want to communicate, but not with each other, the result will be a deadlock. This is illustrated by the following equivalence

```
$x := c_1? ╫ c_2!e \equiv$
  **stop**
```

The corresponding equivalence for the normal parallel combinator is as follows

```
$x := c_1? \parallel c_2!e \equiv$
  $(x := c_1? ; c_2!e) \;[]\; (c_2!e ; x := c_1?)$
```

That is: since the two expressions cannot communicate with each other, they can only communicate with the surroundings.

The interlocking combinator is well suited for illustrating the difference between external choice and internal choice. Recall the equivalence given above for external choice and then compare with the following one for internal choice

```
$(x := c_1 \sqcap c_2!e_2) ╫ c_1!e_1 \equiv$
  $x := e_1 \sqcap$ **stop**
```

That is: if the internal choice falls on the expression $x := c_1$, then a communication takes place (resulting in $x := e_1$). If on the other hand, the internal choice falls on $c_2!e_2$, then both interlocked expressions want to communicate, but not with each other, and the result will be a deadlock.

The interlocking combinator is commutative but it is not associative like the normal parallel combinator.

The interlocking combinator can now be used to correctly write the *head_add* axiom

```
**axiom**
  **forall** i : **Int**, l : List •
  [head_add]
    list(l) ╫ (add!i ; head_res := head?) ≡
      list(l) ╫ (add!i ; head_res := i)
```

We shall, however, prefer to write this axiom in a slightly different way. Note first that the interlocking combinator satisfies the following property

```
expr₁ ╫ (expr₂ ; id₁ := id₂) ≡
   (expr₁ ╫ expr₂) ; id₁ := id₂
```

That is: since the assignment does not communicate with $expr_1$ it can be “sequentialised”. We shall thus write our axiom instead as follows

```
axiom
   forall i : Int, l : List •
   [head_add]
      list(l) ╫ (add!i ; head_res := head?) ≡
         (list(l) ╫ add!i) ; head_res := i
```

This presentation has the advantage of separating what the effect on the process is, namely *list*(*l*) ╫ *add*!*i*, from what the returned value is, namely *head_res* := *i*.

The complete concurrency-based specification of lists is as follows.

Example 22.2

```
LIST =
   class
      type
         List
      channel
         empty : Unit,
         is_empty : Bool,
         add : Int,
         head : Int,
         tail : Unit
      variable
         is_empty_res : Bool,
         head_res : Int
      value
         list : List → in empty,add,tail out is_empty,head Unit
   axiom
         forall i : Int, l : List •
         [is_empty_empty]
            list(l) ╫ (empty!() ; is_empty_res := is_empty?) ≡
```

```
        (list(l) ⫳ empty!()) ; is_empty_res := true,
    [is_empty_add]
      list(l) ⫳ (add!i ; is_empty_res := is_empty?) ≡
        (list(l) ⫳ add!i) ; is_empty_res := false,
    [head_add]
      list(l) ⫳ (add!i ; head_res := head?) ≡
        (list(l) ⫳ add!i) ; head_res := i,
    [tail_add]
      list(l) ⫳ (add!i ; tail!()) ≡
        list(l)
end
```

□

The parameter type of the *list* process is *List* which is a sort. Nothing has thus been said about representation. The axioms likewise assume no particular representation of lists.

22.3 Being Implicit about Channels

Until now we have been abstract only about data representation. There is, however, a possibility of being even more abstract than that. In the third approach we shall additionally be implicit about what the channels are, by simply not defining any. The technique is to instead define the interface to the *list* process as a set of interaction processes (see example 19.7) and then to state their properties in terms of the interlocking combinator.

We can thus modify example 22.2 by removing the following definitions

```
channel
  empty : Unit,
  is_empty : Bool,
  add : Int,
  head : Int,
  tail : Unit
```

The *list* process type must now be modified such that it does not mention the channels. Recall that it had the following definition

```
list : List → in empty,add,tail out is_empty,head Unit
```

Instead of channel names one can write **any** in the access description to indicate that "any" channel defined my be communicated on. An access description can thus have one of the forms

in any

out any

The definition of the type of *list* becomes

list : List → **in any out any Unit**

The interaction processes must also have **any** accesses. As an example let us consider the *add* and *head* interaction processes which could be given the types

add : **Int** → **in any out any Unit**
head : **Unit** → **in any out any Int**

Since the interlocking combinator requires its argument expressions to have the type **Unit** we shall need a variable to hold the value returned from the *head* process

variable
  head_res : **Int**

The axiom *head_add* can now be written as follows

axiom
  forall i : **Int**, l : List •
  [head_add]
    list(l) ╫ (add(i) ; head_res := head()) ≡
      (list(i) ╫ add(i)) ; head_res := i

An alternative style is to let the interaction process *head* itself write to the result variable, thus giving *head* the type

head : **Unit** → **in any out any write** head_res **Unit**

The axiom will in this case become a bit more elegant

axiom
  forall i : **Int**, l : List •
  [head_add]
    list(l) ╫ (add(i) ; head()) ≡
      (list(i) ╫ add(i)) ; head_res := i

Using this style also makes calls of *head* simpler since one does not have to consider the storing of the result: *head* does it for you. The complete specification of the list module can be written as follows.

Example 22.3

```
LIST =
  class
    type
      List
    value
      empty : Unit → in any out any Unit,
      is_empty : Unit → in any out any write is_empty_res Unit,
      add : Int → in any out any Unit,
      head : Unit → in any out any write head_res Unit,
      tail : Unit → in any out any Unit
    variable
      is_empty_res : Bool,
      head_res : Int
    value
      list : List → in any out any Unit
    axiom
      forall i : Int, l : List •
      [is_empty_empty]
        list(l) ⫲ (empty() ; is_empty()) ≡
          (list(l) ⫲ empty()) ; is_empty_res := true,
      [is_empty_add]
        list(l) ⫲ (add(i) ; is_empty()) ≡
          (list(l) ⫲ add(i)) ; is_empty_res := false,
      [head_add]
        list(l) ⫲ (add(i) ; head()) ≡
          (list(l) ⫲ add(i)) ; head_res := i,
      [tail_add]
        list(l) ⫲ (add(i) ; tail()) ≡
          list(l)
  end
```

□

The following has been gained by being implicit about channels and using interaction processes

- We have avoided deciding what the channels shall be and what their types shall be.

- Suppose we later develop an implementation of the *LIST*-module from example 22.3. Our specification then places no restriction on what the channels of an implementation shall be.
- The specification places no restrictions on what channels the processes are allowed to access. The implementation of abstract interaction processes in terms of concrete interaction processes that do explicit channel communication is sometimes referred to as event refinement.

Note that **any**-accesses can also be used in process types even if channels have been defined in the context. It then allows the processes to access any of the defined channels. Again, one can see this as giving freedom to an implementation.

A natural question is when to be implicit about channels and when to be explicit. It is difficult to give exact rules. Very roughly, one may be implicit in the following situations.

- One will not be bothered with what the channels are.
- One wants to leave freedom to a later development that is expected to be an implementation in the formal sense.

Being explicit, however, has its benefits.

- One can from the type of a process see exactly what channels may be accessed and how they may be accessed. This makes concurrency-based specifications a lot easier to read and prove properties about.
- One may simply prefer the explicit style since it perhaps more clearly expresses "what goes on".

A more detailed description of **any**-accesses will be given in part four on modules.

22.4 Example

Example 22.4

Consider an algebraic specification of a concurrency-based database. We will be implicit about channels by not defining any. Consequently, we must define a set of interaction processes.

DATABASE =

```
class
   type
      Key, Data,
      Database
   value
      empty : Unit → in any out any Unit,
      insert : Key × Data → in any out any Unit,
      remove : Key → in any out any Unit,
      defined : Key → in any out any write defined_res Unit,
      lookup : Key → in any out any write lookup_res Unit
   variable
      defined_res : Bool,
      lookup_res : Data
   value
      database : Database → in any out any Unit
   axiom
      forall k,k1 : Key, d : Data, db : Database •
      [remove_empty]
         database(db) ⫲ (empty() ; remove(k)) ≡
            database(db) ⫲ empty(),
      [remove_insert]
         database(db) ⫲ (insert(k1,d) ; remove(k)) ≡
            if k = k1 then
               database(db) ⫲ remove(k)
            else
               database(db) ⫲ (remove(k) ; insert(k1,d))
            end,
      [defined_empty]
         database(db) ⫲ (empty() ; defined(k)) ≡
            (database(db) ⫲ empty()) ; defined_res := false,
      [defined_insert]
         database(db) ⫲ (insert(k1,d) ; defined(k)) ≡
            if k = k1 then
               (database(db) ⫲ insert(k1,d)) ; defined_res := true
            else
               database(db) ⫲ (defined(k) ; insert(k1,d))
            end,
      [lookup_insert]
         database(db) ⫲ (insert(k1,d) ; lookup(k)) ≡
            if k = k1 then
               (database(db) ⫲ insert(k1,d)) ; lookup_res := d
            else
               database(db) ⫲ (lookup(k) ; insert(k1,d))
            end
end
```

The concurrency-based database example illustrates the "constructor" technique for inventing axioms, which we until now have seen applied in the state-based case as well as in the applicative case. The technique used in the concurrency-based case with interaction processes can be characterised as follows

1. Identify the "constructor interaction processes" by which any database can be constructed. These are the processes *empty* and *insert*. Any database can thus be generated by an expression of the form

   database(db) ╫ (empty() ; insert(k_1,d_1) ; ... ; insert(k_n,d_n))

2. Define the remaining processes "by case" over the constructor processes called with identifiers as parameters. In the above axioms, *remove*, *defined* and *lookup* are thus defined over the two constructor-expressions

   empty()

   insert(k1,d)

   We thus get "for free" all the left-hand sides of the axioms we must write. That is

   database(db) ╫ (empty() ; remove(k))
   database(db) ╫ (insert(k1,d) ; remove(k))

   database(db) ╫ (empty() ; defined(k))
   database(db) ╫ (insert(k1,d) ; defined(k))

   database(db) ╫ (empty() ; lookup(k))
   database(db) ╫ (insert(k1,d) ; lookup(k))

   Note, however, that due to the partiality of a lookup communication we don't bother with giving the right-hand side corresponding to *database*(*db*) ╫ (*empty*() ; *lookup*(*k*)).

The list-axioms in example 22.3 actually have the same form. The list-axioms in example 22.2 illustrate a similar technique applied in the case where the process interface is a set of channels and not a set of interaction processes.

The technique is useful in many applications, but there are of course applications where one must be more imaginative when writing axioms.

□

Part IV

Composing Systems from Modules

To be written.

References

[1] *RSL Reference Manual*
RAISE/CRI/DOC/2/V1

A Syntax Summary

The syntax defines the syntactically correct strings of the language. The strings are divided into syntax categories with the top syntax category containing all syntactically correct RSL specifications. Each syntax category is defined by a rule.

Each rule is of the form

category_name ::=
  $\text{alternative}_1|$
  ...
  alternative_n

where $n \geq 1$. This rule introduces the syntax category named category_name and defines that category as the union of the strings generated by the alternatives. As an example consider

set_type_expr ::=
  finite_set_type_expr|
  infinite_set_type_expr

Each alternative consists of a sequence of tokens where a token is of one of three kinds

- A keyword in bolded font such as '**Bool**'
- A symbol such as '('.
- A sub-category name such as 'expr', possibly prefixed with a text such as '*logical-*' in italics. Text in italics states a context condition that syntactically correct strings must satisfy in order to be statically correct.

The strings generated by an alternative are those obtained by concatenating keywords, symbols and strings from sub-categories – in the order of appearance. As examples consider

finite_set_type_expr ::=
  type_expr-**set**

map_type_expr ::=
  type_expr $\underset{m}{\rightarrow}$ type_expr

The below convention is used for defining optional presence (ϵ represents absence): For any syntax category name 'x' the following rule is assumed.

opt_x ::=
  ϵ|
  x

The below conventions are used for defining repetition: For any syntax category name ‘x’ the following rules are assumed.

x−string ::=
  x|
  x x−string

x-list ::=
  x|
  x , x-list

x-list2 ::=
  x , x-list

x−choice ::=
  x|
  x | x−choice

x−choice2 ::=
  x | x−choice

x−product2 ::=
  x × x−product

x−product ::=
  x|
  x × x−product

Specifications

specification ::=
  module_decl-string

module_decl ::=
  object_decl |
  scheme_decl

Object declarations

object_decl ::=
  object object_def-list

object_def ::=
  opt-comment-string id opt-formal_array_parameter : class_expr

formal_array_parameter ::=
  [typing-list]

Scheme declarations

scheme_decl ::=
  scheme scheme_def-list

scheme_def ::=
  opt-comment-string id opt-formal_scheme_parameter = class_expr

formal_scheme_parameter ::=
  (formal_scheme_argument-list)

formal_scheme_argument ::=
  object_def

Class expressions

class_expr ::=
  basic_class_expr |
  importing_class_expr |
  extending_class_expr |
  hiding_class_expr |

renaming_class_expr |
scheme_instantiation

Basic class expressions

basic_class_expr ::=
class opt-decl-string **end**

Importing class expressions

importing_class_expr ::=
import object_expr-list **in** class_expr

Extending class expressions

extending_class_expr ::=
extend class_expr-list **with** opt-decl-string **end**

Hiding class expressions

hiding_class_expr ::=
hide defined_item-list **in** class_expr

Renaming class expressions

renaming_class_expr ::=
use rename_pair-list **in** class_expr

Scheme instantiations

scheme_instantiation ::=
scheme-name opt-actual_scheme_parameter

actual_scheme_parameter ::=
(object_expr-list)

Object expressions

object_expr ::=
  object-name |
  element_object_expr |
  array_object_expr |
  fitting_object_expr

Element object expressions

element_object_expr ::=
  array-object_expr actual_array_parameter

actual_array_parameter ::=
  [*pure*-expr-list]

Array object expressions

array_object_expr ::=
  [| typing-list • *element*-object_expr |]

Fitting object expressions

fitting_object_expr ::=
  object_expr renaming

Renamings

renaming ::=
  { rename_pair-list }

rename_pair ::=
  defined_item **for** defined_item

defined_item ::=
  id_or_op |
  disambiguated_item

disambiguated_item ::=
  id_or_op : type_expr

Declarations

decl ::=
  object_decl |
  scheme_decl |
  type_decl |
  value_decl |
  variable_decl |
  channel_decl |
  axiom_decl

Type declarations

type_decl ::=
  type commented_type_def-list

commented_type_def ::=
  opt-comment-string type_def

type_def ::=
  sort_def |
  variant_def |
  union_def |
  short_record_def |
  abbreviation_def

Sort definitions

sort_def ::=
  id

Variant definitions

variant_def ::=
  id == variant-choice

variant ::=
  constant_variant |
  record_variant

constant_variant ::=

constructor opt-subtype_naming

record_variant ::=
constructor component_kinds opt-subtype_naming

constructor ::=
id_or_op |
_

component_kinds ::=
(component_kind-list)

component_kind ::=
opt-destructor type_expr opt-reconstructor

destructor ::=
id_or_op :

reconstructor ::=
↔ id_or_op

subtype_naming ::=
@ id

Union definitions

union_def ::=
id = *type*-name-choice2

Short record definitions

short_record_def ::=
id :: component_kind-string

Abbreviation definitions

abbreviation_def ::=
id = type_expr

Value declarations

value_decl ::=
  value commented_value_def-list

commented_value_def ::=
  opt-comment-string value_def

value_def ::=
  typing |
  explicit_value_def |
  implicit_value_def |
  explicit_function_def |
  implicit_function_def

Explicit value definitions

explicit_value_def ::=
  single_typing = *pure*-expr

Implicit value definitions

implicit_value_def ::=
  single_typing *pure*-restriction

Explicit function definitions

explicit_function_def ::=
  single_typing formal_function_application ≡ expr opt-pre_condition

formal_function_application ::=
  id_application |
  prefix_application |
  infix_application

id_application ::=
  value-id formal_function_parameter-string

formal_function_parameter ::=
  (opt-binding-list)

prefix_application ::=
  prefix_op id

infix_application ::=
  id infix_op id

pre_condition ::=
  pre *readonly_logical*-expr

Implicit function definitions

implicit_function_def ::=
  single_typing formal_function_application post_condition opt-pre_condition

post_condition ::=
  opt-result_naming **post** *readonly_logical*-expr

result_naming ::=
  as binding

Variable declarations

variable_decl ::=
  variable commented_variable_def-list

commented_variable_def ::=
  opt-comment-string variable_def

variable_def ::=
  single_variable_def |
  multiple_variable_def

single_variable_def ::=
  id : type_expr opt-initialisation

initialisation ::=
  := *pure*-expr

multiple_variable_def ::=
  id-list2 : type_expr

Channel declarations

channel_decl ::=
 channel commented_channel_def-list

commented_channel_def ::=
 opt-comment-string channel_def

channel_def ::=
 single_channel_def |
 multiple_channel_def

single_channel_def ::=
 id : type_expr

multiple_channel_def ::=
 id-list2 : type_expr

Axiom declarations

axiom_decl ::=
 axiom opt-axiom_quantification axiom_def-list

axiom_quantification ::=
 forall typing-list •

axiom_def ::=
 opt-comment-string opt-axiom_naming *pure_logical*-expr

axiom_naming ::=
 [id]

Type expressions

type_expr ::=
  type_literal |
  type-name |
  product_type_expr |
  set_type_expr |
  list_type_expr |
  map_type_expr |
  function_type_expr |
  subtype_expr |
  bracketted_type_expr

Type literals

type_literal ::=
  Unit |
  Bool |
  Int |
  Nat |
  Real |
  Text |
  Char

Product type expressions

product_type_expr ::=
  type_expr-product2

Set type expressions

set_type_expr ::=
  finite_set_type_expr |
  infinite_set_type_expr

finite_set_type_expr ::=
  type_expr-**set**

infinite_set_type_expr ::=
  type_expr-**infset**

List type expressions

list_type_expr ::=
  finite_list_type_expr |
  infinite_list_type_expr

finite_list_type_expr ::=
  type_expr*

infinite_list_type_expr ::=
  type_expr$^\omega$

Map type expressions

map_type_expr ::=
  type_expr $\underset{m}{\rightarrow}$ type_expr

Function type expressions

function_type_expr ::=
  type_expr function_arrow result_desc

function_arrow ::=
  $\overset{\sim}{\rightarrow}$ |
  $\rightarrow$

result_desc ::=
  opt-access_desc-string type_expr

Access descriptions

access_desc ::=
  access_mode access-list

access_mode ::=
  read |
  write |
  in |
  out

access ::=

variable_or_channel-name |
completed_access |
comprehended_access

completed_access ::=
opt-qualification **any**

comprehended_access ::=
{ access-list | *pure*-set_limitation }

Subtype expressions

subtype_expr ::=
{| single_typing *pure*-restriction |}

Bracketted type expressions

bracketted_type_expr ::=
(type_expr)

Expressions

expr ::=
  value_literal |
  value_or_variable-name |
  pre_name |
  basic_expr |
  product_expr |
  set_expr |
  list_expr |
  map_expr |
  function_expr |
  application_expr |
  quantified_expr |
  equivalence_expr |
  post_expr |
  disambiguation_expr |
  bracketted_expr |
  infix_expr |
  prefix_expr |
  comprehended_expr |
  initialise_expr |
  assignment_expr |
  input_expr |
  output_expr |
  structured_expr

Value literals

value_literal ::=
  unit_literal |
  bool_literal |
  int_literal |
  real_literal |
  text_literal |
  char_literal

unit_literal ::=
  ( )

bool_literal ::=
  true |
  false

Pre names

pre_name ::=
  variable-name `

Basic expressions

basic_expr ::=
  chaos |
  skip |
  stop |
  swap

Product expressions

product_expr ::=
  (expr-list2)

Set expressions

set_expr ::=
  ranged_set_expr |
  enumerated_set_expr |
  comprehended_set_expr

Ranged set expressions

ranged_set_expr ::=
  { *readonly_integer*-expr .. *readonly_integer*-expr }

Enumerated set expressions

enumerated_set_expr ::=
  { *readonly*-opt-expr-list }

Comprehended set expressions

comprehended_set_expr ::=
  { *readonly*-expr | set_limitation }

set_limitation ::=
  typing-list opt-restriction

restriction ::=
  • *readonly_logical*-expr

List expressions

list_expr ::=
  ranged_list_expr |
  enumerated_list_expr |
  comprehended_list_expr

Ranged list expressions

ranged_list_expr ::=
  ⟨ *readonly_integer*-expr .. *readonly_integer*-expr ⟩

Enumerated list expressions

enumerated_list_expr ::=
  ⟨ *readonly*-opt-expr-list ⟩

Comprehended list expressions

comprehended_list_expr ::=
  ⟨ *readonly*-expr | list_limitation ⟩

list_limitation ::=
  binding **in** *readonly_list*-expr opt-restriction

Map expressions

map_expr ::=
  enumerated_map_expr |
  comprehended_map_expr

Enumerated map expressions

enumerated_map_expr ::=
  [opt-expr_pair-list]

expr_pair ::=
  readonly-expr ↦ *readonly*-expr

Comprehended map expressions

comprehended_map_expr ::=
  [expr_pair | set_limitation]

Function expressions

function_expr ::=
  λ lambda_parameter • expr

lambda_parameter ::=
  lambda_typing |
  single_typing

lambda_typing ::=
  (opt-typing-list)

Application expressions

application_expr ::=
  list_or_map_or_function-expr actual_function_parameter-string

actual_function_parameter ::=
  (opt-expr-list)

Quantified expressions

quantified_expr ::=
  quantifier typing-list restriction

quantifier ::=
  ∀ |
  ∃ |
  ∃!

Equivalence expressions

equivalence_expr ::=
  expr ≡ expr opt-pre_condition

Post expressions

post_expr ::=
  expr post_condition opt-pre_condition

Disambiguation expressions

disambiguation_expr ::=
  expr : type_expr

Bracketted expressions

bracketted_expr ::=
  (expr)

Infix expressions

infix_expr ::=
  stmt_infix_expr |
  axiom_infix_expr |
  value_infix_expr

Stmt infix expressions

stmt_infix_expr ::=
  expr infix_combinator expr

Axiom infix expressions

axiom_infix_expr ::=
  logical-expr infix_connective *logical*-expr

Value infix expressions

value_infix_expr ::=
  expr infix_op expr

Prefix expressions

prefix_expr ::=
  axiom_prefix_expr |
  value_prefix_expr

Axiom prefix expressions

axiom_prefix_expr ::=
  prefix_connective *logical*-expr

Value prefix expressions

value_prefix_expr ::=
  prefix_op expr

Comprehended expressions

comprehended_expr ::=
  associative_commutative-infix_combinator { expr | set_limitation }

Initialise expressions

initialise_expr ::=
  opt-qualification **initialise**

Assignment expressions

assignment_expr ::=
  variable-name := expr

Input expressions

input_expr ::=
  channel-name ?

Output expressions

output_expr ::=
  channel-name ! expr

Structured expressions

structured_expr ::=
  local_expr |
  let_expr |
  if_expr |
  case_expr |
  for_expr |
  while_expr |
  until_expr

Local expressions

local_expr ::=
  local opt-decl-string **in** expr **end**

Let expressions

let_expr ::=
  let let_def-list **in** expr **end**

let_def ::=
  typing |
  explicit_let |
  implicit_let

explicit_let ::=
  let_binding = expr

implicit_let ::=
  single_typing restriction

let_binding ::=
  binding |
  record_pattern |
  list_pattern

If expressions

if_expr ::=
  if *logical*-expr **then**
    expr
  opt-elsif_branch-string
  opt-else_branch
  end

elsif_branch ::=
  elsif *logical*-expr **then** expr

else_branch ::=
  else expr

Case expressions

case_expr ::=
  case expr **of** case_branch-list **end**

case_branch ::=
  pattern → expr

For expressions

for_expr ::=
  for list_limitation **do** *unit*-expr **end**

While expressions

while_expr ::=
  while *logical*-expr **do** *unit*-expr **end**

Until expressions

until_expr ::=
  do *unit*-expr **until** *logical*-expr **end**

Bindings

binding ::=
  id_or_op |
  product_binding

product_binding ::=
  (binding-list2)

Typings

typing ::=
 single_typing |
 multiple_typing

single_typing ::=
 binding : type_expr

multiple_typing ::=
 binding-list2 : type_expr

Patterns

pattern ::=
   value_literal |
   pure_value-name |
   wildcard_pattern |
   product_pattern |
   record_pattern |
   list_pattern

Wildcard patterns

wildcard_pattern ::=
   _

Product patterns

product_pattern ::=
   (pattern-list2)

Record patterns

record_pattern ::=
   pure_value-name component_patterns

component_patterns ::=
   (inner_pattern-list)

inner_pattern ::=
   binding |
   wildcard_pattern

List patterns

list_pattern ::=
   constructed_list_pattern |
   left_list_pattern |
   right_list_pattern |
   left_right_list_pattern

Constructed list patterns

constructed_list_pattern ::=
  ⟨ opt-inner_pattern-list ⟩

Left list patterns

left_list_pattern ::=
  constructed_list_pattern ⌢ id_or_wildcard

id_or_wildcard ::=
  id |
  wildcard_pattern

Right list patterns

right_list_pattern ::=
  id_or_wildcard ⌢ constructed_list_pattern

Left right list patterns

left_right_list_pattern ::=
  constructed_list_pattern ⌢ id_or_wildcard ⌢ constructed_list_pattern

Names

name ::=
  qualified_id |
  qualified_op

Qualified ids

qualified_id ::=
  opt-qualification id

qualification ::=
  element-object_expr .

Qualified ops

qualified_op ::=
  opt-qualification (op)

Identifiers and operators

id_or_op ::=
  id |
  op

op ::=
  infix_op |
  prefix_op

Infix ops

infix_op ::=
  $=$ |
  $\neq$ |
  $>$ |
  $<$ |
  $\geq$ |
  $\leq$ |
  $\supset$ |

⊂ |
⊇ |
⊆ |
∈ |
∉ |
+ |
− |
\ |
^ |
∪ |
† |
∗ |
/ |
° |
∩ |
↑ |
$

Prefix ops

prefix_op ::=
abs |
it |
rl |
card |
len |
inds |
elems |
hd |
tl |
front |
last |
dom |
rng

Connectives

connective ::=
  infix_connective |
  prefix_connective

Infix connectives

infix_connective ::=
  $\Rightarrow$ |
  $\vee$ |
  $\wedge$

Prefix connectives

prefix_connective ::=
  $\sim$ |
  $\Box$

Infix combinators

infix_combinator ::=
  ⌷ |
  ⌈⌉ |
  ∥ |
  ╫ |
  ;