The RAISE Specification Language

The RAISE Language Group

Part No.: LACOS/CRI/DOC/8/V1

Date: 20 January 1992

The RAISE Language Group:

Chris George,
Peter Haff,
Klaus Havelund,
Anne E. Haxthausen,
Robert Milne,
Claus Bendix Nielsen,
Søren Prehn,
Kim Ritter Wagner

Contents

Editorial Preface

The aim of the BCS Practitioner Series is to produce books which are relevant for practising computer professionals across the whole spectrum of Information Technology activities. We want to encourage practitioners to share their practical experience of methods and applications with fellow professionals. We also seek to disseminate information in a form which is suitable for the practitioner who often has only limited time to read widely within a new subject area or to assimilate research findings.

The role of the BCS is to provide advice on the suitability of books for the Series, via the Editorial Panel, and to provide a pool of potential authors upon which we can draw. Our objectitve is that this Series will reinforce the drive within the BCS to increase professional standards in IT. The other partners in this venture, Prentice Hall, provide the publishing expertise and international marketing capabilities of a leading publisher in the computing field.

The response when we set up the Series was extremely encouraging. However, the success of the Series depends on there being practitioners who want to learn as well as those who feel they have something to offer! The Series is under continual development and we are always looking for ideas for new topics and feedback on how to further improve the usefulness of the Series. If you are interested in writing for the Series then please contact us.

The use of **formal methods** for software development was one of the topics which the Editorial Panel identified as of great importance for the Series. This book is one of two about RAISE which provides a complete method, together with supporting tools, for the application of a formal approach to software specification, design and implementation. RAISE was developed with the practitioner in mind and is the result of collaborative effort within the ESPRIT programme.

Ray Welland
Computing Science Department, University of Glasgow

Preface

A formal specification language like the RAISE Specification Language aims at providing a sound notation — with a semantics and a proof system — for capturing requirements and expressing the functionality of software.

Formal methods are no longer just the subject of academic study; they are increasingly being accepted by industry. Together with a sound development method, also provided by RAISE, and to be treated in a separate volume, use of RAISE aims at providing the software industry with a mature means of developing software correct with respect to its specification.

The name RAISE stands for Rigorous Approach to Industrial Software Engineering. RAISE was the name of a CEC funded ESPRIT project, and is now the name for a wide spectrum specification and design language, an associated method, and a commercially available tool set.

RAISE caters for a full spectrum of specification features — parameterizable abstract data types, modularity, concurrency, non-determinism, subtypes — for full development from abstraction to programming languages like Ada and C++, and for formal correctness proofs.

This book is primarily aimed at professional programmers, but can also be used by students at a late undergraduate or early graduate level. It shows how specifications may be written in any of the styles permitted by RSL: applicative or imperative; sequential or concurrent; direct (explicit) or axiomatic (implicit); with abstract data types (algebraic) or with concrete data types (model-oriented). Each combination of paradigmatic styles fits specific external contexts and also allows for a progression of increasingly concrete designs.

The RAISE language, method and tools have been developed as collective efforts in the ESPRIT RAISE (315) and LaCoS (5383) projects. Part I of the book was written by Klaus Havelund, and part II of the book was written by Anne E. Haxthausen and Klaus Havelund.

Acknowledgements

During the design of RSL many people adviced, influenced and reviewed the design process.

We would like to thank the following external advisors and consultants: *A. Blikle*, who gave us good advice in the early days of the RAISE project, in particular on type (domain) theories and logic, and introduced us to the ideas of `MetaSoft`; *M. Broy* and *C.B. Jones*, who, as consultants to the project, reviewed early to mid-term RAISE and RSL research and development, and gave much appreciated insight and advice; *D. Sannella* and *A. Tarlecki*, who gave lectures and advice on modularity concepts and type (domain) models, during the mid-term RAISE project; *A. Hill* and *H. Langmaack*, who, as external reviewers assigned by the CEC, followed the RAISE project throughout, and carefully scrutinized the progress of the project.

In addition to the RAISE Language Group, many people involved in the RAISE project directly or indirectly contributed to the design of RSL: *Erik Meiling*, *Ida Ørding Hansen*, *Holger Morell Heerfordt*, *Jesper Jørgensen*, *Steen Lynenskjold*, *Ole Frost Mikkelsen*, *Leif Sandegaard Nielsen*, *Mogens Nielsen*, *Ole N. Oest*, *Steen U. Palm*, *Jan Storbank Pedersen*, *Peter Sestoft*, *Harald Søndergaard* (with DDC/CRI); *Peter Olsen*, *Theodor Norup Petersen* (with NBB/SYPRO); *Tim Denvir*, *Tony Evans*, *Patrick Goldsack*, *David Grosvenor*, *Mel I. Jackson*, *Hamid Lesan* and *Roger Shaw* (with STC Technology Ltd).

Final editing of this book was done by Dines Bjørner, Chris George and Søren Prehn.

CHAPTER 1

Introduction

The construction of computer-based systems is a rather young profession. It is concerned firstly with evolving techniques, tools and engineering practices in order to cope with and exploit the underlying, rapidly evolving hardware technology. Additionally, very basic techniques and methods for specifying, developing, operating and maintaining computer software systems are still being developed.

As in any field of engineering, it turns out that construction of large and complex systems is very difficult and requires a complementary set of techniques to be deployed. In fields of engineering such as mechanical engineering and electrical engineering, mathematical techniques based on geometry, calculus and complex function theory have been developed and put to invaluable industrial use for more than two hundred years.

Mathematical techniques for specification, development and verification of software systems, often termed formal methods, are now coming into use for the construction of real systems. First of all, suitably mature theories and formalisms are now being turned into usable techniques. Secondly, a growing number of (young) professionals have been taught such techniques. Thirdly, there is a growing insistence that such techniques be used, in particular for the development of systems whose function or malfunction may seriously affect lives, property and society.

This book is about such mathematical techniques. It teaches a mathematically based notation, the RAISE Specification Language, which is useful for formal specification, design and development of software. RSL is, we believe, the most versatile and comprehensive language of its kind available today. It includes many of the ideas on formal methods that have been researched and discussed over the last two decades, in a unified framework. RSL permits abstract, property-oriented specification of sequential as well as concurrent systems. RSL permits specifications and designs of large systems to be modularized and permits separate subsystems to be separately developed. RSL also permits low-level operational designs to be expressed, to a level of detail from which extraction of final code is straightforward. That is: most of the construction of a system, from specification to design, may be done using one and the same formalism, thus facilitating precise, mathematical

arguments for correctness of development steps and of other critical properties.

The book has two main parts: an RSL Tutorial, and an RSL Reference Description. The Tutorial is intended as an introduction to RSL. Readers with a working knowledge of contemporary programming languages may use the Tutorial for self-study. Also, the Tutorial could be used in study-groups, or as text book in classes. The Reference Description is intended for use when reading or writing RSL: for looking up precise definitions of constructs in RSL.

This book does *not* teach formal development and proofs. These areas will be covered in a subsequent volume on the RAISE Method.

1.1 Structure of the Book

Part I, chapters 2–33, is the RSL Tutorial. This part covers RSL in a manner suitable for sequential reading.

Part II, chapters 34–48, is the RSL Syntax and Semantics Reference Description.

Part III consists of five appendices. Appendix A contains a syntax summary for RSL. The syntax summary may be used as an overview of available constructs, as well as a quick reference on how to write specific constructs. Appendix B describes the precedence and associativity of RSL combinators and operators. Appendix C describes lexical matters (microsyntax) for RSL. Appendix D contains an extensive bibliography of the technical and scientific literature that influenced the design of RSL. Appendix E contains an Index. The index lists, in four groups, all RSL symbols, the RSL literals, syntax categories (i.e. non-terminals of the syntax) and important concepts and, finally, the examples of the Tutorial part.

The reader is well advised to study the contents listing of the book carefully. The entire book is systematically organized with respect to the structure and facilities of RSL; hence the contents listing may serve as an index to 'areas' of RSL.

1.2 The RAISE Background

RAISE stands for Rigorous Approach to Industrial Software Engineering, and was initially coined as the name of an industrial R & D project carried out in the context of the CEC ESPRIT programme during the years 1985 through 1990.

The aims of the RAISE project were to develop notation, techniques and tools that would enable industrial usage of 'formal methods' in the construction of software systems. The project was motivated by early successes on projects employing methods such as the Vienna Development Method (VDM), as well as by a realization of the problems and relative immaturity of such methods.

The results of the RAISE project include RSL, as well as a large number of techniques and strategies for doing formal development and proofs (together termed the RAISE Method). Also, the RAISE project resulted in comprehensive tool support, which has by now been further developed into commercially supported products.

The RAISE project was carried out by four companies: Dansk Datamatik Center (DDC), which was taken over by Computer Resources International (CRI) in 1988; STC Technology Ltd (STL) (now incorporated in BNR Europe Limited), Nordisk Brown Boveri (now SYPRO) and ICL. DDC/CRI and STL/BNR were the main developers of RAISE, while NBB/SYPRO and ICL were the main industrial trialists.

1.3 The LaCoS Continuation

The LaCoS ESPRIT project, planned for the period 1990–1995, serves to further refine the RAISE technology. An important aspect of the project is a series of real-life industrial applications of RAISE. The *producer* partners of LaCoS, who provide and refine RAISE technology, are: CRI, BNR Europe and SYPRO. The *consumer* partners, who apply RAISE and report critically on RAISE technology are: Bull (F), MATRA Transport (F), INISEL Espacio (E), SSI (I), Technisystems (H), and Lloyd's Register of Shipping (UK).

The LaCoS (Large scale Correct Systems using formal methods) project is now well underway.

1.4 Related Work

RAISE builds upon ideas reflected in a number of formal methods and specification languages.

As a precursor to the RAISE project, DDC/CRI and STL carried out the 'Formal Methods Appraisal' study ([47]) in 1983. That study reviewed a large number of approaches, in order to identify the best possible basis for the RAISE project. The most important conclusion of that study was that so far the model-oriented approaches, in particular VDM, had proven most viable for industrial usage, and that the RAISE project would be well-advised to build on such a basis. Another important conclusion was, however, that the meta-language of VDM, colloquially called Meta-IV, was insufficient. In particular Meta-IV did not deal with structuring and concurrency. Structuring was found to be better addressed in 'algebraic specification languages'. Also, these languages enable a different level of abstraction to that possible with model-oriented languages. This difference is found in, for example, the (model-oriented approach to) concrete types compared with the (algebraic approach to) abstract types. For concurrency, it was concluded that the most mature approaches which were likely to blend well with the general ideas of methods like VDM were process algebras like CCS and CSP, based on synchronous communication.

The task facing the designers of RSL was to construct a unifying framework in which the basic features of VDM could be extended with facilities for property-oriented specification, structuring and concurrency.

A mid-term report on this design process is given in [46], reflecting an early version of RSL ([43]).

We now review briefly various features of RSL and RAISE with respect to predecessor methods, languages and ideas.

1.4.1 The Vienna Development Method

The VDM has continued to develop ever since it was conceived in the early seventies, and the VDM meta-language has been used in a number of different versions ([6, 29, 30, 31]). In the very early phase of the RAISE project, an attempt was made to consolidate the different versions of VDM notation ([25]). Several aspects of VDM notation which are more or less universally agreed on in the VDM community were included in early as well as final versions of RSL: type constructors for mappings, sets, lists, Cartesian products, etc., together with the applicative forms of expressions and function definitions (including pre-/post- style).

A substantial body of case studies using VDM has been published ([9, 32, 8, 11, 5, 10, 1]). Together with more systematic accounts of the method aspect of using VDM ([7, 31, 3, 4]) they present a large collection of techniques and strategies for modelling, specifying and developing software; techniques and strategies which may all readily be used with RSL.

1.4.2 Property-oriented Specifications

In 'algebraic specifications languages', such as ACT ONE, Clear, OBJ and Larch, the possible denotations of type and value names are constrained by means of axioms. In the usual algebraic specification style, individual axioms may affect the possible denotations of both type and value names: the structure of the denotation of a type name ('sort') is implicitly derived from the axioms. Also, the use of axioms, in contrast to explicit definitions of types and values (including functions) may lead to under-specification (looseness).

An important realization during the design of RSL was that, since sorts eventually have to be implemented, the possible denotations should be drawn from types which could be explicitly defined (as in VDM). In this manner, property-oriented specifications may easily be linked formally to model-oriented designs. Moreover, by combining the two techniques, intermediate techniques may be obtained: some types in a specification may be just sorts while others may be explicitly defined; functions may be (mutually) (under-)specified by axioms, although they may apply to values of explicitly defined types. Early formalizations of these ideas are given in [27].

1.4.3 Modularity Constructs

The facilities for structuring specifications in RSL reflect awareness of [45, 44] and are closely related to those found in other languages such as Clear, ACT ONE, ASL, OBJ, Larch and Standard/Extended ML [16, 51, 20, 13, 24, 23, 21, 19, 14, 22, 48, 42].

In RSL there are two kinds of 'modules': objects and schemes. A scheme represents a (possibly parameterized) class of models while an object represents either a single model belonging to a specified class or an array of such models.

The schemes, classes and objects of RSL are like the functors, signatures and structures of Standard ML ([42]) and Extended ML ([48]), except in their treatments of results and sharing. The schemes of RSL take objects as parameters and return classes as results; they achieve sharing between two parameters by ensuring that the classes of the parameters are applications of schemes to shared objects. The functors of Standard ML and Extended ML take structures as parameters and return structures as results; they achieve sharing between two parameters by imposing explicit sharing constraints.

The operations on classes are similar to the specification-building operations in algebraic specification languages. Basic class expressions in RSL correspond to theory presentations (signature + axioms) in algebraic specification languages. Other operations such as 'hide' and 'rename' are special cases of 'derive' in Clear and ASL. 'Extend' corresponds to 'enrich' in Clear except for the sharing properties. (However, the use in RSL of schemes which take objects as parameters can require the declaration of extra objects to provide actual parameters.)

The use of objects, and of the access **any** qualified by object names, allows interference between procedures to be delimited succinctly in specifications without the use of detailed states. The resulting style of specification has as its nearest equivalent the 'object-oriented' style of COLD-K ([17, 33]), which goes further by permitting dynamic object creation, using dynamically varying types.

1.4.4 Sequencing and Concurrency

Early versions of RSL ([43]) included constructs for expressing side-effects on a state, and communication on channels. The imperative facet was based on a simplified version of the imperative features available in 'Danish VDM' ([25]), while the concurrent facet was closely modelled on CSP ([28]). Although sequential and concurrent constructs could easily refer to types and values, irrespective of whether these were defined directly or in terms of axioms, sequential and concurrent constructs came with their own sets of specification means; for example, processes could be implicitly defined by means of trace assertions, or explicitly in terms of 'CSP programs'. Thus, the applicative, sequential and concurrent facets were, in fact, rather disjoint.

A tighter integration of the facets of the language was achieved in [35] by emphasizing the role of equational axioms. Handling side-effects in equational axioms involved introducing equivalences between the effects of expressions (not just equalities between the results of expressions). Post-conditions for partial correctness (but not for total correctness) can be defined in terms of equivalence.

To extend the use of equational axioms to concurrent processes, interlocking was introduced in [36]. This can be a powerful means of specifying properties of processes in terms of their mutual behaviour, although the proof rules governing its

interactions with concurrency are complicated. As interlocking is difficult to use in the presence of 'silent transitions', the binary non-deterministic choice of CSP ([28]) was adopted instead of the unary non-deterministic choice of CCS ([39, 40, 41]). However, using the techniques of [15], concurrency was taken to be essentially that of CCS, not that of CSP, in order to capture intuitions about synchronization, communication and deadlock. The combination of the binary non-deterministic choice of CSP with the concurrency of CCS is satisfactory in theory but in practice leads to formal manipulations that are more intricate than those in CSP or CCS.

This integration between the facets of the language does more than make imperative specifications analogous with applicative ones: it formalizes the relation between specification in different styles. In particular, when an imperative specification is analogous with an applicative one, then (under certain weak conditions) the imperative specification (or, strictly, a conservative extension of it) implements the applicative one and thereby provides an interpretation of it. The proof that this is so depends on the ability to pass imperative functions as parameters in RSL.

The treatment of all functions as values (including imperative functions that may access variables and processes that may also communicate on channels) means that one can quantify over them as well as pass them as parameters and return them as results. In particular, it is possible to state induction axioms in the language.

1.4.5 Development Relations

During the development of a software system, many aspects may be recorded formally, using RSL, and then relations between the various, individual RSL documents may themselves be recorded using RSL. However, the notion of implementation relation has a special status.

The formal implementation relation defined for RAISE ensures that an implementation can be substituted for its specification. As such, the implementation relation is crucial for separate development: the specification of a subsystem which is to be separately developed is a contract between the users and producers of the system, and the users will expect the implementation to be substituted for the specification at system integration time.

A class expression signifies a theory and denotes a class of models. In this respect RSL resembles ASL ([51]) and Extended ML ([49]). However, the implementation relation is more limited than that of ASL and Extended ML. In RAISE one class expression implements or refines another if every provable consequence of the latter is a provable consequence of the theory of the former. This is equivalent to subclassing of models, but only on the assumption that equality is not implementable using some other function or relation. This choice of implementation relation allows reasoning to be conducted using the proof rules rather than the semantics. In particular it allows the assertion of an implementation relation between two class expressions to be expanded into a finite collection of axioms about the implementing theory. However, it requires behavioural equivalence to be treated by the use of explicit abstraction functions and relations.

It would be possible to make the RAISE implementation relation subsume behavioural equivalence by allowing equalities to be implementable using functions or relations. Doing this in a logic like that of RAISE would still permit the horizontal and vertical composition of theories ([34, 50]). However, allowing equalities to be implementable would add considerable complications to the proof rules. For example, the property that the implementation relation can be expressed as a finite collection of axioms would be lost.

1.4.6 Other RAISE Documents

The formal, mathematical description of RSL is covered in the following documents: [37], which contains a set of proof rules which axiomatize RSL, and [38] which provides a mathematical model. The earlier document [26] provides a simplified, but more approachable mathematical model.

[12], the RAISE Method Manual, provides a large number of techniques and strategies for doing development and proof in RAISE.

[18], the RAISE Justification Handbook, provides a collection of proof rules presented in a style suited for doing justifications. The rules are based on those in [37].

A sequel volume to this book on the RAISE Method will be based on [12] and [18].

Part I

RSL Tutorial

CHAPTER 2

Introduction to Tutorial

This tutorial presents the RAISE Specification Language, which is a wide-spectrum language for specifying and designing software systems.

The term 'wide-spectrum' refers to the fact that RSL allows for abstract property oriented specification styles as well as for concrete algorithm oriented styles. These different styles make it possible to do refinement within RSL. That is, one may write an abstract RSL specification suited for human reading, and then in one or more steps refine it into a concrete RSL specification suited for translation into some programming language.

In this tutorial RSL is presented in a way that as far as possible avoids forward references; it tries to start with the more simple and generally familiar concepts and builds on them towards the more complex and less familiar.

Chapters 3–17 present applicative features. The basic concepts are those of a function and function composition. A function is basically a simple mapping from values of one type to values of another type. Functions in this sense are well known from mathematics. Note that the concepts of value and type are central. Examples of values are integers $1, 2, ...$, but also functions themselves are values. A type contains a set of values of the 'same shape'. RSL provides value expressions for representing values and type expressions for representing types.

Chapters 18–23 present sequential imperative features. The basic concepts are those of a variable, variable assignment and sequential composition. 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. Value expressions that change the value of a variable are said to have side-effects on that variable.

Value expressions with side-effects may be composed sequentially, meaning that they are to be evaluated in a specified order. This order of evaluation is of course important since changing the order may give different side-effects.

Chapters 24–27 present concurrency. The basic concepts are those of a channel, channel communication and concurrent composition. A channel is a medium along which values of a particular type can be communicated. Value expressions can

be composed to be evaluated in parallel, in which case they may communicate with each other through channels. The basic channel communication primitives are 'send a value to a channel' and 'receive a value from a channel'.

Chapters 28–33 present modules. The basic concepts are those of class expressions, schemes and objects. Class expressions are collections of declarations and represent classes of models. Schemes are named class expressions, possibly parameterized. Objects are named instances (or arrays of instances) of class expressions, so they represent single models (or arrays of single models). Schemes and objects, together termed modules, allow for the composition and reuse of specifications.

CHAPTER 3

Some Basic Concepts

This chapter 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 in subsequent sections. The annotations introduce the basic concepts as they occur in the specification.

3.1 Specification of an Election Database

3.1.1 Informal Requirements

Consider the following requirements for an election database.

The database is supposed to support the administration of an election by identifying all the people who are currently registered as voters.

The database must provide the following functions:

1. *register*: Registers a person in the database.
2. *check*: Checks whether a person has been registered in the database.
3. *number*: Returns the number of people currently registered in the database.

3.1.2 Formal Specification

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

```
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
```

That is, an RSL module captures the types, the values, and the axioms of some part of a system. Please read this module and relate it to the informal requirements presented in section 3.1.1, recalling that *number* is not yet formally specified.

The subsequent sections explain the contents of this module.

3.2 Modules

In general a module definition has the form:

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

for $n \geq 0$, where a declaration begins with a keyword (**type**, **value**, **axiom**) indicating the kind of declaration to come, followed by one or more definitions of that kind, separated by commas.

The example 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.

Here *empty* is a constant *Database* value. *register* is a function value from *Person*s and *Database*s to *Database*s and *check* is a function value from *Person*s and *Database*s to Booleans.

The module concept in its full power will be explained in chapters 28–33 of this tutorial.

3.3 Type Declarations

A type is a collection of logically related values. Some types are already built-in, i.e. predefined within RSL. An example of a built-in type is **Nat**. It contains all the natural numbers represented by the literals: 0,1,2, ...

In addition to the built-in types, one is allowed to define one's own types.

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

type
type_definition$_1$,
⋮
type_definition$_n$

for $n \geq 1$. In our example specification there are two such definitions.

The first type definition has the form:

id

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

Each type is associated with an equality operator = and an inequality operator $\neq$, which are applicable to values of the type.

The fact that *Person* is defined as an abstract type reflects the requirements, where no information is given about how people are identified in terms of their name and the like. We abstract 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 has 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 people. The type operator **-set** when applied to the type *Person* gives a new type containing as values all (finite) subsets of the set of values in *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 since the order of registration is irrelevant and no person may be registered more than once.

3.4 Value Declarations

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

value
value_definition$_1$,
⋮
value_definition$_n$

for $n \geq 1$. In our example specification there are three such definitions.

A value definition has in the simplest case the form:

id : type_expr

That is, the identifier *id* is defined to represent a value within the type represented by the type expression. Such a definition is sometimes referred to as a value signature. We often for convenience say that such a value definition 'defines the value *id*' instead of saying that it 'defines the identifier *id* to represent a value'.

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

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

The second value definition defines the function *register* that adds a person to the database. Suppose we want to register the person *Hamid* in a database *db*, then:

register(Hamid,db)

represents the database after having made the registration.

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

Person × Database → Database

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

(Person × Database) → Database

The type operator × (Cartesian product) is thus applied to the pair *Person* and *Database*, and the type operator → (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 $p : Person$ and $db : Database$.

The third value definition defines the function *check*, that, when applied to a person and a database, returns a Boolean value within the built-in type **Bool**. This type contains two values represented by the literals **true** and **false**. The function is supposed to return **true** if and only if the person is registered in the database.

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

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

3.5 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 people). Remember that the type of *empty* is *Person*-**set**.

Note the use of the symbol ≡ (equivalence) instead of = (equality). These two operators have the same meaning in applicative contexts, but their meanings are fundamentally different in sequential imperative and concurrent contexts (see section 18.7 and section 25.2). For reasons of consistency the ≡ symbol will normally be used as the outermost comparing operator in axioms.

The first axiom states that two value expressions are equivalent, namely *empty* and {}. A value expression evaluates to a value. The term 'expression' will in the rest of the tutorial be used as short for 'value expression', when no confusion arises from doing so. 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 and s_2 to be the same.

In fact the whole axiom:

empty ≡ {}

is itself an expression of type **Bool**. All axioms are Boolean expressions which, by definition, must evaluate to **true**.

An axiom declaration (in the simplest case) thus has the form:

axiom
value_expr$_1$,
⋮
value_expr$_n$

for $n \geq 1$. The second axiom in our example 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 people *p* and for all databases *db*, *register* applied to the pair (*p*, *db*) must be equivalent to $\{p\} \cup db$.

The third axiom defines the function *check*. A person is registered if that person belongs to the set representing the database.

The collection of axioms is complete in the sense that for each value identifier the axioms state exactly what value within its type each identifier represents. Thus, for example, *empty* is defined to represent nothing but the empty set. Likewise, the function *register* represents the one and only function that adds its first argument to its second argument.

Axioms do not, however, need to be complete. The ultimate extreme is the situation where there are no axioms at all. In that case the value identifier may represent any value within its type.

An example of an incomplete axiom is the following. Suppose that we want to respecify the function *check* in such a way that it returns **true** for any person *p* and database *db* if *p* is in the database and some additional, yet unknown, condition is satisfied. This can be expressed with the following axiom:

axiom ∀ p : Person, db : Database • check(p,db) ⇒ p ∈ db

The axiom says that for all people *p* and for all databases *db*, if the predicate

check(*p*, *db*) holds, then $p \in db$.

This axiom is incomplete since several *check* functions within *Person* × *Database* → **Bool** satisfy it. One such function is the one originally specified by the axiom:

axiom ∀ p : Person, db : Database • check(p,db) ≡ p ∈ db

Another function is one that satisfies the following axiom:

axiom ∀ p : Person, db : Database • check(p,db) ≡ p ∈ db ∧ old_enough(p)

Here the function *old_enough* is supposed to return **true** if a person is old enough to vote, according to some rule. It must have the following type:

value old_enough : Person → **Bool**

An identifier that is not completely specified through the axioms is said to be under-specified.

Axioms may be named for documentation purposes and for reference in justifications. The axioms defining *empty*, *register* and *check* can for example be written as follows, where axiom names bracketed with [and] precede the axioms:

axiom
  [empty_axiom]
    empty ≡ {},
  [register_axiom]
    ∀ p : Person, db : Database • register(p,db) ≡ {p} ∪ db,
  [check_axiom]
    ∀ p : Person, db : Database • check(p,db) ≡ p ∈ db

The three axioms have here been named *empty_axiom*, *register_axiom* and *check_axiom*. Axiom namings do not add to the properties of a specification.

The more generalized form of an axiom declaration now is:

axiom
  opt-axiom_naming$_1$ value_expr$_1$,
  ⋮
  opt-axiom_naming$_n$ value_expr$_n$

for $n \geq 1$. An *axiom_naming* has the form [id] and is optional, as indicated by the opt- prefix in the syntax.

3.6 Module Extension

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

ELECTION_DATABASE =
  extend DATABASE **with**

```
class
  value number : Database → Nat
  axiom ∀ db : Database • number(db) ≡ card db
end
```

The typical form of an extending module definition is:

```
id =
  extend id0 with
  class
    declaration1
    ⋮
    declarationn
  end
```

where id_0 is the name of some module.

The function *number*, when applied to a database, returns a natural number — the number of people registered in the database. The axiom defining *number* makes use of the cardinality (**card**) operator which is applicable to any finite set.

3.7 Combining Value and Axiom Declarations

RSL provides shorthands for combining value and axiom declarations when the axioms have particular forms. Two of these shorthands are described here, to illustrate the concept of shorthand.

First, the following value declaration containing an explicit value definition:

```
value id : type_expr = value_expr
```

is short for:

```
value id : type_expr
axiom id ≡ value_expr
```

The expression *value_expr* must evaluate to a value within the type represented by *type_expr*. As an example, we could have defined the constant *empty* as follows:

```
value empty : Database = {}
```

Second, the following value declaration containing an implicit value definition:

```
value id : type_expr • value_expr
```

is short for:

```
value id : type_expr
axiom value_expr
```

The expression *value_expr* must be a Boolean expression. As an example, we could have defined the constant *empty* as follows:

```
value empty : Database • empty = {}
```

Many other kinds of shorthands are provided, as will be described throughout the tutorial.

3.8 Comments in Specifications

A specification may include comments, where a comment is a piece of text that has no influence on the formal meaning of the specification. A comment is enclosed with the symbols /∗ and ∗/, and comes in front of a definition, be it a type definition, a value definition or an axiom definition.

Below follows our *ELECTION_DATABASE* module with an added comment.

```
ELECTION_DATABASE =
  extend DATABASE with
  class
    value
      /* The function 'number' returns the number of people registered */
      number : Database → Nat
    axiom ∀ db : Database • number(db) ≡ card db
  end
```

CHAPTER 4

Built-in Types

Three kinds of types have been presented: built-in types, like **Nat**, abstract types (sorts), like *Person*, and compound types, like *Database*, which was defined as containing sets of *Person*s.

In this chapter all the built-in types provided by RSL are described. These are represented by the type literals: **Bool** (Booleans), **Int** (integers), **Nat** (natural numbers), **Real** (real numbers), **Char** (characters), **Text** (texts) and **Unit** (the singleton type).

For each built-in type, the value literals and the operators associated with that type are described. Note that every type is associated with an equality operator = and an inequality operator ≠.

4.1 Booleans

The Boolean type literal **Bool** represents the type containing the two truth values **true** and **false**.

4.1.1 If Expressions

A Boolean expression *value_expr* can be used to choose between the evaluation of two alternative expressions *value_expr*$_1$ and *value_expr*$_2$ in an if expression of the form:

if value_expr **then** value_expr$_1$ **else** value_expr$_2$ **end**

If *value_expr* evaluates to **true** *value_expr*$_1$ is evaluated; if *value_expr* evaluates to **false** *value_expr*$_2$ is evaluated.

The two expressions *value_expr*$_1$ and *value_expr*$_2$ must have the same type which is also the type of the whole if expression.

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

if x > y **then** x − y **else** y − x **end**

RSL is a specification language in which the behaviour of programs running on computers can be specified. A characteristic of programs is that they may fail to terminate. Such non-termination, or more generally, chaotic behaviour, is represented by the RSL expression **chaos**.

An important property of if expressions is non-strictness in the sense that when applied to **chaos** they do not necessarily themselves evaluate to **chaos**. The following rules hold:

if true then value_expr **else chaos end** ≡ value_expr
if false then chaos else value_expr **end** ≡ value_expr

If, however, the condition of an if expression evaluates to **chaos**, then the if expression itself evaluates to **chaos**:

if chaos then value_expr$_1$ **else** value_expr$_2$ **end** ≡ **chaos**

Note that **chaos** can appear in positions where expressions of different types are expected. **chaos** is not, however, a member of any type, such as **Bool**. Types only contain the values of terminating expressions.

Often one may want to nest if expressions, as in the following example:

```
if x < 0 then x − 1
else
  if x > 0 then x + 1
  else 0
  end
end
```

A shorthand syntax allows us to avoid the nesting and instead to write the expression as follows:

```
if x < 0 then x − 1
elsif x > 0 then x + 1
else 0
end
```

A multiple if expression of the form:

```
if value_expr1 then value_expr1′
elsif value_expr2 then value_expr2′
⋮
elsif value_exprn then value_exprn′
else value_exprn+1′
end
```

for $n \geq 2$, is short for:

```
if value_expr1 then value_expr1′
else
  if value_expr2 then value_expr2′
```

```
    else
      ⋮
      if value_expr_n then value_expr_n'
      else value_expr_{n+1}'
      end
      ⋮
    end
  end
```

4.1.2 Prefix Connectives

No operators other than = and ≠ are defined on **Bool**. Instead, a number of connectives are defined, which together with their Boolean argument expressions are short for certain if expressions.

A Boolean expression *value_expr* can be negated with the connective ∼ (not):

∼ value_expr

This is read as 'not *value_expr*' and is short for:

if value_expr **then false else true end**

4.1.3 Infix Connectives

Two Boolean expressions *value_expr*$_1$ and *value_expr*$_2$ can be combined with any of the binary connectives ∧ (and), ∨ (or) and ⇒ (implies) as shown below. The resulting expressions are short for certain if expressions.

The expression:

value_expr$_1$ ∧ value_expr$_2$

is short for:

if value_expr$_1$ **then** value_expr$_2$ **else false end**

The expression:

value_expr$_1$ ∨ value_expr$_2$

is short for:

if value_expr$_1$ **then true else** value_expr$_2$ **end**

The expression:

value_expr$_1$ ⇒ value_expr$_2$

is short for:

if value_expr$_1$ **then** value_expr$_2$ **else true end**

As examples consider the following Boolean expressions which are tautologies (evaluating to **true**) for any integer x:

$(\mathrm{x} \leq 0) \vee (\mathrm{x} > 0)$

$\sim ((\mathrm{x} < 0) \wedge (\mathrm{x} > 0))$

$(\mathrm{x} > 0) \Rightarrow (\mathrm{x} \geq 1)$

The explanation of the Boolean connectives in terms of if expressions emphasizes that one must consider evaluation order when using the infix connectives. Consider for example the following expression where x and *epsilon* are real numbers:

$(\mathrm{x} \neq 0.0) \wedge (1.0/\mathrm{x} < \mathrm{epsilon})$

and suppose that $x = 0.0$. The evaluation of the constituent expression $1.0/x$ is under-specified since 'real division' has the pre-condition that the second argument must be different from zero (0.0). The expression $1.0/x$ where $x = 0.0$ might evaluate to **chaos**. Fortunately, with the interpretation of $\wedge$ in terms of an if expression, this constituent expression will never be evaluated when $x = 0.0$.

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

$(\mathrm{x} \neq 0.0) \wedge (1.0/\mathrm{x} < \mathrm{epsilon})$
$\equiv$ **if** $(\mathrm{x} \neq 0.0)$ **then** $(1.0/\mathrm{x} < \mathrm{epsilon})$ **else false end**
$\equiv$ **if** $(0.0 \neq 0.0)$ **then** $(1.0/0.0 < \mathrm{epsilon})$ **else false end**
$\equiv$ **if false then** $(1.0/0.0 < \mathrm{epsilon})$ **else false end**
$\equiv$ **false**

The logic obtained is a conditional logic where the second constituent expression is evaluated only if the value of the first constituent expression is insufficient to determine the value of the composite expression. In this way partiality of the second constituent expression (here due to 1.0/0.0) need not cause a problem since evaluation may be avoided.

The meanings of the Boolean infix connectives and in particular their meanings when applied to **chaos** are summarized in the following truth tables. The left column in each table indicates the left argument of the connective whilst the top row indicates the right argument.

$\wedge$	**true**	**false**	**chaos**
true	**true**	**false**	**chaos**
false	**false**	**false**	**false**
chaos	**chaos**	**chaos**	**chaos**

$\vee$	**true**	**false**	**chaos**
true	**true**	**true**	**true**
false	**true**	**false**	**chaos**
chaos	**chaos**	**chaos**	**chaos**

$\Rightarrow$	**true**	**false**	**chaos**
true	**true**	**false**	**chaos**
false	**true**	**true**	**true**
chaos	**chaos**	**chaos**	**chaos**

Note that $\wedge$ and $\vee$ are not commutative if their arguments do not terminate.

4.1.4 Quantifiers

The following expression is an example of a quantified expression:

$\forall$ x : **Nat** • (x = 0) $\vee$ (x > 0)

and it reads: 'for all natural numbers x, either x is equal to 0 or x is greater than 0'.

The quantifier $\forall$ binds the identifier x, and the x immediately following $\forall$ is therefore called a binding. The x : **Nat** is called a typing.

We say that x is 'bound' within the quantified expression. On the other hand, x is 'free' within the expression:

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

A quantified expression has the general form:

quantifier typing$_1$,...,typing$_n$ • value_expr

for $n \geq 1$. The expression *value_expr* must be of type **Bool**. A quantifier is one of the following:

$\forall$, $\exists$, $\exists!$

read as 'for all', 'there exists' and 'there exists exactly one'.

A typing in the simple case has the form:

id$_1$,...,id$_m$: type_expr

for $m \geq 1$. Some more examples of quantified expressions which evaluate to **true** are:

$\exists$ x : **Nat** • x > 99
$\forall$ x,y : **Nat** • $\exists!$ z : **Nat** • x + y = z
$\exists$ x,y : **Nat**, b : **Bool** • b = (x = y)

Note that the scope of quantifiers extends to the end of any expression following the •. For example:

$\forall$ x : **Nat** • (x = 0) $\vee$ (x > 0)

is equivalent to:

$\forall$ x : **Nat** • ((x = 0) $\vee$ (x > 0))

and not to:

($\forall$ x : **Nat** • (x = 0)) $\vee$ (x > 0)

This is decided by the relative precedence of ∀ and (in this case) ∨. There is more on precedence in section 4.8.1.

4.1.5 Axiom Quantifications

Sometimes several axioms within an axiom declaration are quantified over a common set of value names as is the case in our election *DATABASE* module (section 3.1.2):

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

The axioms get somewhat clumsy due to the repeated quantifications. One may instead make an axiom quantification as follows:

axiom forall p : Person, db : Database •
  empty ≡ {},
  register(p,db) ≡ {p} ∪ db,
  check(p,db) ≡ p ∈ db

Note the commas separating axioms. This corresponds to the following single axiom:

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

but from a presentational point of view it has the advantage of containing three clearly separated axioms which may be given individual names.

An axiom declaration of the form:

axiom forall typing_list •
  opt-axiom_naming$_1$ value_expr$_1$,
  ⋮
  opt-axiom_naming$_n$ value_expr$_n$

for $n \geq 1$, is short for:

axiom
  opt-axiom_naming$_1$ ∀ typing_list • value_expr$_1$,
  ⋮
  opt-axiom_naming$_n$ ∀ typing_list • value_expr$_n$

4.2 Integers

The integer type literal **Int** represents the type containing the negative as well as non-negative whole numbers, i.e. integers:

...,−2,−1,0,1,2,...

4.2.1 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 argument is returned. The operator is the identity on non-negative numbers. The result is a natural number (section 4.3).

Some examples are:

abs −5 = 5
abs 5 = 5

4.2.2 Infix Operators

A number of binary infix operators are defined on integers.

There are the relational operators 'greater than', 'less than', 'greater than or equal to' and 'less than or equal to':

$>$: **Int** × **Int** → **Bool**
$<$: **Int** × **Int** → **Bool**
$\geq$: **Int** × **Int** → **Bool**
$\leq$: **Int** × **Int** → **Bool**

Some examples are:

$5 > 2$
$1 \leq 1$

There are the four arithmetic operators for addition, subtraction, multiplication and integer division:

$+$: **Int** × **Int** → **Int**
$-$: **Int** × **Int** → **Int**
$*$: **Int** × **Int** → **Int**
$/$: **Int** × **Int** $\xrightarrow{\sim}$ **Int**

Note that the integer division operator returns an integer, the absolute value of which is the whole number of times that the absolute value of the second argument divides into the absolute value of the first. The sign of the result is the product of the signs of the arguments.

The integer division operator is partial, in that its result is under-specified if the second argument is zero. A special type operator $\xrightarrow{\sim}$ is provided for generating the type of partial functions. Chapter 7 will describe the function type operators in more detail.

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:

$\backslash : \mathbf{Int} \times \mathbf{Int} \xrightarrow{\sim} \mathbf{Int}$

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, if b is not zero:

a = (a/b)*b + (a\b)

Some examples are:

$5 \setminus 2 = 1$
$5 \setminus -2 = 1$
$-5 \setminus 2 = -1$
$-5 \setminus -2 = -1$

There is finally the exponentiation operator:

$\uparrow : \mathbf{Int} \times \mathbf{Int} \xrightarrow{\sim} \mathbf{Int}$

which raises the first integer to the power of the second integer.

The integer exponentiation operator is partial in that its result is under-specified if the second argument is negative or if both arguments are zero.

An example is:

$3 \uparrow 2 = 9$

4.3 Natural Numbers

The natural number type literal **Nat** represents the type containing the non-negative integers:

0,1,2,...

The natural number type is a subtype (chapter 11) of the integer type. Consequently, all the integer operators are defined for natural numbers. (Though the results may not be natural numbers, as in '$1 - 5$'.)

4.4 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. Not all real numbers are representable by literals as one cannot write literals with infinitely many decimals (digits to the right of the decimal point).

4.4.1 Conversion Operators

The integer type is not a subtype (chapter 11) 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 4.2) and another set of operators is defined for the reals (see below). There is for example both an 'integer addition' operator and a 'real addition' operator.

Since the two types are separated and since there will sometimes be a need in calculations to switch from one type to the other, two conversion operators 'real to integer' and 'integer to real' are defined:

int : **Real** → **Int**
real : **Int** → **Real**

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

Some examples are:

int 4.6 = 4
int −4.6 = −4
real 5 = 5.0
real((**int** 5.2)/2) = 2.0

4.4.2 Other Prefix Operators

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

abs : **Real** → **Real**

4.4.3 Infix Operators

A number of binary infix operators are defined on real numbers. They correspond to the similar infix integer operators:

$>$: **Real** × **Real** → **Bool**
$<$: **Real** × **Real** → **Bool**
$\geq$: **Real** × **Real** → **Bool**
$\leq$: **Real** × **Real** → **Bool**

$+ : \mathbf{Real} \times \mathbf{Real} \to \mathbf{Real}$
$- : \mathbf{Real} \times \mathbf{Real} \to \mathbf{Real}$
$* : \mathbf{Real} \times \mathbf{Real} \to \mathbf{Real}$
$/ : \mathbf{Real} \times \mathbf{Real} \xrightarrow{\sim} \mathbf{Real}$
$\uparrow : \mathbf{Real} \times \mathbf{Real} \xrightarrow{\sim} \mathbf{Real}$

Note that the real division operator performs arithmetic division without truncation.

The real exponentiation operator is partial in that its result is under-specified if the first argument is zero and the second argument is not positive, or if the first argument is negative and the second argument is not a whole number.

4.5 Characters

The character type literal **Char** represents the type containing the characters:

$'\mathtt{A}', '\mathtt{B}', ..., '\mathtt{a}', '\mathtt{b}', ...$

Note that a character begins and ends with $'$.

4.6 Texts

The text type literal **Text** represents the type containing strings of characters. A text begins and ends with the symbol $''$ and has the general form:

$''c_1 \ ... \ c_n''$

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

Some examples are:

$''\mathtt{this\ is\ a\ text}''$
$''\mathtt{Formal\ Methods}''$
$''''$

In chapter 9 more will be said about texts.

4.7 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 tutorial.

4.8 Precedence and Associativity

4.8.1 Precedence

To avoid having to write too many brackets in expressions, operators and some other symbols like ∀ are given *precedences*. Increasing precedence means that things bind more closely together. For instance, ∗ in RSL has a higher precedence than + (as is common in arithmetic), so that:

$$x + y * z \equiv x + (y * z)$$

In other words, either of these expressions may be written in RSL, and they are equivalent, but the brackets in the latter are unnecessary.

Similarly, the symbols +, =, ≡ and ∀ are in decreasing order of precedence, so:

$$\forall\, i : \mathbf{Int} \bullet i = 0 \equiv i + 1 = 1$$

is equivalent to:

$$\forall\, i : \mathbf{Int} \bullet ((i = 0) \equiv ((i + 1) = 1))$$

The precedence rules are designed to make brackets unnecessary as often as possible. A table of them can be found in appendix B.

4.8.2 Associativity

Precedence deals with expressions involving different operators; what happens when they are the same? For example, is $x - y - z$ equivalent to:

$$x - (y - z)$$

or

$$(x - y) - z$$

The answer is given by associativity rules, also given in the table in appendix B. Here − is defined to associate to the left, i.e. the latter form with the brackets on the left is the correct one. Typically the arithmetic operators (like −) associate to the left and the others to the right. For instance:

$$x \Rightarrow y \Rightarrow z \equiv x \Rightarrow (y \Rightarrow z)$$

Where the table gives no associativity it is because the construct would be ill typed however it were bracketed, like:

$$s \subseteq s' \subseteq s''$$

or because there is no standard convention for what it means, like:

$$x\uparrow y\uparrow z$$

which must be written either:

$$x\uparrow(y\uparrow z)$$

or

$$(x\uparrow y)\uparrow z$$

CHAPTER 5

Products

A product is an ordered finite collection of values of possibly different types. Examples of products are:

(1,2)
(1,**true**,″`John`″)

The first product is a pair where the first value is 1 and the second value is 2. Note that one can speak about 'the first value', 'the second value', etc. The second product consists of three values, all of different types. Functions with n arguments are really functions of single products which have n component values.

5.1 Product Type Expressions

The Cartesian product type expression:

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

for $n \geq 2$, represents the type containing products of length n:

$(v_1,...,v_n)$

where each v_i is a value of type *type_expr*$_i$.

As an example consider the type expression:

Bool × **Bool**

The type represented by this 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**)
⋮

5.2 Product Value Expressions

An expression of the form:

(value_expr$_1$,...,value_expr$_n$)

for $n \geq 2$, evaluates to a product:

(v$_1$,...,v$_n$)

where v_i is the value of *value_expr*$_i$.

Some examples of product value expressions together with their types are:

(**true**, p ⇒ q) : **Bool** × **Bool**
(x + 1, 0, ″`this is a text`″) : **Nat** × **Nat** × **Text**

5.3 Example: A System of Coordinates

A system of coordinates provides a set of positions:

(x,y)

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

The distance between two positions is obtained by Pythagoras' theorem.

```
SYSTEM_OF_COORDINATES =
  class
    type
      Position = Real × Real
    value
      origin : Position,
      distance : Position × Position → Real
    axiom
      origin ≡ (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 — 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 of which is a pair of real numbers.

CHAPTER 6

Bindings and Typings

In this chapter we develop further the concepts of binding and typing. The concepts were briefly introduced in connection with quantified Boolean expressions (section 4.1.4) and they will be used extensively in later chapters. As an example, consider the quantified expression:

$\forall$ x : **Nat** • (x = 0) $\vee$ (x > 0)

The x : **Nat** following $\forall$ is a typing consisting of the binding x and the type expression **Nat**.

A typing is a generalized declarative construct for defining identifiers with associated types. In section 4.1.4 a typing was defined to have the simplified form:

id_1,...,id_n : type_expr (for n $\geq$ 1)

A typing can, however, have other forms. In general we distinguish between two kinds of typings: single typings and multiple typings, the latter being a short form of the former. In addition we can even have lists of typings. Later it is explained what single and multiple typings are, and how multiple typings and typing lists can be expanded into single typings.

First, however, we explain the fundamental concept of a binding.

6.1 Bindings

A binding is a structure of identifiers, possibly grouped by parentheses. Examples are:

x
(x,y)
(x,(y,z))

The purpose of bindings is to give names to values, extended to giving names to components of product values. That is, a value can be matched against a binding, resulting in a collection of definitions.

The matching of values against bindings takes place for example in a let expression, of which the following is an example:

let (x,y) = v **in** x + 1 **end**

The value v is matched against the binding (x, y) before the expression $x + 1$ is evaluated. Let us assume the following definition of the value v:

value v : **Int** × (**Bool** × **Bool**)
axiom v ≡ (1,(**true**,**false**))

The expression $x + 1$ in the let expression is evaluated in the scope of x and y where x is bound to 1 and where y is bound to (**true**, **false**). The result of the expression is therefore 2. Formulated otherwise, the expression $x + 1$ is evaluated within the scope of the following definitions obtained by matching v against (x, y):

value
  x : **Int**,
  y : **Bool** × **Bool**
axiom
  x ≡ 1,
  y ≡ (**true**,**false**)

Let expressions will be explained in more detail in chapter 14.

The value v can be matched against any of the bindings x, (x, y) and $(x, (y, z))$ given above. Matching v against the binding x corresponds to the definitions:

value x : **Int** × (**Bool** × **Bool**)
axiom x ≡ (1,(**true**,**false**))

Matching v against the binding $(x, (y, z))$ corresponds to the definitions:

value
  x : **Int**,
  y : **Bool**,
  z : **Bool**
axiom
  x ≡ 1,
  y ≡ **true**,
  z ≡ **false**

Note that matching a value against a binding does not generate the definitions in a syntactic sense. What happens is that within the scope of the binding, it is as if the definitions had been given.

A binding either has the form:

id

or (in the syntax recursively defined) the form:

$(\text{binding}_1,...,\text{binding}_n)$ (for $n \geq 2$)

6.2 Single Typings

A single typing has the form:

binding : type_expr

Examples of single typings are:

x : **Int** × (**Bool** × **Bool**)
(x,y) : **Int** × (**Bool** × **Bool**)
(x,(y,z)) : **Int** × (**Bool** × **Bool**)

Such a single typing associates identifiers with types in the obvious way. In the second single typing above, x is associated with **Int** while y is associated with **Bool** × **Bool**.

A single typing is thus a way of defining identifiers together with their types.

6.3 Multiple Typings

A multiple typing has the form:

binding_1,...,binding_n : type_expr

for $n \geq 2$. It is short for the single typing:

(binding_1,...,binding_n) : type_expr × ... × type_expr

where the product type expression has length n. An example of a multiple typing is:

y,z : **Bool**

This is short for the following single typing:

(y,z) : **Bool** × **Bool**

6.4 Typing Lists

We shall often meet typing lists of the form:

typing_1,...,typing_n

where $n > 1$. Such a typing list is expanded into a single typing in two steps.

First, the individual typings are expanded into single typings, thereby obtaining:

binding_1 : type_expr_1,...,binding_n : type_expr_n

Secondly, this list of single typings is expanded as one single typing, taking the form:

(binding_1,...,binding_n) : type_expr_1 × ... × type_expr_n

As an example, consider the typing list:

x : **Int**, y,z : **Bool**

This is expanded into a single typing as follows.

First, the individual typings are expanded into single typings, thereby obtaining:

x : **Int**, (y,z) : **Bool** × **Bool**

Secondly, this list of single typings is expanded as one single typing:

(x,(y,z)) : **Int** × (**Bool** × **Bool**)

CHAPTER 7

Functions

A function is essentially a mapping from values of one type to values of another type. Functions are central to the specification of a system. As an example, the activities within a system may be modelled as functions, such as for example the activity: 'register a person in a database'. This form of activity may be modelled by a function which, when applied to a pair consisting of a person and a database, returns an augmented database containing the person.

A function, say f, that maps values of a type T_1 to values of a type T_2 is total if for every value in T_1, f returns a unique value in T_2. A function is partial if there exists a value within T_1 for which f might not return a value in T_2 (i.e. might not terminate), or for which different applications of the function might return different values in T_2. (In the second case we say that the function is 'non-deterministic', see chapter 16.)

A function that is not known to be total is considered partial. Hence the total functions are included in the partial functions, and the term partial means 'not specified to be total' rather than 'specified to be non-terminating or non-deterministic'. For example, the integer and real division operators in RSL are partial because they are under-specified for division by zero.

The properties of functions may be defined in a variety of styles, with abstract property oriented styles at one end of the spectrum, and concrete algorithm oriented styles at the other.

7.1 Total Functions

A function type expression of the form:

type_expr$_1$ $\rightarrow$ 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:

f : type_expr$_1$ $\rightarrow$ type_expr$_2$

has the following property:

$\forall$ x : type_expr$_1$ • $\exists!$ y : type_expr$_2$ • f(x) $\equiv$ y

That is, for any value of the first type the function may be applied and will give a unique value in the second type. Functions may also be partial, see section 7.4.

We have already seen some examples of functions. In the election database (section 3.1.2 and section 3.6) we defined:

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

and in the system of coordinates (section 5.3) we defined:

value distance : Position × Position → **Real**

We have also seen how functions are applied:

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

A function is applied via an application expression of the general form:

value_expr(value_expr$_1$,...,value_expr$_n$)

for $n \geq 0$, where (for $n > 0$) *value_expr* represents a function of the type:

$T_1 \times ... \times T_n \to T$

and where each *value_expr*$_i$ is of type T_i. The result is of type T.

The following sections illustrate how functions may be defined using a variety of styles.

7.2 Definitions by Axioms

For the purpose of illustration we choose one example which we specify in several ways in order to show different possibilities. The function has the signature (name and type):

value fraction : **Real** → **Real**

and is supposed to return $1.0/x$ for any real number argument $x \neq 0.0$, and to return 0.0 for the argument $x = 0.0$. A first solution is:

axiom $\forall$ x : **Real** • fraction(x) $\equiv$ **if** x = 0.0 **then** 0.0 **else** 1.0/x **end**

Alternatively one could define the function through two axioms, one for the zero case and one for the non-zero case:

axiom
  fraction(0.0) $\equiv$ 0.0,

∀ x : **Real** • x ≠ 0.0 ⇒ (fraction(x) ≡ 1.0/x)

7.3 Explicit Definition of Total Functions

A shorter way of writing:

value fraction : **Real** → **Real**
axiom ∀ x : **Real** • fraction(x) ≡ **if** x = 0.0 **then** 0.0 **else** 1.0/x **end**

is:

value
fraction : **Real** → **Real**
fraction(x) ≡ **if** x = 0.0 **then** 0.0 **else** 1.0/x **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 associates the axiom more closely with the signature, a style typical in programming languages. This style can make it easier for a reader of a specification to detect the properties associated with a particular function identifier.

The explicit function definition is an instance of the form:

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

for $n \geq 1$, which is typically 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$_n$) ≡ value_expr

7.4 Partial Functions

A function type expression of the form:

type_expr$_1$ $\xrightarrow{\sim}$ type_expr$_2$

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

A function:

f : type_expr$_1$ $\xrightarrow{\sim}$ type_expr$_2$

is partial if there exists a value v : *type_expr*$_1$ such that f might fail to return a value when applied to v, i.e. might not terminate, or such that the result is non-deterministic. Non-termination is a common feature of programs and in RSL this is

also possible. As we shall see later, RSL provides a range of loop constructs which all can lead to non-termination. Non-determinism will be discussed in chapter 16.

The fact that the function f definitely does not terminate for some v : *type_expr*$_1$ can be written as follows:

f(v) ≡ **chaos**

where **chaos** is the non-terminating expression.

As an example, suppose that we make the *fraction* function partial at zero. That is, we do not specify how the function behaves for the argument $x = 0.0$. This can be done as follows:

value partial_fraction : **Real** $\xrightarrow{\sim}$ **Real**
axiom ∀ x : **Real** • x ≠ 0.0 ⇒ (partial_fraction(x) ≡ 1.0/x)

The axiom only defines the behaviour of the function for arguments different from zero. This is done by preceding the defining equivalence with a test for inequality with 0.0. The function may therefore evaluate to **chaos** when applied to 0.0. That is, one possible function satisfying the axiom is the one that also satisfies the following property:

partial_fraction(0.0) ≡ **chaos**

The original axiom does, however, not imply this property, so another function satisfying the original axiom may therefore instead satisfy the alternative property:

∃ r : **Real** • partial_fraction(0.0) ≡ r

Remember that the partial functions are simply those that are not known to be total, not those that are known to be non-terminating or non-deterministic. Hence many partial functions are potentially total because they are partial as a result of under-specification.

7.5 Explicit Definition of Partial Functions

A shorter way of writing:

value partial_fraction : **Real** $\xrightarrow{\sim}$ **Real**
axiom ∀ x : **Real** • x ≠ 0.0 ⇒ (partial_fraction(x) ≡ 1.0/x)

is:

value
  partial_fraction : **Real** $\xrightarrow{\sim}$ **Real**
  partial_fraction(x) ≡ 1.0/x
  pre x ≠ 0.0

This is an explicit function definition with a pre-condition following the keyword **pre**.

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

value partial_fraction : **Real** $\xrightarrow{\sim}$ **Real**

axiom ∀ x : **Real** • partial_fraction(x) ≡ 1.0/x **pre** x ≠ 0.0

where the expression following the • has the form:

value_expr$_1$ ≡ value_expr$_2$ **pre** value_expr$_3$

As will be described in section 18.8 such a conditional equivalence expression is short for:

(value_expr$_3$ ≡ **true**) ⇒ (value_expr$_1$ ≡ value_expr$_2$)

The above explicit definition of *partial_fraction* is an instance of the form:

value
id : type_expr$_1$ × ... × type_expr$_n$ $\xrightarrow{\sim}$ type_expr
id(id$_1$,...,id$_n$) ≡ value_expr$_1$ **pre** value_expr$_2$

7.6 Function Expressions

In the following, *fraction* is defined as the function represented by the function expression occurring on the right hand side of ≡:

value fraction : **Real** → **Real**
axiom fraction ≡ λ x : **Real** • **if** x = 0.0 **then** 0.0 **else** 1.0/x **end**

The function expression evaluates to a function. The form of a function expression, sometimes called a lambda abstraction, consists of a single typing (a binding and a type expression) and an expression:

λ binding : type_expr • value_expr

representing a function of type:

type_expr $\xrightarrow{\sim}$ T

where *T* is the type of *value_expr*. The *binding* must match the type represented by *type_expr*.

Some other examples are:

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

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

axiom
add ≡ λ (x : **Int**, y : **Int**) • x + y,

cond ≡ λ (b : **Bool**, x,y : **Nat**) • **if** b **then** x **else** y **end**

A function expression can in addition to the previous form also have the following form:

λ (typing$_1$,...,typing$_n$) • value_expr

where $n \geq 0$. This form can be expanded into the previous form using the rules presented in chapter 6 and as exemplified by the above axioms. That is, the typing list:

typing$_1$,...,typing$_n$

can be expanded into a single typing of the form:

binding : type_expr

So the function expression can be expanded into:

λ (binding : type_expr) • value_expr

and therefore into:

λ binding : type_expr • value_expr

The case where $n = 0$ in the 'typing list' form represents function expressions of the form:

λ () • value_expr

which are functions of type:

Unit $\xrightarrow{\sim}$ T

where T is the type of *value_expr*. Such a function expression can of course instead be written as:

λ dummy : **Unit** • value_expr

where *dummy* is then not referred to within *value_expr*. The () version, however, saves one from inventing a parameter name, and thereby from confusing the reader.

Functions with parameter type **Unit** are, however, primarily interesting when *value_expr* has side-effects, as will be described later in this tutorial.

7.7 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 : (**Int** $\xrightarrow{\sim}$ **Int**) → **Int** $\xrightarrow{\sim}$ **Int**
  twice(f) ≡ λ i : **Int** • f(f(i))

Function arrows associate to the right, so the type of *twice* could (equivalently) have been written in the following way:

twice : (**Int** $\xrightarrow{\sim}$ **Int**) → (**Int** $\xrightarrow{\sim}$ **Int**)

The function *twice* when applied to a function f returns a function (represented by the function expression) that when applied to an integer i applies f twice.

Some examples of *twice* applications are:

twice(λ i : **Int** • i + 1) = λ i : **Int** • i + 2
twice(λ i : **Int** • i + 1)(1) = 3

Note that *twice* can be applied to one argument, as in the first example, or to two, as in the second.

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

We do not need to use a function expression to define *twice*. We can also write an axiom like:

axiom ∀ f : **Int** $\xrightarrow{\sim}$ **Int**, i : **Int** • twice(f)(i) ≡ f(f(i))

In yet another way of defining *twice* we could use the built-in operator ° for function composition. For arbitrary types T_1, T_2 and T_3 this has the type:

° : ($T_2 \xrightarrow{\sim} T_3$) × ($T_1 \xrightarrow{\sim} T_2$) → $T_1 \xrightarrow{\sim} T_3$

The result of composing two functions f_1 and f_2 is defined as follows:

$f_1 \circ f_2 = \lambda x : T_1 \bullet f_1(f_2(x))$

Our axiom for *twice* would then be:

axiom ∀ f : **Int** $\xrightarrow{\sim}$ **Int** • twice(f) ≡ f ° f

7.8 Explicit Definition of Curried Functions

A shorter way of writing:

value twice : (**Int** $\xrightarrow{\sim}$ **Int**) → **Int** $\xrightarrow{\sim}$ **Int**
axiom ∀ f : **Int** $\xrightarrow{\sim}$ **Int**, i : **Int** • twice(f)(i) ≡ f(f(i))

is:

value
  twice : (**Int** $\xrightarrow{\sim}$ **Int**) → **Int** $\xrightarrow{\sim}$ **Int**
  twice(f)(i) ≡ f(f(i))

This explicit function definition is an instance of the form:

value
  id : $type\_expr_1$ → ... → $type\_expr_n$ → type_expr
  $id(id_1)...(id_n)$ ≡ value_expr

7.9 Currying and Uncurrying

The function *twice* above is curried. Alternatively, we can define an uncurried version *twice′* as:

value
twice′ : (**Int** $\xrightarrow{\sim}$ **Int**) × **Int** $\xrightarrow{\sim}$ **Int**
twice′(f,i) ≡ f(f(i))

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 applications of the form:

twice(f)

must now be written:

λ i : **Int** • twice′(f,i)

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

7.10 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 predicative in the sense of just saying what properties the result must have.

Consider the following specification of the square root function:

value
square_root : **Real** $\xrightarrow{\sim}$ **Real**
axiom
∀ x : **Real** • x ≥ 0.0 ⇒
∃ s : **Real** •
square_root(x) = s ∧
s ∗ s = x ∧
s ≥ 0.0

If the predicate did not include the property that $s \geq 0.0$ then the *square_root* function would be under-specified, returning either a positive or negative result on application — one would not know.

7.11 Implicit Definition of Functions

A shorter way of writing the above is:

value
square_root : **Real** $\xrightarrow{\sim}$ **Real**
square_root(x) **as** s

post s ∗ s = x ∧ s ≥ 0.0
pre x ≥ 0.0

This is an implicit function definition reading as follows.

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

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

The implicit definition is short for:

value
square_root : **Real** $\xrightarrow{\sim}$ **Real**
axiom
∀ x : **Real** •
square_root(x) **as** s
post s ∗ s = x ∧ s ≥ 0.0
pre x ≥ 0.0

where the expression following the • has the form:

value_expr$_1$ **as** id **post** value_expr$_2$ **pre** value_expr$_3$

This is the general form of a post-expression, and in the simple case it is equivalent to the following expression, assuming that the type of *value_expr*$_1$ can be represented by the type expression *type_expr*:

value_expr$_3$ ⇒
∃ id : type_expr •
id = value_expr$_1$ ∧ value_expr$_2$

This equivalence only holds, however, if the expressions are applicative and terminate with a unique result. In chapter 23 the general meaning of post-expressions will be explained in detail.

The above implicit definition of *square_root* 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 value_expr$_1$
pre value_expr$_2$

for $n \geq 1$, where *value_expr*$_2$ can refer to the arguments $id_1, ..., id_n$ and where *value_expr*$_1$ in addition can refer to id_r.

7.12 Algebraic Definition of Functions

Most of the function definitions we have seen up to now are of the form:

id(id$_1$,...,id$_n$) ≡ value_expr

The important point here is that between the brackets (and) is a list of identifiers. This corresponds to the way of defining functions in many programming languages.

RSL, however, also allows for an algebraic style of function definitions. Using this style, axioms typically have the form:

id(value_expr$_1$,...,value_expr$_n$) ≡ value_expr

where the expressions between (and) typically themselves contain calls of functions, perhaps even including *id*. For example, axioms stating that, for an argument x, a function f is idempotent and the inverse of a function g can be written in this style:

f(f(x)) ≡ f(x),
f(g(x)) ≡ x

Consider the specification of integer lists. A list is an ordered sequence of elements. One can construct a new list by adding an element to an old list. 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.

```
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 do not explicitly say how lists are represented.

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

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

The *head* and *tail* functions are partial in that they are not necessarily 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 gives the element i.

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

So we have as consequences of these axioms:

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

7.13 Example: A Database

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 is already 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 $\xrightarrow{\sim}$ Data
    axiom forall k,k_1 : Key, d : Data, db : Database •
      [remove_empty]
        remove(k,empty) ≡ empty,
      [remove_insert]
        remove(k,insert(k_1,d,db)) ≡
          if k = k_1 **then** remove(k,db) **else** insert(k_1,d,remove(k,db)) **end**,
      [defined_empty]
        defined(k,empty) ≡ **false**,
      [defined_insert]
        defined(k,insert(k_1,d,db)) ≡ k = k_1 ∨ defined(k,db),
      [lookup_insert]
        lookup(k,insert(k_1,d,db)) ≡ **if** k = k_1 **then** d **else** lookup(k,db) **end**
        pre k = k_1 ∨ defined(k,db)
  end

The *Database* type is given as an abstract type since we do not want to say anything about how databases are represented. Likewise, nothing is said about keys and data.

The *lookup* function is partial and is under-specified when applied to a key and a database not associating that key with a data element. There is only one axiom for *lookup*, namely *lookup_insert*, and that only applies when its pre-condition is satisfied. There is no axiom *lookup_empty* and hence the value of $lookup(k, empty)$ is under-specified.

The *remove_insert* axiom is the most elaborate of the axioms. The right hand side is an if expression with two branches:

- If the key k to be removed equals the inserted key k_1, then the association of k with d is 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 be:

  remove(k,insert(k_1,d,db)) ≡ **if** k = k_1 **then** db **else** ... **end**

  This is, however, wrong. 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 identifying axioms. The technique can be characterized 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:

   insert(k_1,d_1,insert(k_2,d_2,...insert(k_n,d_n,empty)...))

2. Define the remaining functions by case over the constructors using new identifiers as parameters. In the above axioms, *remove*, *defined* and *lookup* are defined over the two constructor expressions:

   empty
   insert(k_1,d,db)

   We thus get immediately all the left hand sides of the axioms that we need. That is:

   remove(k,empty)
   remove(k,insert(k_1,d,db))
   defined(k,empty)
   defined(k,insert(k_1,d,db))
   lookup(k,empty)
   lookup(k,insert(k_1,d,db))

Note, however, that we choose to under-specify *lookup*; its signature includes the partial function arrow, we do not include an axiom with left hand side $lookup(k, empty)$ and the axiom *lookup_insert* has a pre-condition — it only applies to defined keys.

The list axioms (section 7.12) have the same form. The technique is useful in many applications, but there are of course applications where one must be more inventive when writing axioms.

Chapter 12 describes a very simple way of identifying the constructors of a type.

7.14 Example: The Natural Numbers

Consider the specification of natural numbers. This specification is not really needed since RSL provides the built-in type **Nat**. The example is given only for illustration. Functions are 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 the 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'. Note the quantification over predicates.

What might 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 could have been stated in section 7.12 and section 7.13 (and would have been needed for us to do inductive proofs about the lists and databases specified there). In chapter 12 a shorthand for such induction axioms is described.

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

```
NATURAL_NUMBERS =
  extend PEANO with
  class
    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
```

There are two questions that one can ask about the axioms for *plus* and *mult*:

- Are they correct, i.e. do they conform to the standard rules of arithmetic?
- Are they adequate, i.e. can we use them to add or multiply any two natural numbers?

The first question we can answer with some confidence by studying the axioms, perhaps trying a few examples. The second we can answer by observing that the constructor technique for inventing axioms outlined in section 7.13 has been used. That is, the axioms for *plus* and *mult* are given by case over the constructors of N: *zero* and *succ*. If this technique is used then any expression involving non-constructors (*plus* and *mult*) can be shown to be equivalent to one only involving the constructors (*zero* and *succ*). The only other requirement is that the axioms for non-constructors must 'make some progress'. For example, the *plus_zero* axiom would be no help in defining *plus* if it were:

plus(n,zero) ≡ plus(n,zero)

because we could then never use the axiom to evaluate the left hand side as an expression only involving constructors — we would just go round in circles. So, to show the axioms make progress in defining the non-constructors we need one of

the two conditions on the right hand side of an axiom whose left hand side has a non-constructor as its outermost function:

- The right hand side does not involve the non-constructor. *plus_zero* and *mult_zero* are examples.
- The right hand side applies non-constructors only to terms with fewer constructors. *plus_succ* and *mult_succ* are examples. In each case the second argument of the non-constructor is reduced from $succ(n_2)$ to n_2. (If there are several arguments then none must increase and at least one must decrease in the number of constructors.)

So with these conditions all expressions can be reduced to expressions only involving constructors. Hence adequacy is assured if the constructors are adequate.

CHAPTER 8

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. The set concept is widely accepted to be very useful in formal specification. Many real-life aspects can be modelled as sets: 'the set of inhabitants of a town', 'the set of participants on a course', etc.

8.1 Set Type Expressions

A type expression of the form *type_expr*-**set** represents a type of finite sets. Each set is a subset of the set of all the values in 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 type containing all finite subsets of the set of all the natural numbers (note that there are infinitely many finite subsets):

{}
{0}
{1}
{0,1}
{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 set of all the values 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 the set of values of a finite type like **Bool**.

The type **Nat-infset** however, contains infinite sets in addition to the finite ones:

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

The dots ... indicate infinity (note that this is not a proper RSL expression).

An example of an infinite set is the set of all the values in **Nat** 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.

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.

8.2 Set Value Expressions

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:

{value_expr$_1$,...,value_expr$_n$}

for $n \geq 0$, where the *value_expr*$_i$ have a common maximal type (see section 11.3).

Each expression is evaluated to a value which is included in the resulting set. Sets are unordered, as is illustrated by the following equality between two set expressions:

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

A set contains distinct values, so the following equality holds:

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

An important 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 comprehended set expression is:

{2∗n | n : **Nat** • n ≤ 3}

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', so it is equal to {0,2,4,6}.

Other examples are:

{n | n : **Nat** • is_a_prime(n)} = {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:

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

for $n \geq 1$, where *value_expr*$_2$ must be a Boolean expression.

A ranged set expression gives a set of integers in a range from a lower bound to an upper bound:

{3 .. 7} = {3,4,5,6,7}
{3 .. 3} = {3}
{3 .. 2} = {}

The general form of a ranged set expression is:

{value_expr$_1$ .. value_expr$_2$}

where *value_expr*$_1$ and *value_expr*$_2$ are integer valued expressions. The expression represents the set of integers between and including the two bounds, *value_expr*$_1$ being the lower bound. Note that if *value_expr*$_1$ is greater than *value_expr*$_2$, the set is empty.

8.3 Infix Operators

Basic operators on sets are 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:

e ∉ s = ∼(e ∈ s)

Some examples are:

$3 \in \{1,3\} = \textbf{true}$
$2 \notin \{1,3\} = \textbf{true}$
$2 \in \{1,3\} = \textbf{false}$

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

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

These operators can be defined in terms of test for membership:

$s_1 \cup s_2 \equiv \{e \mid e : T \bullet e \in s_1 \vee e \in s_2\}$
$s_1 \cap s_2 \equiv \{e \mid e : T \bullet e \in s_1 \wedge e \in s_2\}$

Some examples are:

$\{1,3,5\} \cup \{5,7\} = \{1,3,5,7\}$
$\{1,3,5\} \cap \{5,7\} = \{5\}$
$\{1,3,5\} \cap \{7,8\} = \{\}$

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

$\setminus : \text{T-}\textbf{infset} \times \text{T-}\textbf{infset} \to \text{T-}\textbf{infset}$

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:

$\{1,3,5\} \setminus \{1\} = \{3,5\}$
$\{1,3,5\} \setminus \{7\} = \{1,3,5\}$
$\{1,3,5\} \setminus \{n \mid n : \textbf{Nat} \bullet \text{is\_a\_prime}(n)\} = \{1\}$

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

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

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:

$\{1,3,5\} \subseteq \{1,3,5\} = \textbf{true}$
$\{1,3\} \subset \{1,3,5\} = \textbf{true}$
$\{1,3,5\} \subset \{1,3,5\} = \textbf{false}$
$\{1,3\} \subseteq \{3,5\} = \textbf{false}$

For convenience there are reversed versions of the comparison operators:

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

$\supset$: T-**infset** $\times$ T-**infset** $\rightarrow$ **Bool**

8.4 Prefix Operators

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

card : T-**infset** $\xrightarrow{\sim}$ **Nat**

Some examples are:

card {1,4,67} = 3
card {} = 0

card is a total function when applied to finite sets. The application of **card** to an infinite set gives **chaos**. An example is:

card {n | n : **Nat**} $\equiv$ **chaos**

Note that since the result is **chaos** and not just under-specified, one can always test whether some set *s* is infinite by writing:

card s $\equiv$ **chaos**

8.5 Example: A Resource Manager

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 sending it back to the pool.

```
RESOURCE_MANAGER =
  class
    type
      Resource,
      Pool = Resource-set
    value
      obtain : Pool ~→ Pool × Resource,
      release : Resource × Pool ~→ Pool
    axiom forall r : Resource, p : Pool •
      obtain(p) as (p1,r1) post r1 ∈ p ∧ p1 = p\{r1}
      pre p ≠ {},
      release(r,p) ≡ p ∪ {r}
      pre r ∉ 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 finite set of resources.

The definition of *obtain* reads as follows. When applied to a pool p that is non-empty, a pair (p_1, r_1) is returned. The resource r_1 must be a member of the old pool p. The new pool p_1 is equal to the old p except for r_1 which has been removed. Note that which resource is obtained from a pool containing more than one resource is under-specified.

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

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 $\cup$ which represents the intended behaviour.

8.6 Example: A Database

Consider a set version of the database algebraically specified in section 7.13.

```
SET_DATABASE =
  class
    type
      Record = Key × Data,
      Database = {| rs : Record-set • is_wf_Database(rs) |},
      Key, Data
    value
      is_wf_Database : Record-set → 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, rs : Record-set, db : Database •
      is_wf_Database(rs) ≡
        (∀ k : Key, d1,d2 : Data • ((k,d1) ∈ rs ∧ (k,d2) ∈ rs) ⇒ d1 = d2),
      empty ≡ {},
      insert(k,d,db) ≡ remove(k,db) ∪ {(k,d)},
      remove(k,db) ≡ db \ {(k,d) | d : Data • true},
      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 finite set of records, where a record consists of a key and a data element.

Not all set of records are 'well-formed' as databases. We are not interested in those holding more than one record with the same key. The function *is_wf_Database* defines when a set of records is well-formed. This function is used to define a type *Database*, of sets of records:

type Database = {| rs : Record-**set** • is_wf_Database(rs) |}

Such a type definition, called a subtype definition, defines a new type (here *Database*) containing all the values of a type expression (here *Record*-**set**) that satisfy a predicate (here *is_wf_Database*). Subtypes are described in chapter 11.

The functions *insert*, *remove*, *defined* and *lookup* all have parameters of type *Database*. This means that the property of *Database* values, that they satisfy *is_wf_Database*, may be assumed in their definitions.

The constant *empty* is of type *Database* and the functions *insert* and *remove* return values of type *Database*. This means that the following axioms are implicit in the specification:

axiom forall k : Key, d : Data, db : Database •
  is_wf_Database(empty),
  is_wf_Database(insert(k,d,db)),
  is_wf_Database(remove(k,db))

The specification would be inconsistent if the axioms explicitly given were inconsistent with these.

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 well-formed.

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 (defined) 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 section 7.13. We do not give a detailed definition of the implementation relation here, but just outline a strategy for proving the implementation.

SET_DATABASE implements *DATABASE* because:

1. *SET_DATABASE* defines all the types that *DATABASE* defines, the only change being that one sort (*Database*) has been replaced by a concrete definition (of a subtype).
2. *SET_DATABASE* defines (with the same signatures) all the constants and functions that are defined by *DATABASE*.
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**⌟
unfold empty:
⌞defined(k,{}) ≡ **false**⌟
unfold defined:
⌞(∃ d : Data • (k,d) ∈ {}) ≡ **false**⌟
isin_empty:
⌞(∃ d : Data • **false**) ≡ **false**⌟
exists_introduction:
⌞**false** ≡ **false**⌟
is_annihilation:
⌞**true**⌟
qed

The annotations to the steps in this justification, like *isin_empty*, are references to proof rules of those names in [18].

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

8.7 Example: Equivalence Relations

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 equivalence classes. All the elements of an equivalence class are considered equivalent.

An essential function on equivalence relations is *make_equivalent* for making two elements equivalent. Basically this function joins 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-infset,
      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(e_1,e_2,r) ≡
{c | c : Class • c ∈ r ∧ {e_1,e_2} ∩ c = {}} ∪
{c_1 ∪ c_2 | c_1,c_2 : Class • c_1 ∈ r ∧ c_2 ∈ r ∧ e_1 ∈ c_1 ∧ e_2 ∈ c_2},
are_equivalent(e_1,e_2,r) ≡ (∃ c : Class • c ∈ r ∧ e_1 ∈ c ∧ e_2 ∈ c)
end

An equivalence class *Class* is a set of elements. A *Relation* is a set of equivalence classes. Both these sets are possibly infinite (which also, of course, allows them to be finite).

A relation is well-formed, according to *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 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 either of the two elements. Such classes 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 unchanged.

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

We could have used a subtype to define *Relation* as we did with *Database* in the previous example in section 8.6. Readers might like to consider how the specification would be changed.

CHAPTER 9

Lists

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

⟨1,3,3,1,5⟩
⟨**true**,**false**,**true**⟩

The first list is an integer list and the second is a Boolean list. Lists are ordered and may contain duplicates, and one can therefore speak about 'the first value' in a list, 'the number of occurrences of a particular value' in a list, etc. An example of a list is 'the list of events in order of occurrence'.

9.1 List Type Expressions

A type expression of the form *type_expr** represents a type of finite lists. Each list contains only values from the type represented by *type_expr*.

Consider for example the type expression **Bool***. This type contains infinitely many finite lists of Booleans:

⟨⟩
⟨**true**⟩
⟨**false**⟩
⟨**true**,**false**⟩
⟨**false**,**true**⟩
⟨**true**,**true**⟩
⟨**false**,**false**⟩
⟨**true**,**false**,**true**⟩
⋮

Note that the empty list ⟨⟩ is included. The reader should compare the above Boolean lists with the Boolean sets contained in **Bool-set** (chapter 8).

A type expression of the form *type_expr*$^\omega$ represents the type of infinite as well as finite lists. The type **Bool**$^\omega$ thus contains infinite Boolean lists in addition to

the finite ones:

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

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

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

9.2 List Value Expressions

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

The general form of an enumerated list expression is:

$\langle \text{value\_expr}_1,...,\text{value\_expr}_n \rangle$

for $n \geq 0$, where the *value_expr*$_i$ have a common maximal type (see section 11.3).

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 the following inequality holds:

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

A list may contain duplicates, so the following inequality holds:

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

An important list is that with no members, the empty list $\langle\rangle$.

A ranged list expression represents a list of integers in a range from a lower bound to an upper bound:

$\langle 3 \mathinner{..} 7 \rangle = \langle 3,4,5,6,7 \rangle$
$\langle 3 \mathinner{..} 3 \rangle = \langle 3 \rangle$
$\langle 3 \mathinner{..} 2 \rangle = \langle\rangle$

The general form of a ranged list expression is:

$\langle \text{value\_expr}_1 \mathinner{..} \text{value\_expr}_2 \rangle$

where *value_expr*$_1$ and *value_expr*$_2$ are integer valued expressions. The expression represents the list of increasingly ordered integers between and including the two bounds, *value_expr*$_1$ being the lower bound. Note that if *value_expr*$_1$ is greater than *value_expr*$_2$, the list is empty.

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 comprehended list expression is:

$\langle 2{*}\text{n} \mid \text{n}$ **in** $\langle 0 \,..\, 3\rangle\rangle$

The comprehended list expression reads: 'the list of values $2 * n$ where n ranges over the list $\langle 0..3\rangle$', so it is equal to $\langle 0,2,4,6\rangle$. 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:

$\langle \text{n} \mid \text{n}$ **in** $\langle 1 \,..\, 100\rangle \bullet \text{is\_a\_prime(n)}\rangle = \langle 2,3,5,7,...,97\rangle$

This comprehended list expression reads as follows: 'the list of values n where n ranges over the list $\langle 1..100\rangle$, 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*:

$\langle$(k,transformation(d)) | (k,d) **in** db • is_interesting(k)$\rangle$

The general form of a comprehended list expression is:

$\langle$value_expr$_1$ | binding **in** value_expr$_2$ • value_expr$_3\rangle$

where *value_expr*$_2$ is a list expression and *value_expr*$_3$ is a Boolean expression. The *binding* must match the elements of the list represented by *value_expr*$_2$.

9.3 List Indexing

A particular element of a list may be extracted by indexing, where the index must be a natural number which is at least one and, for finite lists, at most the length of the list. As an example consider the list l defined by:

value l : **Nat***
axiom l = ⟨10,20,30⟩

Then indexing l with index 2 gives the second element in the list:

l(2) = 20

Indexing may be regarded as applying the list to the index, just like applying a function. The general form of an application expression is:

value_expr$_1$(value_expr$_2$)

where *value_expr*$_1$ is a list expression and *value_expr*$_2$ is an integer expression evaluating to a value between one and the length of the list.

9.4 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**$^\omega$
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:

⟨n | n **in** all_natural_numbers • is_a_prime(n)⟩ = ⟨2,3,5,7,... ⟩

9.5 Infix Operators

The concatenation operator concatenates two lists:

⌢ : T* × T$^\omega$ → T$^\omega$

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

⟨e$_1$,...,e$_n$⟩ ⌢ ⟨e$_{n+1}$,... ⟩ = ⟨e$_1$,...,e$_n$,e$_{n+1}$,... ⟩

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 may, however, be infinite as in:

$\langle 0\rangle \frown$ all_natural_numbers $= \langle 0,0,1,2,3,4,5,\ldots\ \rangle$

where *all_natural_numbers* is defined above.

9.6 Prefix Operators

Two basic operators on lists are head and tail:

hd : $T^\omega \xrightarrow{\sim} T$
tl : $T^\omega \xrightarrow{\sim} T^\omega$

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

hd $\langle e_1,e_2,\ldots\ \rangle = e_1$

The tail of a list is the list that remains after the head element is removed:

tl $\langle e_1,e_2,\ldots\ \rangle = \langle e_2,\ldots\ \rangle$

Some examples are:

hd $\langle 1,2,3\rangle = 1$
tl $\langle 1,2,3\rangle = \langle 2,3\rangle$
tl $\langle 1\rangle = \langle\rangle$
hd all_natural_numbers $= 0$
tl all_natural_numbers $= \langle 1,2,3,4,\ldots\ \rangle$

The head and tail operators are only defined for non-empty list arguments.

The 'length' operator returns the length of a finite list:

len : $T^\omega \xrightarrow{\sim}$ **Nat**

Some examples are:

len $\langle 2,4,2\rangle = 3$
len $\langle\rangle = 0$

len is a total function when applied to finite lists. The application of **len** to an infinite list gives **chaos**. An example is:

len all_natural_numbers $\equiv$ **chaos**

Note that since the result is **chaos** and not just under-specified, one can always test whether some list l is infinite by writing:

len l $\equiv$ **chaos**

This expression evaluates to **true** if the list l is infinite, and to **false** if l is finite.

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

inds : $T^\omega \to$ **Nat-infset**
elems : $T^\omega \to$ T**-infset**

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

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**}

9.7 Texts are Character Lists

The type **Text** is short for **Char***. That is, one can apply all the list operators to text values. Some examples are:

$''\mathtt{abc}'' = \langle'\mathtt{a}','\mathtt{b}','\mathtt{c}'\rangle$
$'''' = \langle\rangle$
hd $''\mathtt{abc}'' = '\mathtt{a}'$
$''\mathtt{abc}'' \frown ''\mathtt{de}'' \frown \langle'\mathtt{f}'\rangle = ''\mathtt{abcdef}''$

9.8 Example: A Queue

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 $\xrightarrow{\sim}$ 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 characterized by having an ordering on its members, just like lists. Only finite lists are 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, returning the augmented queue.

To *get* an element from the queue corresponds to removing the head of the list, returning the reduced queue and the element removed.

9.9 Example: Sorting Integer Lists

Consider the specification of a sorting function that sorts an integer list to give a list of increasing integers. We do 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 select i must be equal to the number of indices in the other list which select i'.

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

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

```
SORTING =
  extend LIST_PROPERTIES with
  class
    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.

9.10 Example: A Database

Consider a list version of the database from section 7.13. The database will now be a list of records, corresponding to the standard 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 greater than or equal to k is found. If the key found is greater than k, the search has failed. This algorithm saves time (on average) if the key is not contained in the database.

We also make the function *lookup* total (for 'well-formed' databases) by returning an error value when looking up a key not in the database. We therefore define 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 separate modules. The decomposition into sub-modules reduces the size, and thereby increases the readability, of each module.

```
KEY =
  class
    type
      Key
    value
      less_than : Key × Key → Bool
    axiom forall k,k1,k2,k3 : Key •
      [anti_reflexive]
        ∼less_than(k,k),
      [transitive]
        less_than(k1,k2) ∧ less_than(k2,k3) ⇒ less_than(k1,k3),
      [total_order]
        less_than(k1,k2) ∨ less_than(k2,k1) ∨ k1 = k2
```

```
  end
DATA =
  class
    type Data
    value not_found : Data
  end
```

The error element *not_found* is under-specified — we do not care about the particular value at this point.

The function *less_than* defines a strong 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 <.

To specify records, we make an abstraction, 'hiding' the fact that they are pairs of keys and data. For that purpose we define functions for generating new records (*new_record*), and for decomposing records (*key_of* and *data_of*).

A new module which is an extension of *KEY* and *DATA* defines these functions.

```
RECORD =
  extend KEY with extend DATA with
  class
    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 strange since it is the identity function, taking a pair and returning a pair. We have, however, ensured that we don't need to bother any more 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
  class
    type
      Database = {| rl : Record* • is_wf_Database(rl) |}
    value
      is_wf_Database : Record* → 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, rl : Record*, db : Database •
    is_wf_Database(rl) ≡
      (∀ r1,r2 : Record, left,right : Record* •
        rl = left ^ ⟨r1,r2⟩ ^ 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)},
    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
      elsif key_of(hd db) = k then data_of(hd db)
      else lookup(k,tl db)
      end
end
```

The type *Database* is defined to be a subtype of lists of records — those that are 'well-formed' (see chapter 11 for a description of subtype definitions).

A list of records is well-formed, according to *is_wf_Database*, if for any two successive records, the key of the 'left' record is less than the key of the 'right' record. Note that this well-formedness condition also prevents duplicate keys, i.e. two records having the same key. This is actually a consequence of the *anti_reflexive* axiom in the module *KEY*.

The function *insert* is defined implicitly by saying that the result of an insertion must contain the correct set of records. Note the fact that the result must be well formed (i.e. sorted and without duplicates) is implicit in the result type *Database* in the signature of *insert*. Although we are trying to be implementation oriented, the implicit style has been used in the definition of *insert*, since our particular aim at this point is to optimize the functions *defined* and *lookup*.

The function *remove* is defined by a list comprehension expression that removes all the records having the specified key. Note that the result will be well-formed as the database argument is.

The functions *defined* and *lookup* are defined by nearly the same prescription. 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. This algorithm depends on the database argument being well-formed, ensured by their signatures.

Note in *defined* that due to the conditional interpretation of $\vee$, the function *defined* will not be applied recursively if the key is found.

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

An interesting point to note here is that *LIST_DATABASE* implements *DATABASE* from section 7.13.

Achieving this implementation relation has been our aim, but at the cost of introducing a problem: the constant *not_found* is a value of type *Data* just like any other value of *Data*. It is therefore possible to insert it into the database by *insert*. This is probably not the intention and users of *LIST_DATABASE* should not do this. (We could, for example, make sure that calls of *insert* do not have *not_found* as an argument.)

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: the original function *insert* was defined for all *Data* values.

The list database specification above 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 high-level specifications.

CHAPTER 10

Maps

A map is a table-like structure, very similar to a function, that maps values of one type into values of another type. Examples of maps are:

$[3 \mapsto \textbf{true}, 5 \mapsto \textbf{false}]$
$[\text{"}\texttt{Klaus}\text{"} \mapsto 7, \text{"}\texttt{John}\text{"} \mapsto 2, \text{"}\texttt{Mary}\text{"} \mapsto 7]$

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

$\{\text{"}\texttt{Klaus}\text{"},\text{"}\texttt{John}\text{"},\text{"}\texttt{Mary}\text{"}\}$

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

$\{2,7\}$

Maps are similar to functions in that a map can be applied to a domain value to return an associated range value. The difference between functions and maps lies primarily in the kinds of operators which may be applied. Maps can be viewed as finite or infinite sets of domain/range pairs which may be merged, restricted, augmented, reduced, overridden etc. Functions, on the other hand, may only be composed and applied to arguments: in particular we cannot (unless they also depend on variables) change the set of domain values for which they are defined, or change the results of applying them to particular domain values.

As an example of a map, consider a file directory which is the mapping from file identifiers into files. Such a map could typically be subject to the following manipulations:

1. List all names of existing files.
2. Add a file.
3. Change a file.
4. Delete a file.

10.1 Map Type Expressions

A type expression of the form:

$\text{type\_expr}_1 \ \overrightarrow{m} \ \text{type\_expr}_2$

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 set of values of the type represented by *type_expr*$_1$.

Consider for example the type expression:

Text $\overrightarrow{m}$ **Nat**

This type contains infinitely many maps:

$[\,]$
$[''\texttt{3}'' \mapsto 3]$
$[''\texttt{Klaus}'' \mapsto 7, ''\texttt{John}'' \mapsto 2, ''\texttt{Mary}'' \mapsto 7]$
$\vdots$

Note that the empty map $[\,]$ is included.

Maps may be infinite, in the sense of having infinite domains. The above map type thus also contains infinite maps.

10.2 Map Value Expressions

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

$[3 \mapsto \textbf{true}, 5 \mapsto \textbf{false}]$
$[''\texttt{Klaus}'' \mapsto 7, ''\texttt{John}'' \mapsto 2, ''\texttt{Mary}'' \mapsto 7]$

The general form of an enumerated map expression is:

$[\text{value\_expr}_1 \mapsto \text{value\_expr}_1', ..., \text{value\_expr}_n \mapsto \text{value\_expr}_n']$

for $n \geq 0$. Each expression pair (*value_expr*$_i$,*value_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:

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

An important map is that with no associations, called the empty map: $[\,]$.

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

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

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:

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

It is also possible to create a map that will give non-deterministic results when applied . Consider for example:

$[x \mapsto y \mid x,y : \mathbf{Nat} \bullet \{x,y\} \subseteq \{1,2\}]$

This map maps 1 to 1 as well as to 2, and similarly for 2. Such maps should usually be avoided in specifications but it is possible to create them. See chapter 16 for a discussion about non-determinism.

The general form of a comprehended map expression is:

$[value\_expr_1 \mapsto value\_expr_2 \mid typing_1,\ldots,typing_n \bullet value\_expr_3]$

for $n \geq 1$, where *value_expr*$_3$ is a Boolean expression.

10.3 Application of a Map

A map can be applied to a value if the value belongs to the domain of the map. As an example consider the map m defined by:

value m : **Text** $\overrightarrow{m}$ **Nat**
axiom m = [″`Klaus`″ $\mapsto$ 7, ″`John`″ $\mapsto$ 2, ″`Mary`″ $\mapsto$ 7]

Then applying m to ″`John`″ gives the value 2:

m(″`John`″) = 2

The general form of an application expression is:

$value\_expr_1(value\_expr_2)$

where *value_expr*$_1$ is a map expression and *value_expr*$_2$ must return a value within the domain of the map.

An enumerated map expression where a domain value is mapped to more than one range value may give a non-deterministic result when applied. As an example, consider the following map application:

[3 $\mapsto$ **true**, 3 $\mapsto$ **false**](3)

This expression evaluates to the non-deterministic expression '**true** $\sqcap$ **false**'. See chapter 16 for a discussion about non-determinism.

10.4 Prefix Operators

A basic operator on maps is the domain operator which for a particular map returns its domain, the set of values for which it is defined:

dom : (T_1 $\overrightarrow{m}$ T_2) $\rightarrow$ T_1**-infset**

Some examples are:

dom [″Klaus″ ↦ 7, ″John″ ↦ 2] = {″Klaus″, ″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 returns the range of a map:

rng : $(T_1 \ \overrightarrow{m} \ T_2) \to T_2$**-infset**

It is defined as follows:

rng m = {m(d) | d : T_1 • d ∈ **dom** m}

Some examples are:

rng [″Klaus″ ↦ 7, ″John″ ↦ 2] = {7,2}
rng [n ↦ 2∗n | n : **Nat** • is_a_prime(n)] = {2∗n | n : **Nat** • is_a_prime(n)}
rng [ ] = {}

10.5 Infix Operators

The override operator overrides one map with another:

† : $(T_1 \ \overrightarrow{m} \ T_2) \times (T_1 \ \overrightarrow{m} \ T_2) \to (T_1 \ \overrightarrow{m} \ T_2)$

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

[3 ↦ **true**, 5 ↦ **false**] † [5 ↦ **true**] = [3 ↦ **true**, 5 ↦ **true**]
[3 ↦ **true**] † [5 ↦ **false**] = [3 ↦ **true**, 5 ↦ **false**]
[3 ↦ **true**] † [ ] = [3 ↦ **true**]

The union operator combines two maps in a way similar to the override operator:

∪ : $(T_1 \ \overrightarrow{m} \ T_2) \times (T_1 \ \overrightarrow{m} \ T_2) \to (T_1 \ \overrightarrow{m} \ T_2)$

An example is:

[3 ↦ **true**] ∪ [5 ↦ **false**] = [3 ↦ **true**, 5 ↦ **false**]

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

There are two operators for restricting the domain of a map, namely 'restriction by' and 'restriction to':

$\setminus$: $(T_1 \ \overrightarrow{m} \ T_2) \times T_1$**-infset** $\to (T_1 \ \overrightarrow{m} \ T_2)$
/ : $(T_1 \ \overrightarrow{m} \ T_2) \times T_1$**-infset** $\to (T_1 \ \overrightarrow{m} \ T_2)$

Restriction by removes a set of domain values from a map; restriction to restricts the domain to a set of domain values. They are defined as follows:

m $\setminus$ s = [d ↦ m(d) | d : T_1 • d ∈ **dom** m ∧ d ∉ s]
m / s = [d ↦ m(d) | d : T_1 • d ∈ **dom** m ∧ d ∈ s]

Some examples are:

$[3 \mapsto \textbf{true}, 5 \mapsto \textbf{false}] \setminus \{3\} = [5 \mapsto \textbf{false}]$
$[3 \mapsto \textbf{true}, 5 \mapsto \textbf{false}] / \{3\} = [3 \mapsto \textbf{true}]$
$[3 \mapsto \textbf{true}, 5 \mapsto \textbf{false}] \setminus \{3, 5, 7\} = [\,]$
$[3 \mapsto \textbf{true}, 5 \mapsto \textbf{false}] / \{3, 5, 7\} = [3 \mapsto \textbf{true}, 5 \mapsto \textbf{false}]$

The map composition operator makes it possible to compose two maps:

$$\circ : (\mathrm{T}_2 \xrightarrow[m]{} \mathrm{T}_3) \times (\mathrm{T}_1 \xrightarrow[m]{} \mathrm{T}_2) \to (\mathrm{T}_1 \xrightarrow[m]{} \mathrm{T}_3)$$

The result of composing two maps m_1 and m_2 is defined as follows:

$$\mathrm{m}_1 \circ \mathrm{m}_2 = [\mathrm{x} \mapsto \mathrm{m}_1(\mathrm{m}_2(\mathrm{x})) \mid \mathrm{x} : \mathrm{T}_1 \bullet \mathrm{x} \in \textbf{dom}\ \mathrm{m}_2 \wedge \mathrm{m}_2(\mathrm{x}) \in \textbf{dom}\ \mathrm{m}_1]$$

Some examples are:

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

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

10.6 Example: A Database

Consider a map version of the database from section 7.13. The map data type is very well suited for modelling the database since the database manipulations correspond closely to map operators.

```
MAP_DATABASE =
  class
    type
      Database = Key ~m~> Data,
      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 : 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 is 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.

10.7 Example: Equivalence Relations

Consider a map version of the equivalence relation specification from section 8.7. Remember 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).

We now model a relation 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 ~m~> Partition_Id
    value
      is_wf_Relation : Relation → Bool,
      initial : Relation,
      make_equivalent : Element × Element × Relation ~~> Relation,
      are_equivalent : Element × Element × Relation ~~> 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 well-formed, according to *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 well-formed (the second axiom).

The *initial* relation must in addition map different elements to different partition identifiers (the third axiom). So initially no elements are equivalent.

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 function *make_equivalent* and to the function *are_equivalent*. We could have strengthened these to:

pre is_wf_Relation(r)

In chapter 11 we shall see how a well-formedness predicate can be used to restrict a type via a subtype expression. This makes it possible to avoid pre-conditions like those above. Indeed, the second axiom defining *initial* to be well-formed can also be avoided.

10.8 Example: A Bill of Products

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

The subproducts of a product are all the immediate ones plus their subproducts. Each product therefore defines a hierarchy with itself as the top node.

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

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

A function must therefore be provided that for any product returns the set of its subproducts.

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 subproduct relations between existing products.

```
BILL_OF_PRODUCTS =
  class
    type
      Product,
      Bop = Product ~m~> Product-set
    value
      is_wf_Bop : Bop → Bool,
      empty : Bop,
      sub_products : Product × Bop ~~> Product-infset,
      depends_on : Product × Product × Bop ~~> Bool,
```

```
    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) ≡
      {p_sub | p_sub : Product • depends_on(p,p_sub,bop)}
    pre p ∈ dom bop,
    depends_on(p1,p2,bop) ≡
      p2 ∈ bop(p1) ∨
      (∃ p : Product • (p ∈ bop(p1) ∧ depends_on(p,p2,bop)))
    pre p1 ∈ 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 $\{p_1, ..., p_n\}$ if it consists of the immediate subproducts p_1 ...p_n. A basic product is mapped to the empty set.

A bill of products is well-formed, according to *is_wf_Bop*, if:

1. Every subproduct is in the domain of the map. That is, every product mentioned must occur in the domain,
2. No product is a subproduct of itself. The function application expression *sub_products*(p, bop) returns all the subproducts of p.

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

The subproduct, *sub_products*, of a product is the set of products that the original product depends on as specified by the auxiliary function *depends_on*: a product p_1 depends on a product p_2 if either p_2 is an immediate subproduct of p_1, or if there exists an immediate subproduct of p_1 which depends on p_2.

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 subproducts corresponds to directly adding that association to the map. The pre-condition says that the product must not already exist but that all the immediate subproducts 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 subproduct of some other product.

To *add* an immediate subproduct 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 subproduct must exist.
2. They must be different.
3. The subproduct must not already be an immediate subproduct of the product.
4. The product must not be a subproduct of the subproduct. This prevents violation of the well-formedness condition that a bill of products must be acyclic.

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

CHAPTER 11

Subtypes

A type T_1 is a subtype of another type T_2, if all the values contained in T_1 are also contained in T_2. The type T_2 may contain values that are not in T_1. As an example, the type **Nat** of natural numbers is a subtype of **Int** of integers. That is, **Nat** contains all the integers in **Int** that are non-negative. This is an example of a built-in subtype relationship. The subtype expression construct makes it possible to define arbitrary subtypes.

11.1 Subtype Expressions

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

{| t : **Text** • **len** t > 0 |}

It represents the type of 'those t of type **Text** where the length of t is greater than zero' (remember that a text is a finite list of characters). We thus get a type containing non-empty texts.

Another example is:

{| (x,y) : **Int** × **Int** • x < 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:

{| binding : type_expr • value_expr |}

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

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

T_1 = {| x : T_2 • p(x) |}

The subtype relation is transitive. That is, for any types T_1, T_2 and T_3: if T_1 is a subtype of T_2 and T_2 is a subtype of T_3 then T_1 is a subtype of T_3.

11.2 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 and specifying them in axioms.

Assume the definition:

type NonEmpty = {| l : **Int*** • l ≠ ⟨⟩ |}

Consider then the definition:

value non_empty : NonEmpty

This could also have been written as:

value non_empty : **Int***
axiom non_empty ≠ ⟨⟩

As another example, consider the following function definition:

value last_element : NonEmpty → **Int**
axiom forall l : NonEmpty • last_element(l) ≡ l(**len** l)

This could equivalently have been written as:

value last_element : **Int*** $\xrightarrow{\sim}$ **Int**
axiom forall l : **Int*** • last_element(l) ≡ l(**len** l) **pre** l ≠ ⟨⟩

Although we can give such equivalences between subtypes and axioms when there is just one value of the type (or even a few of them) subtypes can be useful both as shorthands and as capturing a particular concept — here that of the non-empty list. They are also very useful in some specifications when functions can be defined to be total on subtypes although they are partial on any larger type, as we have seen in the database specifications in sections 8.6 and 9.10.

11.3 Maximal Types

We would like to have our specifications automatically type checked. That is, expressions must have the expected types according to a set of rules. Since the type concept of RSL involves general predicates used to construct of subtypes, type checking must be simplified in order to make it automatic. Proving relations between RSL predicates cannot in general be made automatic.

The concept of maximal type is the basis of this simplification. The idea is to simply ignore the predicates. All types are turned into maximal types (by ignoring subtype predicates) before type checking takes place.

Intuitively, a type is maximal if it is not a subtype of any type other than itself. The maximal type of a type T is the largest type of which T is a subtype. We now define the maximal types for the different kinds of types introduced up to this point.

The built-in types **Bool**, **Int**, **Real**, **Char** and **Unit** have themselves as maximal types.

The maximal type of **Nat** is **Int**. **Nat** contains those integers that are greater than or equal to zero. We can in fact write **Nat** as:

{| n : **Int** • n ≥ 0 |}

A sort has itself as maximal type.

The maximal type of a composite type consisting of the application of a type operator to a sequence of argument types is obtained by applying a maximal version of the type operator to the maximal types of the argument types. As an example, the maximal type of the type $\mathbf{Nat}^*$ is $\mathbf{Int}^\omega$. That is, the type of finite natural number lists is a subtype of the type of all (infinite as well as finite) integer lists: those that are finite and which only contain natural numbers.

For the complete description of composite types we assume that for any type T, $maximal(T)$ is the corresponding maximal type. Note that *maximal* is not an RSL function; it is a meta-notation. Then we can list the following rules:

$maximal(\mathrm{T}_1 \times ... \times \mathrm{T}_n) = maximal(\mathrm{T}_1) \times ... \times maximal(\mathrm{T}_n)$
$maximal(\mathrm{T}\textbf{-set}) = maximal(\mathrm{T})\textbf{-infset}$
$maximal(\mathrm{T}\textbf{-infset}) = maximal(\mathrm{T})\textbf{-infset}$
$maximal(\mathrm{T}^*) = maximal(\mathrm{T})^\omega$
$maximal(\mathrm{T}^\omega) = maximal(\mathrm{T})^\omega$
$maximal(\{|\ \mathrm{id} : \mathrm{T} \bullet \mathrm{expr}\ |\}) = maximal(\mathrm{T})$
$maximal(\mathrm{T}_1 \xrightarrow[m]{} \mathrm{T}_2) = maximal(\mathrm{T}_1) \xrightarrow[m]{} maximal(\mathrm{T}_2)$
$maximal(\mathrm{T}_1 \xrightarrow{\sim} \mathrm{T}_2) = maximal(\mathrm{T}_1) \xrightarrow{\sim} maximal(\mathrm{T}_2)$
$maximal(\mathrm{T}_1 \to \mathrm{T}_2) = maximal(\mathrm{T}_1) \xrightarrow{\sim} maximal(\mathrm{T}_2)$

Note that the last rule says that the maximal type of a total function is a partial function. This is necessary because type checking is to be mechanical, and one cannot mechanically decide if an arbitrary function is total.

As an example of applying the above rules to find a maximal type, we can apply this last rule to show:

$maximal(\mathbf{Nat} \to \mathbf{Nat}) = \mathbf{Int} \xrightarrow{\sim} \mathbf{Int}$

So the type of total functions from **Nat** to **Nat** is the type of those partial functions from **Int** to **Int** that for arguments within **Nat** terminate and return a **Nat**.

As another example:

$maximal(\mathbf{Text}) = \mathbf{Char}^\omega$

Recall that **Text** is short for $\mathbf{Char}^*$, and that (following the rules):

$maximal(\mathbf{Char}^*) = maximal(\mathbf{Char})^\omega = \mathbf{Char}^\omega$

In type checking an RSL specification, only maximal types are considered. As an example, consider the definitions:

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

```
    four : Int
  axiom
    four ≡ 4,
    is_a_prime(four) ≡ false
```

In the second axiom, the function *is_a_prime* expecting a **Nat** argument is applied to the value *four*, which is an **Int** argument. Although the types **Nat** and **Int** are different, the axiom is well-typed since their maximal types are the same, namely **Int**. From a pragmatic point of view, it is reasonable that the above specification is well-typed, since *four* really represents a natural number (4).

Stated in another way: a specification is type checked by first transforming all types into maximal types, and then doing the type checking. Transforming all types into maximal types in the above specification gives:

```
  value
    is_a_prime : Int ~→ Bool,
    four : Int
  axiom
    four ≡ 4,
    is_a_prime(four) ≡ false
```

It should be easy to see that the specification is well-typed.

Any type is a subtype of its corresponding maximal type. That is, for any type T, there exists a predicate $p : \mathit{maximal}(T) \to \mathbf{Bool}$ such that:

```
T = {| t : maximal(T) • p(t) |}
```

Observe that the concept of maximal type may be a useful tool when proving that one type is a subtype of another type. One just transforms both types into maximal types, and then proves that the one predicate (that of the expected subtype) implies the other predicate.

11.4 Example: Equivalence Relations

We have seen a number of examples where a well-formedness function expresses which values of a particular type are well-formed (sections 8.7, 10.7 and 10.8). The form used has been:

```
type T = ...
value is_wf_T : T → Bool
```

The function *is_wf_T* has not, however, been used in these examples to actually eliminate the undesired values from T. This can be done through a subtype expression, as was done in the examples in sections 8.6 and 9.10.

Consider the map version of the equivalence relation specification from section 10.7. Recall that a relation is a mapping from elements to partition identifiers. A relation is well-formed 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 well-formedness 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 made compared to the specification in section 10.7:

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 a different signature in that the argument type has become:

   (Element ~m~> Partition_Id)

   instead of *Relation*. This has been necessary in order to avoid a recursion between *Relation* and the type of *is_wf_Relation*.
3. The functions *make_equivalent* and *are_equivalent* have been made total. They no longer need pre-conditions due to the well-formedness of their arguments.
4. The axiom saying that *initial* must be well-formed has been removed. It is 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, thereby removing the need to define the function explicitly:

```
  type
    Relation = {| r : Element ~m~> 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 concealed by the 'secondary information', which is the well-formedness predicate. One has to analyse the subtype expression in order to find the primary information.

An alternative solution could be to first give the primary information in one type definition, and then to give the secondary in another one, using a function definition to express the detail:

```
type
  Loose_Relation = Element ~m~> Partition_Id,
  Relation = {| r : Loose_Relation • is_wf_Relation(r) |}
value
  is_wf_Relation : Loose_Relation → Bool
axiom forall r : Loose_Relation •
  is_wf_Relation(r) ≡ (∀ e : Element • e ∈ dom r)
```

11.5 Example: A Bounded Queue

Consider a bounded version of the queue from section 9.8. 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 are defined for extensible queues (with a size less than *max*) and for reducible 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 |},
      Extensible_Queue = {| q : Queue • len q < max |},
      Reducible_Queue = {| q : Queue • q ≠ empty |}
    value
      max : Nat,
      empty : Extensible_Queue,
      put : Element × Extensible_Queue → Reducible_Queue,
      get : Reducible_Queue → Extensible_Queue × Element
    axiom forall e : Element, eq : Extensible_Queue, rq : Reducible_Queue •
      max > 0,
      empty ≡ ⟨⟩,
      put(e,eq) ≡ eq ⌢ ⟨e⟩,
      get(rq) ≡ (tl rq,hd rq)
  end
```

11.6 Empty Subtypes

What happens if a subtype is empty, because its restriction reduces to **false**? We probably don't want this to happen, but if it does we need to know what it means.

Defining a type by an abbreviation to an empty type does no harm on its own:

type
  NoInt = {| i : **Int** • **false** |}

This declaration is useless but only causes problems when *NoInt* is used. If we use *NoInt* as the value of a constant or the result type of a function that can be applied then we generate a contradiction. Suppose we add to the declaration of *NoInt*:

value
  no_constant : NoInt,
  no_fun : **Unit** → NoInt

Then these are equivalent to the following declarations:

value
  no_constant : **Int**
  no_fun : Unt → **Int**
axiom
  no_constant ∈ { i | i : **Int** • **false** },
  no_fun() ∈ { i | i : **Int** • **false** }

Both axioms are equivalent to **false** and so a specification containing these declarations will be contradictory. Formally, it will have no models, which in practice means that we can never implement it.

But it is possible (though not encouraged!) to refer to empty types without causing contradictions. For example, consider the following declarations:

value
  no_arg_fun : NoInt → **Int**
axiom
  ∀ x : NoInt • **false**,
  ∼(∃ x : NoInt),
  { x | x : NoInt } = {}

The value declaration does no harm because such a function can never be applied.

One would normally expect the first axiom to reduce to the restriction, i.e. to **false**, since the bound identifier x does not appear in the restriction. However, this rule only applies if the type involved is not empty. If it is empty, as here, the universal quantification is defined to be **true** and so the axiom is **true**.

The second axiom looks more reasonable, and one might hope it reduces to **true** — it seems to say that *NoInt* is empty. And indeed there is a rule for existential quantification that when the type involved is empty then it is automatically **false**, making the axiom **true**.

The set comprehension in the third axiom similarly reduces to the empty set and the axiom is **true**, as we might expect.

CHAPTER 12

Variant Definitions

Using a variant definition, one can conveniently define a sort together with a number of functions and constants over that sort. That is, a variant definition is short for a sort definition, some value definitions and some axioms.

Among the defined values are constructors for generating values of the sort, destructors for decomposing values of the sort and finally reconstructors for modifying values of the sort.

The axioms define the properties of the constructors, destructors and reconstructors. An important axiom is the induction axiom which states that the sort is generated by the constructors: any value within the sort is the result of a finite number of constructor applications.

12.1 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 constants *black* and *white*, also referred to as the constructors of the type. In the scope of this definition one can 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. Ignoring, for a moment, one of the axioms (the induction axiom), it is short for:

type

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

The *disjoint* axiom says that *black* is different from *white*. This implies that the type *Colour* contains at least two values. Note, however, that the axiom does not prevent *Colour* from containing more than two values. An extra axiom is needed to express the intended limit on the size of *Colour*. This axiom should ensure that *Colour* contains only the values represented by *black* and *white*.

The second axiom, the induction axiom, which the *Colour* type definition is also short for, states this in a slightly different way:

```
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'.

We can use the induction axiom to show that *Colour* only contains the values *black* and *white*. We take the predicate p to be defined by:

```
p = λ c : Colour • (c = black) ∨ (c = white)
```

Then clearly $p(black)$ is **true** and $p(white)$ is **true**, so the induction axiom says that $p(c)$ is true for all colours c, i.e. that all colours are *black* or *white*.

The reason that this axiom is called an induction axiom is that it makes inductive proofs over the type *Colour* possible. That is, if one can prove a property about both *black* and *white*, then one has proven that property for all values within *Colour*. This is a very simple example of inductive proof. In section 12.2 a more interesting and general case of inductive proof is described.

12.2 Record Constructors

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

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

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

1. The value *empty*.
2. Values of the form $add(e, c)$ where e : *Elem* and c : *Collection*. The function *add* is a record constructor. The term 'record' stems from the fact that such constructors may be used to generate records, where a record is a collection of named fields. This is explained in section 12.3 and, in particular, in

section 15.3.

The definition is short for a sort definition, two value definitions and two axioms. Ignoring the induction axiom, it is short for:

type
  Collection
value
  empty : Collection,
  add : Elem × Collection → Collection
axiom
  [disjoint]
    ∀ e : Elem, c : Collection • empty ≠ add(e,c)

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

The induction axiom is somewhat more complex than in the *Colour* case:

axiom
  [induction]
    ∀ p : Collection → **Bool** •
      (p(empty) ∧ (∀ e : Elem, c : Collection • p(c) ⇒ p(add(e,c)))) ⇒
        (∀ c : Collection • p(c))

The axiom says: 'for all predicates p, if p holds for *empty* and p holding for a collection c implies p holding for $add(e, c)$ for any element e, then p holds for all collections'.

Observe that the truth of this induction axiom makes inductive proofs over the type *Collection* possible. If one can prove a property about *empty*, and one can prove the property for any *add* extension of a *Collection* satisfying the property, then one can prove that property for all values within *Collection*.

Formally speaking we have now said what there is to say about the variant type definition of *Collection*. In the remaining part of this section (12.2) we elaborate on how definitions that put further constraints on the *add* function may be given.

The *disjoint* axiom says that *empty* differs from $add(e, c)$ for any element e and collection c. In fact, nothing is said about *add* beyond the disjointness from *empty* and the generatedness of *Collection* by *empty* and *add*. In particular, there are no axioms stating that different applications of *add* return different collections in *Collection*. Given two different elements e_1 and e_2, the following property is thus not necessarily a consequence — although we might like it to be:

add(e_1,empty) ≠ add(e_2,empty)

To obtain this, we can define an observer function, *is_in*, that tests whether a particular element is in a collection:

value
  is_in : Elem × Collection → **Bool**
axiom forall e,e_1 : Elem, c : Collection •

is_in(e,empty) ≡ **false**
is_in(e,add(e_1,c)) ≡ e = e_1 ∨ is_in(e,c)

This function will now distinguish the two collections above. That is, for $e_1 \neq e_2$:

is_in(e_1,add(e_1,empty)) = **true**
is_in(e_1,add(e_2,empty)) = **false**

and as a consequence of this we get the desired property:

add(e_1,empty) ≠ add(e_2,empty)

Note that we can deduce this due to the rule that for any total function f:

f(x) ≠ f(y) ⇒ x ≠ y

The definition of the function *is_in* thus implies that those collections are distinguishable which we want to be distinguishable. The definition of the function *is_in* implies that collections show the expected behaviour in this respect: one can decide whether a particular element has been added or not.

Suppose we wanted to use collections to model sets. Normally we think of a set as an unordered collection of distinct elements. None of the above axioms, however, prevent collections from being ordered or containing duplicates. To obtain this we may add the following axioms:

axiom forall e,e_1,e_2 : Elem, c : Collection •
[unordered]
add(e_1,add(e_2,c)) ≡ add(e_2,add(e_1,c)),
[no_duplicates]
add(e,add(e,c)) ≡ add(e,c)

12.3 Destructors

Recall that constructors are functions for generating values of a variant type. Destructors are functions for extracting components from values of a variant type. Consider the following variant definition:

type List == empty | add(head : Elem, tail : List)

The difference between this definition and the previous *Collection* definition is, besides the new name *List* instead of *Collection*, the existence of the destructors *head* and *tail*.

Like *Collection*, the type *List* contains two kinds of values:

1. the value *empty*
2. values of the form $add(e, l)$ where e : *Elem* and l : *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 *Collection*:

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 ~→ Elem,
  tail : List ~→ 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 under-specified for the *empty* list.

The destructors can be used to decompose *List* values generated by the *add* constructor. As an example consider the following definition of a function that replaces the head of a list:

```
value
  replace_head : Elem × List ~→ 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 decompose 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 $tail(l_1)$.

As a consequence, the *unordered* axiom:

axiom forall e$_1$,e$_2$: Elem, l : List •
  [unordered]
    add(e$_1$,add(e$_2$,l)) = add(e$_2$,add(e$_1$,l))

is inconsistent with the destructor axiom *head_add* if *Elem* contains more than one element. To see this consider the following deduction. Given two different elements:

e$_1$ ≠ e$_2$

By using the *head_add* axiom, then the *unordered* axiom and then again the *head_add* axiom we can deduce the following:

e$_1$ = head(add(e$_1$,add(e$_2$,l))) = head(add(e$_2$,add(e$_1$,l))) = e$_2$

This is obviously inconsistent with the assumption that e_1 and e_2 are different. There is a similar inconsistency with the *no_duplicates* axiom.

The destructors thus really make *List* values into ordered collections of elements, with the possibility of duplicates.

12.4 Reconstructors

The function *replace_head* defined in the previous section can be defined 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 $\xrightarrow{\sim}$ 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.

12.5 Forming Disjoint Unions of Types

The types *Collection* and *List* represent 'containers', where a container informally speaking is a collection of elements. Variant definitions can also be used to form a type as a disjoint union of other types. 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**)

12.6 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 containing 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 therefore 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 not sure 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 uncertainty can be captured by 'minimizing' what is said about triangles 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 no value definition of a (triangle) constructor is generated. As a consequence, no axioms for the destructors, in this case *base_line*, are generated. (Recall that axioms for destructors relate these to corresponding constructors.) As a third consequence, no induction axiom is generated, as was also the case with the former use of wildcard.

An implementation of the latter 'minimized' definition of *Figure* is the former one with *box*, *circle* and *triangle* constructors and with no wildcards.

The uncertainty about the representation of triangles may alternatively be captured by 'maximizing' what is said about triangles as follows:

```
type
  Figure ==
    box(length : Real, width : Real) |
    circle(radius : Real) |
    _(base_line : Real, left_angle : Real, right_angle : Real,
      left_line : Real, right_line : Real)
```

That is, the triangle alternative contains more components than are necessary in order to represent triangles: all components we can think of are included. An implementation of this 'maximized' definition of *Figure* is the following:

```
type
  Figure ==
    box(length : Real, width : Real) |
    circle(radius : Real) |
    triangle(base_line : Real, left_angle : Real, right_angle : Real)
value
  left_line : Figure ~→ Real,
  right_line : Figure ~→ Real
```

Here we have chosen to keep *base_line*, *left_angle* and *right_angle* as destructors and to define *left_line* and *right_line* as separate functions. These functions can be given axioms defining their behaviour in terms of the constructor and destructors.

12.7 The General Form of a Variant Definition

A variant definition has the general form:

type id == variant$_1$ | ... | variant$_n$

for $n \geq 1$. Each variant is either a constant variant of the form:

id_or_wildcard

or a record variant of the form:

id_or_wildcard(
destructor$_1$: type_expr$_1$ $\leftrightarrow$ reconstructor$_1$,
$\vdots$
destructor$_n$: type_expr$_n$ $\leftrightarrow$ reconstructor$_n$)

for $n \geq 1$. Destructors and reconstructors both take the form *id_or_op* and are optional.

12.8 Example: Sets

Consider a specification of sets with functions for choosing and removing elements. First we give a summary of the basic definitions following the style of section 12.2. The exception is that an axiom is given that defines what equality means for sets. This axiom then replaces the axioms *unordered* and *no_duplicates* since these then become consequences.

SET =
class
type
Set == empty | add(Elem,Set),
Elem
value
is_in : Elem $\times$ Set $\rightarrow$ **Bool**
axiom forall e,e$_1$,e$_2$: Elem, s,s$_1$,s$_2$: Set •
[is_in_empty]
is_in(e,empty) $\equiv$ **false**,
[is_in_add]
is_in(e,add(e$_1$,s)) $\equiv$ e = e$_1$ $\vee$ is_in(e,s),
[equality]
(s$_1$ = s$_2$) $\equiv$ ($\forall$ e : Elem • is_in(e,s$_1$) = is_in(e,s$_2$))
end

The following axioms are consequences of the above definitions:

axiom forall e,e$_1$,e$_2$: Elem, s : Set •
[unordered]
add(e$_1$,add(e$_2$,s)) $\equiv$ add(e$_2$,add(e$_1$,s)),
[no_duplicates]
add(e,add(e,s)) $\equiv$ add(e,s)

We can now define a function for choosing an arbitrary element from a set and a function for removing an element from a set.

CHOOSE_REMOVE =
extend SET **with**
class
value

```
    choose : Set ~→ Elem,
    remove : Elem × Set → Set
  axiom forall e,e1 : Elem, s : Set •
    [choose_elem]
      choose(s) as e post is_in(e,s)
      pre s ≠ empty,
    [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
end
```

Note in particular the axiom for *choose*. It returns some arbitrary element which is just required to be in 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, but it is not (at least if we want genuine sets). It says that the *choose* function returns the last inserted element. For two different elements e_1 and e_2 we have:

$\text{choose}(\text{add}(e_1,\text{add}(e_2,s))) = e_1$
$\text{choose}(\text{add}(e_2,\text{add}(e_1,s))) = e_2$

implying that:

$\text{add}(e_1,\text{add}(e_2,s)) \neq \text{add}(e_2,\text{add}(e_1,s))$

We thus get an inconsistency with the *unordered* axiom if *Elem* contains more than one element. The *choose_add* axiom implies that a set must contain information about which element has been added as the last one.

Of course, we are not forced to adopt the *equality* axiom and thereby the implied *unordered* axiom. If we really want to have the *choose_add* axiom we drop the *equality* axiom. We no longer have sets, which do not have any concept of the most recently added element, but this is just a consequence of desiring to define a *choose* function that can identify the most recently added element.

12.9 Example: Keys and Data

Consider the functions needed for records in the module *RECORD* in section 9.10. These functions identify records with pairs of keys and data. However, usually we wish to avoid making this identification, as we thereby become free to implement

records using other representations. We therefore want to say that records can be constructed from, and decomposed into, keys and data, but not that records are necessarily pairs of keys and data. This can be achieved using a variant definition for the type *Record*.

Additionally, in the module *DATA* in the same section is the definition of one element of the type *Data*, namely *not_found*. This, too, can be expressed using a variant definition, which in this case expresses the fact that *not_found* is one (but possibly not the only) element of *Data*.

Putting these definitions together gives us the following.

```
DATA =
  class
    type Data == not_found | _
  end

RECORD =
  extend DATA with extend KEY with
  class
    type
      Record == new_record(key_of : Key, data_of : Data)
  end
```

This is equivalent with the following, in which the variant definitions are expanded into the equivalent value definitions and axioms.

```
DATA =
  class
    type Data
    value not_found : Data
  end

RECORD =
  extend DATA with extend KEY with
  class
    type
      Record
    value
      new_record : Key × Data → Record,
      key_of : Record → Key,
      data_of : Record → Data
    axiom
      ∀ k : Key, d : Data • key_of(new_record(k, d)) = k,
      ∀ k : Key, d : Data • data_of(new_record(k, d)) = d,
      ∀ p : Record → Bool •
        (∀ (k, d) : Key × Data • p(new_record(k, d))) ⇒
          (∀ r : Record • p(r))
  end
```

No induction axiom is generated for *Data* since its variant definition included a wildcard. The expansion is exactly the same as the original definition in section 9.10.

There is an induction axiom for *Record*. This can be rephrased as a statement that that *new_record* generates all possible records. Doing this gives us the following.

```
RECORD =
  extend DATA with extend KEY with
  class
    type
      Record
    value
      new_record : Key × Data → Record,
      key_of : Record → Key,
      data_of : Record → Data
    axiom
      ∀ k : Key, d : Data • key_of(new_record(k, d)) = k,
      ∀ k : Key, d : Data • data_of(new_record(k, d)) = d,
      ∀ r : Record • ∃ k : Key, d : Data • r = new_record(k, d)
  end
```

12.10 Example: Ordered Trees

Consider the specification of ordered binary trees of elements. A binary tree is either:

1. Empty.
2. Composed of an element and two subtrees: 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 unordered 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* returns all the elements contained in a tree.

When the goal is execution-time efficiency of membership test, 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 subtrees can be ignored if they only contain elements smaller than or bigger than the element in question.

Note that we allow ourselves to talk about execution-time efficiency, although RSL is not a programming language. RSL is, however, a wide-spectrum specification language supporting algorithm design. Since algorithmic RSL specifications typically will be translated into programs in some programming language, we shall feel free to consider efficiency already at the RSL level. Observe, though, that there is no formal necessity to do so.

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_FUNCTIONS =
  extend ORDERED_TREE with
  class
    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 subtree is ignored in the search of an element if all the elements in that subtree are either less than or greater than the element being sought. This improves execution-time efficiency of a search.

The efficiency of *is_in* could further be improved if trees were always balanced. A tree is balanced if its two subtrees have depths that at most differ by a chosen fixed maximum. The *add* function should then make sure that the resulting tree is balanced (resulting in a loss of efficiency of element addition).

Choosing the maximum to be one (1) we can obtain balanced trees as follows.

```
BALANCED_SET_FUNCTIONS =
  extend SET_FUNCTIONS with
  class
    type
      Balanced_Tree = {| t : Ordered_Tree • is_balanced(t) |}
    value
      is_balanced : Ordered_Tree → Bool,
      depth : Ordered_Tree → Nat,
      add_balanced : Elem × Balanced_Tree → Balanced_Tree
    axiom forall e : Elem, t1,t2 : Ordered_Tree, t : Balanced_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_elem]
      add_balanced(e,t) as rt
      post extract_elems(rt) = extract_elems(t) ∪ {e}
  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. Note that since the type *Balanced_Tree* only includes balanced trees (that are also ordered by definition), the resulting tree must be both ordered and balanced 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 specification of a concrete insertion algorithm is rather more complicated.

We do not have to re-specify the function *is_in* since the one coming from *SET_FUNCTIONS* is still sufficient.

12.11 Example: A Database

Consider a version of the map database from section 10.6. In that example, the database manipulations *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 a result it returns a possibly changed database and an output.

The type of input commands is defined as the union of the different kinds of input command. Likewise, the type of outputs is defined as the union of the different kinds of output. 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 : *Data*, are results of evaluating a *mk_lookup*(k) command, where k : *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 : *Key* and d : *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 there, and writing them just increases the size of the specification and forces us to invent new names for them. There are therefore good reasons for not writing them in this example.

As stated earlier (section 12.3), 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. In the current case, however, the other axioms make *mk_insert* and *mk_remove* information-preserving, so the destructors are largely superfluous.

Leaving out destructors usually makes descriptions more under-specified. Descriptions should generally be kept as under-specified as possible, in order not to reduce the range of permitted implementations. Leaving out destructors, however, needs to be done with care, because the users of a description may expect the constructors to be information-preserving.

Likewise the type *Output* could instead be written as follows:

```
type
  Output ==
    lookup_failed | lookup_succeeded(sel_data : Data) | change_done
```

Another observation concerns 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 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:

```
type
  Input ==
    mk_empty |
    mk_insert(Key, Data) |
    mk_remove(Key) |
    mk_lookup(Key)
```

12.12 Example: A File Directory

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 a directory.

```
FILE_DIRECTORY =
  class
    type
      Id, File,
      Directory = Id -m-> Entry,
      Entry == mk_file(sel_file : File) | mk_dir(sel_dir : Directory)
  end
```

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 full″,
      2 → ″buffer is empty″,
      _ → ″error″
    end
```

The evaluation of the case expression is done by first evaluating the expression *error_code*. Depending on the obtained value, one of the texts ″buffer is full″ (if 1), ″buffer is empty″ (if 2) or ″error″ (otherwise) is returned.

The general form of a case expression is:

```
case value_expr of
  pattern1 → value_expr1,
  ⋮
  patternn → value_exprn
end
```

for $n \geq 1$. 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 *value_expr* is matched against the patterns from top to bottom (so that the order is significant) until a successful match is obtained whereupon the corresponding expression is evaluated. If none of the matches are successful, the result is under-specified.

13.1 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 recapitulate the literals 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.

13.2 Wildcard Patterns

A pattern may be a wildcard pattern '_' as already illustrated in the introductory example. Any value matches a wildcard pattern successfully. 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.

13.3 Name Patterns

A pattern may be a value name. A value matches a name pattern successfully if the value equals the value represented by the name. The most typical situation is where the name is a constant constructor defined in a variant definition.

As an example recall the definition of type *Colour* in chapter 12 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
```

13.4 Record Patterns

Consider the following example which defines a function for inverting all colours in a list of colours (previous section):

```
type
  List == empty | add(head : Colour, tail : List)
value
  invert_list : List → List
axiom forall l : List •
  invert_list(l) ≡
    case l of
      empty → empty,
      add(c,l1) → add(invert(c),invert_list(l1))
    end
```

The pattern $add(c, l_1)$ is a record pattern. The value l matches this pattern successfully if l is a non-empty *List* value generated by *add*. The names c and l_1 are bound to the components of l (head and tail) and the succeeding expression is then evaluated within the scope of these two bindings.

This can be re-stated more fomally, observing that *add* is a function: the value l matches this pattern successfully if there exist values x : *Colour* and y : *List* such that $l = add(x, y)$. If this is the case, c is bound to x while l_1 is bound to y, and the succeeding expression is then evaluated within the scope of these two bindings.

Note that there will at most exist one pair (x, y) : *Colour* × *List* such that $l = add(x, y)$. This is a consequence of the occurrence of the destructors *head* and *tail* in the definition of the variant type *List*. As an example where this is not the case, consider a function for choosing an arbitrary element from a set. The set type is defined as follows:

```
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))
```

The function for choosing an element can be defined as follows:

```
type
  Choose_Result == set_is_empty | element(sel_element : Elem)
value
  choose : Set ~→ Choose_Result
axiom forall s : Set •
  choose(s) ≡
    case s of
      empty → set_is_empty,
      add(e,s1) → element(e)
    end
```

The interesting observation to make here is 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, an arbitrary choice is made between the pairs, whereupon e and s_1 are bound to the chosen x and y. We say that the record pattern is non-deterministic. In general, any pattern containing record patterns may be non-deterministic.

As a consequence, an application of the function *choose* to a set s results in the non-deterministic choice of a member of s.

An (applicative) expression is non-deterministic if two different occurrences of that expression may give different results. An expression is deterministic if it is not non-deterministic. The concept of non-determinism and its relation to under-specification is elaborated on in chapter 16.

Note that since the function *choose* may give non-deterministic results, its type must involve a partial arrow. As will be explained in chapter 16, the total function space contains only functions whose results must be deterministic, while the partial function space also contains functions that may return non-deterministic results. The *choose* function is, however, defined for all values in the *Set* type.

It is worth noting that the above definition of *choose* is **not** equivalent to the following:

```
value
  choose : Set ~→ Choose_Result
axiom forall e : Elem, s : Set •
  [choose_empty]
    choose(empty) ≡ set_is_empty,
  [choose_add]
    choose(add(e,s)) ≡ element(e)
```

The *choose_add* axiom says that *choose* selects the most recently 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.

A record pattern has the general form:

id(inner_pattern$_1$,...,inner_pattern$_n$)

for $n \geq 1$, where *id* represents some function of the type:

$T_1 \times ... \times T_n \xrightarrow{\sim} T$

T is the type of the expression whose value is matched against the record pattern. Each T_i represents values that can be matched against *inner_pattern*$_i$.

An inner pattern is basically a pattern, but with a different meaning associated with name patterns. In an inner pattern of the form of an identifier *id*, the *id* is

a 'defining occurrence' in that it is bound as part of the pattern matching. This is in contrast to name patterns at the outermost level: an outermost *id* must have been defined somewhere else. Such an occurrence is called an 'applied occurrence'.

In an inner pattern, an identifier can be turned into an applied occurrence by prefixing it with =, so an inner pattern can have the form:

=id

This is called an equality pattern. As an example, consider the following function for inverting all colours in a list of colours:

```
type
  List == empty | add(head : Colour, tail : List)
value
  invert_list : List → List
axiom forall l : List •
  invert_list(l) ≡
    case l of
      empty → empty,
      add(=white,l1) → add(black,invert_list(l1)),
      add(=black,l1) → add(white,invert_list(l1))
    end
```

13.5 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 the next pattern $\langle i\rangle^\frown l_1$ successfully if an x : **Int** and a y : **Int*** exist such that $l = \langle x\rangle^\frown y$. 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 l_1 is bound to y.

A list pattern has one of two forms. That is, it has either the form of an enumerated list pattern:

$\langle$inner_pattern$_1$,...,inner_pattern$_n\rangle$ (for n $\geq$ 0)

or the form of a concatenated list pattern:

$\langle$inner_pattern$_1$,...,inner_pattern$_n\rangle$ ⌢ inner_pattern (for n ≥ 0)

13.6 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
```

A product pattern has the general form:

(inner_pattern$_1$,...,inner_pattern$_n$) (for n ≥ 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 = Elem*
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⟩ ⌢ t1,⟨e2⟩ ⌢ t2) →
        elements_match(e1,e2) ∧ lists_match(t1,t2),
      _ → false
    end
```

13.7 Example: Ordered Trees

Consider a version of the ordered tree specification from section 12.10. 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
```

The functions *is_ordered* and *extract_elems* have deterministic results, because there are destructors for the components of the record variant of *Tree*.

13.8 Example: A Database

Consider a version of the database from section 12.11. 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
```

The function *evaluate* has deterministic results, because there are destructors for the components of the record variants of *Input*.

CHAPTER 14

Let Expressions

By a let expression one can define local names for particular values. There are two kinds of let expressions, namely explicit and implicit let expressions.

14.1 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* contains 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 the forms:

let let_binding = value_expr$_1$ **in** value_expr$_2$ **end**

where *let_binding* takes one of the the forms *binding*, *record_pattern* or *list_pattern*.

See chapter 13 for a description of record patterns and list patterns.

The example above is an instance of the first form. The expression *value_expr*$_1$ is evaluated to return a value which is then matched against the *binding*, *record_pattern* or *list_pattern*. If the match is successful, the expression *value_expr*$_2$ is then evaluated within the scope of the bindings that occurred as part of the match. If the match is not successful, the value of the whole let expression is under-specified.

An explicit let expression cannot be recursive. That is, the identifiers defined by the *binding*, *record_pattern* or *list_pattern* cannot be referred to within *value_expr*$_1$. Occurrences of these identifiers within *value_expr*$_1$ refer to definitions at an outer level.

The following equivalences hold between let expressions and case expressions:

```
let record_pattern = value_expr1 in value_expr2 end ≡
  case value_expr1 of record_pattern → value_expr2 end

let list_pattern = value_expr1 in value_expr2 end ≡
  case value_expr1 of list_pattern → value_expr2 end
```

Another way of defining the *square_head* function using a let expression with a (product) binding is:

```
value
  square_head : Int* ~→ Int*
axiom forall l : 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 in:

```
type
  Set == empty | add(Elem,Set)
value
  choose : Set ~→ Elem
axiom forall s : Set •
  choose(s) ≡ let add(e,_) = s in e end
  pre s ≠ empty
```

Note that implementations of the choose function may non-deterministically choose 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
```

See chapter 13 on case expressions for a discussion of this non-determinism.

An example of a let expression using a list pattern is in:

```
value
  square_head : Int* ~→ Int*
axiom forall l : Int* •
  square_head(l) ≡ let ⟨h⟩ ⌢ t = l in ⟨h∗h⟩ ⌢ t end
  pre l ≠ ⟨⟩
```

14.2 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* contains 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 typing in value_expr end
let binding : type_expr • value_expr1 in value_expr2 end
```

The above example is an instance of the second form. An implicit let expression of the second form is evaluated by evaluating $value\_expr_2$ in the scope of the *binding* where the identifiers in *binding* are non-deterministically bound to values that make the Boolean expression $value\_expr_1$ hold.

An implicit let expression of the first form is evaluated by evaluating the expression *value_expr* within the scope of the identifiers defined in the *typing*. These identifiers are only specified via their type and are therefore non-deterministically bound to values within their respective types.

An example of an implicit let expression using a typing is in:

```
value some_char : Unit ~→ Char
axiom some_char() ≡ let c : Char in c end
```

The function *some_char* has been defined so that a non-deterministic choice of the character is made at the moment of application. It follows that the following expression is not **true**:

```
some_char() = some_char()
```

Note if we had that defined *some_char* as a constant, we would not obtain this non-determinism:

```
value some_char : Char
```

In this case *some_char* is under-specified, but always represents the same character. See chapter 16 for more discussion on non-determinism.

14.3 Nested Let Expressions

Often one may want to nest let expressions. Suppose that a list is represented by a map from natural numbers (list indices) 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 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
```

A multiple let expression of the form:

let let_def$_1$,...,let_def$_n$ **in** value_expr **end**

for $n > 1$, is short for:

```
let let_def1 in
  ⋮
  let let_defn in
    value_expr
  end
  ⋮
end
```

which means that identifiers bound in *let_def*$_i$ may occur in *let_def*$_j$ if $j > i$.

14.4 Example: A Resource Manager

Consider a version of the resource manager from section 8.5. 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 section 8.5 the function *obtain* was defined in terms of a post-condition. As is explained in chapter 23, post-conditions imply determinism. That is, applied twice to the same pool, the function *obtain* will 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
      obtain : Pool ~→ Pool × Resource,
      release : Resource × Pool ~→ Pool
    axiom forall r : Resource, p : Pool •
      obtain(p) ≡ let r1 : Resource • r1 ∈ p in (p\{r1},r1) end
      pre p ≠ {},
      release(r,p) ≡ p ∪ {r}
      pre r ∉ p
  end
```

Note that we can even avoid the problem of whether *obtain* is deterministic (as specified in section 8.5 using a post-condition) or non-deterministic (as specified above using an implicit let expression). We can use an axiom of the form:

```
axiom forall r : Resource, p : Pool •
  let (p1,r1) = obtain(p) in p1 = p\{r1} ∧ r1 ∈ p end
  pre p ≠ {}
```

The signature of *obtain* indicates that it is partial — which means that it is not specified whether it is total or not, and hence that it is not specified whether it is non-deterministic or deterministic. The axiom still leaves the issue open: while following closely the post-expression from section 8.5 in form it is not actually expressed as a post-condition and so does not enforce determinism. This is probably the best specification as it expresses the required property of *obtain* without forcing the issue of determinism — it leaves as much as choice as possible to the developer.

CHAPTER 15

Union and Short 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 characterized by a flight identification and a landing time. A *take off* is characterized by a flight identification.
3. A *passenger event* is either a *reservation*, a *check in* or a *cancellation*. A *reservation* is characterized by a passenger identification and a flight identification. A *check in* is characterized by a passenger identification, a flight identification and a seat number. A *cancellation* is characterized by a passenger identification and a flight identification.

15.1 Using a Layered Variant Definition

Using variant definitions the above requirements specification can be expressed as follows. Assume the following basic types.

```
BASIC_AIRPORT_TYPES =
  class
    type
      Flight, Passenger, Time, Seat
  end
```

The airport types are then defined as an extension as follows.

```
AIRPORT_TYPES =
  extend BASIC_AIRPORT_TYPES with
  class
    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_cancellation(sel_cancellation : Cancellation),
    Landing = Flight × Time,
    Take_Off = Flight,
    Reservation = Passenger × Flight,
    Check_In = Passenger × Flight × Seat,
    Cancellation = Passenger × Flight
end
```

An immediate observation is that two layers of constructors are defined. That is, given an f : *Flight* and a t : *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.

```
FLIGHT_IDENTIFICATION =
  extend AIRPORT_TYPES with
  class
    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_off(flight) → flight
            end,
          mk_passenger_event(passenger_event) →
            case passenger_event of
              mk_reservation(_,flight) → flight,
              mk_check_in(_,flight,_) → flight,
              mk_cancellation(_,flight) → flight
            end
        end
  end
```

The above example has two layers of constructors. One can imagine examples with three or more layers, which may become even more tedious.

The example can of course be modified by lifting the *Flight* component to the top level, thereby being directly accessible (and in practice this would probably be done). The goal here is, however, to motivate the concept of union definitions described in the following section.

15.2 Union Definitions

RSL provides a way of avoiding the constructors from layered variant definitions. Assume that the identifiers id_1 ...id_n are names for types, then a union definition of the form:

type id = id_1 | ... | id_n

with $n \geq 2$ is short for:

type id == $\text{id\_from\_id}_1(\text{id\_to\_id}_1 : \text{id}_1)$ | ... | $\text{id\_from\_id}_\text{n}(\text{id\_to\_id}_\text{n} : \text{id}_\text{n})$

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 and patterns. We refer to the constructors as 'implicit constructors' since they can be left out.

Observe that before one replaces the union definition with the corresponding variant definition, one must insert the implicit constructors where they have been left out.

Let us re-specify the above example using union definitions.

```
AIRPORT_TYPES =
  extend BASIC_AIRPORT_TYPES with
  class
    type
      Airport_Event = Airplane_Event | Passenger_Event,
      Airplane_Event = Landing | Take_Off,
      Passenger_Event = Reservation | Check_In | Cancellation,
      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),
      Cancellation ==
        mk_cancellation(sel_passenger : Passenger, sel_flight : Flight)
  end
```

Note the definition of the types *Landing*, *Take_Off*, *Reservation*, *Check_In* and *Cancellation*. They are all defined by variant definitions, each with only a single alternative. We need to define these types as constructed and not as abbreviations of Cartesian products for the following reason.

Our intention is to avoid referring to the implicit constructors introduced by the definitions of *Airport_Event*, *Airplane_Event*, respectively *Passenger_Event*. That is, no references will be made to the constructors:

```
Airport_Event_from_Airplane_Event
Airport_Event_from_Passenger_Event
Airplane_Event_from_Landing
Airplane_Event_from_Take_Off
Passenger_Event_from_Reservation
Passenger_Event_from_Check_In
Passenger_Event_from_Cancellation
```

So in order to be able to distinguish values of the type *Airport_Event*, constructors must be defined 'at the lowest level'. Otherwise, for example, we could not distinguish the types *Reservation* and *Cancellation*.

Note also that we have chosen to include destructors, and some of these are identically named. There are, for example, five destructors named *sel_flight*, corresponding to the following value definitions:

```
value
  sel_flight : Landing → Flight,
  sel_flight : Take_Off → Flight,
  sel_flight : Reservation → Flight,
  sel_flight : Check_In → Flight,
  sel_flight : Cancellation → Flight
```

This situation, where an identifier is defined more than once but with distinct maximal types, is called overloading, and is described in chapter 17.

One can now define the function *flight_identification* as follows, recalling that implicit constructors can be left out in patterns.

```
FLIGHT_IDENTIFICATION =
  extend AIRPORT_TYPES with
  class
    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_off(flight) → flight,
          mk_reservation(_,flight) → flight,
          mk_check_in(_,flight,_) → flight,
```

```
      mk_cancellation(_,flight) → flight
    end
end
```

Note that one cannot just write:

```
axiom forall airport_event : Airport_Event •
  flight_identification(airport_event) ≡ sel_flight(airport_event)
```

as it is only constructors that can be left out, and not destructors which in this case should turn the *airport_event* into a *Landing*, *Take_Off*, *Reservation*, *Check_In* or *Cancellation* before *sel_flight* could be applied. It would not be statically decidable what destructors to apply.

15.3 Short Record Definitions

Referring back to the definitions of types *Landing*, *Take_Off*, *Reservation*, *Check_In* and *Cancellation* 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 short 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)
```

A type definition of the form:

```
type
  id ::
    destr_id1 : type_expr1 ↔ recon_id1
    ⋮
    destr_idn : type_exprn ↔ recon_idn
```

for $n \geq 1$, is short for:

```
type
  id ==
    mk_id(
      destr_id1 : type_expr1 ↔ recon_id1,
      ⋮
      destr_idn : type_exprn ↔ recon_idn)
```

Note that the constructor *mk_id* cannot be omitted when writing functions over the type *id*. As for variant definitions, destructors and reconstructors are optional.

We can now finally write the airport types as follows.

```
AIRPORT_TYPES =
  extend BASIC_AIRPORT_TYPES with
  class
    type
      Airport_Event = Airplane_Event | Passenger_Event,
      Airplane_Event = Landing | Take_Off,
      Passenger_Event = Reservation | Check_In | Cancellation,
      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,
      Cancellation :: sel_passenger : Passenger  sel_flight : Flight
  end
```

The function *flight_identification* can now be defined as follows.

```
FLIGHT_IDENTIFICATION =
  extend AIRPORT_TYPES with
  class
    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_Off(flight) → flight,
          mk_Reservation(_,flight) → flight,
          mk_Check_In(_,flight,_) → flight,
          mk_Cancellation(_,flight) → flight
        end
  end
```

15.4 Wildcards in Union Definitions

Union definitions are allowed to contain wildcards ('_') to signify that not all alternatives are known by the specifier when the union definition is written. This is completely analogous to writing wildcards in variant definitions (section 12.6).

As an example, a different definition of the type *Airplane_Event* could be given as follows:

```
type Airplane_Event = Landing | Take_Off | _
```

This means that an *airplane event* may either be a *landing*, a *take_off* or *something else*. At the time of writing the definition, the specifier is uncertain what *something else* is.

A union definition may consequently have the form:

type id = id_1 | ... | id_n | _

with $n \geq 1$, and this is short for:

type id == $\text{id\_from\_id}_1(\text{id\_to\_id}_1 : \text{id}_1)$ | ... | $\text{id\_from\_id}_n(\text{id\_to\_id}_n : \text{id}_n)$ | _

Section 12.6 describes what a wildcard in a variant definition means.

Note that the union definition including a wildcard still allows one to omit the constructors when writing functions over the type *id*, just as is the case for a normal union definition without wildcard (section 15.2).

15.5 Using a Flat Variant Definition

An alternative to using a union definition is of course to define the type *Airport_Event* as a flat variant definition and then ignore the concepts of *airplane event* and *passenger event* in the formal specification. This is done below (leaving out destructors for convenience).

```
AIRPORT_TYPES =
  extend BASIC_AIRPORT_TYPES with
  class
    type
      Airport_Event ==
        mk_landing(Flight, Time) |
        mk_take_off(Flight) |
        mk_reservation(Passenger, Flight) |
        mk_check_in(Passenger, Flight, Seat) |
        mk_cancellation(Passenger, 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*. For example, for *Airplane_Event*:

```
  type
    Airplane_Event = {| e : Airport_Event • is_Airplane_Event(e) |}
  value
    is_Airplane_Event : Airport_Event → Bool
  axiom forall e : Airport_Event •
    is_Airplane_Event(e) ≡
      case e of
        mk_landing(_) → true,
        mk_take_off(_) → true,
        _ → false
      end
```

15.6 Example: A Database

Consider a rewriting of the database from section 13.8 using union definitions instead of variant definitions of the types *Input* and *Output*.

```
UNION_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 can generally be used as an alternative to variant definitions. Union definitions are of particular use when there are too many layers to represent clearly in a single variant.

CHAPTER 16

Under-specification and Non-determinism

The concepts of under-specification and non-determinism have already been introduced in previous sections. Under-specification was introduced in section 3.5 and non-determinism was introduced in section 13.4. In this chapter the two concepts are summarized.

16.1 Under-specification

A value identifier with the definition:

value id : T

is under-specified if the associated axioms do not identify exactly one value within T which id represents.

As an example, consider the following definitions:

value x : **Int**
axiom x $\neq$ 0

The identifier x is under-specified. Every occurrence of x, however, evaluates to the same value. For instance, the expression $x - x$ always evaluates to 0.

An example of an under-specified function is the following:

value increase : **Int** $\rightarrow$ **Int**
axiom forall i : **Int** • increase(i) $>$ i

The function increases its argument by some amount, but is under-specified with respect to what the increase is for each argument.

16.2 Non-determinism

An (applicative) expression is non-deterministic if two different occurrences of that expression may return different results.

An example of a non-deterministic expression is the following:

let x : **Nat** • x < 3 **in** x **end**

Two occurrences of this expression may evaluate to different values. Thus, the following expression does not necessarily evaluate to 0:

(**let** x : **Nat** • x < 3 **in** x **end**) − (**let** x : **Nat** • x < 3 **in** x **end**)

but in fact to $-2 \sqcap -1 \sqcap 0 \sqcap 1 \sqcap 2$. (See section 24.7 for a description of the internal choice operator $\sqcap$.)

An example of a function that may return non-deterministic results is the following:

value choose : **Int-set** $\xrightarrow{\sim}$ **Int**
axiom forall s : **Int-set** • choose(s) ≡ **let** i : **Int** • i ∈ s **in** i **end**

Note that the total function space $T_1 \rightarrow T_2$ only contains functions whose results must be deterministic, while the partial function space $T_1 \xrightarrow{\sim} T_2$ also contains functions that may return non-deterministic results.

The *choose* function can be defined to return deterministic but under-specified results by defining it with a post-condition. As will be explained in chapter 23, post-conditions imply determinism:

value
  choose : **Int-set** $\xrightarrow{\sim}$ **Int**
axiom forall s : **Int-set** •
  choose(s) **as** i **post** i ∈ s
  pre s ≠ {}

16.3 Unbounded non-determinism

A non-deterministic expression is said to be unboundedly non-deterministic if there are an infinite number of choices. For example, the following expressions are unboundedly non-deterministic:

let x : **Int in** x **end**
let x : **Nat in true end**

In RSL, unbounded non-determinism is equivalent to completely chaotic behaviour, so these expressions are both equivalent to **chaos**. This may seem strange in the second case, which might appear to reduce to **true**, but the set of choices of values for x is evaluated before the expression following **in**.

16.4 Predicates Must be Deterministic

Specification writers are advised to ensure that expressions are deterministic in certain places. As a first example, a non-deterministic axiom always evaluates to **false**. The following axiom therefore evaluates to **false**:

axiom let b : **Bool in** b **end**

An expression must be deterministic if it occurs as a restricting Boolean predicate and is to evaluate to **true**. As a second example, the predicate following the • within a quantified expression must be deterministic if it is to evaluate to **true**. The following existential quantification therefore evaluates to **false**:

∃ x : **Char** • x = **let** y : **Char in** y **end**

As a third example, the predicate within a set comprehension must be deterministic if it is to evaluate to **true**. The following set comprehension therefore evaluates to the empty set:

{x | x : **Char** • x = **let** y : **Char in** y **end**}

The other places where we find predicates (considering the part of RSL described so far) are in list comprehensions, map comprehensions, subtype expressions and implicit let expressions. In the syntax such predicates are named 'restrictions'.

The reason why axioms and restrictions have to be deterministic if they are to result in **true** is that:

axiom value_expr

is short for:

axiom value_expr ≡ **true**

and:

• value_expr

is short for:

• value_expr ≡ **true**

where:

value_expr ≡ **true**

is **true** if *value_expr* evaluates to **true** and **false** otherwise.

CHAPTER 17

Overloading and User-defined Operators

RSL allows for the overloading of value identifiers and operators. An identifier or operator is overloaded at a certain point if there are several definitions of that identifier or operator visible at that point, but with different maximal types. Some of the predefined operators are already overloaded. As an example, consider the less than or equal operator $\leq$. There is an **Int** $\leq$ and a **Real** $\leq$:

$\leq$: **Int** $\times$ **Int** $\to$ **Bool**
$\leq$: **Real** $\times$ **Real** $\to$ **Bool**

The two operators have different maximal types (recall that **Int** is not a subtype of **Real**). The $\leq$ operator can occur in different contexts as illustrated by the following two occurrences:

$3 \leq 7 =$ **true**
$3.0 \leq 7.0 =$ **true**

The first occurrence refers to the **Int** $\leq$ since the arguments are integers (without decimal points). The second occurrence refers to the **Real** $\leq$. The task of finding the right definition for an occurrence of an overloaded identifier or operator is called overload resolution. Overload resolution fails unless it identifies exactly one definition.

17.1 Overloading of Value Identifiers

Value identifiers may be overloaded. A simple example is the definition of two constants, both named *empty*, one representing the empty integer set and one representing the empty integer list:

value
  empty : **Int-set**,
  empty : **Int***

axiom
  empty = {},
  empty = ⟨⟩

The two definitions are compatible since the maximal types are different. That is, the maximal types are **Int-infset** and $\mathbf{Int}^{\omega}$.

Overload resolution will find the right definition for each of the following two occurrences:

empty ∪ {1} = {1}
empty ⌢ ⟨1⟩ = ⟨1⟩

The first occurrence of *empty* refers to the empty integer set while the second occurrence refers to the empty integer list.

Functions may be overloaded. A simple example is the definition of two functions, both named *max*, one finding the maximum of two integers, and one finding the maximum of the members of a list:

value
  max : **Int** × **Int** → **Int**,
  max : $\mathbf{Int}^{*} \xrightarrow{\sim}$ **Int**
axiom forall i,j : **Int**, l : $\mathbf{Int}^{*}$ •
  max(i,j) ≡
    if i ≥ j **then** i **else** j **end**,
  max(l) ≡
    case l **of**
      ⟨i⟩ → i,
      ⟨i⟩ ⌢ l_1 → max(i,max(l_1))
    end
  pre l ≠ ⟨⟩

The two axioms contain four occurrences of *max*. These are the following:

1. max(i,j)
2. max(l)
3. max(l_1)
4. max(i,max(l_1))

The first and the last refer to the function defined on integer pairs. The second and third refer to the function defined on integer lists. Overload resolution will find the right definition for each of these occurrences, and for the following two occurrences:

max(1,7) = 7
max(⟨1,7,5⟩) = 7

The two definitions of *max* are valid since their maximal types are different. The following two definitions are not valid:

```
value
  max : Nat × Nat → Nat,
  max : Int × Int → Int
```

This is because they have the same maximal type, namely **Int** × **Int** $\xrightarrow{\sim}$ **Int**.

As a final remark, the reader should note that overloaded definitions can arise indirectly, for example as a result of several short record definitions defining identically named destructors. As an example, consider the following two definitions taken from section 15.3:

```
type
  Landing :: sel_flight : Flight  sel_time : Time,
  Take_Off :: sel_flight : Flight
```

These definitions are short for, amongst others, the following overloaded definitions:

```
value
  sel_flight : Landing → Flight,
  sel_flight : Take_Off → Flight
```

17.2 User-defined Operators

RSL allows operators to be overloaded with meanings other than the predefined ones. Assume as a basis for an example the definition of a type of trees:

```
type Tree == empty | node(left : Tree, elem : Int, right : Tree)
```

One can then define the **elems** operator to return all the elements in a tree:

```
value
  elems : Tree → Int-set
axiom forall t : Tree •
  elems t ≡
    case t of
      empty → {},
      node(l,e,r) → elems l ∪ {e} ∪ elems r
    end
```

In the scope of this definition, two kinds of **elems** operators are visible, one predefined on lists with type:

```
elems : T^ω → T-infset
```

for any type T, and one user-defined on trees. Overload resolution will find the right definition for each of the following two occurrences:

```
elems ⟨1,2⟩ = {1,2}
elems node(empty,1,empty) = {1}
```

A shorthand for the above definition of **elems** is the following:

```
value
  elems : Tree → Int-set
  elems t ≡
    case t of
      empty → {},
      node(l,e,r) → elems l ∪ {e} ∪ elems r
    end
```

The syntax for RSL describes the form of such shorthand definitions of prefix operators. All the prefix operators may be overloaded. Predefined meanings of prefix operators must, however, not be hidden by a definition. As an example, one cannot define **elems** to apply to an integer list and return an integer set, since **elems** already has a predefined meaning of this type. The following value definition is therefore not legal:

```
value elems : Int^ω → Int-infset
```

More formally, this value definition is illegal because the type of **elems** is an instance of the predefined type which in section 9.6 was given to be:

```
elems : T^ω → T-infset
```

Note that, due to this rule, the following definition is allowed, since the type is not an instance of the predefined type:

```
value elems : Int^ω → Bool
```

Infix operators may also be overloaded. As an example, consider the following definition of the ∈ operator that tests whether an element is in a tree:

```
value
  ∈ : Int × Tree → Bool
axiom forall i : Int, t : Tree •
  i ∈ t ≡ i ∈ elems t
```

Overload resolution will find the right definition for each of the following two occurrences of ∈:

```
1 ∈ {1} = true
1 ∈ node(empty,1,empty) = true
```

A shorthand for the above type and axiom for ∈ is the following:

```
value
  ∈ : Int × Tree → Bool
  i ∈ t ≡ i ∈ elems t
```

The syntax for RSL describes the form of such shorthand definitions of infix operators, which may all be overloaded provided, as was the case for prefix operators, their predefined meanings are not hidden by the new definitions. Note also that infix operators cannot be overloaded as prefix operators and vice versa.

17.3 Turning Operators into Expressions

Operators cannot directly occur as expressions. The term $\leq$ is thus not an expression representing a less than or equal function. Any user-defined operator *op* can, however, be turned into an expression by placing brackets around it as follows: (*op*). As an example, assume the following value definition:

value $\leq$: **Char** $\times$ **Char** $\to$ **Bool**

Within the scope of this definition, the term $(\leq)$ is an expression representing this user-defined less than or equal function on characters.

To continue the example, assume further the definition of a function with the following signature:

value is_sorted : (**Char** $\times$ **Char** $\to$ **Bool**) $\times$ **Char*** $\to$ **Bool**

That is, *is_sorted* takes as first argument a predicate on character pairs. The following expression applies *is_sorted* to the user-defined $\leq$ operator and some character list:

is_sorted$((\leq),\langle'\mathtt{a}','\mathtt{b}','\mathtt{c}'\rangle)$

The brackets turn an operator into a function that must be applied using function application notation. That is, for example:

$(\leq)('\mathtt{a}','\mathtt{b}') \equiv '\mathtt{a}' \leq '\mathtt{b}'$

For any user-defined infix operator *infix_op*, the following equivalence holds:

(infix_op)(value_expr$_1$,value_expr$_2$) $\equiv$ value_expr$_1$ infix_op value_expr$_2$

A similar rule holds for any user-defined prefix operator *prefix_op*:

(prefix_op)(value_expr) $\equiv$ prefix_op value_expr

17.4 Occurrences of Operators

Operators are allowed to occur in many places where value identifiers representing functions are allowed to occur. Some examples will illustrate this. The first example is a variant definition with operators occurring as constructors, destructors and reconstructors:

type List == empty | ^(**hd** : Elem $\leftrightarrow$ † , **tl** : List)

Within the scope of this definition, the following expressions are well-formed for any *e* : *Elem* and *l* : *List*:

e ^ l
hd l
tl l
e † l

The second example illustrates how the formal parameter of a function in an explicit function definition may be an operator. The function *is_sorted* from the previous section can be defined as follows:

```
value
  is_sorted : (Char × Char → Bool) × Char* → Bool
  is_sorted(≤,l) ≡
    ∀ idx1,idx2 : Nat •
      {idx1,idx2} ⊆ inds l ∧ idx1 < idx2 ⇒
        l(idx1) ≤ l(idx2)
```

This definition is short for the following signature and axiom, where the reader should note the bracketing of $\leq$ on the left hand side of $\equiv$:

```
value
  is_sorted : (Char × Char → Bool) × Char* → Bool
axiom forall ≤ : Char × Char → Bool, l : Char* •
  is_sorted((≤),l) ≡
    ∀ idx1,idx2 : Nat •
      {idx1,idx2} ⊆ inds l ∧ idx1 < idx2 ⇒
        l(idx1) ≤ l(idx2)
```

17.5 Type Disambiguation

Recall that overload resolution must always result in identifying exactly one definition for each applied occurrence of an identifier. The following example illustrates a situation where overload resolution fails to identify a single definition for each identifier:

```
value
  empty : Int-set,
  empty : Int*,
  is_empty : Int-set → Bool,
  is_empty : Int* → Bool
axiom
  is_empty(empty)
```

Overload resolution of the axiom will lead to identifying the following candidate pairs of matching definitions. Either:

```
value
  empty : Int-set,
  is_empty : Int-set → Bool
```

or:

```
value
  empty : Int*,
  is_empty : Int* → Bool
```

That is, it is not possible to identify a single definition for each identifier, and the overload resolution of the axiom therefore fails. The example can be corrected by adding some extra type information as part of the expression constituting the axiom. In general, a type ambiguous expression, say *value_expr*, can be disambiguated by a disambiguation expression as follows:

value_expr : type_expr

The above example can be corrected by changing the axiom into the following (for instance):

axiom
  is_empty(empty : **Int-set**),
  is_empty(empty : **Int***)

17.6 Example: The Rational Numbers

Consider the specification of rational numbers. A rational number is written a/b where a and b are integers. The a is the numerator and the b is the denominator.

RATIONAL =
  class
    type
      Rational
    value
      / : **Int** × **Int** → Rational,
      + : Rational × Rational → Rational,
      − : Rational × Rational → Rational,
      ∗ : Rational × Rational → Rational,
      / : Rational × Rational → Rational,
      real : Rational $\xrightarrow{\sim}$ **Real**
    axiom forall n,n_1,n_2,d,d_1,d_2 : **Int**, r_1,r_2 : Rational •
      $(n_1 / d_1) + (n_2 / d_2) \equiv (n_1 * d_2 + d_1 * n_2) / (d_1 * d_2)$,
      $r_1 - r_2 \equiv r_1 + (r_2 * ((0-1)/1))$,
      $(n_1 / d_1) * (n_2 / d_2) \equiv (n_1 * n_2) / (d_1 * d_2)$,
      $(n_1 / d_1) / (n_2 / d_2) \equiv (n_1 * d_2) / (d_1 * n_2)$,
      real (n / d) ≡ (**real** n) / (**real** d)
      pre d ≠ 0
  end

Note that / is used to construct values of type *Rational* but it is not specified whether, for example:

$1/2 = 2/4$

We could add such a property by an axiom but it might well only constrain possible implementations unnecessarily. If **real** is the only observer of rational values then the constraint will certainly be unnecessary.

CHAPTER 18

Variables and Sequencing

RSL allows declaration of variables as known from programming languages like Ada and Pascal. 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 the change.

```
COUNTER =
  class
    variable counter : Nat := 0
    value increase : Unit → write counter Nat
    axiom increase() ≡ counter := counter + 1 ; counter
  end
```

The following sections explain the individual declarations of the module.

18.1 Variable Declarations

A variable declaration has the form:

variable
variable_definition$_1$,
⋮
variable_definition$_n$

for $n \geq 1$. In our specification there is one such definition.

A variable definition has the form:

id : type_expr := value_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

value_expr. The initialisation is optional and if not given explicitly, the initial value is some arbitrary value within the specified type.

The variable *counter* in the example is defined to contain values of type **Nat**, with the initial value 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

for $n \geq 2$, which is short for:

id_1 : type_expr,
⋮
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.

18.2 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 see later examples of operations with result type **Unit**, where the only interesting effect of the operations is the way they change the state.

A function type expression for total operations has the general form:

$\mathrm{type\_expr}_1$ → $\mathrm{access\_desc}_1$ ... $\mathrm{access\_desc}_n$ $\mathrm{type\_expr}_2$

for $n \geq 1$. An operation 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 descriptions $access\_desc_i$ is either of the form:

write id_1,...,id_n

for $n \geq 1$, expressing which variables may be written to (as well as read from), or of the form:

read id_1,...,id_n

for $n \geq 1$, expressing which variables may only be read from. Since totality implies determinism (chapter 16), a total operation must change the state in a deterministic manner, just as it must return a value in a deterministic manner.

An expression with variable access is non-deterministic if two different occurrences of the expression, evaluated with the same arguments and in the same state, may give different results or states.

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.

```
RETURN_COUNTER =
  extend COUNTER with
  class
    value return_counter : Unit → read counter Nat
    axiom return_counter() ≡ counter
  end
```

A function type expression for partial operations has the general form:

type_expr$_1$ $\xrightarrow{\sim}$ access_desc$_1$ ... access_desc$_n$ type_expr$_2$ (for n $\geq$ 1)

18.3 Assignment Expressions

A variable *id* can be assigned to by an assignment expression of the form:

id := value_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 RSL there is no distinction between expressions and statements as seen in 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 return a value of a certain type. The value returned by an assignment expression is the value () of type **Unit**.

18.4 Sequencing Expressions

Two expressions can be combined with the sequencing combinator giving a new composite expression:

value_expr$_1$; value_expr$_2$

The composite expression is evaluated by first evaluating *value_expr*$_1$ for the purpose of its possible side-effect on variables, and then by evaluating *value_expr*$_2$ in the changed state. The value returned by the composite expression is the value returned by *value_expr*$_2$. The type of *value_expr*$_1$ must be **Unit**.

Our example contains the following sequencing expression:

counter := counter + 1 ; counter

18.5 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 therefore from now on often use the terms pure expression and read-only expression.

A pure expression is an expression that does not access variables. That is, a pure expression neither reads from nor 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
```

Examples of expressions that are neither pure nor read-only are:

```
x := x + 1
x := x + 1 ; x
```

An example of an expression which is required to be pure is the initialisation expression in a variable definition.

18.6 Quantification over States

How do we interpret 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 variable with a value within the variable's type.

The 'always' combinator □ performs this universal quantification over states. An always expression has the form:

```
□ value_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 *value_expr* evaluates deterministically to **true**. Otherwise, the always expression evaluates to **false**.

The expression *value_expr* must not change the state, but it may depend on the state by reading from variables. That is, *value_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 value_expr$_1$

is therefore short for:

axiom □ value_expr

Since □ requires the expression to be read-only (see above), axioms must be read-only.

In the general case, an axiom declaration of the form:

axiom forall typing$_1$,...,typing$_m$ •
  opt_axiom_naming$_1$ value_expr$_1$,
  ⋮
  opt_axiom_naming$_n$ value_expr$_n$

for $m, n \geq 1$. is short for:

axiom
  opt_axiom_naming$_1$ □ ∀ typing$_1$,...,typing$_m$ • value_expr$_1$,
  ⋮
  opt_axiom_naming$_n$ □ ∀ typing$_1$,...,typing$_m$ • value_expr$_n$

18.7 Equivalence Expressions

Our example contains a single axiom:

increase() ≡ counter := counter + 1 ; counter

which is an equivalence expression of the form:

value_expr$_1$ ≡ value_expr$_2$

Since this equivalence expression occurs as an axiom, it is short for:

□(value_expr$_1$ ≡ value_expr$_2$)

The left hand side of the equivalence (*value_expr*$_1$) is the expression:

increase()

which represents the application of the operation *increase* to the unit value (). The right hand side of the equivalence (*value_expr*$_2$) is the sequencing expression:

counter := counter + 1 ; counter

We now explain in more detail what equivalence ≡ means.

The expression:

value_expr$_1$ ≡ value_expr$_2$

is a Boolean expression which is evaluated in the current state. It evaluates to **true** if and only if the effect of *value_expr*$_1$ evaluated in the current state is exactly

the same as the effect of *value_expr*$_2$ evaluated in the same state. That is, the two expressions must have the same side-effects on variables as well as return the same value. If this is not the case, the equivalence expression evaluates to **false**.

The equivalence also requires equivalent effects concerning **chaos**. That is, if one of the expressions evaluates to **chaos**, the other one must also do so. Note that an equivalence expression always evaluates to either **true** or **false**; it will never be **chaos** itself.

Finally, if one of the expressions is non-deterministic, the other one must show exactly the same 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 constituent expressions are only utilized in the comparison of effects, and are ignored thereafter. The value of the equivalence expression may, however, depend on the state if variables are accessed. An equivalence expression is therefore defined to be read-only.

When an equivalence expression occurs as an axiom, it says that for all states satisfying the variable definitions, the effects of the two expressions must be the same. This implies that any occurrence of *value_expr*$_1$ within the scope of the variable definitions can be replaced by *value_expr*$_2$ and vice versa (assuming that the replacement does not cause any name clashes).

The axiom from our example above says that the *increase* 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 chapters we shall see 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 described in chapter 7 for applicative functions.

18.8 Conditional Equivalence Expressions

An equivalence may be conditional. Such an equivalence contains a pre-condition:

value_expr$_1$ ≡ value_expr$_2$ **pre** value_expr$_3$

where *value_expr*$_3$ must be a read-only Boolean expression. This is short for:

(value_expr$_3$ ≡ **true**) ⇒ (value_expr$_1$ ≡ value_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.

```
DECREASE =
  extend COUNTER with
  class
    value
      decrease : Unit ~→ write counter Nat
    axiom
      decrease() ≡ counter := counter − 1 ; counter
```

pre counter > 0
end

We can paraphrase this axiom: In any state in which *counter* is greater than zero, evaluating *decrease*() is equivalent to decrementing *counter* and then returning its new value.

18.9 Equivalence and Equality

Consider two expressions $value\_expr_1$ and $value\_expr_2$. If these have no side-effects on variables, do not evaluate to **chaos**, and are both deterministic, equality = and equivalence ≡ mean the same. That is, the expression:

$\text{value\_expr}_1 = \text{value\_expr}_2$

has the same meaning as:

$\text{value\_expr}_1 \equiv \text{value\_expr}_2$

If one of the expressions has side-effects, evaluates to **chaos** or is non-deterministic, the meaning of equivalence is different from the meaning of equality. The expression:

$\text{value\_expr}_1 = \text{value\_expr}_2$

is a Boolean expression evaluated as follows.

If one of the expressions evaluates to **chaos**, the equality expression itself evaluates to **chaos**. That is to say, equality is a strict operator. Alternatively, both expressions (possibly non-deterministically) give a side-effect and return 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 combination of the side-effects of the constituent expressions, but is not involved in the comparison.

Evaluation order is from left to right: first $value\_expr_1$ is evaluated, and then $value\_expr_2$ is evaluated. This means that possible side-effects of $value\_expr_1$ may influence the effect of $value\_expr_2$. There is a more comprehensive discussion on expression evaluation order in section 19.2.

To summarize:

- An equivalence expression compares effects as well as results; equality only compares results.
- An equivalence expression does not evaluate its constituent expressions, so it itself has no effects; equality evaluates its constituent expressions (left to right).
- An equivalence expression always evaluates to either **true** (if the effects and results are equivalent) or **false** (otherwise). Equality may evaluate to any (Boolean) expression, including non-deterministic expressions and **chaos**.

For example, assuming the current value of variable x is *0*:

(x := x + 1 ; 1) ≡ (x := x + 1 ; x) is equivalent to **true**

(x := x + 1 ; 1) = (x := x + 1 ; x) is equivalent to x := 2 ; **false**
(1 ⌈⌉ 2) ≡ (1 ⌈⌉ 2) is equivalent to **true**
(1 ⌈⌉ 2) = (1 ⌈⌉ 2) is equivalent to **true** ⌈⌉ **false**

Equality is intended to be close to the behaviour of the equality in programming languages.

18.10 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 increases the counter and returns a Boolean value depending on a comparison of the value of the resulting counter and the parameter.

```
TEST_COUNTER =
  extend COUNTER with
  class
    value increase_and_test : Nat → write counter Bool
    axiom forall n : Nat • increase_and_test(n) ≡ increase() ≤ n
  end
```

Suppose that we want to specify an operation for increasing the counter twice and that we want to specify it in terms of two calls of the *increase* operation. We observe that the following expression is not allowed:

```
increase() ; increase()
```

due to the rule (see section 18.4) that the expression before the semicolon must have the type **Unit**. Instead we can specify the *increase_twice* function as follows.

```
INCREASE_TWICE =
  extend COUNTER with
  class
    value increase_twice : Unit → write counter Nat
    axiom increase_twice() ≡ let dummy = increase() in increase() end
  end
```

We have to introduce a *dummy* name for the result returned by the first application of *increase*. 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 a semicolon.

In our example one could easily separate the operations for increasing the counter and for reading the counter. This is done below.

```
COUNTER =
  class
    variable
      counter : Nat := 0
    value
```

```
    increase : Unit → write counter Unit,
    return_counter : Unit → read counter Nat
  axiom
    increase() ≡ counter := counter + 1,
    return_counter() ≡ counter
end
```

The operation *increase_twice* (with unchanged type) then looks as follows.

```
INCREASE_TWICE =
  extend COUNTER with
  class
    value increase_twice : Unit → write counter Nat
    axiom increase_twice() ≡ increase() ; increase() ; return_counter()
  end
```

18.11 Example: A Database

Consider an imperative version of the database from section 10.6. A variable containing the database is defined, 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 are several reasons for writing imperative specifications instead of applicative specifications. Some reasons are:

1. A specification that is to be implemented in an imperative programming language may be more naturally written in an imperative style.
2. The imperative style of specification reduces the number of parameters to functions. Thus, a call of *insert* has the form:

   insert(k,d)

   for some $k : Key$ and $d : Data$. A call of the applicative *insert* from section 10.6 has an extra parameter, namely the database:

   insert(k,d,db)

   for some $k : Key$, $d : Data$ and $db : Database$. Recall that the applicative version of *insert* had the type:

   value insert : Key × Data × Database → Database

   This argument for imperative 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 an imperative nature, like the database example. One may then prefer to model them as such.

CHAPTER 19

Expressions Revisited

All expressions are evaluated in a state. This also holds for applicative expressions. In this chapter we briefly revisit expressions in the light of their evaluation in a state.

19.1 Pure and Read-only Expressions

Any expression may access variables. We have already seen examples of both if expressions and let expressions accessing variables.

There are, however, general restrictions on how variables can be accessed, as already indicated by the introduction of pure and read-only expressions in chapter 18. We do not revisit all expressions here but just give some examples. The syntax for RSL 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 (chapter 11):

{| binding : type_expr • value_expr |}

That is, *value_expr* must be pure. Examples of expression occurrences that are required to be read-only are the constituent expressions of a comprehended set expression (chapter 8):

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

That is, *value_expr*$_1$ and *value_expr*$_2$ must be read-only.

19.2 Expression Evaluation Order

Recall from chapter 18 that the two constituent expressions of an equality expression:

value_expr$_1$ = value_expr$_2$

are evaluated from left to right. The general rule is that the constituent expressions of a value infix expression of the form:

value_expr$_1$ infix_op value_expr$_2$

are evaluated from left to right for all infix operators.

As an example, assume the variable definition:

variable x : **Int**

In the scope of this definition, the following equivalences hold:

(x := 1 ; x) + (x := 2 ; x) ≡ x := 2 ; 3
(x := 2 ; x) + (x := 1 ; x) ≡ x := 1 ; 3

The rule can be generalized even more: Unless otherwise stated, any list of expressions is evaluated from left to right. An example is the constituent expressions of a product expression (chapter 5):

(value_expr$_1$,...,value_expr$_n$)

19.3 If Expressions

Recall that an if expression has been described as having the form:

if value_expr$_1$ **then** value_expr$_2$ **else** value_expr$_3$ **end**

It thus contains both a then-branch and an else-branch. In sequential specifications, a form without else-branch is often useful:

if value_expr$_1$ **then** value_expr$_2$ **end**

This is short for:

if value_expr$_1$ **then** value_expr$_2$ **else skip end**

where **skip** is a predefined side-effect free expression of type **Unit**. In fact:

skip ≡ ()

The reason for introducing **skip** when () is available is for ease of reading.

Note that since both branches of an if expression must have the same type, the type of *value_expr*$_2$ must also be **Unit**.

As an example illustrating an if expression without an 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**

CHAPTER 20

Repetitive Expressions

A repetitive expression specifies that a certain expression is repeatedly evaluated for the purpose of its side-effect. There are three forms, all typical of 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.

20.1 While Expressions

A while expression evaluates an expression as long as some predicate is satisfied. A while expression has the form:

while value_expr$_1$ **do** value_expr$_2$ **end**

The expression *value_expr*$_1$ is the controlling expression which must be of type **Bool**. The expression *value_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, *value_expr*$_1$ is evaluated. If it evaluates to **true**, *value_expr*$_2$ is evaluated, and a new iteration is begun. If, on the other hand, *value_expr*$_1$ evaluates to **false**, the while expression terminates. Note the importance of considering the concept of non-termination for repetitive expressions.

A while expression of the above form is equivalent to:

if value_expr$_1$ **then**
  value_expr$_2$; **while** value_expr$_1$ **do** value_expr$_2$ **end**
else skip
end

Consider an operation, *fraction_sum*, for calculating the number:

1 + 1/2 + ... + 1/n

for some non-zero 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 forall n : Nat •
      fraction_sum(n) ≡
        counter := n ;
        result := 0.0 ;
        while counter > 0 do
          result := result + 1.0/(real 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.

Note also that the pre-condition would not be necessary here (if we extended our definition of a fraction sum to be zero for zero n) as a while expression is equivalent to **skip** if its first value expression is **false**.

20.2 Until Expressions

An until expression evaluates an expression until some predicate is satisfied. An until expression has the form:

do value_expr$_1$ **until** value_expr$_2$ **end**

The expression *value_expr*$_2$ is the controlling expression which must be of type **Bool**. The expression *value_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 *value_expr*$_2$ evaluates to **true**, and is evaluated at least once.

An until expression of the above form is equivalent to:

value_expr$_1$; **while** ∼value_expr$_2$ **do** value_expr$_1$ **end**

Consider a re-formulation 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 forall n : Nat •
      fraction_sum(n) ≡
        counter := n ;
        result := 0.0 ;
        do
          result := result + 1.0/(real counter) ;
          counter := counter − 1
        until counter = 0 end
      pre n > 0
  end
```

Note that the pre-condition is necessary here: the body of an until expression is always executed at least once.

20.3 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** value_expr$_1$ **do** value_expr$_2$ **end**

The expression *value_expr*$_1$ must be of a list type, T^*, for some type T. The expression *value_expr*$_2$ is the one to be repeatedly evaluated and must have type **Unit**.

The for expression is evaluated as follows:

1. *value_expr*$_1$ is evaluated to return a (possibly empty) list $\langle e_1,...,e_n \rangle$.
2. For each value, e_i, in the list, processed from left to right, *value_expr*$_2$ is evaluated in the scope of the definitions obtained by matching e_i against the *binding*.

Consider a re-formulation 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/(real i)
        end
    pre n > 0
end
```

As with the while version the pre-condition may not be necessary here: a for expression is equivalent to **skip** if the list expression is empty.

In an extended form of the for expression, one can state a predicate, $value\_expr_p$ of type **Bool**, that specifies which elements from the list $\langle e_1,...,e_n \rangle$ returned by $value\_expr_1$ lead to an evaluation of $value\_expr_2$. The extended version has the form:

for binding **in** value_expr_1 • value_expr_p **do** value_expr_2 **end**

An element e_i from the list returned by $value\_expr_1$ only leads to an evaluation of $value\_expr_2$ if the predicate $value\_expr_p$ deterministically evaluates to **true** (in the scope of the bindings obtained by matching e_i against the *binding*).

The expressions $value\_expr_1$ and $value\_expr_p$ must be read-only.

Consider the specification of a database as 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
  class
    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
```

CHAPTER 21

Local Expressions

A collection of declarations can be made local to an expression by means of a local expression of the form:

local
  declaration$_1$ ... declaration$_n$
in value_expr **end**

The local expression is evaluated by evaluating *value_expr* in the scope of the declarations. Recall that we have seen how declarations can define types, values, variables and axioms.

Consider a re-formulation of the module *FRACTION_SUM* from section 20.1. A function *fraction_sum* is defined to calculate the number:

1 + 1/2 + ... + 1/n

for some positive natural number n. The function is defined in terms of a while expression working on two local variables.

```
FRACTION_SUM =
  class
    value
      fraction_sum : Nat ~→ Real
    axiom forall n : Nat •
      fraction_sum(n) ≡
        local
          variable
            counter : Nat := n,
            result : Real := 0.0
          value
            calc_fraction : Unit → write counter, result Unit
          axiom
            calc_fraction() ≡
              result := result + 1.0/(real counter) ; counter := counter − 1
```

```
        in
          while counter > 0 do calc_fraction() end ; result
        end
      pre n > 0
  end
```

The variable *result* holds the current sum and the variable *counter* controls the iteration. The local operation *calc_fraction* performs the calculation of a single iteration.

A local expression is capable of introducing non-determinism. The following function non-deterministically chooses a value from a set of natural numbers.

```
CHOOSE =
  class
    value
      choose : Nat-set ~→ Nat
    axiom forall s : Nat-set •
      choose(s) ≡
        local
          value n : Nat
          axiom n ∈ s
        in n end
      pre s ≠ {}
  end
```

With this definition, we can not assume that, for arbitrary s:

choose(s) = choose(s)

Note, however, that the following holds:

choose(s) ≡ choose(s)

Values of a local type cannot be returned. As an example, the following expression is not well-formed:

```
  local
    type Local_T
    value l : Local_T
  in l end
```

The two previous examples were well-formed: *fraction_sum* returns the value in a local variable *result*, and *choose* returns the value of a local value n, but the types **Real** and **Nat** involved are not local.

Note that local expressions can be used in purely applicative specifications to introduce local variables.

CHAPTER 22

Algebraic Definition of Operations

Chapter 7 described how applicative functions can be defined abstractly in terms of algebraic equivalences. Recall in particular the algebraic specification of the *LIST* module from section 7.12, which is repeated below. The names of constants and functions have been suffixed with an *a* to indicate that they are applicative.

```
LIST_A =
  class
    type
      List
    value
      empty_a : List,
      add_a : Int × List → List,
      head_a : List ~→ Int,
      tail_a : List ~→ List
    axiom forall i : Int, l : List •
      [head_add]
        head_a(add_a(i,l)) = i,
      [tail_add]
        tail_a(add_a(i,l)) = 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 an imperative sequential specification of lists can be given that ignores representation details in a similar way. There are three main styles of doing this and we treat each of them below.

22.1 Extending an Applicative Module

The first approach is to use the entities from the applicative *LIST_A* module in defining the imperative sequential module. The imperative sequential module becomes an extension of the *LIST_A* module.

```
LIST =
  extend LIST_A with
  class
    variable
      list : List
    value
      empty : Unit → write list Unit,
      is_empty : Unit ~→ read list Bool,
      add : Int → write list Unit,
      head : Unit ~→ read list Int,
      tail : Unit ~→ write list Unit
    axiom forall i : Int •
      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, *list*, of type *List* is defined. This type comes from the *LIST_A* module and is a sort. Nothing has been said about the representation of its values.

The operations working on the *list* variable are defined using 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 defined 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:

```
l = empty_a
```

if we wanted to test whether l was empty. In the imperative sequential 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 *list* to be done through operation calls (which is a reasonable requirement), we must define *is_empty*.

The approach of using an applicative specification in defining an imperative sequential one may seem tedious, especially if the applicative one does not exist already.

22.2 Algebraic Equivalences

The second approach to abstractly specifying the imperative sequential list module is to give algebraic equivalences between operation calls in a way very similar to the equivalences in the applicative *LIST_A* module.

As an example, consider the applicative axiom *head_add* from *LIST_A*:

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 gives the element just added. The corresponding imperative sequential axiom is:

axiom forall i : **Int** •
  [head_add]
    add(i) ; head() ≡ add(i) ; i

The 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 imperative sequential specification of lists is as follows.

```
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 ~→ read list Int,
      tail : Unit ~→ 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 therefore

been said about representation. The operations are defined without assuming any particular representation of lists.

The *head_add* axiom says that adding an element ($add(i)$) followed by examining the head ($head()$) is equivalent to adding an element ($add(i)$) followed by returning the element (i).

The *tail_add* axiom says that adding an element ($add(i)$) followed by removing the head ($tail()$) is equivalent to doing nothing (**skip**).

22.3 Being Implicit about Variables

An interesting observation about the *LIST* module in section 22.2 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 therefore 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 are totally implicit about what the variables are, by simply not defining them. We can modify the *LIST* module in section 22.2 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** $\xrightarrow{\sim}$ **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 may 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** $\xrightarrow{\sim}$ **read any Bool**

A write access description can similarly 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 need occur in a single operation type.

After having performed these changes, and leaving the axioms unchanged, the imperative sequential list module becomes as follows.

```
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 variables there will be and what their types will be.
- Suppose we later develop an implementation of the *LIST* module. Our specification then places no restriction on what the variables of an implementation will be. In particular we are free to use one, two or more variables in an implementation.
- 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, or more variables to be added. 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 is not interested (yet) in what variables there are.
- One wants to leave freedom to a later development that is expected to be an implementation in the formal sense.
- In large specifications it is sometimes necessary to add extra variables to be able to define certain operations. If these operations are called by others, the extra access(es) must be added to these other operations' types, and this can in turn require more access(es) to be added to yet other operations' types. The use of **any** can avoid this problem.

Being explicit, however, has its benefits. From the type of an operation one can see exactly what variables may be accessed and how they may be accessed. This can make imperative specifications easier to read.

A more detailed description of **any** accesses will be given in section 33.3.

22.4 Initialise Expressions

Consider the *LIST* module from the previous section. Since no variables have been explicitly defined, it has not been possible to specify what their initial values are. It is, however, possible to specify such initialisation properties in axioms irrespective of whether any variables have been defined or not. This is demonstrated below.

Note that section 18.1 describes how the initial value of a variable can be specified. Section 28.2 defines more precisely what initialisation means. An intuitive understanding of the concept will do for now.

Suppose that we want to add the following property (axiom) to the *LIST* module in the previous section: 'The initial values of variables must be such that *is_empty*() evaluates to **true**'. In other words, the initial list must be empty.

For the formulation of this axiom, the **initialise** expression is used. This initialises all variables represented by **any** to their initial value. The axiom is added as an extension of the *LIST* module.

```
INITIAL_EMPTY_LIST =
  extend LIST with
  class
    axiom
      initialise ; is_empty() ≡ initialise ; true
  end
```

The axiom says that initialising all variables to their initial value followed by evaluating *is_empty*() is equivalent to initialising all variables and then returning **true**.

22.5 Example: A Database

Consider an algebraic specification of the imperative sequential database from section 18.11. We are implicit about variables by not defining any. As a consequence, all access descriptions 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
      pre k = k1 ∨ defined(k)
end
```

The reader should compare this specification with the algebraic specification of the corresponding applicative module from section 7.13.

The imperative sequential database example illustrates the constructor technique for inventing axioms, which we also saw in section 7.13. The technique used in the imperative sequential case can be characterized as follows.

1. Identify the 'constructor operations' with 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, using new identifiers as parameters. In the above axioms, *remove*, *defined* and *lookup* are defined over the two constructor expressions:

   empty()
   insert(k_1,d)

   We thus get immediately all the left hand sides of the axioms we need:

   empty() ; remove(k)
   insert(k_1,d) ; remove(k)
   empty() ; defined(k)
   insert(k_1,d) ; defined(k)
   empty() ; lookup(k)
   insert(k_1,d) ; lookup(k)

Note, however, that we choose to under-specify *lookup*; its signature includes the partial function arrow, we do not include an axiom with left hand side *empty*() ; *lookup*(k) and the axiom *lookup_insert* has a pre-condition — it only applies to defined keys.

The right hand sides of the axioms *defined_insert* and *lookup_insert* are somewhat different from the corresponding applicative ones. This is due to the requirement that the effect on the state of the left hand side of an equivalence must be the same as the effect on the state of the right hand side. More specifically, the call *insert*(k_1, d) (or its equivalent) must occur on the right hand side since it occurs on the left hand side and since it has a non-trivial effect on the state. Note also the use of let expressions in the two axioms. These are necessary in order to ensure that *defined*(k) and *lookup*(k) are evaluated before *insert*(k_1, d).

The *LIST* axioms (section 22.2 and section 22.3) actually have the same form. The technique is useful in many applications, but there are of course applications where one must be more inventive when writing axioms.

22.6 Refining Applicative Specifications into Imperative Ones

The analogy between sequential imperative specifications like that in section 22.3 and applicative specifications can be formalized: there is a sense in which such imperative specifications are essentially refinements of the applicative ones. Here we illustrate the formalization in the case of lists.

An applicative specification of this kind of lists is as follows:

```
LIST_A =
  class
    type
      List
    value
      empty_a : List,
      add_a : Int × List → List,
      is_empty_a : List ~→ Bool,
      head_a : List ~→ Int,
      tail_a : List ~→ List
    axiom forall i : Int, l : List •
      [is_empty_empty]
        is_empty_a(empty_a) = true,
      [is_empty_add]
        is_empty_a(add_a(i,l)) = false,
      [head_add]
        head_a(add_a(i,l)) = i,
      [tail_add]
```

```
        tail_a(add_a(i,l)) = l
  end
```

A sequential imperative specification produced by analogy with this applicative specification is as follows.

```
LIST =
  class
    value
      empty : Unit → write any Unit,
      is_empty : Unit ~→ write any Bool,
      add : Int → write any Unit,
      head : Unit ~→ write 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
```

This is slightly more general than the corresponding specification in section 22.3 in that is does not require that *is_empty* and *head* only read from variables: they are allowed to write to them as well. (In this instance little is gained from this generality, but it can be useful.) *LIST* can be extended by defining new types, constants and functions in the following manner.

```
LIST_B =
  extend LIST with
  class
    type
      List = {| l : Unit → write any Unit • is_list(l) |}
    value
      is_list : (Unit → write any Unit) → Bool,
      empty_a : List,
      add_a : Int × List → List,
      is_empty_a : List ~→ Bool,
      head_a : List ~→ Int,
      tail_a : List ~→ List
    axiom forall i : Int, l : List •
      empty_a = empty,
      add_a(i,l) = λ() • l() ; add(i),
```

```
    is_empty_a(l) =
      let b : Bool • (λ() • l() ; is_empty()) = (λ() • l() ; b) in b end,
    head_a(l) =
      let i : Int • (λ() • l() ; head()) = (λ() • l() ; i) in i end,
    tail_a(l) = λ() • l() ; tail()
end
```

Though it might appear that these definitions say nothing about *is_list*, they do in fact constrain it; in particular the types given to *empty_a* and *add_a* ensure that *is_list*(*empty*) is **true** and that so is:

$$\forall\ i : \mathbf{Int}, l : \mathbf{Unit} \to \mathbf{write\ any\ Unit} \bullet is\_list(l) \Rightarrow is\_list(\lambda() \bullet l() ; add(i))$$

By contrast, because they are explicitly defined, these definitions do not constrain the functions introduced in *LIST*: nothing can be proved about the functions introduced in *LIST* with the aid of this extension of it that could not be proved without the extension. For instance, because *empty* and *add* are stated in *LIST* to be total functions, and because the sequential composition of two applications of total functions is itself total, nothing new is said about *empty* or *add* by stating that all the functions in type *List* are total.

Because all the functions in the type *List* are total it is the case that:

$$\forall\ b_1, b_2 \bullet \mathbf{Bool} \bullet (\lambda() \bullet empty() ; b_1) = (\lambda() \bullet empty() ; b_2) \Rightarrow b_1 = b_2$$

and that:

$$\forall\ b_1, b_2 \bullet \mathbf{Bool}, i : \mathbf{Int}, l : List \bullet (\lambda() \bullet l() ; add(i) ; b_1) = (\lambda() \bullet l() ; add(i) ; b_2) \Rightarrow b_1 = b_2$$

and similarly that:

$$\forall\ i_1, i_2 \bullet \mathbf{Int}, i : \mathbf{Int}, l : List \bullet (\lambda() \bullet l() ; add(i) ; i_1) = (\lambda() \bullet l() ; add(i) ; i_2) \Rightarrow i_1 = i_2$$

From this it follows that the functions *empty_a*, *add_a*, *is_empty_a*, *head_a* and *tail_a* defined explicitly in *LIST_B* satisfy all the axioms in the original applicative *LIST_A*. Indeed, *LIST_B* is a refinement of *LIST_A*, as everything that can be proved about the latter can be proved about the former. So, by refining an abstract type *List* from *LIST_A* into (a subtype of) an operation type in *LIST_B* we have refined an applicative specification into a sequential imperative one.

CHAPTER 23

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 already seen several examples of this style in the applicative case. See for instance section 7.11.

Consider the following specification of a *choose* operation that returns an arbitrary element from a set contained in a variable. The returned element is at the same time removed from the set, thereby changing the contents of the variable.

CHOOSE =
  class
    variable
      set : **Int-set**
    value
      choose : **Unit** $\xrightarrow{\sim}$ **write** set **Int**
    axiom
      choose() **as** i **post** i $\in$ set` $\wedge$ set = set`\{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. A 'hooked' variable like *set*` 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:

value_expr$_1$ **as** binding **post** value_expr$_2$

where the result naming '**as** *binding*' is optional.

The post-condition *value_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 *value_expr*$_1$ is evaluated in the current state, the pre-state, thereby returning 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. *value_expr*$_1$ is defined and deterministic.
2. *value_expr*$_2$ $\equiv$ **true** when evaluated in the post-state and in the scope of the *binding*. Hooked variables of the form *id*`, however, refer to the pre-state (and are consequently called pre-names).

The post-condition *value_expr*$_2$ must be read-only. Concerning the post-expression itself, the side-effect obtained by evaluating *value_expr*$_1$ is only used for evaluating the post-condition and is ignored thereafter. The post-expression is therefore read-only. A post-expression is always defined and deterministic.

Condition 1 above says that *value_expr*$_1$ is defined and deterministic. In the above *CHOOSE* module, *value_expr*$_1$ corresponds to *choose*(). The *choose* operation is therefore specified by the post-condition to be defined where the pre-condition holds and, moreover, to be deterministic. That is, two applications of *choose* in the same state return the same result state and result value.

It is worth noting that as the pre-name is a technique for accessing the pre-state its use can be replced with a let expression. For instance, the axiom for *choose* could be written:

axiom
  let s = set **in**
    choose() **as** i **post** i $\in$ s $\wedge$ set = s\{i}
    pre set $\neq$ {}
  end

A post-expression may include a pre-condition, which is a read-only expression of type **Bool**:

value_expr$_1$ **as** binding **post** value_expr$_2$ **pre** value_expr$_3$

This is short for:

(value_expr$_3$ $\equiv$ **true**) $\Rightarrow$ value_expr$_1$ **as** binding **post** value_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 preceded by the always combinator

□ implying a universal quantification over all states.

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 efficient 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 returns **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 may be 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 call read-only operations in post-conditions. Such operation calls are 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.

It is possible to use post-conditions in specifications without mentioning variables. Sequential imperative lists, for example, can be specified as follows.

```
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() post (is_empty() = true),
    [is_empty_add]
      add(i) post (is_empty() = false),
    [head_add]
      add(i) post (head() = i)
    [tail_add]
      add(i) ; tail() ≡ skip
end
```

This specification illustrates how post-conditions for functions such as *add* may be split over more than one axiom.

Specifications involving post-conditions can be somewhat more stringent than the corresponding specifications involving equivalences because they are interpreted as expressing both equivalences and requirements that make functions total. The axioms above, however, say nothing beyond what is said in the corresponding specification in section 22.3 because that requires *empty* and *add* to be total and hence, from *head_add* and *tail_add* that *head* following *add* and *tail* following *add* will be total.

CHAPTER 24

Channels and Communication

RSL provides means for specifying concurrent systems. More precisely, combinators are provided for specifying the concurrent evaluation of expressions. Moreover, communication primitives are provided so that expressions evaluating concurrently can communicate with each other through channels.

Concurrency becomes relevant in 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 concurrently to save time. This kind of concurrency could be called 'efficiency concurrency'.

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.

```
ONE_PLACE_BUFFER =
  class
    type Elem
    channel add, get : Elem
    value opb : Unit → in add out get Unit
    axiom opb() ≡ let v = add? in get!v end ; opb()
  end
```

The following sections explain the individual declarations of the module.

24.1 Channel Declarations

A channel declaration has the form:

```
  channel
    channel_definition1,
```

⋮
channel_definition$_n$

for $n \geq 1$. In our example specification there are two such definitions.

A channel definition has the form:

id : type_expr

That is, the channel *id* is defined to carry 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:

id$_1$,...,id$_n$: type_expr

for $n \geq 2$, which is short for:

id$_1$: type_expr,
⋮
id$_n$: type_expr

24.2 Functions with Channel Access

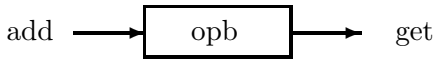
The function *opb* from the example has the type:

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 its ability to communicate through *add* and *get*, and therefore its parameter type and result type are **Unit**. We shall see examples of more interesting parameter and result types later.

The process *opb* can be illustrated as follows:



A type expression for total processes has the general form:

type_expr$_1$ → access_desc$_1$ ... access_desc$_n$ type_expr$_2$

for $n \geq 1$. 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 may access the channels mentioned in the access descriptions.

Each of the access descriptions *access_desc*$_i$ can be of the form:

in id$_1$,...,id$_n$

for $n \geq 1$, expressing which channels processes of the type may input from, or it can be of the form:

out id_1,...,id_n

for $n \geq 1$, expressing which channels they may output to. In addition, since processes can also access variables, access descriptions can describe access to variables as explained in chapter 18 and in chapter 22.

A type expression for partial processes has the general form:

$\text{type\_expr}_1 \xrightarrow{\sim}$ access_desc_1 ... access_desc_n type_expr_2 (for n $\geq$ 1)

24.3 Communication Expressions

RSL provides two communication primitives: one for the input of a value from a channel and one for the output of a value to a channel. In the scope of the channel declaration:

channel id : T

an expression may specify the input of a value from a channel by an input expression of the form:

id?

Evaluation of the expression waits until a value is output along channel *id* by another process. Upon input from channel *id*, the received value is returned as the result of executing the input expression. That is, the type of the input expression is the same type T, as that of the channel.

An expression may specify the output of a value to a channel by an output expression of the form:

id!value_expr

The expression, *value_expr*, is evaluated, and the result is output to channel *id*. Evaluation of the expression waits until the value is input on channel *id* by another process. The type of the expression must be the same as the type of the channel. The type of the output expression itself is **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 introduced for expressing communication, just as no special syntax category is introduced for expressing assignment.

24.4 Composing Expressions Concurrently

Communication through channels is the means by which expressions evaluating concurrently interact. Two expressions may be composed concurrently by being put in parallel as follows:

value_expr$_1$ ‖ value_expr$_2$

The two expressions are evaluated concurrently until the evaluation of one of them comes to an end, whereupon evaluation continues with the other. The two expressions 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 since 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 is the following:

x := 5

Communication is synchronized: the output-specifying expression only specifies output to the channel if the input-specifying expression simultaneously specifies input from the channel.

Parallel attempts to input from a channel and to output to the channel do not, however, 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 combinator is commutative as well as associative. That is:

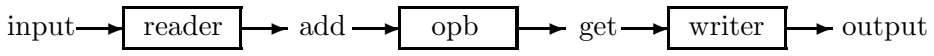
value_expr$_1$ ‖ value_expr$_2$ ≡ value_expr$_2$ ‖ value_expr$_1$

value_expr$_1$ ‖ (value_expr$_2$ ‖ value_expr$_3$) ≡
(value_expr$_1$ ‖ value_expr$_2$) ‖ value_expr$_3$

Two expression evaluations occurring concurrently should be state-independent: if one expression has write access to a variable, the other should not have access to that variable (neither read from it, nor write to it). RSL type checking does not enforce state-independency, but it is highly recommended.

What happens is that the two expressions are evaluated with a copy of the state each, so a state change within one expression cannot be observed within the other expression. Upon termination of the two expressions, a non-deterministic choice is made between the two resulting states, provided that both are different from the original one. If only one of the two resulting states is different from the original state, this one is chosen.

As an example of a concurrent system, suppose that we want to use the one place buffer as a connection between two processes called *reader* and *writer*. The following figure illustrates the concurrent processes and the channels that connect them:



The *reader* process inputs values from the *input* channel and the *writer* process outputs values to the *output* channel. Values move from the *reader* process to the *writer* process via the one place buffer *opb*.

The *reader* and *writer* processes can be specified as follows.

```
READER_WRITER =
  extend ONE_PLACE_BUFFER with
  class
    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 and the *output* channel, respectively. We are abstract about the types since we want to illustrate the concurrency 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 under-specified.

We can now put the processes *reader*, *opb* and *writer* together in parallel, calling the composed process *system*.

```
SYSTEM =
  extend READER_WRITER with
  class
    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 illustrate this better, 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 then 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()
```

24.5 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. The following figure illustrates how we really would like to regard the *system* process from the outside:

That is, we want to hide the channels *add* and *get*. 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 not visible outside the local expression. The effect of leaving the scope (moving beyond 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, so the effect is:

```
  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 get the following module.

```
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, get : Elem
        value
          opb : Unit → in add out get Unit,
          transform1 : Input → Elem,
          transform2 : Elem → Output,
          reader : Unit → in input out add Unit,
          writer : Unit → in get out output Unit
        axiom
          opb() ≡ let v = add? in get!v end ; opb(),
          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.

It may seem tedious to be forced to define all subprocesses of a process within a local expression, especially when a system consists of many subprocesses and these perhaps themselves are composite. Chapter 32 illustrates how the module concept can be used in combination with the local expression to model a hierarchy of processes.

24.6 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 begins by offering 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 communication at the same time. The *system* process defined in the module *SYSTEM* in section 24.4 did in fact 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 specify a choice between different kinds of communications explicitly. 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 to the surroundings (i.e. other expressions executing concurrently with this one) 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, thereby resulting in:

x := 1

Recall, however, that parallel composition only forces communication to happen when channels are hidden in a local expression.

The external choice combinator puts expressions together as follows:

value_expr$_1$ [] value_expr$_2$

The two expressions must have the same type which is also the type of the whole expression. Typically, *value_expr*$_1$ and *value_expr*$_2$ each begin 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 at any one time it 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 possibly modified list as actual parameter.

```
MANY_PLACE_BUFFER =
  class
    type
      Elem,
      Buffer = Elem*
    channel
      empty : Unit,
      add, 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 for *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 deadlock — it will have no further effects. When placed in an external choice, however, by deadlocking one choice it forces one of the others to be chosen. This is because **stop** is the unit for external choice — it has the property that for any expression *value_expr*, the following equivalence holds:

  value_expr [] **stop** ≡ value_expr

  From the axiom for *mpb* 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
```

An empty many place buffer can put in parallel with an expression *value_expr* as follows:

```
mpb(⟨⟩) || value_expr
```

24.7 Internal Choice

Recall that the external choice combinator expresses a choice between two expressions. The term 'external' says that the surroundings may influence which expression is selected. As an example, consider the expression:

```
(x := c? [] d!5) || c!1
```

If, out of $x := c?$ and $d!5$, $x := c?$ is chosen for evaluation it will be because of some event external to $(x := c? [] d!5)$, such as $c!1$.

In addition to the external choice combinator, RSL provides an internal choice combinator $\sqcap$ that specifies an internal choice between two expressions:

```
value_expr1 ⌈⌉ value_expr2
```

In this case, whether $value\_expr_1$ is evaluated or $value\_expr_2$ is evaluated depends on an internal choice, which the surroundings cannot influence. The two expressions must have the same type which is also the type of the whole expression.

The internal choice combinator is commutative and associative. 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 results in:

```
x := c? || c!1
```

thereby potentially leading to a communication over c. If the internal choice on the other hand falls on $d!5$, the expression results in:

```
d!5 || c!1
```

thereby preventing any internal communication from taking place.

The internal choice combinator is not typically used in specifications because of its generally undesirable behaviour. It often occurs proofs about concurrent RSL specifications. One can, however, use the combinator when writing specifications. Consider for example the specification of a die-thrower:

```
type Face_Of_Die == one | two | three | four | five | six
value throw_die : Unit ~→ Face_Of_Die
axiom throw_die() ≡ one ⌈⌉ two ⌈⌉ three ⌈⌉ four ⌈⌉ five ⌈⌉ six
```

The function *throw_die* non-deterministically returns a face of the 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**

24.8 Example: A Database

Consider a concurrent version of the database from section 10.6. A database process, *database*, is defined together with channels for communicating with it.

```
DATABASE =
  class
    type
      Key, Data,
      Database = Key ~m~> Data
    channel
      empty : Unit,
      insert : Key × Data,
      remove, defined, lookup : Key,
      defined_res : Bool,
      lookup_res : Data
    value
      database : Database → in empty, insert, remove, defined, lookup
                            out defined_res, lookup_res Unit,
      not_found : Data
    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 lookup_res!not_found ; database(db) end
        end
  end
```

An essential task when specifying a process is to decide what the channels are and what the protocol is for their use. The specification above illustrates for example how certain channels should be used in particular sequences: 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 informative to compare the channel definitions from the example above with the function and constant types from the applicative database in section 10.6. 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

24.9 Example: An Interfaced Database

Suppose we want to lookup a key k in the concurrent database defined in the previous example. We 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 interface functions 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 interface functions.

```
INTERFACED_DATABASE =
  extend DATABASE with
  class
    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 interface functions *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 : *Key* and a *db* : *Database*:

(lookup!k ; x := lookup_res?) ‖ database(db)

is written as follows when using the interface function *Lookup*:

(x := Lookup(k)) ‖ database(db)

The use of interface functions shortens specifications. In addition, they represent information hiding so that the user of (in this case) *INTERFACED_DATABASE* is not required to know about the details of channel communication; how this can be done without any mention of channels at all is discussed in section 27.3.

24.10 Imperative Processes

The many place buffer process, *mpb*, defined in the module *MANY_PLACE_BUFFER* in section 24.6 is parameterized with respect to the buffer contents. That is, it has the type:

value mpb : Buffer → **in** empty, add **out** get **Unit**

Changes to the buffer contents are reflected in recursive calls of *mpb* with different actual parameter values.

An alternative is to keep the buffer contents in a variable which the process then continuously modifies by means of assignments. We could make this variable global to the module or make it local to the *mpb* process. We will give the latter case as an example. The former is only a small adaptation, but would also suggest the need to be able to hide the variable within the module so that only the process *mpb* could access it. Hiding will be described in section 29.2.

```
MANY_PLACE_BUFFER =
  class
    type
      Elem
    channel
      empty : Unit,
      add, 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 ≠ ⟨⟩ then get!(hd buffer) ; buffer := tl buffer
              else stop
              end
            end
          end
  end
```

The parameter type of *mpb* has become **Unit**.

We could have defined *mpb* in terms of a local recursive process, but it is natural when using assignments and sequencing to use a while expression.

CHAPTER 25

Expressions Revisited

In this chapter we very briefly revisit some of the applicative and imperative expressions introduced earlier in the context of concurrency.

25.1 Pure and Read-only Expressions

In chapter 18 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, however, read from variables.

The syntax for RSL describes the occurrences of expressions that must either be pure or read-only.

25.2 Equivalence Expressions

An equivalence expression (chapter 18) of the form:

$$\text{value\_expr}_1 \equiv \text{value\_expr}_2$$

requires the two constituent expressions to represent the same possible communication behaviour in order to hold. The equivalence expression itself does not communicate since it is only the potential communications of the two constituent expressions that are compared. An equivalence expression is therefore still read-only.

Comprehended Expressions

The combinators $[]$, $\bigsqcap$ and $\|$ are all associative and commutative. It is therefore possible to define their distribution through a set of values. These distribution are defined in RSL as comprehended expressions, and take the general form:

infix_combinator {value_expr$_1$ | typing$_1$,...,typing$_n$ • value_expr$_2$}

for $n \geq 1$, where the *infix_combinator* is one of the three infix combinators $\|$, $[]$ and $\bigsqcap$.

The main use of comprehended expressions is with object arrays (presented in chapter 32), but we give here simple examples of them that do not use object arrays. The function *throw_die* from section 24.7 could be written:

```
type Face_Of_Die == one | two | three | four | five | six
value throw_die : Unit ~→ Face_Of_Die
axiom throw_die() ≡ ⨅ { t | t : Face_Of_Die }
```

(The missing restriction in the comprehended expression is equivalent to • **true**.)

However, as noted previously, the internal choice operator is rarely used in specifications because we usually want our systems to behave more predictably.

Consider the following definitions.

```
channel
  c1, c2 : Text
type
  Index = {| i : Int • i ∈ {1,2} |}
value
  put : Index → Text → out {c1,c2} Unit,
  get : Index → in {c1,c2} Text,
  put_all : Text → out {c1,c2} Unit,
  get_one : Unit → in {c1,c2} Text
axiom forall t : Text, i : Index •
    put(i)(t) ≡
      case i of
```

```
        1 → c1!t,
        2 → c2!t
    end,
  get(i)(t) ≡
    case i of
        1 → c1?,
        2 → c2?
    end,
  put_all(t) ≡ ‖ { put(i)(t) | i : Index },
  get_one() ≡ [] { get(i) | i : Index }
```

The function *put_all* is intended to output a text to all the channels. It does this using a parallel comprehended expression.

The function *get_one* is intended to get a text from one channel or the other. It does this using an external choice comprehended expression.

Note that any one evaluation of *put_all(t)* will (if it terminates) output t to all the channels in some order. Any one evaluation of *get_one()* will (if it terminates) input from only one channel.

If the set in a comprehended expression is empty then the comprehended expression reduces to the unit for the combinator — **stop** for [], **swap** for ⫿ and **skip** for ‖. **stop** (deadlock) was introduced in section 24.6; **skip** (the value in **Unit**) was introduced in section 19.3; **swap** is a completely under-specified expression — it may terminate or not, may deadlock, may behave non-deterministically.

If the set in a comprehended expression is a singleton the comprehended expression is equivalent to the single expression in the set.

CHAPTER 27

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 (chapter 7) and for operations (chapter 22).

In order to be able to compare with the applicative and imperative case, we shall specify a concurrent list module abstractly. Recall the algebraic specification of the *LIST_A* module from section 7.12, which is repeated below. Constants and functions have been suffixed with an *a* to indicate that they are applicative.

```
LIST_A =
  class
    type
      List
    value
      empty_a : List,
      add_a : Int × List → List,
      head_a : List ~→ Int,
      tail_a : List ~→ List
    axiom forall i : Int, l : List •
      [head_add]
        head_a(add_a(i,l)) = i,
      [tail_add]
        tail_a(add_a(i,l)) = l
  end
```

There are four main styles of specifying a concurrent list module abstractly and we treat each of them below. The first two styles (extending an applicative module and using algebraic equivalences with explicit channels) correspond closely to the first two styles outlined for the imperative sequential case in chapter 22. The third and fourth styles (being implicit about channels and using subtypes) correspond (the latter more than the former) to the third style in the imperative sequential case.

27.1 Extending an Applicative Module

The first approach is to use the entities from the applicative *LIST_A* module in the definition of the concurrent module. The concurrent module becomes an extension of the *LIST_A* module.

```
LIST =
  extend LIST_A with
  class
    channel
      is_empty : Bool,
      add, head : Int,
      tail : Unit
    value
      list : List → in add, tail out is_empty, head Unit
    axiom forall l : List •
      list(l) ≡
        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*, which communicates with its surroundings via the channels *is_empty*, *add*, *head* and *tail* is defined.

The process is parameterized with its 'state', which has the type *List*. This type comes from the *LIST_A* module and is a sort. Nothing has been said about the representation of its values.

The process behaviour following communication on the channels is defined using calls of the corresponding applicative functions from *LIST_A*. Since these are defined without assuming any particular representation, the process shares 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 beforehand.

27.2 Algebraic Equivalences

The second approach to abstract specification of the concurrent list module is to give algebraic equivalences between process communications in a way very similar to the equivalences in the applicative *LIST_A* module.

As an example, consider the applicative axiom *head_add* from *LIST_A*:

```
  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 gives the element just added.

In the concurrent case, we have to write a bit more. First of all, we define a variable to hold the value returned from the *head* channel:

variable head_res : **Int**

We could then attempt to state the axiom 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: the *head* channel always gives the element last added on the *add* channel. In addition, a communication on the *head* channel does not affect the state of the process.

The variable *head_res* is introduced to ensure that the two arguments of ‖ have the type **Unit**.

The axiom is, however, inappropriate. The reason is that ‖ does not prevent external communication, so it allows other expressions evaluating concurrently to interfere. Such an interfering expression could perform an $add!i_1$ 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,i_1 : **Int**, l : List •
add!i_1 ‖ (list(l) ‖ (add!i ; head_res := head?)) ≡
add!i_1 ‖ (list(l) ‖ (add!i ; head_res := i))

When *list* is as in the concurrent *LIST* module in section 27.1, for the left hand side of this derived equivalence, a possible evaluation sequence is:

1. $list(l)$ accepts the communication $add!i$, thereby resulting in:
   add!i_1 ‖ list(add_a(i,l)) ‖ head_res := head?
2. $list(add\_a(i, l))$ accepts the communication $add!i_1$, thereby resulting in:
   list(add_a(i_1,add_a(i,l))) ‖ head_res := head?
3. $list(add\_a(i_1, add\_a(i, l)))$ accepts the communication *head*?, thereby resulting in:

head_res := i_1 ; list(add_a(i_1,add_a(i,l)))

This final expression is obviously not acceptable as an evaluation sequence of the right hand side of the derived equivalence — *head_res* contains the wrong value. The conclusion must therefore be that the original axiom should not hold.

The solution is to introduce a new combinator that is more 'aggressive' than the parallel combinator in forcing communication between the two expressions to happen. The interlocking combinator ⫴ does exactly that. An expression of the form:

value_expr$_1$ ⫴ value_expr$_2$

is evaluated by evaluating the two constituent expressions (both having type **Unit**) interlocked concurrently: the two expressions are evaluated concurrently until the evaluation of one of them comes to an end, whereupon evaluation continues with the other (just like ||). However: during the concurrent evaluation, any external communication is prevented. In our example above, $add!i_1$ is the external communication that should be prevented. The type of the whole expression is **Unit**.

The interlocking combinator may best be explained by stating some equivalences between expressions using it.

Assume the following definitions:

value e,e_1,e_2 : T
channel c,c_1,c_2 : 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 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 to 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 := c_1? [] c_2!e_2) ⫴ c_1!e_1 ≡ x := e_1

That is: the interlocking combinator forces 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 expres-

sions want to communicate, but not with each other, the result is a deadlock. This is illustrated by the following equivalence:

$x := c_1? \parallel c_2!e \equiv \textbf{stop}$

The corresponding equivalence for the 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:

$(x := c_1? \sqcap c_2!e_2) \parallel c_1!e_1 \equiv x := e_1 \sqcap \textbf{stop}$

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 is a deadlock.

The interlocking combinator is commutative but it is not associative (by contrast with the 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
```

Note that the expression $list(l) \parallel add!i$ is itself a processs, namely the process representing the list which has had integer i added to its 'state' l.

The complete concurrent specification of lists becomes:

```
LIST =
  class
    type
      List
    channel
      is_empty : Bool,
      add, head : Int,
      tail : Unit
    variable
      is_empty_res : Bool,
      head_res : Int
    value
      empty : List,
      list : List → in add, tail out is_empty, head Unit
    axiom forall i : Int, l : List •
```

```
    [is_empty_empty]
      list(empty) ⫴ is_empty_res := is_empty? ≡
        list(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),
    [add_list]
      ∃ l′ : List • □ list(l) ⫴ add!i ≡ list(l′)
end
```

The last axiom is special to this kind of specification. We noted above that the expression $list(l) \between add!i$ is intended to be the list process whose 'state' is the result of adding i to the state l. But there is nothing in the specification to ensure that it is a process of the appropriate type, i.e. willing to do the communications that the *list* process is willing to do. In the corresponding applicative specification we had an *add_a* function with result type *List*. The axiom *add_list* represents this information about the 'result type' of the interactions on this channels.

So why do we need this axiom for *add* only, and not for *tail*, say? It turns out that we are using interactions on *add* like a 'constructor' for the list process, as we can see from the structure of the axioms (just as we used the constant *empty_a* and the function *add_a* as constructors for applicative lists). Hence these are sufficient; everything else is defined in terms of them. The constant *empty* is defined to be of type *List*; for the function *add* we need this axiom.

The parameter type of the *list* process is *List* which is a sort. Nothing has therefore been said about representation. The axioms likewise assume no particular representation of lists.

27.3 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 are, additionally, implicit about what the channels are, by simply not defining them. Instead, the technique is to define the interface to the *list* process as a set of interface functions (see section 24.9) and then to state their properties in terms of the interlocking combinator.

In this way we can modify the *LIST* module in section 27.2 by removing the following definitions:

```
  channel
    is_empty : Bool,
```

add, head : **Int**,
tail : **Unit**

The *list* process type must now be modified so that it does not mention the channels. Recall that it had the following definition:

list : List → **in** 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 may be communicated on. An access description can thus have one of the forms **in any** and **out any**.

The definition of the type of *list* becomes:

list : List → **in any out any Unit**

The interface functions must also have **any** accesses. As an example let us consider the *add* and *head* interface functions which could be given the types:

add : **Int** → **in any out any Unit**
head : **Unit** → **in any out any Int**

The interlocking combinator (unlike ';') requires its argument expressions to both have type **Unit**. This results in the awkward use of extra 'result' variables in the *LIST* module in section 27.2.

This use of extra variables can be eliminated quite easily by noting that we are actually interested in what happens when the results of applying *is_empty* and *head* are used subsequently. This subsequent use can be described by quantifying over 'test functions' having the types **Bool** $\xrightarrow{\sim}$ **Unit** (in the case of *is_empty*) and **Int** $\xrightarrow{\sim}$ **Unit** (in the case of *head*). These types contain enough functions to discriminate between all possible Booleans and integers; for instance if b_1 : **Bool** and b_2 : **Bool** then, if l is not $\lambda() \bullet$ **chaos**, the definition

value
test_bool : **Bool** $\xrightarrow{\sim}$ **Unit**
axiom
test_bool = λ b_2 : **Bool** • **if** $b_1 = b_2$ **then skip else chaos end**

provides a function *test_bool* such that:

$(\Box\ (l() \parallel \text{test\_bool}(b_1) \equiv l() \parallel \text{test\_bool}(b_2))) \Rightarrow b_1 = b_2$

or, equivalently, such that:

$(\lambda() \bullet l() \parallel \text{test\_bool}(b_1)) = (\lambda() \bullet l() \parallel \text{test\_bool}(b_2)) \Rightarrow b_1 = b_2$

Though it would be possible to introduce test functions in the axioms handling functions (such as *tail*) returning results of type **Unit**, doing so is unnecessary.

It would have been possible to use test functions in the corresponding sequential specification in section 22.3, but they would merely obscure it. The concurrent version is as follows:

LIST =
class

```
  type
    List
  value
    empty : List,
    is_empty : Unit ~→ in any out any Bool,
    add : Int ~→ in any out any Unit,
    head : Unit ~→ in any out any Int,
    tail : Unit ~→ in any out any Unit,
    list : List → in any out any Unit
  axiom forall i : Int, l : List,
                  test_bool : Bool ~→ Unit, test_int : Int ~→ Unit •
    [is_empty_empty]
      list(empty) ‖ test_bool(is_empty()) ≡
        list(empty) ‖ test_bool(true),
    [is_empty_add]
      (list(l) ‖ add(i)) ‖ test_bool(is_empty()) ≡
        (list(l) ‖ add(i)) ‖ test_bool(false),
    [head_add]
      (list(l) ‖ add(i)) ‖ test_int(head()) ≡
        (list(l) ‖ add(i)) ‖ test_int(i),
    [tail_add]
      (list(l) ‖ add(i)) ‖ tail() ≡ list(l),
    [add_list]
      ∃ l′ : List • □ list(l) ‖ add(i) ≡ list(l′)
end
```

27.4 Using Subtypes

The previous specification in section 27.3 holds the 'state' in a type *List* which appears as a parameter to the function *list*. There was no such corresponding type in the corresponding imperative sequential specification in section 22.3.

We can eliminate this type by introducing instead a type of list processes, which is a subtype of all the processes that may access the variables and channels identified by **any**. We can imagine different values of this type representing different 'states'. This we do next, for our fourth version.

```
LIST =
  class
    type
      List = {| l : Unit → in any out any write any Unit • is_list(l) |}
    value
      is_list : (Unit → in any out any write any Unit) → Bool,
      empty : List,
      is_empty : Unit ~→ in any out any Bool,
```

```
    add : Int ~> in any out any Unit,
    head : Unit ~> in any out any Int,
    tail : Unit ~> in any out any Unit
  axiom forall i : Int, l : List,
                test_bool : Bool ~> Unit, test_int : Int ~> Unit •
    [is_empty_empty]
      empty() ||| test_bool(is_empty()) ≡
        empty() ||| test_bool(true),
    [is_empty_add]
      (l() ||| add(i)) ||| test_bool(is_empty()) ≡
        (l() ||| add(i)) ||| test_bool(false),
    [head_add]
      (l() ||| add(i)) ||| test_int(head()) ≡
        (l() ||| add(i)) ||| test_int(i),
    [tail_add]
      (l() ||| add(i)) ||| tail() ≡ l(),
    [add_list]
      ∃ l' : List • □ l() ||| add(i) ≡ l'()
end
```

The following has been gained by being implicit about channels and using interface functions:

- We have avoided deciding what channels there will be and what their types will be.
- Suppose we later develop an implementation of the *LIST* module from section 27.3 or 27.4. Our specification then places no restriction on what the channels of an implementation will be.
- The specification places no restrictions on what channels the processes are allowed to access. The implementation of abstract interface functions in terms of concrete interface functions that do explicit channel communication is sometimes referred to as event refinement. This is particularly useful in that we can decide later what protocols to use, such as designing external choices over inputs only if the programming language we want to finally implement in enforces such a restriction. (We could design in this way from the start, but only at the cost of a less abstract and less re-usable specification.)

Note that **any** accesses can also be used in process types even if channels have been defined. 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 is not interested (yet) in what channels there are.
- One wants to leave freedom to a later development which is expected to be-

come an implementation in the formal sense. The freedom to do event refinement is particularly important.

Being explicit, however, has its benefits. From the type of a process one can see exactly what channels may be accessed and how they may be accessed. This can make concurrent specifications easier to read.

A more detailed description of **any** accesses will be given in section 33.3.

27.5 Example: A Database

Consider an algebraic specification of a concurrent database. We will be implicit about channels by not defining any. Consequently, we must define a set of interface functions.

```
DATABASE =
  class
    type
      Key, Data,
      Database =
        {| db : Unit → in any out any write any Unit • is_database(db) |}
    value
      is_database : (Unit → in any out any write any Unit) → Bool,
      empty : Database,
      insert : Key × Data ~→ in any out any Unit,
      remove : Key ~→ in any out any Unit,
      defined : Key ~→ in any out any Bool,
      lookup : Key ~→ in any out Data
    axiom forall k,k₁ : Key, d : Data, db : Database,
                  test_bool : Bool ~→ Unit, test_data : Data ~→ Unit •
      [remove_empty]
        empty() ‖ remove(k) ≡ empty(),
      [remove_insert]
        (db() ‖ insert(k₁,d)) ‖ remove(k) ≡
          if k = k₁ then db() ‖ remove(k)
          else (db() ‖ remove(k)) ‖ insert(k₁,d)
          end,
      [defined_empty]
        empty() ‖ test_bool(defined(k)) ≡
          empty() ‖ test_bool(false),
      [defined_insert]
        (db() ‖ insert(k₁,d)) ‖ test_bool(defined(k)) ≡
          if k = k₁ then (db() ‖ insert(k₁,d)) ‖ test_bool(true)
          else (db() ‖ test_bool(defined(k))) ‖ insert(k₁,d)
          end,
      [lookup_insert]
```

(db() ⫴ insert(k_1,d)) ⫴ test_data(lookup(k)) ≡
if k = k_1 **then** (db() ⫴ insert(k_1,d)) ⫴ test_data(d)
else (db() ⫴ test_data(lookup(k))) ⫴ insert(k_1,d)
end,
[insert_database]
∃ db′ : Database • □ db() ⫴ insert(k,d) ≡ db′()
end

The concurrent database example illustrates the constructor technique for inventing axioms, which we have previously seen applied in the imperative sequential case as well as in the applicative case. The technique used in the concurrent case with interface functions can be characterized as follows:

1. Identify the 'constructor functions' 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:

   empty() ⫴ insert(k_1,d_1) ⫴ ... ⫴ insert(k_n,d_n)

2. Define the remaining processes by case over the constructor processes, using new identifiers as parameters. In the above axioms, *remove*, *defined* and *lookup* are defined over the two constructor expressions:

   empty()
   db() ⫴ insert(k_1,d)

   We thus get immediately all the left hand sides of the axioms we need. That is:

   empty() ⫴ remove(k)
   (db() ⫴ insert(k_1,d)) ⫴ remove(k)
   empty() ⫴ test_bool(defined(k))
   (db() ⫴ insert(k_1,d)) ⫴ test_bool(defined(k))
   empty() ⫴ test_data(lookup(k))
   (db() ⫴ insert(k_1,d)) ⫴ test_data(lookup(k))

   Note, however, that we choose to under-specify the effect of *lookup* and so we do not include an axiom with left hand side *empty*() ⫴ *test_data*(*lookup*(*k*)).

   The right hand sides of the axioms *defined_insert* and *lookup_insert* are somewhat different from the corresponding applicative ones. This is due to the requirement that the effect on the state of the left hand side of an equivalence must be the same as the effect on the state of the right hand side. More specifically, the call *insert*(k_1, *d*) (or its equivalent) must occur on the right hand side since it occurs on the left hand side and since it has a non-trivial effect on the state. The use of test functions allows values which do not have type **Unit** to be returned by function calls which appear as arguments of ⫴.

The *LIST* axioms in section 27.4 actually have the same form.

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

27.6 Refining Applicative Specifications into Imperative Ones

The analogy between concurrent imperative specifications like that in section 27.4 and applicative specifications can be formalized: there is a sense in which such concurrent imperative specifications are essentially refinements of the applicative ones. Here we illustrate the formalization in the case of lists.

An applicative specification of this kind of list is as follows.

```
LIST_A =
  class
    type
      List
    value
      empty_a : List,
      add_a : Int × List → List,
      is_empty_a : List ~→ Bool,
      head_a : List ~→ Int,
      tail_a : List ~→ List
    axiom forall i : Int, l : List •
      [is_empty_empty]
        is_empty_a(empty_a) = true,
      [is_empty_add]
        is_empty_a(add_a(i,l)) = false,
      [head_add]
        head_a(add_a(i,l)) = i,
      [tail_add]
        tail_a(add_a(i,l)) = l
  end
```

A concurrent imperative specification produced by analogy with this applicative specification is as follows.

```
LIST =
  class
    type
      List = {| l : Unit → in any out any write any Unit • is_list(l) |}
    value
      is_list : (Unit → in any out any write any Unit) → Bool,
      empty : List,
      is_empty : Unit ~→ in any out any Bool,
      add : Int ~→ in any out any Unit,
      head : Unit ~→ in any out any Int,
      tail : Unit ~→ in any out any Unit
    axiom forall i : Int, l : List,
```

```
                    test_bool : Bool ~→ Unit, test_int : Int ~→ Unit •
    [is_empty_empty]
      empty() ‖ test_bool(is_empty()) ≡
        empty() ‖ test_bool(true),
    [is_empty_add]
      (l() ‖ add(i)) ‖ test_bool(is_empty()) ≡
        (l() ‖ add(i)) ‖ test_bool(false),
    [head_add]
      (l() ‖ add(i)) ‖ test_int(head()) ≡
        (l() ‖ add(i)) ‖ test_int(i),
    [tail_add]
      (l() ‖ add(i)) ‖ tail() ≡ l(),
    [add_list]
      ∃ l′ : List • □ l() ‖ add(i) ≡ l′()
end
```

LIST can be extended by defining new types, constants and functions in the following manner.

```
LIST_B =
  extend LIST with
  class
    value
      empty_a : List,
      add_a : Int × List → List,
      is_empty_a : List ~→ Bool,
      head_a : List ~→ Int,
      tail_a : List ~→ List
    axiom forall i : Int, l : List •
      empty_a = empty,
      add_a(i,l) = λ() • l() ‖ add(i),
      is_empty_a(l) =
        let b : Bool • ∀ test_bool : Bool ~→ Unit •
          (λ() • l() ‖ test_bool(is_empty())) = (λ() • l() ‖ test_bool(b))
        in b end,
      head_a(l) =
        let i : Int • ∀ test_int : Int ~→ Unit •
          (λ() • l() ‖ test_int(head())) = (λ() • l() ‖ test_int(i))
        in i end,
      tail_a(l) = λ() • l() ; tail()
  end
```

The axioms of *LIST_B* ensure that *is_list*(*empty*) is **true** and that so is:

```
∀ i : Int, l : Unit → in any out any write any Unit •
  is_list(l) ⇒ is_list(λ() • l() ‖ add(i))
```

LIST_B does not constrain the functions introduced in *LIST*: nothing can be proved about the functions introduced in *LIST* with the aid of this extension of it that could not be proved without the extension.

Because all the functions in the type *List* are total, so in particular $\lambda() \bullet \textbf{chaos}$ is not in the type *List*, it is the case that:

$$\begin{array}{l}\forall\ b_1, b_2 \bullet \textbf{Bool} \bullet \\ \quad (\forall\ \text{test\_bool} : \textbf{Bool} \xrightarrow{\sim} \textbf{Unit} \bullet \\ \qquad (\lambda() \bullet \text{empty}() \parallel \text{test\_bool}(b_1)) = (\lambda() \bullet \text{empty}() \parallel \text{test\_bool}(b_2)) \\ \quad) \Rightarrow b_1 = b_2\end{array}$$

and that:

$$\begin{array}{l}\forall\ b_1, b_2 \bullet \textbf{Bool}, i : \textbf{Int}, l : \text{List} \bullet \\ \quad (\forall\ \text{test\_bool} : \textbf{Bool} \xrightarrow{\sim} \textbf{Unit} \bullet \\ \qquad (\lambda() \bullet (l() \parallel \text{add}(i)) \parallel \text{test\_bool}(b_1)) = \\ \qquad (\lambda() \bullet (l() \parallel \text{add}(i)) \parallel \text{test\_bool}(b_2)) \\ \quad) \Rightarrow b_1 = b_2\end{array}$$

and similarly that:

$$\begin{array}{l}\forall\ i_1, i_2 \bullet \textbf{Int}, i : \textbf{Int}, l : \text{List} \bullet \\ \quad (\forall\ \text{test\_int} : \textbf{Int} \xrightarrow{\sim} \textbf{Unit} \bullet \\ \qquad (\lambda() \bullet (l() \parallel \text{add}(i)) \parallel \text{test\_int}(i_1)) = \\ \qquad (\lambda() \bullet (l() \parallel \text{add}(i)) \parallel \text{test\_int}(i_2)) \\ \quad) \Rightarrow i_1 = i_2\end{array}$$

From this it follows that the functions *empty_a*, *add_a*, *is_empty_a*, *head_a* and *tail_a* defined explicitly in *LIST_B* satisfy all the axioms in the original applicative *LIST_A*. Indeed, *LIST_B* is a refinement of *LIST_A*, as everything that can be proved about the latter can be proved about the former. So, by refining an abstract type *List* from *LIST_A* into (a subtype of) a process type in *LIST* we have refined an applicative specification into a concurrent imperative one.

CHAPTER 28

Modules

Modules are the means by which to decompose specifications into comprehensible and reusable units. A module is basically a named collection of declarations. A module M_1 can be used to define another module M_2, meaning that the declarations of M_1 are used to define M_2.

As we shall see, there are two kinds of modules: objects and schemes. All the modules shown so far in this tutorial are actually schemes. Modules are the only entities that can be defined at the outermost level of a specification. In other words, a specification is a collection of module declarations.

Before introducing objects and schemes, the concept of a basic class expression is described. Class expressions are fundamental in the definition of both objects and schemes.

28.1 Basic Class Expressions

The kernel module concept is that of a class expression. The basic form of a class expression is:

class
  declaration_1
  ⋮
  declaration_n
end

for $n \geq 0$. That is, a collection of declarations enclosed by the keywords **class** and **end**.

We have seen many examples of basic class expressions, and we have seen examples of type, value, variable, channel and axiom declarations. As we shall see, class expressions are entities in their own right, and can therefore also be defined in declarations (of schemes).

A class expression represents a class (essentially a set) of models. Each model associates an entity (value, type, variable, channel or module) with each identifier

defined within the class expression. There may be more than one model because identifiers may be under-specified.

As an example consider the following class expression:

class
  value i : **Int**
  axiom i = 1 ∨ i = 2
end

This class expression represents the class containing two models, corresponding to the fact that i is under-specified. The class can be illustrated as follows:

{ [i ↦ 1] , [i ↦ 2] }

The class represented by a basic class expression is the class containing all models that satisfy the declarations. Each of the two models above satisfies the declarations. Take for example the first model:

[i ↦ 1]

This model satisfies the declaration of i as an integer since i is bound to a value within **Int**. The model also satisfies the axiom that i must be either 1 or 2 since i is bound to 1. Similarly for the second model.

In fact, this description is a simplification. In order to allow extension (adding new definitions) as implementations we define the models of a class expression to include all possible extensions. This means that the class of models of any class expression is infinite. But we can characterize the classes of models for our simple example as all containing an association of the value i to either 1 or 2.

Class expressions may also have other forms, as we shall see in subsequent sections. The other forms can, however, usually be expanded into equivalent basic class expressions. We refer to the category of class expressions as *class_expr*.

In the following we see how one can give names to models (giving objects) and class expressions (giving schemes).

28.2 Objects

An object is essentially a named model chosen from a class of models represented by some class expression. As an example, consider the following object.

object
  LIST :
    class
      variable
        list : **Int***
      value
        empty : **Unit** → **write** list **Unit**,
        is_empty : **Unit** → **read** list **Bool**,
        add : **Int** → **write** list **Unit**,

```
      head : Unit ~→ read list Int,
      tail : Unit ~→ write list Unit
    axiom forall i : Int •
      empty() ≡ list := ⟨⟩,
      is_empty() ≡ list = ⟨⟩,
      add(i) ≡ list := ⟨i⟩ ⌢ list,
      head() ≡ hd list
      pre ~is_empty(),
      tail() ≡ list := tl list
      pre ~is_empty()
  end
```

The object has a name, *LIST*, and it provides a number of named entities, *list*, *empty*, *is_empty*, *add*, *head* and *tail*. One may say that the object is an 'instance' of the class expression, a phrasing that perhaps supports the reader's intuition.

All variables defined in the class expression are initialised to their initial value. In the above class expression no specification of an initial *list* value has been given, so an arbitrary value within $\mathbf{Int}^*$ is assigned to the *list* variable. If, instead, the variable definition is given as:

```
variable list : Int* := ⟨⟩
```

then the initial value will be the empty list.

The entities within one object can be referred to from another object. All references must be prefixed with the object name, in this case *LIST*. In the case of, for example, the entity *empty*, the reference is *LIST.empty*. Such a reference is called a qualified identifier. Note that this entity is an operation. If one wants to evaluate this operation one has to write *LIST.empty*(). In general, a reference may be of the form $id_1.id_2$ where id_1 is the object name and where id_2 is the entity name.

As an example of an object that refers to the *LIST* object, consider an object providing an operation that applies an integer-to-integer function to all members of the list currently contained in the *LIST.list* variable.

```
object
  LIST_APPLY :
    class
      value
        apply : (Int → Int) → write LIST.list Unit
      axiom forall f : Int → Int •
        apply(f) ≡
          if ~ LIST.is_empty() then
            let first = LIST.head() in
              LIST.tail() ; apply(f) ; LIST.add(f(first))
            end
          end
  end
```

The operation *LIST_APPLY.apply* accesses the variable *LIST.list* via the operations defined within *LIST*. It is important to note that there is only one variable defined above, namely *LIST.list*, and that this variable is the one accessed and eventually modified by *LIST_APPLY.apply* as well as by all the *LIST* operations.

An object may also define operators. As an example, recall the specification of rational numbers from section 17.6, which is repeated below, here defined as an object.

```
object RATIONAL :
  class
    type
      Rational
    value
      / : Int × Int → Rational,
      + : Rational × Rational → Rational,
      − : Rational × Rational → Rational,
      ∗ : Rational × Rational → Rational,
      / : Rational × Rational → Rational,
      real : Rational ~→ Real
    axiom forall n,n1,n2,d,d1,d2 : Int, r1,r2 : Rational •
      (n1 / d1) + (n2 / d2) ≡ (n1 ∗ d2 + d1 ∗ n2) / (d1 ∗ d2),
      r1 − r2 ≡ r1 + (r2 ∗ ((0−1)/1)),
      (n1 / d1) ∗ (n2 / d2) ≡ (n1 ∗ n2) / (d1 ∗ d2),
      (n1 / d1) / (n2 / d2) ≡ (n1 ∗ d2) / (d1 ∗ n2),
      real (n / d) ≡ (real n) / (real d)
      pre d ≠ 0
  end
```

All the operators defined within *RATIONAL* can only be accessed by prefixing them with *RATIONAL*, just as is the case for identifiers, although the syntax is slightly different. In the case of for example the + operator, the reference is *RATIONAL.*(+). Such a reference is called a qualified operator. The obtained entity is a function that must be applied using function application notation. Recall from section 17.3 how user-defined operators are turned into functions by bracketing them. Thus, an application of + to the two rational numbers r_1 and r_2 has the form *RATIONAL.*(+)(r_1, r_2). In general, a reference may be of the form *id.*(*op*) where *id* is the object name and where *op* is the operator.

As an example of an object that refers to the *RATIONAL* object, consider one providing a function for multiplying all the rational numbers which are members of a list.

```
object
  RATIONAL_MULT :
    class
      value
        multiply : RATIONAL.Rational* → RATIONAL.Rational
```

```
      axiom forall l : RATIONAL.Rational* •
        multiply(l) ≡
          case l of
            ⟨⟩ → RATIONAL.(/)(1,1),
            ⟨r⟩ ⌢ l1 → RATIONAL.(*)(r,multiply(l1))
          end
    end
```

Objects are essentially entities just like types, values, variables and channels. They are therefore defined in declarations. An object declaration has the form:

object
  object_definition$_1$,
  ⋮
  object_definition$_n$

for $n \geq 1$. An object definition (in the simple form we have seen so far) has the form:

  id : class_expr

That is, the identifier *id* is defined to represent some arbitrary model belonging to the class of models represented by the *class_expr*.

Since objects are defined in declarations, they may be defined inside class expressions in a nested manner. Chapter 31 illustrates this in more detail.

28.3 Schemes

An object represents a model, arbitrarily chosen from the class of models represented by some class expression. In some situations it is convenient to be able to manipulate class expressions on their own before defining objects. A prerequisite for doing this is that one can name class expressions.

A named class expression is called a scheme. As an example we can define a scheme corresponding to the class expression defining the *LIST* object above.

```
scheme
  LIST_S =
    class
      variable
        list : Int*
      value
        empty : Unit → write list Unit,
        is_empty : Unit → read list Bool,
        add : Int → write list Unit,
        head : Unit ~→ read list Int,
        tail : Unit ~→ write list Unit
      axiom forall i : Int •
```

```
    empty() ≡ list := ⟨⟩,
    is_empty() ≡ list = ⟨⟩,
    add(i) ≡ list := ⟨i⟩ ⌢ list,
    head() ≡ hd list
    pre ~is_empty(),
    tail() ≡ list := tl list
    pre ~is_empty()
end
```

The suffix S is used in the scheme-name so that we are able to distinguish it from the object. Note that this is just a convention used in this particular context, and is therefore neither a naming required by RSL nor a suggested style.

The scheme *LIST_S* is defined to be an abbreviation for its defining class expression. Every occurrence of the name *LIST_S*, called a scheme instantiation, can thus be replaced with the class expression (and vice versa).

As an example we can define two list objects, each an instance of the *LIST_S* scheme.

```
object
  LIST1 : LIST_S,
  LIST2 : LIST_S
```

These definitions are exactly equivalent to the following, where the two occurrences of the scheme-name *LIST_S* have been replaced with its defining class expression.

```
object
  LIST1 : class ... end,
  LIST2 : class ... end
```

In this way we have defined two objects defining two different variables. That is, the variable *LIST1.list* is different from *LIST2.list*.

It is important to note that the two object definitions above could have been obtained without introducing a scheme. One would then instead be forced to write the expanded forms sketched above. Such a style both requires more writing and blurs the fact that the two objects are instances of the same class expression.

Schemes are entities just like types, values, variables, channels and objects. They are therefore defined in declarations. A scheme declaration has the form:

```
scheme
  scheme_definition1,
  ⋮
  scheme_definitionn
```

for $n \geq 1$. A scheme definition (in the simple form we have seen so far) has the form:

```
  id = class_expr
```

That is, the identifier *id* is defined to be an abbreviation for the *class_expr*.

As mentioned earlier, all the modules shown in chapters 3–27 of this tutorial are schemes.

28.4 Extension

One can in RSL build a class expression in successive steps, at each step adding declarations with the **extend** operator.

As an example, suppose we want to gradually build up the scheme *LIST_S*. In the first step we decide what the state is. That is, we define the basic scheme *LIST_STATE*.

```
scheme
  LIST_STATE =
    class
      variable list : Int*
    end
```

In the second step we identify all the operations on the state, but we ignore any axioms defining properties of the operations. The addition of the operations is expressed as an extension of the *LIST_STATE* scheme.

```
scheme
  LIST_OPERATIONS =
    extend LIST_STATE with
    class
      value
        empty : Unit → write list Unit,
        is_empty : Unit → read list Bool,
        add : Int → write list Unit,
        head : Unit ~→ read list Int,
        tail : Unit ~→ write list Unit
    end
```

Such an extension can be expanded into a basic class expression by simply combining the declarations of the extended class expression with the new ones. The above one can therefore be expanded into the following scheme definition.

```
scheme
  LIST_OPERATIONS =
    class
      variable
        list : Int*
      value
        empty : Unit → write list Unit,
        is_empty : Unit → read list Bool,
        add : Int → write list Unit,
        head : Unit ~→ read list Int,
```

```
      tail : Unit ~→ write list Unit
  end
```

In the third and final step we add axioms defining properties of the operations.

```
scheme
  LIST_S =
    extend LIST_OPERATIONS with
    class
      axiom forall i : Int •
        empty() ≡ list := ⟨⟩,
        is_empty() ≡ list = ⟨⟩,
        add(i) ≡ list := ⟨i⟩ ^ list,
        head() ≡ hd list
        pre ~is_empty(),
        tail() ≡ list := tl list
        pre ~is_empty()
    end
```

This scheme-definition is exactly equivalent to the original definition of *LIST_S*, where the state, the operations and the axioms were defined in one basic class expression.

The general form of an extending class expression is:

extend class_expr$_1$ **with** class_expr$_2$

The class expression *class_expr*$_1$ is typically a scheme name, while the class expression *class_expr*$_2$ is typically a basic class expression of the form:

```
class
  declaration_1
  ⋮
  declaration_n
end
```

The constituent class expressions must be compatible. They are compatible if all their definitions are compatible. Two definitions are compatible if they define distinct identifiers and operators or if they are both value definitions defining the same identifier or operator but with distinguishable maximal types.

So if for example the scheme S_1 defines a type named T, and if the scheme S_2 also defines a type named T, then the following extension is not well-formed:

extend S_1 **with** S_2

Suppose each of the class expressions *class_expr*$_i$ ($i \in \{1, 2\}$) can be expanded into a basic class expression of the form:

```
class
  declaration_i,1
```

$\vdots$
$\text{declaration}_{\text{i,n}_\text{i}}$
end

Then the extension above can be expanded into:

class
$\text{declaration}_{1,1}$
$\vdots$
$\text{declaration}_{1,\text{n}_1}$
$\text{declaration}_{2,1}$
$\vdots$
$\text{declaration}_{2,\text{n}_2}$
end

CHAPTER 29

Renaming and Hiding

In addition to the **extend** operator there are two more operators available on class expressions, namely renaming and hiding.

29.1 Renaming

A class expression may be renamed giving a new class expression with old names replaced by new names. As an example, consider the following renaming of the *LIST_S* scheme making it into a stack scheme.

scheme
  STACK_S =
    use stack **for** list, push **for** add, top **for** head, pop **for** tail
    in LIST_S

A renaming class expression can be expanded into a basic class expression. The result of the expansion is as follows.

scheme
  STACK_S =
    class
      variable
        stack : **Int***
      value
        empty : **Unit** $\to$ **write** stack **Unit**,
        is_empty : **Unit** $\to$ **read** stack **Bool**,
        push : **Int** $\to$ **write** stack **Unit**,
        top : **Unit** $\xrightarrow{\sim}$ **read** stack **Int**,
        pop : **Unit** $\xrightarrow{\sim}$ **write** stack **Unit**
      axiom forall i : **Int** •
        empty() $\equiv$ stack := $\langle\rangle$,
        is_empty() $\equiv$ stack = $\langle\rangle$,
        push(i) $\equiv$ stack := $\langle$i$\rangle$ $^\frown$ stack,

```
        top() ≡ hd stack
        pre ~is_empty(),
        pop() ≡ stack := tl stack
        pre ~is_empty()
    end
```

The general form of a renaming class expression is:

```
use
  id_new1 for id_old1, ... ,id_newn for id_oldn
in class_expr
```

for $n \geq 1$. The new names appear to the left of **for** while the old names appear to the right. In the case where an overloaded identifier is to be renamed, a type expression must be given in order to disambiguate it:

```
use
  ..., id_new for id_old : type_expr, ...
in class_expr
```

29.2 Hiding

Identifiers defined within a class expression may be hidden so that one cannot refer to them outside the class expression.

As an example we can hide make the variable *list* in the *LIST_S* scheme. This is a common encapsulation technique which prevents variables from being accessed except through their associated operations.

```
scheme ENCAPSULATED_LIST_S = hide list in LIST_S
```

In order to illustrate what hiding means, two illegal uses of the *ENCAPSULATED_LIST_S* are shown below. In the first example, we extend the scheme with an operation that illegally refers to the hidden list variable.

```
scheme
  ILLEGAL_S =
    extend ENCAPSULATED_LIST_S with
    class
      value length : Unit → read list Nat
      axiom length() ≡ len list
    end
```

In the second example, an object is first defined as an instance of the scheme *ENCAPSULATED_LIST_S*.

```
object ENCAPSULATED_LIST : ENCAPSULATED_LIST_S
```

Then another object is defined that illegally refers to the hidden list variable.

```
object
  ILLEGAL :
```

```
class
  value length : Unit → read ENCAPSULATED_LIST.list Nat
  axiom length() ≡ len ENCAPSULATED_LIST.list
end
```

Suppose an operation is to be defined, which may call the operations defined in *ENCAPSULATED_LIST*. The type of this operation must include an access description stating the fact that the variables (in this case one variable) of *ENCAPSULATED_LIST* are accessed. This description must not mention *list*, which is hidden in *ENCAPSULATED_LIST*. The variables of *ENCAPSULATED_LIST* can instead be referred to in accesses by:

ENCAPSULATED_LIST.**any**

That is, a reference to the particular *list* variable is avoided. The type of the calling operation must therefore include an access description of the form:

read ENCAPSULATED_LIST.**any**

if it just calls the read-only operations (*is_empty* and *head*). Alternatively it must include an access description of the form:

write ENCAPSULATED_LIST.**any**

if it calls operations modifying the list variable (*empty*, *add* and *tail*). As an example, consider the following object.

```
object
  LIST_APPLY :
    class
      value
        apply : (Int → Int) → write ENCAPSULATED_LIST.any Unit
      axiom forall f : Int → Int •
        apply(f) ≡
          if ∼ ENCAPSULATED_LIST.is_empty() then
            let first = ENCAPSULATED_LIST.head() in
              ENCAPSULATED_LIST.tail() ;
              apply(f) ;
              ENCAPSULATED_LIST.add(f(first))
            end
          end
    end
```

The general form of a hiding class expression is:

hide id_1,...,id_n **in** class_expr

for $n \geq 1$. In the case where an overloaded identifier is to be hidden, a type expression must be given in order to disambiguate it:

hide ..., id : type_expr, ... **in** class_expr

A hiding class expression cannot immediately be expanded into a basic class expression, since hiding is not a concept expressible in basic class expressions. One can therefore say that the general form of class expression is:

hide id_1,...,id_m **in**
class
  $declaration_1$
  ⋮
  $declaration_n$
end

The above definition of the *ENCAPSULATED_LIST_S* scheme is thus equivalent to the following definition.

```
scheme
  ENCAPSULATED_LIST_S =
    hide list in
    class
      variable
        list : Int*
      value
        empty : Unit → write list Unit,
        is_empty : Unit → read list Bool,
        add : Int → write list Unit,
        head : Unit ~→ read list Int,
        tail : Unit ~→ write list Unit
      axiom forall i : Int •
        empty() ≡ list := ⟨⟩,
        is_empty() ≡ list = ⟨⟩,
        add(i) ≡ list := ⟨i⟩ ⌢ list,
        head() ≡ hd list
        pre ~is_empty(),
        tail() ≡ list := tl list
        pre ~is_empty()
    end
```

The other operators on class expressions (extension and renaming) should really have been defined on this general form. This has, however, been avoided to obtain simpler explanations.

Another typical use of hiding is the hiding of auxiliary functions by means of which more central functions are defined.

Note that hiding in RSL only restricts visibility. The names hidden are not available outside the class expression but its properties are unchanged.

CHAPTER 30

Parameterized Schemes

The modules in chapter 29 specify lists with element type **Int**. A more general solution would be to turn the element type into a parameter, thereby making it possible to specify lists with arbitrary element types.

RSL allows for such parameterization by allowing schemes to be parameterized with objects.

30.1 Simple Parameterization and Instantiation

A parameterized version of lists is the following.

```
scheme
  PARAM_LIST(E : class type Elem end) =
    class
      variable
        list : E.Elem*
      value
        empty : Unit → write list Unit,
        is_empty : Unit → read list Bool,
        add : E.Elem → write list Unit,
        head : Unit ~→ read list E.Elem,
        tail : Unit ~→ write list Unit
      axiom forall e : E.Elem •
        empty() ≡ list := ⟨⟩,
        is_empty() ≡ list = ⟨⟩,
        add(e) ≡ list := ⟨e⟩ ^ list,
        head() ≡ hd list
        pre ~is_empty(),
        tail() ≡ list := tl list
        pre ~is_empty()
    end
```

The *PARAM_LIST* scheme is parameterized over the list element type *Elem*. More precisely, the scheme may be instantiated with any object, that at least defines a type named *Elem*. (See section 30.5 for details of the necessary relation between formal and actual scheme parameters.) When instantiated with such an object, the result will be a class expression referring to that object.

A definition of a parameterized scheme can have the form:

$\mathrm{id}(\mathrm{id}_1 : \mathrm{class\_expr}_1, ..., \mathrm{id}_n : \mathrm{class\_expr}_n) = \mathrm{class\_expr}$

for $n \geq 1$. Such a scheme has n parameters. If oid_1 ... oid_n are identifiers representing objects, then the scheme may be instantiated with these objects in a scheme instantiation as follows:

$\mathrm{id}(\mathrm{oid}_1, ..., \mathrm{oid}_n)$

Note, that one must ensure that each oid_i satisfies the parameter requirement $class\_expr_i$. What that means in general will be explained later. In case of *PARAM_LIST*, this scheme may be instantiated with an object that at least provides a type named *Elem*.

Let us instantiate the scheme with an object. Consider for example the following object providing a type *Elem* representing the integers.

object INTEGER : **class type** Elem = **Int end**

The *PARAM_LIST* scheme can now be instantiated with *INTEGER* as follows.

object INTEGER_LIST : PARAM_LIST(INTEGER)

That is, the *INTEGER_LIST* object is defined to represent some arbitrary model within the class represented by the class expression *PARAM_LIST*(*INTEGER*). This class expression can be expanded into a basic class expression by replacing all occurrences of the formal parameter name E within the class expression defining *PARAM_LIST* with *INTEGER*. The whole object definition can therefore be expanded into the following.

object
  INTEGER_LIST :
    class
      variable
        list : INTEGER.Elem*
      value
        empty : **Unit** → **write** list **Unit**,
        is_empty : **Unit** → **read** list **Bool**,
        add : INTEGER.Elem → **write** list **Unit**,
        head : **Unit** $\xrightarrow{\sim}$ **read** list INTEGER.Elem,
        tail : **Unit** $\xrightarrow{\sim}$ **write** list **Unit**
      axiom forall e : INTEGER.Elem •
        empty() ≡ list := ⟨⟩,
        is_empty() ≡ list = ⟨⟩,

```
        add(e) ≡ list := ⟨e⟩ ^ list,
        head() ≡ hd list
        pre ∼is_empty(),
        tail() ≡ list := tl list
        pre ∼is_empty()
    end
```

The *INTEGER_LIST* object can now be used to maintain a list of integers. Using the following expression, one can, for example, initialise the list to contain the single integer 1:

INTEGER_LIST.empty() ; INTEGER_LIST.add(1)

The instantiation of the *PARAM_LIST* scheme with the *INTEGER* object can be done safely since the scheme requires an object defining a type *Elem* and since the object actually provides such a type. Section 30.5 will explain in more detail when instantiations of parameterized schemes are safe.

We have seen how a scheme can be instantiated with objects represented by identifiers. There are, however, other ways of representing objects as we shall see later. Therefore we introduce the category *object_expr* of object expressions. The generalized syntax for instantiation of parameterised schemes is therefore:

id(object_expr$_1$, ..., object_expr$_n$) (for n $\geq$ 1)

30.2 Naming of Parameter Requirements

The parameter requirement of the *PARAM_LIST* scheme is the class expression:

class type Elem **end**

Typically one defines such requirements as schemes and one then writes scheme names as parameter requirements. We can define a scheme in this style as follows.

scheme ELEMENT = **class type** Elem **end**

We can now define the parameterized scheme as follows.

scheme PARAM_LIST(E : ELEMENT) = **class** ... **end**

This style is somewhat more pleasant than writing a basic class expression (including keywords **class** and **end**) at the place where *ELEMENT* occurs.

30.3 Object Fittings

There are situations where the object with which we want to instantiate a parameterized scheme provides names different to the ones required by the parameter requirement. In this case the object must be subjected to a fitting at instantiation time.

Suppose for example that we wish to define a list of database commands. We first define a database command object that provides the type of commands. The database is assumed to associate natural numbers with texts.

```
object COMMAND :
  class
    type
      Key = Nat,
      Data = Text,
      Command ==
        mk_empty | mk_insert(Key, Data) | mk_remove(Key) | mk_lookup(Key)
    end
```

Before instantiating the *PARAM_LIST* scheme with this object, we notice that the type name *Command* is different from the required type name *Elem*. We further notice that the object defines more names than required. That is, besides *Command*, the following names are defined: *Key*, *Data*, *mk_empty*, *mk_insert*, *mk_remove* and *mk_lookup*.

The extra names are not a problem since the requirement just defines a minimal name space, allowing for extra names to be defined in the actual parameter object.

The difference between the actual name (*Command*) and the required name (*Elem*) is a problem. To resolve it, we must fit the *COMMAND* object at instantiation time to provide the name *Elem* instead of *Command*. This is done below.

```
object COMMAND_LIST : PARAM_LIST(COMMAND{Command for Elem})
```

The interesting part is the object expression:

```
COMMAND{Command for Elem}
```

This object expression represents an object which is equivalent to the *COMMAND* object, except that it provides the name *Elem* instead of the name *Command*: it 'fits' the object expression *COMMAND* to a context expecting the name *Elem*, not the name *Command*.

The general form of a fitting object expression is:

```
object_expr{renaming}
```

where the *renaming* has the form:

$\mathrm{id}_{\mathrm{actual}_1}$ **for** $\mathrm{id}_{\mathrm{formal}_1}$, ..., $\mathrm{id}_{\mathrm{actual}_n}$ **for** $\mathrm{id}_{\mathrm{formal}_n}$

for $n \geq 1$. The object represented by *object_expr* is fitted such that each id_{actual_i} is renamed into id_{formal_i}.

In the case where an overloaded identifier is to be fitted, a type expression must be given in order to disambiguate it as follows:

object_expr{..., $\mathrm{id}_{\mathrm{actual}}$: type_expr **for** $\mathrm{id}_{\mathrm{formal}}$, ...}

Since the fitted object expression itself represents an object, one can access its entities using the 'dot'-notation. One can for example access the type *Elem* as

follows:

COMMAND{Command **for** Elem}.Elem

Due to this property, we can expand the scheme instantiation into a basic class expression by replacing all occurrences of the formal parameter name E with the object expression *COMMAND*{*Command* **for** *Elem*}. The technique of replacing formal parameter names by actual parameters can be performed for all kinds of instantiations of parameterized schemes with objects. Section 30.1 illustrated the technique in the simplest case.

This expansion gives the following object (focusing on the variable definition).

object
  COMMAND_LIST :
    class
      variable list : COMMAND{Command **for** Elem}.Elem*
      ⋮
    end

The next observation is that the following equivalence holds:

COMMAND{Command **for** Elem}.Elem ≡ COMMAND.Command

That is, looking up the name *Command* in the object *COMMAND* is equivalent to looking up the name *Elem* in the fitted object. In general:

id{...,id_a **for** id_f,...}.id_f ≡ id.id_a

Utilizing this equivalence, we can finally expand our object definition into the following, fully expanded definition.

object
  COMMAND_LIST :
    class
      variable
        list : COMMAND.Command*
      value
        empty : **Unit** → **write** list **Unit**,
        is_empty : **Unit** → **read** list **Bool**,
        add : COMMAND.Command → **write** list **Unit**,
        head : **Unit** $\xrightarrow{\sim}$ **read** list COMMAND.Command,
        tail : **Unit** $\xrightarrow{\sim}$ **write** list **Unit**
      axiom forall e : COMMAND.Command •
        empty() ≡ list := ⟨⟩,
        is_empty() ≡ list = ⟨⟩,
        add(e) ≡ list := ⟨e⟩ ⌢ list,
        head() ≡ **hd** list
        pre ∼is_empty(),
        tail() ≡ list := **tl** list

```
      pre ~is_empty()
  end
```

The *COMMAND_LIST* object can now be used to maintain a list of commands. One can for example modify the list by the following expression:

```
COMMAND_LIST.empty();
COMMAND_LIST.add(COMMAND.mk_empty);
COMMAND_LIST.add(COMMAND.mk_insert(1,"one"))
```

30.4 More Complex Parameter Requirements

The parameter requirement *ELEMENT* of the *PARAM_LIST* scheme is simple in the sense that it only requires a single type. A parameter requirement may be more complex; it may, for instance, require some functions to be defined over that type. In fact, since the requirement is a class expression, any kind of entity can be required and axioms can be given that express required properties.

As an example, suppose we want to extend *PARAM_LIST* with an operation, *is_ordered*, for testing whether the current contents of the list variable is an ordered list. That is, whether any list element is 'less than or equal to' the successor element in the list.

In order to specify *is_ordered*, the ordering on list elements must be known. That is, it must be known what 'less than or equal to' means. The following entities are therefore required as parameters:

```
type Elem
value leq : Elem × Elem → Bool
```

The function *leq* is the element ordering function. We would like to require that for any e_1, e_2 : *Elem*, *leq*(e_1, e_2) is **true** if and only if e_1 'is less than or equal to' e_2. This requirement must in RSL be stated more formally, and this is what we do below.

First, it turns out that in order to specify the properties of *leq*, we must know what equivalence means on *Elem*. That is, we must know when two elements are equivalent. The following scheme extends the *ELEMENT* scheme with an equivalence function, *eq*, and defines its required properties.

```
scheme
  EQUIVALENCE =
    extend ELEMENT with
    class
      value
        eq : Elem × Elem → Bool
      axiom forall e,e1,e2,e3 : Elem •
        [eq_reflexive]
          eq(e,e),
        [eq_transitive]
```

```
      eq(e1,e2) ∧ eq(e2,e3) ⇒ eq(e1,e3),
    [eq_symmetric]
      eq(e1,e2) ⇒ eq(e2,e1)
  end
```

The three axioms define three fundamental properties of equivalence. Try to substitute = for *eq* and see that = satisfies the axioms. Equality is thus an example of an equivalence in the above sense.

The next step is to extend *EQUIVALENCE* with an ordering function, *leq*, and define its required properties.

```
scheme
  PARTIAL_ORDER =
    extend EQUIVALENCE with
    class
      value
        leq : Elem × Elem → Bool
      axiom forall e1,e2,e3 : Elem •
        [leq_reflexive]
          eq(e1,e2) ⇒ leq(e1,e2),
        [leq_transitive]
          leq(e1,e2) ∧ leq(e2,e3) ⇒ leq(e1,e3),
        [leq_antisymmetric]
          leq(e1,e2) ∧ leq(e2,e1) ⇒ eq(e1,e2)
  end
```

The axioms define three fundamental properties of a 'partial ordering'; partial in the sense that two elements do not need to be ordered with respect to each other. Try to substitute equality on texts for *eq* and 'is-a-prefix-of' on text pairs for *leq*, and see that 'is-a-prefix-of' then satisfies the axioms, therefore being a partial ordering.

The *PARTIAL_ORDER* scheme constitutes the complete parameter requirement. The parameterized list scheme can now be written as follows.

```
scheme
  PARAM_ORDERED_LIST(T : PARTIAL_ORDER) =
    extend PARAM_LIST(T) with
    class
      value
        is_ordered : Unit → read list Bool
      axiom
        is_ordered() ≡
          (∀ idx1,idx2 : Nat •
            {idx1,idx2} ⊆ inds list ∧ idx1 < idx2 ⇒
              T.leq(list(idx1),list(idx2)))
  end
```

PARAM_ORDERED_LIST(*T* : *PARTIAL_ORDER*) is, interestingly enough, defined as an extension of *PARAM_LIST*(*T*). The above scheme definition is equivalent to the following, where the instantiation *PARAM_LIST*(*T*) has been expanded, followed by an expansion of the **extend**.

```
scheme
  PARAM_ORDERED_LIST(T : PARTIAL_ORDER) =
    class
      variable
        list : T.Elem*
      value
        empty : Unit → write list Unit,
        is_empty : Unit → read list Bool,
        add : T.Elem → write list Unit,
        head : Unit ~→ read list T.Elem,
        tail : Unit ~→ write list Unit
      axiom forall e : T.Elem •
        empty() ≡ list := ⟨⟩,
        is_empty() ≡ list = ⟨⟩,
        add(e) ≡ list := ⟨e⟩ ^ list,
        head() ≡ hd list
        pre ~is_empty(),
        tail() ≡ list := tl list
        pre ~is_empty()
      value
        is_ordered : Unit → read list Bool
      axiom
        is_ordered() ≡
          (∀ idx1,idx2 : Nat •
            {idx1,idx2} ⊆ inds list ∧ idx1 < idx2 ⇒
              T.leq(list(idx1),list(idx2)))
    end
```

A list of texts can be defined in the following way. First, a text object is defined that provides all the entities *Elem*, *eq* and *leq*.

```
object
  TEXT :
    class
      type
        Elem = Text
      value
        eq : Text × Text → Bool,
        leq : Text × Text → Bool
      axiom forall t1,t2 : Text •
        eq(t1,t2) ≡ t1 = t2,
```

leq(t$_1$,t$_2$) ≡ (∃ t : **Text** • t$_1$ ⌢ t = t$_2$)
end

The *eq* function is defined to be just text equality. This function obviously satisfies the required *eq* axioms in *EQUIVALENCE*.

The *leq* function is defined such that $leq(t_1, t_2)$ is **true** if and only if t_1 is a prefix of t_2. This function satisfies all the required *leq* axioms in *PARTIAL_ORDER*.

An object defining a list of texts can be defined as follows.

object TEXT_LIST : PARAM_ORDERED_LIST(TEXT)

The *TEXT_LIST* object can now be used to maintain a list of texts. As an example, consider the following expression that inserts four texts into an empty list and then evaluates to **true** since the resulting list is ordered:

```
TEXT_LIST.empty();
TEXT_LIST.add("formalism");
TEXT_LIST.add("formal");
TEXT_LIST.add("form");
TEXT_LIST.add("for");
TEXT_LIST.is_ordered()
```

30.5 Actual versus Formal Parameters

There is a relation that must hold between an actual and a formal parameter for a scheme application to be well formed. Note that a formal scheme parameter takes the form *identifier* : *class_expression* and an actual parameter is an object, which has an associated class expression. The relation is that the class expression of the actual parameter must 'statically implement' the class expression of the formal parameter. (The term statically implement is chosen because static implementation is the statically decidable part of the RSL implementation relation.)

To explain the static implementation relation we first generalize the notion of 'signature' to the signature of a class. We have already used the term for the association of a name and its type in a value definition. We can similarly have a signature for a variable or channel — they have names and types. For type definitions we can say the signature of a sort definition (which is just a new type name) is merely the name of the new type. For a type abbreviation the signature associates the name of the new type with the type it is an abbreviation for. Since variant type definitions can be expanded to sort definitions plus some value and axiom declarations, and union and short record definitions can be expanded to variant definitions, we can give signatures to all type definitions. Axiom definitions have no signatures.

For schemes and objects we create a signature by associating the name of the scheme or object with, instead of a type, the class represented by its class expression, together with the classes of parameters for parameterized schemes, and the type of the index for object arrays (see chapter 32).

Having defined a notion of signature for all kinds of definition we can collect these separate signatures (tagged with the kind of definition they come from) and generate a signature for any basic class expression. If we also include in the notion of signature a list of those entities that are hidden we can thus generate a signature for any class expression.

There is also a notion of 'maximal class' that corresponds directly to the notion of 'maximal type'. One can obtain the maximal class of a class expression basically by ignoring all axioms and taking the maximal versions of all types and classes mentioned, just as one can obtain the maximal type of a type expression by ignoring the predicates that define subtypes and taking the maximal versions of any types mentioned. So we have a notion of 'maximal signature' involving only maximal classes and types.

The static implementation relation is now simple to formulate: $class\_expr_2$ statically implements $class\_expr_1$ if the maximal signature of $class\_expr_1$ is included in the maximal signature of $class\_expr_2$. That is $class\_expr_2$ must provide (at least) all the schemes, objects, types, variables, channels and values that $class\_expr_1$ does, with the same maximal classes or types.

If a fitting is applied to an actual scheme parameter, the condition is that the class expression of the object to which the fitting is applied must statically implement the class expression of the formal parameter when the fitting is applied to it as a renaming. For example, we can use:

COMMAND{Command **for** Elem}

as an actual parameter when the formal was *E : ELEMENT* if the class expression of the object *COMMAND* statically implements:

use Command **for** Elem **in** ELEMENT

This last class expression expands to:

class type Command **end**

and this is indeed statically implemented by the class expression defining *COMMAND*.

If there are several formal parameters then there must be the same number of actuals and formals, the first actual statically implementing the first formal and so on.

30.6 The Implementation Relation

We have described in this chapter a notion of 'static implementation'. There is a corresponding notion of (full) implementation or 'refinement' for which static implementation can be seen as just a part. In fact, $class\_expr_2$ implements or refines $class\_expr_1$ if both the following hold:

- $class\_expr_2$ statically implements $class\_expr_1$.
- The theory of $class\_expr_1$ is provable in $class\_expr_2$.

The 'theory' of a class expression is the collection of axioms that can be deduced from its definitions (including its axioms). To take a simple example, consider the following schemes:

S = **class type** T **value** x, y : T **axiom** x ≠ y **end**
S_1 = **class type** T = **Int value** x : T = 1, y : **Int** = 2 **end**
S_2 = **class value** x : **Int** = 1, y : **Int** = 2 **end**
S_3 = **class type** T = **Int value** x : T = 1, y : T = 1 **end**
S_4 = **class type** T, U **value** x : T, y : U **end**
S_5 = **class type** T = **Int**, U = **Int value** x : T = 1, y : U = 2 **end**

S_1 implements S; its signature includes that of S and it satisfies the theory of S that x is different from y.

S_2 does not statically implement S (it defines no type T) and therefore S_2 cannot implement S. S_2 is implemented by S_1.

S_3 statically implements S but does not implement it since it does not satisfy the theory of S that x is different from y.

S_4 does not statically implement S as its signature is different (x and y have different types) and therefore S_4 cannot implement S.

S_5 implements S; its signature includes that of S and it satisfies the theory of S that x is different from y. S_5 also implements S_1, S_2 and S_4.

It is obviously a good idea to check that an actual parameter fully implements the formal, just as it is a good idea to check that a negative integer is not used as an actual parameter to a function defined only for natural numbers, but in neither case is the specification technically ill-formed. Well-formedness is defined as abiding by the static conditions that can be checked by a type checker.

To characterize implementation in a little more detail we have to describe what the 'theory' of the various kinds of class expression and their various constituent declarations are. Loosely, the 'theory' of a class expression is all the information, expressed as axioms, we ignore in calculating its maximal signature.

Apart from hiding (which does not change the theory), all class expressions can be expanded into basic class expressions — essentially collections of declarations. So most of the definition reduces to the properties of declarations. We outline what is involved for each kind of declaration.

30.6.1 Refinement for schemes and objects

For scheme and object declarations the theory is the defining class expressions (plus any parameters for schemes, which are also expressed in terms of class expressions, and array parameters for object arrays). So checking that a new scheme or object definition implements an old one is a question of checking the relation between their class expressions. Similarly for scheme parameters, except that the relation for these is co-gradient — the class expression of the old parameter must implement the class expression of the new. For object arrays the set of new indices must include the old — we can make the array larger.

30.6.2 Refinement for variables and channels

The only theory for variables and channels is subtype information (if any) in their types and (for variables) any initialisation. Hence the checks are that the type of a new variable or channel must be a subtype of the type of the old variable or channel, and if an old variable had an initialisation then the new one must have an initialisation to an equivalent value.

It may seem strange to allow the reduction of the type of a variable or channel. What happens if we reduce its type while assigning or communicating values now outside that type? The answer is that we get a contradiction, which produces the empty set of models. This is a refinement, according to our definition, but a singularly useless one. So the freedom to reduce the types of variables or channels is rarely used.

30.6.3 Refinement for types

Since variant type definitions expand into sort definitions (just new type names), plus some value and axiom declarations, and union and short record definitions expand into variant definitions, there are really only two kinds of type declaration — sort and abbreviation definitions. For the first there is no theory — it just introduces a new type name. For the second the theory is that the new type name is equivalent to the type expression it is an abbreviation for. Hence the new type declaration must be an abbreviation definition for the same type.

Note that this refinement rule for types is more restrictive than that found in, for example, VDM. In RSL if we write:

type Set = Elem-**set**

we cannot later refine it to:

type Set = Elem*

even though there is a well known retrieve function from lists to sets. The reason for the restriction is that we want to be able to substitute refinements for the originals — this is one of the main purposes of doing refinement, and we want to be able to refine parts of a system separately. Clearly we cannot just replace the first of these with the second — almost any expression involving values of the type *Set* will no longer even type check!

So instead of the first we either have to use an abstract definition for *Set*:

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

or else we can use a 'representation function':

type Set
value rep : Set → Elem-**set**

rep is initially only given a signature (and is hidden). To get an implementation where sets are lists we could implement *rep* as **elems**. The techniques for data

refinement are then very similar to those for VDM.

30.6.4 Refinement for values

A value declaration can always be expanded into a value declaration involving only maximal types and one or more axioms. The theory is then precisely these axioms. For example, the declarations:

value
  x : **Int** = 2,
  y : **Nat** • y < 3

are equivalent to:

value
  x : **Int**,
  y : **Int**
axiom
  x = 2,
  y ≥ 0 ∧ y < 3

For functions there is a little more to do: total functions can only be refined by total functions and if there are accesses in the old then the new can only have a subset of those accesses. (Otherwise, for example, the property that a function cannot affect a variable could be lost.)

Note that as a result of the way that function definitions are expanded into axioms we obtain the refinement rules for functions that we might expect: domains may be increased, pre-conditions weakened, post-conditions strengthened.

30.6.5 Refinement for axioms

The axioms are precisely the theory. So we have to show that the old axioms hold in the new class expression.

Implementation is of most importance in development of specifications, and is described in full in [12]. Its formal definition in terms of proof rules is in [37].

CHAPTER 31

Module Nesting

RSL allows for nesting of modules. That is, schemes and objects can be defined within class expressions. As an example, suppose we want to define a single object, say *INTEGER_LIST*, providing a variable capable of containing integer lists. This can be done as follows by instantiating the parameterized *PARAM_LIST* scheme shown earlier.

```
object
  INTEGER_LIST :
    class
      object
        INTEGER : class type Elem = Int end,
        LIST : PARAM_LIST(INTEGER)
      ⋮
    end
```

The *INTEGER* object (which is necessary since schemes can be instantiated with objects only) is defined nested in the class expression defining *INTEGER_LIST*, as is the *LIST* object itself.

Within the class expression defining *INTEGER_LIST*, at the position of the three dots, one can refer to the *INTEGER* and *LIST* objects and their contents. At the position of the three dots, the expression *LIST.add* therefore represents the add operation on lists.

Outside the class expression, one has to prefix all references by *INTEGER_LIST*. The expression *INTEGER_LIST.LIST.add* thus represents the add operation on lists.

Note that with the possibility of nesting objects one has a choice between extension (with the **extend** operator) and nesting. That is, suppose we want to write a class expression that, among other things, provides all the entities from the scheme *LIST_S*. This can either be done by extending *LIST_S*:

```
  extend LIST_S with class ... end
```

or by defining a nested instance of *LIST_S*:

class object L : LIST_S ... **end**

The local instance implies that the entities of *LIST_S* are all prefixed with the object name *L*. This may be preferred when *LIST_S* is clearly a subconcept of the new class expression.

Alternatively, one may want to avoid the prefixing by using the **extend** operator instead. This may be the case when *LIST_S* is not a clear subconcept of the class expression, but rather a 'step on the way'.

Schemes can also be defined in a nested manner, although this does not occur very often in practice.

CHAPTER 32

Object Arrays

We have seen how one can define a model as a member of a class represented by a class expression. The model may be identified by an object identifier. If several models are wanted, each being a member of the same class, one may define just as many object identifiers.

In some situations it is useful to be allowed to define an arbitrary number of models of the same class. The concept of an object array gives exactly this possibility, each model being identified by an object identifier common to them all and some additional distinct index value.

In order to motivate the concept of an object array we study a small example, first expressed without object arrays.

32.1 Formulation without Object Arrays

Suppose we want to define a scheme that defines two lists plus a function for moving elements between the two lists. We can create the two lists by defining two objects which are models of the *LIST_S* scheme.

```
scheme
  TWO_LISTS =
    class
      object
        L1, L2 : LIST_S
      type
        List_No = {| n : Nat • 1 ≤ n ∧ n ≤ 2 |}
      value
        move : List_No × List_No ~→ write L1.list, L2.list Unit
      axiom forall m,n : List_No •
        move(m,n) ≡
          case (m,n) of
            (1,2) →
```

```
            if ~ L1.is_empty() then
              let head_1 = L1.head() in
                L1.tail() ; L2.add(head_1)
              end
            end,
          (2,1) →
            if ~ L2.is_empty() then
              let head_2 = L2.head() in
                L2.tail() ; L1.add(head_2)
              end
            end
        end
      pre m ≠ n
end
```

The scheme defines two local instances, L_1 and L_2, of the *LIST_S* scheme. The type *List_No* contains the values 1 and 2 corresponding to the two objects.

The operation *move* removes the head of one list and inserts it as the head of the other list. The source and target of the move are parameters to the operation.

The observations to make here are as follows:

1. The axiom defining *move* includes a tedious case expression where the case branches differ only with respect to some indexing.
2. If there had been more than two instances, it would have been even more tedious to write down all the object definitions and case branches.
3. There is no possibility of letting the number of instances depend on some parameter.

32.2 Formulation with Object Arrays

With object arrays, the above three problems can be solved. Assume the type definition from the example above:

type List_No = {| n : **Nat** • 1 ≤ n ∧ n ≤ 2 |}

An object array corresponding to the above situation may then be defined as follows.

object L[i : List_No] : LIST_S

This definition defines two models of the *LIST_S* scheme, identified by $L[1]$ and $L[2]$, respectively. The two models deal with two different variables $L[1].list$ and $L[2].list$. The variable $L[i].list$, for i : *List_No*, is accessed through the operations $L[i].op$ where *op* is any operation defined in *LIST_S*.

The definition of an object array has the general form:

id[typing$_1$,...,typing$_n$] : class_expr

for $n \geq 1$. Note here that a list of typings can be expanded into a single typing of the form:

binding : type_expr

Any object definition can therefore be expanded into a definition of the form:

id[binding : type_expr] : class_expr

By this definition the identifier *id* is bound to an array of models. The index type of the array is the type represented by *type_expr*. Each index value belonging to the index type is mapped to a model. The model is an arbitrary one belonging to the class represented by the class expression — evaluated in scope of the definitions obtained by matching the index value against the *binding*. That is, the scope of the identifiers defined by the *binding* is *class_expr*.

The application of an object array to an index value within the index type has the form:

id[value_expr]

This is an object expression that represents a model of which the entities can be obtained by the dot-notation. In the syntax such an object expression is called an element object expression, indicating that it represents a particular element (model) of an object array. Note that two applications of an object array to the same index give the same model. That is, the choice of model for each index is made only once.

As a special case, an object application of the form:

id[value_expr$_1$,...,value_expr$_n$]

for $n \geq 2$ is short for:

id[(value_expr$_1$,...,value_expr$_n$)]

The whole specification can now be written as follows, using an object array.

```
scheme
  TWO_LISTS =
    class
      object
        L[i : List_No] : LIST_S
      type
        List_No = {| n : Nat • 1 ≤ n ∧ n ≤ 2 |}
      value
        move : List_No × List_No ~→ write {L[i].list | i : List_No} Unit
      axiom forall m,n : List_No •
        move(m,n) ≡
          if ~ L[m].is_empty() then
            let head_m = L[m].head() in
              L[m].tail() ; L[n].add(head_m)
            end
```

```
        end
      pre m ≠ n
  end
```

The axiom defining the *move* operation is straightforward. The type of the *move* operation includes the **write** access:

{L[i].list | i : List_No}

That is, the operation has **write** access to all variables $L[i].list$ where i is a member of the type *List_No*. Stated another way, the operation has **write** access to the variables $L[1].list$ and $L[2].list$.

32.3 Making the Size a Parameter

The size of an object array may depend on a parameter. As an example, we can parameterize our scheme with the size of the object array. For that purpose, a parameter requirement must be defined, for example as follows.

scheme SIZE = **class value** size : **Nat axiom** size ≥ 2 **end**

The scheme may now be parameterized with the size.

```
scheme
  MANY_LISTS(S : SIZE) =
    class
      object
        L[i : List_No] : LIST_S
      type
        List_No = {| n : Nat • 1 ≤ n ∧ n ≤ S.size |}
      value
        move : List_No × List_No ~→ write {L[i].list | i : List_No} Unit
      axiom forall m,n : List_No •
        move(m,n) ≡
          if ~ L[m].is_empty() then
            let head_m = L[m].head() in
              L[m].tail() ; L[n].add(head_m)
            end
          end
        pre m ≠ n
    end
```

Note that the only change made in the defining class expression is in the definition of the type *List_No* where the upper limit is represented by $S.size$.

It is worth considering for this example what the result would be if we had not included the condition in the parameter *SIZE* that *size* is at least two, thus allowing it to be one or zero.

If *size* is one then *MANY_LISTS* is consistent but of little use. The type *List_no* contains only one value, there is only one object in the array L and the axiom for *move* can never be applied since the pre-condition can never be satisfied.

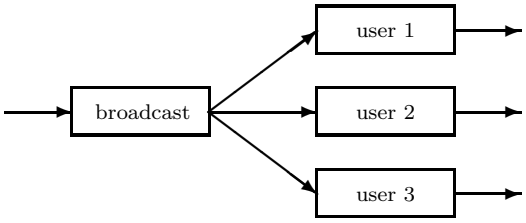
If *size* is zero then the type *List_no* is empty, as is the array L, and the axiom for *move* can never be applied since no arguments can be found for it.

So in this case weakening the parameter requirement did no harm (provided we are happy that the specifications in the cases of 'one list' and of 'no lists' satisfy our interpretation of the requirements). In general, however, one needs to be careful about allowing such degenerate cases.

32.4 Object Arrays as Scheme Parameters

The parameter of a scheme may be an object array. Below we go through an example illustrating this kind of scheme parameterization. Object arrays as scheme parameters typically occur in specification of concurrent systems and therefore our example is of this kind.

Our aim is to specify a general broadcasting process. A broadcasting process is a process that inputs values from a single input channel, and then outputs the values to several output channels, each ending in some user process. The system can be pictured as follows, assuming the number of users to be three:



The first thing we specify is the *broadcast* process. We shall eventually also specify the rest of the system in order to show how things are put together.

The *broadcast* process, or rather the scheme defining it, is chosen to be parameterized with the kind of data transmitted on the channels. We therefore need a parameter requirement of the following form.

scheme DATA = **class type** Data **end**

In addition, the *broadcast* process is chosen to be parameterized with the channel it inputs from and the channels it outputs to. We therefore need a parameter requirement of the form below.

scheme CHANNEL(D : DATA) = **class channel** c : D.Data **end**

This requirement is parameterized with respect to the type of the channel. A specific requirement is obtained by instantiating the parameterized one with an actual channel type as we shall see below.

The number of output channels associated with the broadcast process is also

made a parameter. This is done by parameterizing with an array of output channels where the array index type is an additional parameter satisfying the following requirement.

scheme INDEX = **class type** Index **end**

The specification of the *broadcast* process can now be written as follows.

```
scheme
  BROADCAST(
      I : INDEX,
      D : DATA,
      IN : CHANNEL(D),
      OUT[i : I.Index] : CHANNEL(D)) =
    class
      value
        broadcast : Unit → in IN.c out {OUT[i].c | i : I.Index} Unit
      axiom
        broadcast() ≡
          while true do
            let data = IN.c? in ‖{OUT[i].c!data | i : I.Index} end
          end
    end
```

The parameters of the *BROADCAST* scheme can be described as follows.

The type *I.Index* contains all the indices of the output-channel array *OUT*. The type *D.Data* is the type of the data transmitted on all channels. The channel *IN.c* is the channel that the *broadcast* process inputs from. Each of the channels $OUT[i].c$, where $i : I.Index$, is an output channel of the *broadcast* process.

Note the dependence between the parameters: I and D are referred to in the definition of *IN* and *OUT*. This dependence expresses a required sharing between parameters: the I referred to in the definition of *OUT* is exactly the I given as the first scheme parameter. Likewise for D. This ensures, for example, that the types of input and output data are the same, as there is only one object D and hence only one type *D.Data*.

The general form of a parameterized scheme definition can now be described to be:

scheme id(object_definition$_1$,...,object_definition$_n$) = class_expr

for $n \geq 1$. Recall that an object definition either has the form:

id : class_expr

or the form:

id[typing$_1$,...,typing$_n$] : class_expr

for $n \geq 1$. Let us now examine the definition of the *broadcast* process, starting with its type. The process inputs from the *IN.c* channel and outputs to any of the

$OUT[i].c$ channels where $i : I.Index$. The **out** access is described by the access description:

out {OUT[i].c | i : I.Index}

The axiom expresses that the process repeats the following forever: input a value from the $IN.c$ channel and then output the value on each of the $OUT[i].c$ channels where $i : I.Index$.

The output is expressed by the comprehended expression:

‖{OUT[i].c!data | i : I.Index}

This expression represents the parallel composition of all the processes:

OUT[i].c!data

where $i : I.Index$. Suppose for example that $I.Index$ contains the values i_1, i_2 and i_3, then the above expression is equivalent to the following:

$\text{OUT}[\text{i}_1]$.c!data ‖ $\text{OUT}[\text{i}_2]$.c!data ‖ $\text{OUT}[\text{i}_3]$.c!data

Since the outputs are put in parallel, they will all be carried out at some point in time, and the result is broadcasting.

The expression $value\_expr_2$ is a restriction of type **Bool**. As an example we could have made the broadcasting more selective by restricting the receiving *user* processes to those satisfying some predicate $p : I.Index \to \textbf{Bool}$:

‖{OUT[i].c!data | i : I.Index • p(i)}

We can now illustrate how the *broadcasting* process can be put in parallel with a number of user processes. First, we need to decide on the number of users, and thereby the number of output channels to these. That is, the index type must be determined. Let it be the numbers from 1 to 3.

object ACTUAL_INDEX :
  class type Index = {| i : **Nat** • 1 ≤ i ∧ i ≤ 3 |} **end**

The data transmitted on channels are chosen to be of type **Text**.

object ACTUAL_DATA : **class type** Data = **Text end**

The input channel to the *broadcast* operation is defined as follows.

object ACTUAL_IN : CHANNEL(ACTUAL_DATA)

Each user process inputs from a channel and outputs to another.

scheme USER(D : DATA, IN : CHANNEL(D), OUT : CHANNEL(D)) =
  class value user : **Unit** → **in** IN.c **out** OUT.c **Unit end**

The collection of channels output to by the user processes is represented as an array of channels.

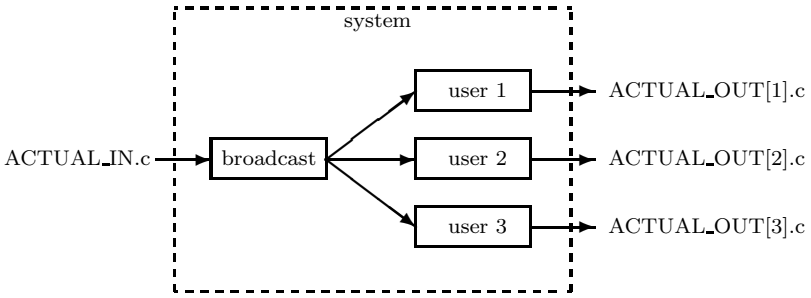
object
  ACTUAL_OUT[i : ACTUAL_INDEX.Index] : CHANNEL(ACTUAL_DATA)

The final system can be specified as follows.

```
object SYSTEM :
  class
    value
      system : Unit → in ACTUAL_IN.c
                      out {ACTUAL_OUT[i].c | i : ACTUAL_INDEX.Index}
                      Unit
    axiom
      system() ≡
        local
          object
            ACTUAL_MID[i : ACTUAL_INDEX.Index] :
              CHANNEL(ACTUAL_DATA),
            ACTUAL_BROADCAST :
              BROADCAST(ACTUAL_INDEX, ACTUAL_DATA,
                        ACTUAL_IN, ACTUAL_MID),
            ACTUAL_USER[i : ACTUAL_INDEX.Index] :
              USER(ACTUAL_DATA, ACTUAL_MID[i], ACTUAL_OUT[i])
        in
          ACTUAL_BROADCAST.broadcast()
          ||
          ||{ACTUAL_USER[i].user() | i : ACTUAL_INDEX.Index}
        end
  end
```

The *system* process is defined in terms of a local expression where all the internal channels, represented by the channel array *ACTUAL_MID*, are local, and thereby hidden. The internal channels are those connecting the *broadcast* process with the *user* processes.

The internals of the *system* process are treated as a 'black box' as indicated by the following figure:



The hiding of internal channels is also reflected in the type of the *system* process which does not mention the hidden channels.

The objects *ACTUAL_BROADCAST* and *ACTUAL_USER* must be defined also in the local expression since they interact through the channels in the locally defined

ACTUAL_MID object.

Note the definition of the object *ACTUAL_USER*:

```
ACTUAL_USER[i : ACTUAL_INDEX.Index] :
  USER(ACTUAL_DATA, ACTUAL_MID[i], ACTUAL_OUT[i])
```

The index identifier i bound to the left of ':' is referred to in the class expression occurring to the right of ':'. Each *ACTUAL_USER*[i] is defined to be an instance of:

```
USER(ACTUAL_DATA,ACTUAL_MID[i],ACTUAL_OUT[i])
```

Note finally that the instantiations of the schemes *BROADCAST* and *USER* can be expanded into basic class expressions by replacing formal parameter names with actual parameters, just as we saw in section 30.1.

32.5 Object Array Fittings

There may be situations where the object array with which we want to instantiate a parameterized scheme provides different names than the ones required by the parameter requirement. In this case the object array must be subjected to a fitting at instantiation time.

As an example, consider the broadcast example from the previous section, and in particular the definition of the *system* process. The parameter requirement of *BROADCAST*, formulated in terms of *CHANNEL*, requires each output channel to be named c. Suppose another name is used instead as actual parameter. Assume, therefore, the following scheme.

```
scheme OTHER_CHANNEL(D : DATA) =
  class channel other_c : D.Data end
```

The definitions local to the *system* process may now be written as follows.

```
object
  ACTUAL_MID[i : ACTUAL_INDEX.Index] :
    OTHER_CHANNEL(ACTUAL_DATA),
  ACTUAL_BROADCAST :
    BROADCAST(ACTUAL_INDEX, ACTUAL_DATA,
                ACTUAL_IN, ACTUAL_MID{other_c for c}),
  ACTUAL_USER[i : ACTUAL_INDEX.Index] :
    USER(ACTUAL_DATA, ACTUAL_MID[i]{other_c for c},
         ACTUAL_OUT[i])
```

The channels connecting the *broadcast* process and the *user* processes are now named *other_c* instead of c. The matching between actual names and required names are performed by the two object expressions:

```
ACTUAL_MID{other_c for c}
```

as parameter to *BROADCAST* and:

ACTUAL_MID[i]{other_c **for** c}

as parameter to *USER*.

The first one fits each model in the object array. The second one fits model i in the object array.

In order to illustrate what the exact meanings of the above definitions are, we can expand the scheme instantiations into basic class expressions by simply replacing formal parameter names with actual parameters. Note that replacement of formal parameter names by actual parameters is always possible for instantiations of parameterized schemes; see also sections 30.1 and 30.3.

In connection with expanding the instantiation of *BROADCAST*, we must do the following replacements:

I → ACTUAL_INDEX
D → ACTUAL_DATA
IN → ACTUAL_IN
OUT → ACTUAL_MID{other_c **for** c}

In connection with expanding the instantiation of *USER*, we must do the following replacements:

D → ACTUAL_DATA
IN → ACTUAL_MID[i]{other_c **for** c}
OUT → ACTUAL_OUT[i]

The expanded object definitions now become as follows.

```
object
  ACTUAL_MID[i : ACTUAL_INDEX.Index] :
    class
      channel other_c : ACTUAL_DATA.Data
    end,
  ACTUAL_BROADCAST :
    class
      value
        broadcast : Unit →
          in ACTUAL_IN.c
          out {ACTUAL_MID{other_c for c}[i].c |
                 i : ACTUAL_INDEX.Index} Unit
      axiom
        broadcast() ≡
          while true do
            let data = ACTUAL_IN.c? in
              ||{ACTUAL_MID{other_c for c}[i].c!data |
                   i : ACTUAL_INDEX.Index}
            end
          end
    end,
```

```
  ACTUAL_USER[i : ACTUAL_INDEX.Index] :
    class
      value
        user : Unit →
          in ACTUAL_MID[i]{other_c for c}.c
          out ACTUAL_OUT[i].c Unit
    end
```

The class expression defining *ACTUAL_BROADCAST* contains a name of the form:

```
ACTUAL_MID{other_c for c}[i].c
```

This is a perfectly valid name in RSL. It represents the channel obtained by first fitting the *ACTUAL_MID* object array, then indexing with the index value i, and then looking up c in the resulting model.

The class expression defining *ACTUAL_USER* contains a name of the form:

```
ACTUAL_MID[i]{other_c for c}.c
```

Assuming the same i, the two names actually represents the same channel. This follows from the fact that:

```
ACTUAL_MID{other_c for c}[i] = ACTUAL_MID[i]{other_c for c}
```

That is, one can just swap the indexing and the fitting. Note further that the following holds:

```
ACTUAL_MID[i]{other_c for c}.c ≡ ACTUAL_MID[i].other_c
```

Based on these observations we can finally expand the object definitions into the following.

```
object
  ACTUAL_MID[i : ACTUAL_INDEX.Index] :
    class
      channel other_c : ACTUAL_DATA.Data
    end,
  ACTUAL_BROADCAST :
    class
      value
        broadcast : Unit →
          in ACTUAL_IN.c
          out {ACTUAL_MID[i].other_c | i : ACTUAL_INDEX.Index} Unit
      axiom
        broadcast() ≡
          while true do
            let data = ACTUAL_IN.c? in
              ||{ACTUAL_MID[i].other_c!data | i : ACTUAL_INDEX.Index}
            end
```

```
        end
  end,
ACTUAL_USER[i : ACTUAL_INDEX.Index] :
  class
    value
      user : Unit →
        in ACTUAL_MID[i].other_c
        out ACTUAL_OUT[i].c Unit
  end
```

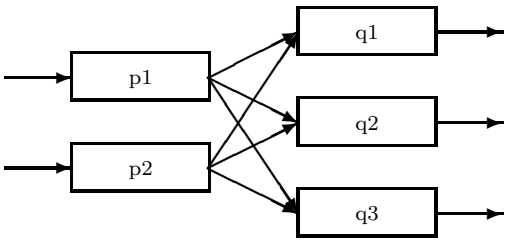
It should now be clear that the *broadcast* process communicates with the *user* processes over the channels $ACTUAL\_MID[i].other\_c$ where i is of type $ACTUAL\_INDEX.Index$.

32.6 Anonymous Object Arrays

So far it has been described how one can define an object array in an object definition. One has to give a name to the object.

RSL provides means of writing an object array without giving it a name. This is appropriate in certain contexts involving scheme instantiation — as is illustrated below.

The example to be used is that of a cross-bar switch. A cross-bar switch can be illustrated by the following figure:



A number of processes, here $p1$ and $p2$, are connected with a number of other processes, here $q1$, $q2$ and $q3$, in the following way. Each p process inputs from one channel and outputs to all the q processes. That is, each p process is connected with every q process. The q processes in turn output on some channels.

The channels of the system can thus be divided into three groups, as follows:

1. The input channels going into the p processes.
2. The connection channels connecting p and q processes.
3. The output channels going out from the q processes.

The specification is parameterized with respect to the number of p processes, say m, and the number of q processes, say n. As we shall see, we need anonymous object arrays to specify the connection between a single p process and all the q processes. Note that there are $m \times n$ connection channels.

First, we specify a scheme for the p processes. Note that we assume the schemes *INDEX*, *DATA* and *CHANNEL* from section 32.4.

```
scheme
  P(N : INDEX, D : DATA, IN : CHANNEL(D),
    OUT[n : N.Index] : CHANNEL(D)) =
    class
      value
        p : Unit → in IN.c out {OUT[n].c | n : N.Index} Unit
    end
```

A p process inputs from a channel and outputs to as many channels as there are q processes, as indicated by the number of elements in the type $N.Index$.

The scheme for q processes becomes very similar.

```
scheme
  Q(M : INDEX, D : DATA, IN[m : M.Index] : CHANNEL(D),
    OUT : CHANNEL(D)) =
    class
      value
        q : Unit → in {IN[m].c | m : M.Index} out OUT.c Unit
    end
```

A q process inputs from as many channels as there are p processes, as indicated by the number of elements in the type $M.Index$.

Let us now create a specific system with two p processes and with three q processes. The actual data transmitted on channels is text.

```
object ACTUAL_DATA : class type Data = Text end
```

The number of p and q processes are represented by the objects M and N respectively.

```
object
  M : class type Index = {| i : Nat • 1 ≤ i ∧ i ≤ 2 |} end,
  N : class type Index = {| i : Nat • 1 ≤ i ∧ i ≤ 3 |} end
```

The input channels going into the p processes and the output channels going out from the q processes are represented by the object arrays *ACTUAL_IN* and *ACTUAL_OUT* respectively.

```
object
  ACTUAL_IN[m : M.Index] : CHANNEL(ACTUAL_DATA),
  ACTUAL_OUT[n : N.Index] : CHANNEL(ACTUAL_DATA)
```

The channels connecting the p and q processes are defined as follows.

```
object
  CONNECTION[m : M.Index, n : N.Index] : CHANNEL(ACTUAL_DATA)
```

The channel:

CONNECTION[x,y].c

connects the p process identified by x with the q process identified by y.

We can now define the collection of actual p processes as follows.

object
ACTUAL_P[m : M.Index] :
P(N, ACTUAL_DATA, ACTUAL_IN[m],
[|n : N.Index • CONNECTION[m,n]|])

The process $ACTUAL\_P[m].p$ inputs from the channel $ACTUAL\_IN[m].c$. The output channels of the process are represented by the anonymous object array:

[|n : N.Index • CONNECTION[m,n]|]

The object array maps each n in $N.Index$ into the object $CONNECTION[m, n]$. Recall that the channel $CONNECTION[m, n].c$ connects the p process identified by m with the q process identified by n.

An anonymous object array, also called an array object expression, has the general form:

[|typing$_1$,...,typing$_n$ • object_expr|]

for $n \geq 1$. Note here that a list of typings can be expanded into a single typing of the form:

binding : type_expr

Any anonymous object array can therefore be expanded into:

[|binding : type_expr • object_expr|]

The index type of the object array is the type represented by *type_expr*. Each index value belonging to the index type is mapped to a model. The model is obtained by evaluating the *object_expr* in the scope of the definitions obtained by matching the index value against the *binding*. That is, the scope of the identifiers defined by the *binding* is *object_expr*.

The collection of actual q processes can defined in a similar way.

object
ACTUAL_Q[n : N.Index] :
Q(M, ACTUAL_DATA, [|m : M.Index • CONNECTION[m,n]|],
ACTUAL_OUT[n])

The p and q processes can now be put in parallel by the following expression:

||{ACTUAL_P[m].p() | m : M.Index}
||
||{ACTUAL_Q[n].q() | n : N.Index}

In order to illustrate the meaning of having anonymous object arrays as scheme parameters we expand the instantiation of the P scheme by replacing formal parameter names by actual parameters. That is, we must do the replacements:

N → N
D → ACTUAL_DATA
IN → ACTUAL_IN[m]
OUT → [|n : N.Index • CONNECTION[m,n]|]

What we obtain is the following.

```
object
  ACTUAL_P[m : M.Index] :
    class
      value
        p : Unit →
          in ACTUAL_IN[m].c
          out {[|n : N.Index • CONNECTION[m,n]|][n].c | n : N.Index} Unit
    end
```

The interesting part is the name:

[|n : N.Index • CONNECTION[m,n]|][n].c

This is a valid name, in RSL, and it represents the channel obtained as follows. Apply the object array:

[|n : N.Index • CONNECTION[m,n]|]

to the index n, and obtain the model:

CONNECTION[m,n]

Then look up c in this model. Following this reasoning we can finally expand the object definition into:

```
object
  ACTUAL_P[m : M.Index] :
    class
      value
        p : Unit →
          in ACTUAL_IN[m].c
          out {CONNECTION[m,n].c | n : N.Index} Unit
    end
```

Similarly the other object definition can be expanded into the following.

```
object
  ACTUAL_Q[n : N.Index] :
    class
      value
        q : Unit →
          in {CONNECTION[m,n].c | m : M.Index}
          out ACTUAL_OUT[n].c Unit
    end
```

CHAPTER 33

The Name Space

The following sections summarize the syntax for names, object expressions and access descriptions, the latter occurring in function types. These categories are related in that they all describe references to named entities: types, values, variables, channels, objects and schemes.

The categories of names and object expressions are defined recursively in terms of each other. The category of access descriptions is defined in terms of names.

33.1 Names

A name can be of one of four forms:

- An identifier representing some type, value, variable, channel, scheme or object:

  id

  Examples:

  head
  LIST

- A user-defined operator turned into a value that may be applied using ordinary function-application notation:

  (op)

  Examples:

  (+)
  (**hd**)

- The looking up of an identifier in a model represented by an object expression:

  object_expr.id

  Examples:

LIST.add
INTEGER_LIST.LIST.add
COMMAND{Command **for** Elem}.Elem
L[m].add
ACTUAL_MID{other_c **for** c}[i].c
ACTUAL_MID[i]{other_c **for** c}.c
[|n : N.Index • CONNECTION[m,n]|][n].c

- The looking up of a user-defined operator in a model represented by an object expression:

  object_expr.(op)

  Example:

  RATIONAL.(+)

33.2 Object Expressions

An object expression represents a model or an array of models. Remember that object expressions also occur as actual parameters to schemes. The description below outlines all the possible object expressions. Note that when an object expression occurs in front of a dot '.' in a name as indicated above, it must represent a model and not an array.

An object expression may have one of four forms:

- A name representing either a model or an array:

  name

  Examples:

  LIST
  INTEGER_LIST.LIST
  ACTUAL_OUT

- The application of an array represented by an object expression to a sequence of index values, the result being a model:

  object_expr[value_expr$_1$,...,value_expr$_n$] (for n $\geq$ 1)

  Example:

  ACTUAL_OUT[i]

- An anonymous array:

  [|typing$_1$,...,typing$_n$ • object_expr |] (for n $\geq$ 1)

  Example:

  [|n : N.Index • CONNECTION[m,n]|]

- The fitting of a model or an array, represented by an object expression:

object_expr{renaming}

Examples:

COMMAND{Command **for** Elem}
ACTUAL_MID{other_c **for** c}
ACTUAL_MID[i]{other_c **for** c}

33.3 Access Descriptions

Remember that function type expressions may include access descriptions. That is, a function type expression with accesses has the general form (in case of total functions):

$\text{type\_expr}_1 \rightarrow \text{access\_desc}_1 \ldots \text{access\_desc}_n \ \text{type\_expr}_2$

for $n \geq 1$. Each access description *access_desc*$_i$ has the form:

$\text{access\_mode access}_1,\ldots,\text{access}_n$

for $n \geq 1$, where an *access_mode* is one of the following:

read
write
in
out

An *access* is of one of five forms:

- The reference to a variable or a channel represented by a name:

  name

  Examples:

  list
  LIST.list
  OUT[i].c

- The reference to any variable or channel defined within the narrowest enclosing class expression:

  any

  It is worth noting that the scope of an unqualified **any** may extend beyond the immediate class expression due to its later extension. For example, consider the following class expression.

  extend
  class value f : **Unit** $\rightarrow$ **write any Unit end**
  with
  class variable v : **Int end**

  Then the the variable v is included in the **write** access of function f. Hence the extent of an unqualified **any** is really only fixed when its class expression

is used to form an object.

This may seem surprising (though it is rarely of concern). There is a proof theoretic reason for it — that extended class expressions should be able to be expanded. The one above is equivalent to:

class
  value f : **Unit** → **write any Unit**
  variable v : **Int**
end

There is also a methodological reason. Part of the idea behind **any** is that it is objects that give structure to the total state (collections of variables) and channels in a system. Hence it is reasonable that the extent of an occurrence of **any** should be fixed when its class expression is used to make an object.

- The reference to any variable or channel defined within a model represented by an object expression:

  object_expr.**any**

  Example:

  ABSTRACT_LIST.**any**

- The reference to a collection of variables or channels through a comprehension:

  {access | typing$_1$,...,typing$_n$ • value_expr} (for n ≥ 1)

  Examples:

  {OUT[i].c | i : I.Index}
  {OUT[i].c | i : I.Index : p(i)}

- The reference to a collection of variables or channels through an enumeration:

  {access$_1$,...,access$_n$}

  where $n \geq 0$.

  The braces { and } have no particular meaning. They just provide a convenient notation for grouping accesses into 'sets' in special cases. The following two type expressions represent exactly the same type, for any types T_1 and T_2 and for any defined variables x and y:

  T_1 → **write** x,y T_2
  T_1 → **write** {x,y} T_2

  Note that an enumerated access may have the form {} representing the 'empty set of variables or channels'. Any function type can thus be represented by a type expression containing **read**, **write**, **in** and **out** access descriptions. As a special case, the following two type expressions represent the same type:

  T_1 → T_2
  T_1 → **read** {} **write** {} **in** {} **out** {} T_2

Part II

RSL Reference Description

CHAPTER 34

Reference Introduction

Sections 34.3–34.4 outline the syntactic structure and documentation conventions of part II, the RSL Reference Description, while sections 34.5–34.6 outline basic static and dynamic semantic concepts.

34.1 Purpose

The purpose of part II is to describe the RAISE Specification Language, RSL. The description is intended for looking up information rather than for sequential reading. It is reference rather than tutorial material.

34.2 Target Group

The target group of part II is users of RSL. But note that this is reference material; some familiarity with part I of this manual will be necessary before part II is useful.

34.3 Structure of Part II Chapters

Part II is formally structured over the syntax of RSL (see below). The introduction is followed by special chapters on declarative constructs, scope, visibility and overloading. After those chapters follow chapters on each of the main syntax categories of RSL:

- Specifications
- Declarations
- Class expressions
- Object expressions
- Type expressions
- Value expressions
- Bindings
- Typings

- Patterns
- Names
- Identifiers and operators
- Infix combinators
- Connectives

34.4 Documentation Conventions

The language description is centred around the syntax of RSL. 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. The rules of the syntax are grouped into chapters in this part (II). Each chapter consists of some or all of the following sections:

Syntax
Terminology
Context-independent Expansions
Scope and Visibility Rules
Context Conditions
Context-dependent Expansions
Attributes
Meaning

The contents of these sections are described below and the conventions used are explained.

Syntax Contains one or more syntax rules each of the form:

category_name ::=
  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 bold font such as '**Bool**'
- A symbol such as '('.
- A subcategory name such as 'value_expr', possibly prefixed with a text such as '*logical-*' in italics.

The strings generated by an alternative are those obtained by concatenating keywords, symbols and strings from subcategories — in the order of appearance. As examples consider:

```
finite_set_type_expr ::=
  type_expr-set

map_type_expr ::=
  type_expr m→ type_expr
```

The convention below is used for defining optional presence ('nil-x' represents absence of 'x'): For any syntax category name 'x' the following rule is assumed:

```
opt-x ::=
  nil-x|
  x
```

The conventions below 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|
  x , x-list2

x-choice ::=
  x|
  x | x-choice

x-choice2 ::=
  x | x|
  x | x-choice2
```

Note that '|' is an RSL symbol in 'x | x' and 'x | x-choice2'.

```
x-product2 ::=
  x × x|
  x × x-product2

x-product ::=
  x|
```

x × x-product

If 'opt' occurs together with 'string' or 'list', 'opt' has the lower precedence. That is, for any syntax category name 'x' the following rules are assumed:

opt-x-string ::=
  nil-x-string|
  x-string

opt-x-list ::=
  nil-x-list|
  x-list

Similarly, if 'nil' occurs together with 'string' or 'list', 'nil' has the lower precedence.

The conventions below are used for indicating context conditions:

If a category name appearing in an alternative is prefixed with a word in italics, then this word indicates a context condition, as explained in the tables 34.1–34.7 on pages 257–259. As an example consider the following syntax rule, where the context condition is that the maximal type of the constituent value expression must be **Bool**:

axiom_prefix_expr ::=
  • *logical*-value_expr

If a category name appearing in an alternative is prefixed with several words in italics separated by underscores, then each of the words indicates a context condition. As an example consider the following syntax rule, where the context conditions are that the constituent value expression must be read-only and have the maximal type **Bool**:

restriction ::=
  • *readonly_logical*-value_expr

If a category name appearing in an alternative is prefixed with a text containing several words in italics separated by '*_or_*', then this text indicates a context condition which is the disjunction of each of the individual context conditions (i.e. one of the context conditions must be fulfilled). As an example consider the following syntax rule, where the context condition is that the constituent name must represent a value or a variable:

value_expr ::=
  value_or_variable-name

Terminology Contains definitions of terms, etc. When a term is being defined it is written in italics.

Context-independent Expansions Contain expansions of constructs. When a construct is given a context-independent expansion in terms of other constructs its scope and visibility rules, context conditions, attributes and meaning are not stated — they are given by the scope and visibility rules, context

conditions, attributes and meaning of its expansion.

It is implicitly assumed that all constructs containing comments (belonging to the syntactic category comment) have a context-independent expansion to the construct obtained by removing the comments.

Scope and Visibility Rules Contain scope and visibility rules. The conventions used in the description of these are explained in sections 35.2–35.3.

Context Conditions Contain a description of the conditions that syntactically correct strings must satisfy in order to be statically correct. Note that as a convenience some of these conditions are also indicated by italicized prefixes in the syntax rules, as described above.

Context-dependent Expansions Contain expansions of constructs. When a construct is given a context-dependent expansion in terms of other constructs its attributes and meaning are not stated — they are given by the attributes and meaning of its expansion. Its context conditions are those stated plus those of its expansion.

Attributes Contain a description of attributes that statically correct strings have. Attributes are used to describe context conditions.

Meaning Contains a description of the meaning of statically correct strings.

prefix	context condition
element	the object_expr must represent a model
array	the object_expr must represent an object array

Table 34.1: Prefixes of object_expr and the context conditions they indicate

prefix	context condition
unit	the maximal type of the value_expr must be **Unit**
logical	the maximal type of the value_expr must be **Bool**
integer	the maximal type of the value_expr must be **Int**
list	the maximal type of the value_expr must be a list type
map	the maximal type of the value_expr must be a map type
function	the maximal type of the value_expr must be a function type
pure	the value_expr must be pure
readonly	the value_expr must be read-only

Table 34.2: Prefixes of value_expr and the context conditions they indicate

34.5 Static Correctness

This section presents an overview of the common context conditions of RSL.

A syntactically correct string is *statically correct* if its context conditions hold. The description *static* indicates that static correctness can be checked decidably

prefix	context condition
pure	the restriction must be pure

Table 34.3: Prefixes of restriction and the context conditions they indicate

prefix	context condition
pure	the set_limitation must be pure

Table 34.4: Prefixes of set_limitation and the context conditions they indicate

(i.e. by a terminating mechanical process). This means that we need to distinguish between what is statically true and what might be proved to be actually true. In particular we need to distinguish between static types (which we call 'maximal') and actual types of expressions, and between static accesses (the variables and channels an expression can access) and the actual accesses.

The main context conditions are:

1. All definitions having the same scope must be compatible.
2. All applied occurrences of operators and identifiers must be within the scopes of their definitions and these definitions must be visible.
3. It must be possible to associate uniquely and consistently a 'maximal' type with each operator and identifier representing a value, variable or channel.
4. The accesses to variables and channels of the bodies of explicit function definitions must be allowed by the access descriptions in their signatures.

1. The first condition about compatible definitions prevents things like:

type
T = **Int**,
T = **Bool**

In general different identifiers must be used for different things. There are some exceptions to this — see chapter 36 on overloading.

prefix	context condition
pure	the maximal type of the name must be a pure function type
type	the name must represent a type
value	the name must represent a value
variable	the name must represent a variable
channel	the name must represent a channel
scheme	the name must represent a scheme
object	the name must represent an object

Table 34.5: Prefixes of name and the context conditions they indicate

prefix	context condition
value	the id must represent a value

Table 34.6: Prefixes of id and the context conditions they indicate

prefix	context condition
associative	the infix_combinator must be associative
commutative	the infix_combinator must be commutative

Table 34.7: Prefixes of infix_combinator and the context conditions they indicate

2. The second condition about definitions being visible is similar to the standard rule in block structured programming languages. But note that RSL does not have any 'define before use' rule. More details about scope and visibility can be found in chapter 35.

3. The third condition about type consistency bans expressions like '1 + **true**'.

The rule is concerned with 'maximal' types since, for example, it is statically undecidable whether a particular integer expression will evaluate to give a natural number. So it is not against the context conditions to, for example, divide by zero or to take the head of an empty list, but the meaning of such an expression is typically under-specified in the semantics of RSL (which means that the specifier may not predict how the final implementation will behave).

4. The fourth condition ensures that the actual accesses made to variables and channels in the bodies of functions correspond to the accesses allowed in their signatures. So a function is only allowed to read a variable if it has read (or write) access to it, only allowed to write to a variable if it has write access to it, only allowed to input from a channel if it has input access to it and only allowed to output to a channel if it has output access to it.

34.6 Semantics

The semantics (meaning) of strings is only defined if they are syntactically and statically correct. Otherwise, they are literally meaningless.

The semantics of RSL is defined in terms of 'classes' of 'models'. This section provides an introduction to these terms.

A basic class expression consists of a string of declarations. Declarations may introduce identifiers or operators of various kinds, and they may express properties of these identifiers or operators. For example, the variable declaration:

variable v : **Int**

introduces the identifier v, says that its kind is variable and that its type is integer. The value declaration:

value x : **Nat**

introduces the identifier x, says that its kind is value and that its type is a natural number, an integer that is not negative. The axiom declaration:

axiom x > 2

introduces no new identifiers, but expresses the property that the value of x is greater than 2.

In the semantic world used to define the semantics of RSL there are denotations, which are things of kinds that correspond to the kinds of things that can be declared in RSL — schemes, objects, types, values, variables and channels — but with precise denotations or meanings. So the RSL integer literal 3 represents a denotation, but the notion of 'some integer greater than 2' does not represent a particular denotation but rather a whole collection of them — that represented by 3, that represented by 4, etc.

A 'model' in the semantic world is an association of identifiers and operators of various kinds with denotations, such as the association of the value identifier x with the denotation represented by 3.

It is clear that many class expressions will have more than one model. For example:

class
  value x : **Int**
  axiom x > 2
end

will have a model in which x is associated with the denotation represented by 3, one in which x is associated with the denotation represented by 4, etc. But the collection of possible models is enlarged further by including possible 'extensions'. The class expression above will also have models in which there is additionally an integer variable v, others in which there is additionally a Boolean variable v, and so on. In fact the models of a class expression are all the possible models that conform to it in associating (at least) the identifiers and operators it declares with denotations of the appropriate kind having the properties it declares. Such a collection of models is known as a 'class', and so the semantics of a class expression is a class of models.

CHAPTER 35

Declarative Constructs, Scope and Visibility Rules

35.1 Declarative Constructs

A *declarative construct* is a language construct representing one or more definitions. A *definition* introduces an identifier or operator for an entity such as a scheme, an object, a type, a value, a variable, a channel or an axiom. Examples of declarative constructs are module_decl, decl, formal_scheme_parameter, formal_array_parameter, formal_function_application, axiom_quantification, class_expr, object_expr, set_limitation, list_limitation, lambda_parameter, result_naming, let_def, binding, single_typing, typing, pattern, qualification

Note that for some declarative constructs the definitions they represent are determined by the context. For instance, for a pattern the definitions are determined by the context in which the pattern occurs. For such constructs the maximal types of the identifiers and/or operators introduced by the definitions are determined by a maximal type given by the context. Such a maximal type is called a *maximal context type* for (or of) the declarative construct.

A definition has an associated region of RSL text, called the *scope* of the definition. Within this scope, and only there, there are points where its entity may be referred to using its identifier or operator. By the scope of a declarative construct is meant the scope of its definitions. The *scope rules* of the language determine the scope of definitions.

A definition is said to be *visible* at a point of an RSL text if its entity may be referred to using its identifier or operator at that point. At such a point the identifier or operator is said to *represent* the entity or to be a *name of* the entity. The *visibility rules* of the language determine the visibility of definitions.

Two definitions are said to be *compatible* if they introduce distinct identifiers or operators or if they are both value definitions introducing the same identifier or operator but with distinguishable maximal types. Two declarative constructs are said to be compatible if all the definitions they represent are compatible.

The context conditions stated in connection with the individual syntax categories ensure that at each point of an RSL text all visible definitions are compatible.

35.2 Scope Rules

We describe conventions governing the description of scope rules.

The scope of a declarative construct may depend on the context in which it occurs. Therefore for each construct containing a declarative construct the scope of this must be given. This is done in the sections called 'Scope and visibility rules' using the following conventions:

1. For a declarative construct occurring immediately within a non-declarative construct the scope is always explicitly stated. This is for instance the case for the declarations in a local expression:

   ```
   local
     value x : Int = 3
   in x + 2 end
   ```

   The scope of the definition of x is the local declaration string and the value expression $x + 2$.
2. For a declarative construct occurring immediately within a declarative construct there are the following possibilities:
   (a) The scope is explicitly stated. This is for instance the case for the typings in an object definition:

   ```
   object O[i : Int] :
     class
       variable v : Int := i − 7
     end
   ```

   The scope of the definition of i is the class expression.
   (b) An immediate scope is stated. This is for instance the case for the declarations in a basic class expression. In this case the scope is the immediate scope plus possible extensions. The extensions depend on the context for the outer declarative construct and are given for all occurrences of it, such as for the first class expression in an extending class expression:

   ```
   scheme S =
     extend
       class
         value x : Int = 3,
       end
     with
       class
         value y : Int = x
       end
   ```

The immediate scope of the definition of x is the declaration string in the first class expression. The total scope of x is this region plus the the declaration string in the second class expression.

(c) No scope is given. This is for instance the case for the value definitions in a value declaration. In this case it is implicitly understood that the scope of the inner construct is given by the scope of the immediately enclosing outer construct in which it occurs. For example:

```
value
  x : Int = y,
  y : Int
```

The scope of the value definition of x is equal to the the scope of the whole value declaration.

35.3 Visibility Rules

The visibility rules are:

1. A definition is not visible outside its scope.
2. A definition is potentially visible throughout its scope. However, there may be places in the scope, where the definition is *hidden*, that is: not visible. For instance, if the identifier or operator introduced by a definition is also introduced by another definition in an inner scope then the outer definition is hidden throughout the scope of the inner definition. If both definitions are value definitions the outer is not hidden if the maximal types of the two values are distinguishable.

First an example with two variables v:

```
class
  variable v : Bool := true
  axiom
    local
      variable v : Int  := 3
    in v = 7 end
end
```

The scope of the variable definition v : **Bool** := **true** is the whole class expression, while the scope of the local variable definition v : **Int** := 3 is just the local declaration string and the value expression $v = 7$. Therefore, according to visibility rule number 2, in the value expression $v = 7$ only the local variable definition is visible. In fact an inner definition of v will always hide an outer one within the inner scope unless they are both values. Consider, for example:

```
class
  value v : Bool = true
  axiom
```

```
      local
        value v : Int = 3
      in v end
  end
```

In the local expression the local value definition does not hide the outer value definition as the maximal types of the two value definitions are distinguishable. Both value definitions are visible in the local expression and in fact the occurrence of v must (since axioms are Boolean expressions) be of the outer definition.

CHAPTER 36

Overloading

36.1 General

An identifier or operator is said to be *overloaded* at a certain point if there are several definitions of that identifier or operator which are visible at that point.

Only value identifiers and operators are allowed to be overloaded.

Note that all operators have one or more predefined meanings which have the whole specification as their scope and which cannot be hidden — except that they are hidden (not visible) within the operator part of qualified operators (see section 46.3). This implies that if the user defines an operator to have a maximal type distinguishable from the maximal types of the predefined meanings of the operator then — at points where both the user definition and the predefined meanings of the operator are visible — the operator is overloaded. The user is not allowed to define an operator to have a maximal type indistinguishable from one of the maximal types of the predefined meanings of the operator — see the context conditions in chapter 47.

36.2 Overload Resolution

For a specification to be useful there must be a unique legal interpretation of each identifier and operator, where by an *interpretation* is meant a corresponding definition. Now, an occurrence of an overloaded identifier or operator has several possible interpretations (namely one for each visible definition of it) and therefore the problem is to find its legal corresponding definition (if it has any).

Considering the context of the identifier or operator, some of the possible interpretations may be illegal according to the context conditions. In general the more context one considers the more information (context conditions) exists to eliminate illegal interpretations. But if the context (a construct) considered is a value expression which has the same maximal type for several different possible interpretations of the constituent overloaded identifiers and operators then further

context will never make it possible to choose one of these interpretations over the other ones. Therefore all such interpretations are illegal.

More formally, for a given construct the *legal interpretations* of the applied occurrences of identifiers and operators are found in the following way:

1. If the construct has no subconstructs then consider all combinations of possible interpretations of the identifiers and operators. Otherwise consider all combinations of interpretations which are legal for subconstructs of the construct.
2. Then remove those combinations which do not satisfy the context conditions for the construct.
3. Finally, if the construct is a value expression (belongs to the syntactic category value_expr) then remove those combinations for which the construct has indistinguishable maximal types.

The combinations of interpretations obtained in this way contain those interpretations of the applied occurrences of the identifiers and operators which are legal for the construct.

The overloading is said to be *resolvable* if there is exactly one legal interpretation of each identifier and operator in its innermost enclosing 'resolving context'.

A *resolving context* is one of the following:

- The value_expr in a list_limitation
- The value_expr in an explicit_let
- The value_expr in a case_expr
- The value_expr in a post_expr
- A defined_item which is just an id_or_op
- A specification

Example 1:

```
class
  value
    v : Int,
    v : Bool
  axiom
    v
end
```

The occurrence of v in the axiom is overloaded — it has two possible interpretations: either it is an integer or it is a Boolean. However, only the latter interpretation satisfies the context condition that an axiom must have the maximal type **Bool**, and hence only this interpretation is legal. As there is exactly one legal interpretation the overloading is resolvable.

Example 2:

```
class
  value
```

```
    + : Bool × Bool → Bool,
    v : Real
  axiom
    true + false ≡ true,
    v ≡ 1.7 + 2.2
end
```

The two occurrences of the operator, +, in the axioms are overloaded — each of the occurrences has three possible interpretations: it is the predefined integer addition (having the maximal type **Int** × **Int** $\xrightarrow{\sim}$ **Int**) or it is the predefined real addition (having the maximal type **Real** × **Real** $\xrightarrow{\sim}$ **Real**) or it is the user-defined Boolean addition (having the maximal type **Bool** × **Bool** $\xrightarrow{\sim}$ **Bool**). Only the user-defined one satisfies the context conditions for the first occurrence, while only the predefined real addition satisfies the context conditions for the second occurrence. As for each of the two occurrences of + there is exactly one legal interpretation the overloading is resolvable.

Example 3:

```
value
  v : Int,
  v : Bool,
  f : Int → Int,
  f : Bool → Nat
axiom
  f(v) ≡ 7
```

There are two combinations of interpretations for f and v satisfying the context conditions. These are:

1. f : **Int** → **Int**, v : **Int**
2. f : **Bool** → **Nat**, v : **Bool**

However, for both combinations the maximal type of $f(v)$ is the same, namely **Int**, and hence the value expression $f(v)$ has no legal interpretations. Therefore the overloading is not resolvable.

Example 4:

```
type
  B,
  C,
  A = B | C
value
  b : B,
  v : B,
  v : C,
  f : A → Bool,
  /* illegal */ a : A = v
axiom
```

/* legal */ f(b),
/* illegal */ f(v)

In the first axiom the identifier b has exactly one legal interpretation, $b : B$. (It is legal as there is an implicit coercion A_from_B which converts the value expression b of maximal type B to a value expression $A\_from\_B(b)$ of maximal type A.) Hence the overloading is resolvable.

In the second axiom the identifier v has two possible interpretations, $v : B$ and $v : C$. Both of these satisfy the context conditions, but for both interpretations the maximal type of the value expression $f(v)$ is the same. Therefore there are no legal interpretations of v in the value expression $f(v)$. Hence the overloading is not resolvable.

In the definition of a the identifier v has two possible interpretations, $v : B$ and $v : C$. Both of these satisfy the context conditions and the maximal types are different. Therefore both interpretations are legal. Hence the overloading is not resolvable.

CHAPTER 37

Specifications

Syntax

specification ::=
  module_decl-string

module_decl ::=
  scheme_decl |
  object_decl

Terminology A *module* is either an object or a scheme.

Scope and Visibility Rules In a specification the scope of the constituent module_decl-string is the module_decl-string itself. Note that this means that the order of definitions is immaterial — an object or a scheme may be used before it is defined.

Context Conditions The constituent module_decls must be compatible.

Meaning A specification stands for one or more module definitions.

CHAPTER 38

Declarations

38.1 General

Syntax

decl ::=
  scheme_decl |
  object_decl |
  type_decl |
  value_decl |
  variable_decl |
  channel_decl |
  axiom_decl

Terminology A declaration is a list of definitions all of the same *kind* – scheme, object, type, value, variable, channel or axiom. Each definition normally introduces an identifier or operator for an *entity* of that kind. Furthermore, it normally states one or more *properties* of that entity.

Attributes Except for axiom definitions, each kind of definition has an associated maximal definition. The specific maximal definition is defined for each kind of definition in the relevant section.

38.2 Scheme Declarations

Syntax

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

Terminology A *scheme* is either a class or a parameterized class.

A *parameterized class* is a mapping from lists of objects to classes: each object list is mapped to a class.

The *maximal class* of a scheme is the maximal class of the class expression in the scheme's definition.

A scheme_def is *cyclic* if the opt-formal_array_parameter or class_expr depends on the scheme introduced by the scheme_def itself.

A construct *depends* on a scheme, S, if it, when disregarding restrictions in subtype expressions, refers to S or to any scheme having a definition in which the opt-formal_scheme_parameter or class_expr depends on S.

Scope and Visibility Rules In a scheme_def the scope of the opt-formal_scheme_parameter is the opt-formal_scheme_parameter itself and the class_expr.

Context Conditions In a scheme_decl the constituent scheme_defs must be compatible.

In a formal_scheme_parameter the constituent formal_scheme_arguments must be compatible.

A scheme_def must not be cyclic.

Attributes In a scheme_def the maximal class of the id is the maximal class of the constituent class_expr and if there is a formal_scheme_parameter present then the id has also a formal scheme parameter which is the formal_scheme_parameter.

The maximal definition of a scheme_def is obtained from the original by replacing the object definitions in its formal_scheme_parameter (if any) by the corresponding maximal object definitions and the constituent class_expr by the corresponding maximal class expression.

Meaning A scheme_def introduces the id for a scheme.

- A scheme_def of the form:

  id = class_expr

  introduces the constituent *id* for a class. The class is the one represented by the *class_expr*.
- A scheme_def of the form:

  id(formal_scheme_argument-list) = class_expr

  introduces the constituent *id* for a parameterized class.

  A parameterized class may be applied to a list of objects in a scheme instantiation as described in section 39.6. That section describes which actual parameters are allowed and what the class resulting from the instantiation is.

38.3 Object Declarations

Syntax

object_decl ::=
  object object_def-list

object_def ::=
  opt-comment-string id opt-formal_array_parameter : class_expr

formal_array_parameter ::=
  [typing-list]

Terminology An *object* is either a model or an array of models.

An *array of models* — also termed an *array* — is a mapping from values to models: each value is mapped to a single model.

The *maximal class* of an object is the maximal class of the class expression in the object's definition.

The *index type* of an array is the type of values, all of which are mapped to a model by the array. An *index value* is a value within the index type.

An array maps any two distinct index values into two models that do not have component objects, variables or channels in common. However, in the context conditions, these are not distinguished.

Scope and Visibility Rules In an object_def the scope of the opt-formal_array-_parameter is the class_expr.

Context Conditions In an object_decl the constituent object_defs must be compatible.

Attributes In an object_def the maximal class of the id is the maximal class of the class_expr and if there is a formal_array_parameter then the id has also a maximal index type which is the maximal index type of the formal_array_parameter.

The maximal index type of a formal_array_parameter is the maximal type of the typing-list.

The maximal definition of an object_def is obtained by replacing its index type expression (if any) by the corresponding maximal type expression and its class_expr by the corresponding maximal class expression.

Meaning An object_def introduces the id for an object.

- An object_def of the form:

  id : class_expr

  introduces the constituent *id* for a model. The model is an arbitrary one belonging to the class represented by the *class_expr*.
- An object_def of the form:

  id[typing-list] : class_expr

introduces the constituent *id* for an array of models. The index type of the array is the type represented by the *typing*-list. Each index value belonging to the index type is mapped to a model. The model is an arbitrary one belonging to the class represented by the *class_expr* — evaluated in scope of the definitions given by matching the index value against the binding also represented by the *typing*-list.

An array may be applied to an index value in an element object expression as described in section 40.3.

No two defined objects have component objects, variables or channels in common.

38.4 Type Declarations

Syntax

type_decl ::=
  type type_def-list

type_def ::=
  sort_def |
  variant_def |
  union_def |
  short_record_def |
  abbreviation_def

Context Conditions In a type_decl the constituent type_defs must be compatible.

38.4.1 Sort Definitions

Syntax

sort_def ::=
  opt-comment-string id

Terminology A *sort* — or synonymously *abstract type* — is a type with no predefined value literals and no predefined operators other than = and ≠.

Attributes The maximal type of the constituent id is the type represented by the id.

A sort definition is maximal.

Meaning A sort_def introduces the id for a sort.

Since a sort is not provided with predefined value literals or operators other than = and ≠ for generating and manipulating its values, the writers of specifications must define such values themselves. Their definitions may indirectly state properties about the sort. If for example two values of the same sort are defined and they

are specified to be different, then indirectly the sort is required to contain at least two values.

38.4.2 Variant Definitions

Syntax

variant_def ::=
  opt-comment-string id == variant-choice

variant ::=
  constructor |
  record_variant

record_variant ::=
  constructor (component_kind-list)

component_kind ::=
  opt-destructor type_expr opt-reconstructor

constructor ::=
  id_or_op |
  —

destructor ::=
  id_or_op :

reconstructor ::=
  ↔ id_or_op

Context-independent Expansions A variant_def is short for defining an abstract type, its constructors, destructors and reconstructors. The following describes how a variant_def of the form:

type id == variant$_1$| ... |variant$_n$

is short for a sort definition, some value definitions and some axioms. The description deals in turn with constructors, destructors, reconstructors, disjointness axioms and induction axioms. The following example is used throughout:

type Tree == empty | node(left : Tree, val : Elem ↔ repl_value, right : Tree)

- *Constructors*
  Tree has two variants, one of which is a constant_variant and one a record_variant.
  The destructors and reconstructors are dealt with below, and so we ignore them here.

A series of declarations can now be constructed that the original declaration is short for. First, there is an abstract type declaration for the variant type being defined:

type id

For the example this declaration would be:

type Tree

Secondly, for each variant, indexed by i say, where the constructor con_i is not a wildcard '_', a value declaration is obtained.

If the variant is a constant_variant, say con_i, the following value declaration is obtained:

value $\mathrm{con_i}$: id

which simply says that con_i is a (constant) value of type id. For the example one obtains the single value declaration:

value empty : Tree

If the variant is a record_variant having n_i component_kinds, say:

$\mathrm{con_i}(\mathrm{T_{i,1}}, ..., \mathrm{T_{i,n_i}})$

the following value declaration is obtained:

value $\mathrm{con_i} : \mathrm{T_{i,1}} \times ... \times \mathrm{T_{i,n_i}} \to \mathrm{id}$

which says that con_i is a total function from the product of its component types to the type id. The function con_i constructs values of type id from values of its component types (as indicated by the name 'constructor' of its syntactic category).

For the example one obtains the single value declaration:

value node : Tree × Elem × Tree → Tree

constant_variants and record_variants which have wildcards for their constructors do not generate any value declarations (except for any destructors or reconstructors attached to the components of a record_variant).

- *Destructors*

  Each destructor $dest_{i,j}$ introduced in a record_variant:

  $\mathrm{con_i}(..., \mathrm{dest_{i,j}} : \mathrm{T_{i,j}}, ...)$

  generates first a value declaration:

  value $\mathrm{dest_{i,j}} : \mathrm{id} \xrightarrow{\sim} \mathrm{T_{i,j}}$

  For the example one obtains the following declaration:

  value
  left : Tree $\xrightarrow{\sim}$ Tree,
  val : Tree $\xrightarrow{\sim}$ Elem,
  right : Tree $\xrightarrow{\sim}$ Tree

If the constructor con_i is not a wildcard, it also generates an axiom of the following form:

axiom
$\forall\ x_1 : T_{i,1}, ..., x_{n_i} : T_{i,n_i} \bullet$
$dest_{i,j}(con_i(x_1, ..., x_{n_i})) \equiv x_j$

For the example one obtains the axioms:

axiom
$\forall\ x_1 : Tree, x_2 : Elem, x_3 : Tree \bullet$
$left(node(x_1, x_2, x_3)) = x_1,$
$\forall\ x_1 : Tree, x_2 : Elem, x_3 : Tree \bullet$
$val(node(x_1, x_2, x_3)) = x_2,$
$\forall\ x_1 : Tree, x_2 : Elem, x_3 : Tree \bullet$
$right(node(x_1, x_2, x_3)) = x_3,$

- *Reconstructors*
  Each reconstructor $recon_{i,j}$ introduced in a record_variant:

  $con_i(..., ... T_{i,j} \leftrightarrow recon_{i,j}, ...)$

  generates first a value declaration:

  value $recon_{i,j} : T_{i,j} \times id \xrightarrow{\sim} id$

  For the example one obtains, for the one reconstructor $repl\_val$:

  value repl_val : Elem × Tree $\xrightarrow{\sim}$ Tree

  If there are destructors associated with the variant then for each destructor $dest_{i,k}$ there is an axiom relating it to the reconstructor $recon_{i,j}$. For the case when j and k are equal one obtains the axiom:

  $\forall\ x_j : T_{i,j}, x : id \bullet$
  $dest_{i,j}(recon_{i,j}(x_j,x)) \equiv x_j$

  which shows that a destructor recovers the component value changed by a corresponding reconstructor.

  When j and k are different one obtains the axiom:

  $\forall\ x_j : T_{i,j}, x : id \bullet$
  $dest_{i,k}(recon_{i,j}(x_j,x)) \equiv dest_{i,k}(x)$

  which expresses the fact that changing a component value by a reconstructor does not affect other components.

  In the example, one obtains the following three axioms:

  axiom
  $\forall\ x_2 : Elem, x : Tree \bullet$
  $left(repl\_val(x_2,x)) \equiv left(x),$
  $\forall\ x_2 : Elem, x : Tree \bullet$
  $val(repl\_val(x_2,x)) \equiv x_2,$

$\forall\ x_2 : Elem, x : Tree \bullet$
$\quad right(repl\_val(x_2,x)) \equiv right(x)$

- *Disjointness axioms*

  Disjointness axioms state that different constructors map into different values. For any two distinct constructors con_i and con_j, the following disjointness axiom is generated:

  axiom
  $\forall\ x_{i,1} : T_{i,1}, ..., x_{i,n_i} : T_{i,n_i} \bullet$
  $\forall\ x_{j,1} : T_{j,1}, ..., x_{j,n_j} : T_{j,n_j} \bullet$
  $\quad con_i(x_{i,1},...,x_{i,n_i}) \neq con_j(x_{j,1},...,x_{j,n_j})$

  (In this definition, for any variant, index k, say, that is a constant_variant, n_k is taken to be zero and one obtains the subexpression con_k in the inequality.)

  The example gives the following disjointness axiom:

  axiom
  $\forall\ x_1 : Tree, x_2 : Elem, x_3 : Tree \bullet$
  $\quad empty \neq node(x_1,x_2,x_3)$

- *Induction axioms*

  Provided there are no variants with wildcard constructors in the type definition one also obtains an induction axiom. (The removal of the induction axiom is the main reason for using wildcard variants — they allow one to add further variants, or components of variants later, and obtain implementation. If there were an induction axiom, making such additions would negate it and so could not give implementation.)

  If there is no recursion in the type, i.e. none of the component types in any variants involve *id*, then the induction axiom is simple:

  axiom
  $\forall\ f : id \to \mathbf{Bool} \bullet$
  $\quad ($
  $\quad\quad (\forall\ x_1 : T_{1,1}, ..., x_{n_1} : T_{1,n_1} \bullet$
  $\quad\quad\quad f(con_1(x_1,...,x_{n_1})))$
  $\quad\quad \wedge\ ... \wedge$
  $\quad\quad (\forall\ x_1 : T_{n,1}, ..., x_{n_n} : T_{n,n_n} \bullet$
  $\quad\quad\quad f(con_n(x_1,...,x_{n_n})))$
  $\quad) \Rightarrow$
  $\quad\quad (\forall\ x : id \bullet f(x))$

  (In this definition, for any variant, index i, say, that is a constant_variant, n_i is taken to be zero so that the quantification in the conjunct disappears and one obtains a conjunct $f(con_i)$.)

  Suppose now that the type is recursive, and that the j'th component in the i'th variant is *id*. Then in the above definition the i'th conjunct becomes:

  $(\forall\ x_1 : T_{i,1}, ..., x_j : id, ..., x_{n_i} : T_{i,n_i} \bullet$

$f(x_j) \Rightarrow f(con_i(x_1,...,x_j,...,x_{n_i})))$

There are obvious extensions to this when there are two or more component types in a variant equal to *id*. For two such one obtains a conjunct of the form:

$(\forall\ x_1 : T_{i,1},\ ...,\ x_j : id,\ ...,\ x_k : id,\ ...,\ x_{n_i} : T_{i,n_i} \bullet$
$\quad (f(x_j) \wedge f(x_k)) \Rightarrow$
$\quad\quad f(con_i(x_1,...,x_j,...,x_k,...,x_{n_i})))$

This is the case in the example, which has the induction axiom:

axiom
$\quad \forall\ f : Tree \to \textbf{Bool} \bullet$
$\quad\quad ($
$\quad\quad\quad f(empty)$
$\quad\quad\quad\quad \wedge$
$\quad\quad\quad (\forall\ x_1 : Tree,\ x_2 : Elem,\ x_3 : Tree \bullet$
$\quad\quad\quad\quad (f(x_1) \wedge f(x_3)) \Rightarrow f(node(x_1,x_2,x_3)))$
$\quad\quad) \Rightarrow$
$\quad\quad\quad (\forall\ x : Tree \bullet f(x))$

So to prove some property of the type *Tree* one proves it for *empty* and one then proves it for a constructed node assuming it is true for the left and right subtrees.

Another extension is when *id* is a component of $T_{i,j}$ instead of equal to it. For instance, suppose $T_{i,j}$ is $U \times id$. Then the conjunct would be:

$(\forall\ x_1 : T_{i,1},\ ...,\ (y,z) : (U \times id),\ ...,\ x_{n_i} : T_{i,n_i} \bullet$
$\quad f(z) \Rightarrow f(con_i(x_1,...,(y,z),...,x_{n_i})))$

This leads to the possibility that *id* is a component of a variant type $T_{i,j}$ and hence to the problem of mutually recursive variant types. The general rule here is fairly complicated, and the reader is referred to the proof rules ([37]) for its formulation. Consider instead an example. Suppose one generalizes trees to have lists of subnodes, and define lists by variants:

type
$\quad Tree == empty\_tree \mid node(val : Elem,\ sub : List),$
$\quad List == empty\_list \mid list(head : Tree,\ tail : List)$

The induction axiom for these is a joint one, formulated as follows:

axiom
$\quad \forall\ tf : Tree \to \textbf{Bool},\ lf : List \to \textbf{Bool} \bullet$
$\quad\quad ($
$\quad\quad\quad tf(empty\_tree) \wedge$
$\quad\quad\quad (\forall\ x_1 : Elem,\ x_2 : List \bullet$
$\quad\quad\quad\quad lf(x_2) \Rightarrow tf(node(x_1,x_2))) \wedge$
$\quad\quad\quad lf(empty\_list) \wedge$

$$
\begin{array}{l}
\quad (\forall\ x_1 : \text{Tree},\ x_2 : \text{List} \bullet \\
\qquad (\text{tf}(x_1) \wedge \text{lf}(x_2)) \Rightarrow \text{lf}(\text{list}(x_1,x_2))) \\
) \Rightarrow \\
\quad (\forall\ x_1 : \text{Tree},\ x_2 : \text{List} \bullet (\text{tf}(x_1) \wedge \text{lf}(x_2)))
\end{array}
$$

So to prove a pair of properties of *Tree* and *List* one proves the appropriate properties for the constants *empty_tree* and *empty_list*, and also proves them for the constructed values assuming the appropriate properties of components.

38.4.3 Union Definitions

Syntax

union_def ::=
  opt-comment-string id = name_or_wildcard-choice2

name_or_wildcard ::=
  type-name |
  _

Context Conditions The constituent names must represent types and the maximal types of these must be distinguishable.

Context-dependent Expansions A union_def of the form:

type id = opt-qualification$_1$ id$_1$| ... | opt-qualification$_n$ id$_n$ | _

is equivalent to the variant definition

type
  id ==
    id_from_id$_1$(id_to_id$_1$: opt-qualification$_1$ id$_1$)| ... |
    id_from_id$_n$(id_to_id$_n$: opt-qualification$_n$ id$_n$)| _

provided all implicit coercions in value expressions and patterns involving the functions *id_from_id*$_i$ ($1 \leq i \leq n$) have already been replaced by actual coercions as explained in sections 42.1 and 45.1.

If the union_def does not contain a wildcard '_' alternative, the variant definition will not do so either.

38.4.4 Short Record Definitions

Syntax

short_record_def ::=
  opt-comment-string id :: component_kind-string

Context-independent Expansions A short_record_def is short for a variant definition with a single variant including a constructor. A short_record_def of the form:

type id :: component_kind$_1$ ... component_kind$_n$

is short for:

type id == mk_id(component_kind$_1$, ... ,component_kind$_n$)

38.4.5 Abbreviation Definitions

Syntax

abbreviation_def ::=
  opt-comment-string id = type_expr

Terminology An abbreviation_def is *cyclic* if the maximal type of its type_expr depends on the type introduced by the abbreviation_def itself.

A maximal type *depends* on a type, t, if it refers to t.

Context Conditions An abbreviation_def must not be cyclic.

Attributes The maximal type of the constituent id is the maximal type of the constituent type_expr.

The maximal definition of an abbreviation_def is obtained by replacing its constituent type_expr by the corresponding maximal type expression.

Meaning An abbreviation_def introduces the id for the type represented by the type_expr.

38.5 Value Declarations

Syntax

value_decl ::=
  value value_def-list

value_def ::=
  commented_typing |
  explicit_value_def |
  implicit_value_def |
  explicit_function_def |
  implicit_function_def

Context Conditions In a value_decl the constituent value_defs must be compatible.

38.5.1 Commented Typings

See chapter 44 on typings.

38.5.2 Explicit Value Definitions

Syntax

explicit_value_def ::=
opt-comment-string single_typing = *pure*-value_expr

Context Conditions The maximal type of the value_expr must be less than or equal to the maximal type of the single_typing.

The constituent value_expr must be pure.

Context-dependent Expansions An explicit_value_def is short for a value definition and an axiom.

Assume the meta-function *express* that turns a binding into a value expression by bracketing all operators and leaving the rest of the binding unchanged. Section 46.3 describes the meaning of bracketed operators.

An explicit_value_def of the form:

value binding : type_expr = value_expr

is short for:

value binding : type_expr
axiom *express*(binding) = value_expr

38.5.3 Implicit Value Definitions

Syntax

implicit_value_def ::=
opt-comment-string single_typing *pure*-restriction

Context Conditions The restriction must be pure.

Context-dependent Expansions An implicit_value_def is short for a value definition and an axiom.

Assume the meta-function *express* that turns a binding into a value expression by bracketing all operators and leaving the rest of the binding unchanged. Section 46.3 describes the meaning of bracketed operators.

An implicit_value_def of the form:

value binding : type_expr • value_expr

is short for:

value binding : type_expr
axiom value_expr

38.5.4 Explicit Function Definitions

Syntax

explicit_function_def ::=
  opt-comment-string single_typing
    formal_function_application ≡ value_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

Terminology The *body* of an explicitly defined function is the value expression in the function's definition.

Scope and Visibility Rules In an explicit_function_def the scope of the formal_function_application is value_expr and opt-pre_condition.

Context Conditions The binding in the single_typing must be an id_or_op (i.e. not a product_binding).

If the id_or_op in the single_typing is an id then the formal_function_application must be an id_application of this id.

If the id_or_op in the single_typing is a prefix_op then the formal_function_application must be a prefix_application of this prefix_op.

If the id_or_op in the single_typing is an infix_op then the formal_function_application must be an infix_application of this infix_op.

The maximal type of the single_typing must be a function type. If the formal_function_application is an infix_application then the parameter part of the function type must be a product type of length 2. If the formal_function_application is an id_application then the function type must be formed using at least as many types, each before $\overset{\sim}{\to}$ or $\to$, as there are formal_function_parameters.

That is: there are the three following legal forms, where any $\overset{\sim}{\to}$ may be replaced by $\to$, and any of the function type expressions may be replaced by names which represent them due to abbreviation definitions:

id : type_expr$_1$ $\overset{\sim}{\to}$ opt-access_desc-string$_1$ type_expr$_2$
  ... $\overset{\sim}{\to}$ opt-access_desc-string$_n$ type_expr$_{n+1}$

id(opt-binding-list$_1$)(opt-binding-list$_2$)...(opt-binding-list$_m$) ≡
  value_expr opt-pre_condition (m ≤ n)

prefix_op : type_expr$_1$ $\xrightarrow{\sim}$ opt-access_desc-string type_expr$_2$
prefix_op id ≡ value_expr opt-pre_condition

infix_op : type_expr$_1$ × type_expr$_2$ $\xrightarrow{\sim}$ opt-access_desc-string type_expr$_3$
id$_1$ infix_op id$_2$ ≡ value_expr opt-pre_condition

In the first form above, the maximal type of the *value_expr* must be less than or equal to the maximal type of either *type_expr*$_{n+1}$ if m = n or:

type_expr$_{m+1}$ ... $\xrightarrow{\sim}$ opt-access_desc-string$_n$ type_expr$_{n+1}$

if m < n.

In the second form above, the maximal type of the *value_expr* must be less than or equal to the maximal type of *type_expr*$_2$.

In the third form above, the maximal type of the *value_expr* must be less than or equal to the maximal type of *type_expr*$_3$.

The *value_expr* and the opt-*pre_condition* can only statically access those variables and channels that are in the static access descriptions of *opt-access_desc-string*$_1$ — *opt-access_desc-string*$_m$ in the first form above and the static access descriptions of *opt-access_desc-string* in the two last forms.

Context-dependent Expansions An explicit_function_def is short for a value definition and an axiom, depending on the form of the formal_function_application.

Assume the meta-function *maximal* that turns a type into its maximal type.

Assume the meta-function *express* that turns a binding into a value expression by bracketing all operators and leaving the rest of the binding unchanged. Section 46.3 describes the meaning of bracketed operators.

An explicit_function_def of the following form where the formal_function_application is an id_application with only one formal_function_parameter:

value
  id : type_expr$_1$ × ... × type_expr$_n$ $\xrightarrow{\sim}$ opt-access_desc-string type_expr$_{n+1}$
  id(binding$_1$,...,binding$_n$) ≡ value_expr opt-pre_condition

where $n \geq 1$, is short for:

value
  id : type_expr$_1$ × ... × type_expr$_n$ $\xrightarrow{\sim}$ opt-access_desc-string type_expr$_{n+1}$
axiom
  ∀ binding$_1$: *maximal*(type_expr$_1$),...,binding$_n$: *maximal*(type_expr$_n$) •
    id(*express*(binding$_1$),...,*express*(binding$_n$)) ≡ value_expr opt-pre_condition

If the binding list binding$_1$,...,binding$_n$ is nil then the type expression type_expr$_1$ × ... × type_expr$_n$ must be **Unit** and the quantification is not needed.

Observe that each formal parameter *binding*$_i$ ranges over the maximal type of the corresponding formal parameter *type_expr*$_i$.

The analogous expansion holds if $\tilde{\rightarrow}$ is replaced by $\rightarrow$.

The case with an id_application containing more than one formal_function_parameter has a similar and obvious explanation.

An explicit_function_def of the following form where the formal_function_application is a prefix_application:

```
value
  prefix_op : type_expr1 ~→ opt-access_desc-string type_expr2
  prefix_op id ≡ value_expr opt-pre_condition
```

is short for:

```
value
  prefix_op : type_expr1 ~→ opt-access_desc-string type_expr2
axiom
  ∀ id : maximal(type_expr1) •
    prefix_op id ≡ value_expr opt-pre_condition
```

Observe that the formal parameter *id* ranges over the maximal type of the formal parameter *type_expr*$_1$.

The analogous expansion holds if $\tilde{\rightarrow}$ is replaced by $\rightarrow$.

An explicit_function_def of the following form where the formal_function_application is an infix_application:

```
value
  infix_op : type_expr1 × type_expr2 ~→ opt-access_desc-string type_expr3
  id1 infix_op id2 ≡
    value_expr
    opt-pre_condition
```

is short for:

```
value
  infix_op : type_expr1 × type_expr2 ~→ opt-access_desc-string type_expr3
axiom
  ∀ id1 : maximal(type_expr1),id2 : maximal(type_expr2) •
    id1 infix_op id2 ≡ value_expr opt-pre_condition
```

Observe that each formal parameter *id*$_i$ ranges over the maximal type of the corresponding formal parameter *type_expr*$_i$.

The analogous expansion holds if $\tilde{\rightarrow}$ is replaced by $\rightarrow$.

38.5.5 Implicit Function Definitions

Syntax

```
implicit_function_def ::=
  opt-comment-string single_typing formal_function_application
    post_condition opt-pre_condition
```

Scope and Visibility Rules In an implicit_function_def the scope of the formal_function_application is the post_condition and opt-pre_condition.

Context Conditions The binding in the single_typing must be an id_or_op (and not a product_binding).

If the id_or_op in the single_typing is an id then the formal_function_application must be an id_application of this id.

If the id_or_op in the single_typing is a prefix_op then the formal_function_application must be a prefix_application of this prefix_op.

If the id_or_op in the single_typing is an infix_op then the formal_function_application must be an infix_application of this infix_op.

The maximal type of the single_typing must be a function type. If the formal_function_application is an infix_application then the parameter part of the function type must be a product type of length 2. If the formal_function_application is an id_application then the function type must be formed using at least as many types, each before $\stackrel{\sim}{\to}$ or $\to$, as there are formal_function_parameters.

That is: there are the three following legal forms, where any of the partial arrows may be replaced by $\to$, and any of the function type expressions may be replaced by names which represent them due to abbreviation definitions:

id : type_expr$_1$ $\stackrel{\sim}{\to}$ opt-access_dec-string$_1$ type_expr$_2$
... $\stackrel{\sim}{\to}$ opt-access_desc-string$_n$ type_expr$_{n+1}$
id(opt-binding-list$_1$)(opt-binding-list$_2$)...(opt-binding-list$_m$)
post_condition opt-pre_condition $(m \leq n)$

prefix_op : type_expr$_1$ $\stackrel{\sim}{\to}$ opt-access_desc-string type_expr$_2$
prefix_op id post_condition opt-pre_condition

infix_op : type_expr$_1$ $\times$ type_expr$_2$ $\stackrel{\sim}{\to}$ opt-access_desc-string type_expr$_3$
id$_1$ infix_op id$_2$ post_condition opt-pre_condition

The post_condition and the opt-pre_condition can only statically read those variables and channels that are in the static access descriptions of *opt-access_desc-string*$_1$ — *opt-access_desc-string*$_m$ in the first form above and the static access descriptions of *opt-access_desc-string* in the two last forms.

Context-dependent Expansions An implicit_function_def is short for a value definition and an axiom, depending on the form of the formal_function_application.

Assume the meta-function *maximal* that turns a type into its maximal type.

Assume the meta-function *express* that turns a binding into a value expression by bracketing all operators and leaving the rest of the binding unchanged. Section 46.3 describes the meaning of bracketed operators.

An implicit_function_def of the following form where the formal_function_application is an id_application with only one formal_function_parameter:

value
id : type_expr$_1$ $\times$ ... $\times$ type_expr$_n$ $\stackrel{\sim}{\to}$ opt-access_desc-string type_expr$_{n+1}$

id(binding$_1$,...,binding$_n$) post_condition opt-pre_condition

where $n \geq 1$, is short for:

value
id : type_expr$_1$ × ... × type_expr$_n$ $\xrightarrow{\sim}$ opt-access_desc-string type_expr$_{n+1}$
axiom
∀ binding$_1$: *maximal*(type_expr$_1$),...,binding$_n$: *maximal*(type_expr$_n$) •
id(*express*(binding$_1$),...,*express*(binding$_n$)) post_condition opt-pre_condition

If the binding list binding$_1$,...,binding$_n$ is nil then the type expression type_expr$_1$ × ... × type_expr$_n$ must be **Unit** and the quantification is not needed.

Observe that each formal parameter *binding*$_i$ ranges over the maximal type of the corresponding formal parameter *type_expr*$_i$.

The analogous expansion holds if $\xrightarrow{\sim}$ is replaced by →.

The case with an id_application containing more than one formal_function_parameter has a similar and obvious explanation.

An implicit_function_def of the following form where the formal_function_application is a prefix_application:

value
prefix_op : type_expr$_1$ $\xrightarrow{\sim}$ opt-access_desc-string type_expr$_2$
prefix_op id post_condition opt-pre_condition

is short for:

value
prefix_op : type_expr$_1$ $\xrightarrow{\sim}$ opt-access_desc-string type_expr$_2$
axiom
∀ id : *maximal*(type_expr$_1$) •
prefix_op id post_condition opt-pre_condition

Observe that the formal parameter *id* ranges over the maximal type of the formal parameter *type_expr*$_1$.

The analogous expansion holds if $\xrightarrow{\sim}$ is replaced by →.

An implicit_function_def of the following form where the formal_function_application is an infix_application:

value
infix_op : type_expr$_1$ × type_expr$_2$ $\xrightarrow{\sim}$ opt-access_desc-string type_expr$_3$
id$_1$ infix_op id$_2$ post_condition opt-pre_condition

is short for:

value
infix_op : type_expr$_1$ × type_expr$_2$ $\xrightarrow{\sim}$ opt-access_desc-string type_expr$_3$
axiom
∀ id$_1$: *maximal*(type_expr$_1$),id$_2$: *maximal*(type_expr$_2$) •
id$_1$ infix_op id$_2$ post_condition opt-pre_condition

Observe that each formal parameter id_i ranges over the maximal type of the corresponding formal parameter $type\_expr_i$.

The analogous expansion holds if $\xrightarrow{\sim}$ is replaced by $\rightarrow$.

38.6 Variable Declarations

Syntax

variable_decl ::=
  variable variable_def-list

variable_def ::=
  single_variable_def |
  multiple_variable_def

single_variable_def ::=
  opt-comment-string id : type_expr opt-initialisation

initialisation ::=
  := *pure*-value_expr

multiple_variable_def ::=
  opt-comment-string id-list2 : type_expr

Terminology A *variable* is an entity in which values of a particular type can be stored. *Assigning* a value to a variable means storing the value in the variable. A value can explicitly be assigned to a variable by an assignment expression (section 42.21).

A *state* of a specification is a particular assignment of values to all variables defined in the constituent objects of the specification.

A single_variable_def is *cyclic* if the maximal type of the type_expr depends on the variable introduced by the single_variable_def itself.

A maximal type *depends* on a variable, v, if it refers to v or to any other variable or channel the maximal type of which depends on v.

Context-independent Expansions A multiple_variable_def is short for two or more single variable definitions. A multiple_variable_def of the form:

variable id_1, ... ,id_n : type_expr

is short for:

variable
  id_1 : type_expr,
  ⋮
  id_n : type_expr

Context Conditions In a variable_decl the constituent variable_defs must be compatible.

A single_variable_def must not be cyclic.

In an initialisation the value_expr must be pure and its maximal type must be less than or equal to the maximal type of the variable.

Attributes In a single_variable_def the maximal type of the constituent id is the maximal type of the type_expr.

The maximal definition of a single_variable_def is obtained by replacing the constituent type_expr by the corresponding maximal type expression and removing the initialisation (if any).

Meaning A single_variable_def introduces the id for a variable in which values of the maximal type of the type represented by the type_expr can be stored. The type of the variable is the type represented by the type_expr.

In addition, an initialisation expression can be given, the value of which is the initial value of the variable. The initial value is the value contained in the variable when its surrounding class expression is used to form an object and restored to it by an initialise expression (section 42.20).

If no initialisation is given, the initial value of the variable is some arbitrarily chosen value within its type.

38.7 Channel Declarations

Syntax

channel_decl ::=
  channel channel_def-list

channel_def ::=
  single_channel_def |
  multiple_channel_def

single_channel_def ::=
  opt-comment-string id : type_expr

multiple_channel_def ::=
  opt-comment-string id-list2 : type_expr

Terminology A *channel* is a medium that concurrently executing value expressions can communicate along.

In order for two value expressions to communicate along a channel, one value expression must output to the channel whilst the other value expression must input from the channel. Communication is *synchronized*: an output value expression only outputs to the channel if an input value expression simultaneously inputs from the channel.

A single_channel_def is *cyclic* if the maximal type of the type_expr depends on the channel introduced by the single_channel_def itself.

A maximal type *depends* on a channel, c, if it refers to c or to any other channel or variable the maximal type of which depends on c.

Context-independent Expansions A multiple_channel_def is short for two or more single channel definitions. A multiple_channel_def of the form:

channel id_1, ... ,id_n : type_expr

is short for:

channel
id_1 : type_expr,
⋮
id_n : type_expr

Context Conditions In a channel_decl the constituent channel_defs must be compatible.

A single_channel_def must not be cyclic.

Attributes In a single_channel_def the maximal type of the constituent id is the maximal type of the type_expr.

The maximal definition of a single_channel_def is obtained by replacing the constituent type_expr by the corresponding maximal type expression.

Meaning A single_channel_def introduces the id for a channel along which values of the type represented by the type_expr can be communicated.

38.8 Axiom Declarations

Syntax

axiom_decl ::=
axiom opt-axiom_quantification axiom_def-list

axiom_quantification ::=
forall typing-list •

axiom_def ::=
opt-comment-string opt-axiom_naming *readonly_logical*-value_expr

axiom_naming ::=
[id]

Context-independent Expansions Any axiom_decl can be expanded into an axiom declaration of the form:

axiom
opt-axiom_naming$_1$ □ value_expr$_1$,

⋮
opt-axiom_naming$_n$ □ value_expr$_n$

An axiom_decl of the form without an axiom_quantification:

axiom
opt-axiom_naming$_1$ value_expr$_1$,
⋮
opt-axiom_naming$_n$ value_expr$_n$

is short for:

axiom
opt-axiom_naming$_1$ □ value_expr$_1$,
⋮
opt-axiom_naming$_n$ □ value_expr$_n$

An axiom_quantification is short for a distributed quantification. That is, an axiom_decl of the form:

axiom forall typing-list •
opt-axiom_naming$_1$ value_expr$_1$,
⋮
opt-axiom_naming$_n$ value_expr$_n$

is short for:

axiom
opt-axiom_naming$_1$ □ ∀ typing-list • value_expr$_1$,
⋮
opt-axiom_naming$_n$ □ ∀ typing-list • value_expr$_n$

Context Conditions In an axiom_decl the constituent axiom_defs must be compatible.

In an axiom_def the value_expr must be read-only and have the maximal type **Bool**.

Meaning An axiom_def states properties of entities introduced elsewhere in terms of an axiom (the constituent Boolean value_expr).

An axiom can be given a name (the id in an axiom_naming).

CHAPTER 39

Class Expressions

39.1 General

Syntax

class_expr ::=
  basic_class_expr |
  extending_class_expr |
  hiding_class_expr |
  renaming_class_expr |
  scheme_instantiation

Terminology A *model* is an association of identifiers and operators with entities. A model *provides* an identifier or operator if it associates that identifier or operator with an entity.

A model *satisfies* a definition if it provides the identifier or operator introduced by that definition and if the entity associated with the identifier or operator has the defined kind and if the properties stated in the definition *hold* in the model.

A *class* is a collection of models.

An identifier or operator is *under-specified* if there exist at least two models in the class in which the identifier or operator is associated with different entities.

A class $class\_expr_1$ is a *subclass* of $class\_expr_2$ if all the models of $class\_expr_1$ are models of $class\_expr_2$.

Some classes are said to be *maximal*. A class expression represents a *maximal class* if and only if the definitions it stands for are all maximal.

A scheme definition is maximal if each object definition in its formal parameter (if any) is maximal and if its class expression is maximal.

An object definition is maximal if its index type expression (if any) is maximal and its class expression is maximal.

A type definition is maximal if the type identifier that it introduces represents a maximal type.

A value definition is maximal if it is a typing whose constituent type expression is maximal.

A variable definition is maximal if it is a single variable definition without initialization, whose constituent type expression is maximal.

A channel definition is maximal if it is a single channel definition whose constituent type expression is maximal.

An axiom definition is not maximal.

Attributes A class expression has an associated maximal class. The specific maximal class is defined for each kind of class expression in the relevant section.

The class represented by a class expression is a subclass of the associated maximal class.

Meaning A class_expr stands for a collection of definitions and represents the class consisting of all models that satisfy each of the definitions. Each model associates the identifiers and operators defined in the class_expr with particular entities. For each alternative we state in the relevant section which definitions the class_expr stands for.

39.2 Basic Class Expressions

Syntax

basic_class_expr ::=
  class opt-decl-string **end**

Scope and Visibility Rules The immediate scope of the opt-decl-string is the opt-decl-string itself. Note, that this means that the order of definitions in the opt-decl-string is immaterial.

Context Conditions The constituent decls must be compatible.

Attributes The maximal class of a basic class expression is represented by a class expression consisting of the maximal definitions obtained from the non-axiom definitions contained in its declarations.

Meaning A basic_class_expr stands for the definitions appearing in the decls.

39.3 Extending Class Expressions

Syntax

extending_class_expr ::=
  extend class_expr **with** class_expr

Scope and Visibility Rules The scope of the first class_expr extends to the second class_expr.

Context Conditions The constituent class_exprs must be compatible.

Attributes The maximal class is represented by an extending class expression formed by making the constituent class expressions maximal.

Meaning An extending_class_expr stands for the definitions which the class_exprs stand for.

39.4 Hiding Class Expressions

Syntax

hiding_class_expr ::=
  hide defined_item-list **in** class_expr

Scope and Visibility Rules The scope of the class_expr extends to the defined_items. Furthermore, the only definitions visible within the defined items are those of the class_expr. From this and the visibility rules it follows that all the id_or_ops occurring immediately within the defined_items must have a corresponding definition in the class_expr. These corresponding definitions are not visible outside the hiding_class_expr.

Context Conditions The id_or_ops in the constituent defined_items must be distinct unless they are disambiguated with distinguishable maximal types.

Attributes The maximal class is represented by a hiding class expression obtained by replacing any type expressions in the defined items by the corresponding maximal type expressions and by making the constituent class expression maximal.

Meaning A hiding_class_expr stands for the definitions that the class_expr stands for, except that the entities that are referred to in the defined_items cannot be referred to outside the class_expr (see the visibility rules).

39.5 Renaming Class Expressions

Syntax

renaming_class_expr ::=
  use rename_pair-list **in** class_expr

Scope and Visibility Rules The scope of the class_expr extends to the rename_pair. Furthermore, the only definitions visible within the defined items are those of the class_expr. (From this and the visibility rules it follows that the old identifiers and operators of the rename_pairs must be defined in the class_expr.)

Context Conditions All old items of the rename_pairs must be distinct. (In other words: there must not be more than one new identifier or operator for each old item.)

All new identifiers and operators in the rename_pairs must be distinct unless they are new identifiers and operators for values of distinguishable maximal types.

A new identifier or operator can only be equal to an identifier or operator of an entity which is defined in the class_expr and which does not get a new name provided both identifiers or operators represent values and they have distinguishable maximal types.

Attributes The maximal class is represented by a renaming class expression formed by replacing any type expressions in the defined items in the renaming pairs by the corresponding maximal type expressions and making its constituent class expression maximal.

Meaning A renaming_class_expr stands for the definitions that the class_expr stands for, but renamed according to the rename_pairs.

39.6 Scheme Instantiations

Syntax

scheme_instantiation ::=
scheme-name opt-actual_scheme_parameter

actual_scheme_parameter ::=
(object_expr-list)

Terminology An object expression list is a *static implementation* of a formal scheme argument list if and only if:

- The number of the object expressions is equal to the number of the formal scheme arguments.
- Each of the object expressions is a static implementation of the corresponding formal scheme argument.

An object expression is a *static implementation* of a formal scheme argument if and only if:

- The object represented by the object expression and the object defined by the formal scheme argument are either both arrays or both models.
- If they are both arrays then the maximal index type of the object expression and the maximal index type of formal array parameter in the formal scheme argument are the same.
- The maximal class of the object_expr is a static implementation of the maximal class of the class expression in the formal scheme argument (object definition).

A maximal class is a *static implementation* of another (old) maximal class if and only if:

- For each non-hidden non-axiom definition in the old class expression there is a non-hidden definition in the new class expression of the same kind statically implementing it.

A maximal type definition is a *static implementation* of another (old) maximal type definition, if and only if:

- They introduce the same identifier.
- If the old type definition is an abbreviation definition then the new one is also an abbreviation definition and the type of the constituent identifier of the new one is equal to the type of the constituent identifier of the old one with all old sorts, variables and channels replaced with their corresponding new types, variables and channels.

A maximal value definition (which is a typing) is a *static implementation* of another (old) maximal value definition, if and only if:

- They introduce the same identifier(s) or operator(s).
- The type of each constituent identifier or operator of the new one is a subtype of the type of same constituent identifier or operator of the old one with all old sorts, variables and channels replaced with their corresponding new types, variables and channels.

A maximal variable/channel definition is a *static implementation* of another (old) maximal variable/channel definition, if and only if:

- They introduce the same identifier.
- The type of the constituent identifier of the new one is equal to the type of the constituent identifier of the old one with all old sorts, variables and channels replaced with their corresponding new types, variables and channels.

A maximal object definition is a *static implementation* of another (old) maximal object definition, if and only if:

- They introduce the same identifier.
- They either both define an array or both define a model.
- If they define arrays then the index type of the formal array parameter in the new one is equal to the index type of the formal array parameter in the old one with all old sorts, variables and channels replaced with their corresponding new types, variables and channels.
- The class expression of the new object definition is a static implementation of the class expression in the old object definition.

A maximal scheme definition is a *static implementation* of another (old) maximal scheme definition, if and only if:

- They introduce the same identifier.
- They have the same number of formal scheme arguments.
- For each formal scheme argument in the old scheme definition and the corresponding formal scheme argument in the new scheme definition the following hold:
  - They either both define an array or both define a model.

  - If they define arrays then the index type of the formal array parameter in the new one is equal to the index type of the formal array parameter in the old one with all old sorts, variables and channels replaced with their corresponding new types, variables and channels.
  - The class expression of the old formal scheme argument is a static implementation of the class expression in the new formal scheme argument.
- The class expression of the new scheme definition is a static implementation of the class expression in the old scheme definition.

Context Conditions In a scheme_instantiation the name must represent a scheme.

There must be an actual_scheme_parameter present if and only if the scheme is a parameterized class, i.e. the name has a formal scheme parameter. In that case the object_expr-list in the actual_scheme_parameter must be a static implementation of the formal scheme argument list in this formal scheme parameter of the name.

The scheme definitions, abbreviation definitions, variable definitions and channel definitions introduced by the scheme_instantiation must not be cyclic and the value definitions must be compatible.

Attributes The maximal class is represented by the the maximal class expression associated with the scheme name. If there is an actual scheme parameter then occurrences of the formal parameter identifiers are bound to objects in the maximal classes of the corresponding object expressions in the actual scheme parameter.

Meaning A scheme_instantiation is either an instantiation of a named class or of a named parameterized class.

- A scheme_instantiation of a named class has the form:

  name

  The *name* has the form opt-*qualification id* and the class must have been defined by a scheme definition (section 38.2) as follows:

  scheme
  id = class_expr

  The scheme_instantiation then stands for the definitions that the *class_expr* stands for.
- A scheme_instantiation of a named parameterized class has the form:

  name(object_expr$_1$, ... ,object_expr$_n$)

  The *name* has the form opt-*qualification id* and the parameterized class must have been defined by a scheme definition (section 38.2) as follows:

  scheme
  id(
  id$_1$ opt-formal_array_parameter$_1$: class_expr$_1$, ... ,
  id$_n$ opt-formal_array_parameter$_n$: class_expr$_n$) =
  class_expr

The scheme_instantiation stands for the definitions that the *class_expr* stands for — evaluated in a model where each id_i has been bound to the object obtained by evaluating $object\_expr_i$, provided that the maximal class of $object\_expr_i$ is a static implementation of the class represented by $class\_expr_i$.

39.7 Rename Pairs

Syntax

rename_pair ::=
  defined_item **for** defined_item

Terminology If a rename_pair occurs in a fitting_object_expr then the id_or_op on the right-hand side of **for** is called a *new* identifier or operator and the id_or_op on the left-hand side of **for** is called an *old* identifier or operator. If it occurs in a renaming_class_expr then the id_or_op on the left-hand side of **for** is called a *new* identifier or operator and the id_or_op on the right-hand side of **for** is called an *old* identifier or operator.

To *rename* something according to a rename_pair means to replace all occurrences of the old identifier or operator with the new identifier or operator.

Context Conditions In a rename_pair there must not be a type_expr in the defined_item which contains a new identifier or operator.

In a rename_pair the maximal type of a new operator must be a function type which is distinguishable from the maximal type(s) of the predefined meanings of the operator. If the operator is an infix operator then the function type must have a parameter type which is a product type of length 2.

Attributes In a rename_pair the attributes of the new identifier or operator are the attributes of the old identifier or operator.

A rename_pair has an associated *old item*. This is the old identifier or operator of the rename_pair together with the maximal type of this old identifier or operator.

39.8 Defined Items

Syntax

defined_item ::=
  id_or_op |
  disambiguated_item

disambiguated_item ::=
  id_or_op : type_expr

Context Conditions In a disambiguated_item the id_or_op must represent a value and its maximal type and the maximal type of type_expr must be the same.

Meaning The type_expr within a disambiguated_item is needed when the id_or_op, due to overloading, represents several values with different maximal types. The type_expr then identifies precisely one of the values.

CHAPTER 40

Object Expressions

40.1 General

Syntax

object_expr ::=
  object-name |
  element_object_expr |
  array_object_expr |
  fitting_object_expr

Attributes An object_expr has an associated *maximal class* (such that the models given by the object belong to the class). If it represents an array, then it also has an associated maximal type, which is called the *maximal index type* of the object expression. The specific associated maximal class and maximal index type are defined for each of the alternatives in the relevant section.

Meaning An object_expr represents an object. The specific objects are defined for each of the alternatives in the relevant section.

40.2 Names

Context Conditions For an object_expr which is a name, this name must represent an object.

Attributes See chapter 46.

Meaning See chapter 46.

40.3 Element Object Expressions

Syntax

element_object_expr ::=
  array-object_expr actual_array_parameter

actual_array_parameter ::=
  [*pure*-value_expr-list]

Context-independent Expansions Any element_object_expr can be expanded into an element object expression of the form:

  object_expr[value_expr]

An element_object_expr of the form:

  object_expr[value_expr$_1$,...,value_expr$_n$]

where $n > 1$, is short for:

  object_expr[(value_expr$_1$,...,value_expr$_n$)]

Context Conditions In an element_object_expr the constituent object_expr must represent an array.

In an element_object_expr the maximal type of the actual_array_parameter must be less than or equal to the maximal index type of the object_expr.

The value_exprs in the actual_array_parameter must be pure.

Attributes The maximal class is the maximal class of the constituent object_expr. Note that in the context conditions, two applications of the same array to distinct actual array parameters are not assumed to define distinct entities.

The maximal type of an actual_array_parameter having one constituent value_expr is the maximal type of the value_expr.

Meaning An element_object_expr of the form:

  object_expr[value_expr]

represents a model obtained as follows. The *object_expr* represents an array and the *value_expr* represents a value, which must be an index value of the array. The model is obtained by applying the array to the value (i.e. the model is that model to which the value is mapped by the array).

40.4 Array Object Expressions

Syntax

array_object_expr ::=
  [| typing-list • *element*-object_expr |]

Scope and Visibility Rules The scope of the constituent typings is the object_expr.

Context Conditions The object_expr must represent a model, not an array.

Attributes The maximal index type is the maximal type of the constituent typing-list. The maximal class is the maximal class of the constituent object_expr.

Meaning An array_object_expr represents an array. The index type of the array is the type represented by the typing-list. Each index value belonging to the index type is mapped to a model. This model is the one obtained by evaluating the object_expr in scope of the definitions given by matching the index value against the binding also represented by the typing-list.

40.5 Fitting Object Expressions

Syntax

fitting_object_expr ::=
  object_expr { rename_pair-list }

Context Conditions The constituent object_expr has a maximal class represented by a *class_expression*$_1$, say. For the fitting_object_expr to be well-formed it must be possible to express *class_expression*$_1$ in the form:

  use rename_pair-list **in** class_expr$_2$

where *class_expr*$_2$ is a class expression representing a maximal class. From this and the scope and visibility rules for renaming class expressions it follows that:

- The old items of the rename_pairs must be defined in *class_expression*$_1$, the maximal class of the object_expr.
- All old items of the rename_pairs must be distinct. (In other words: there must not be more than one new identifier or operator for each old item.)
- All new identifiers and operators in the rename_pairs must be distinct unless they are new identifiers and operators for values of distinguishable maximal types.
- A new identifier or operator can only be equal to an identifier and operator of an entity which is defined in *class_expr*$_2$ and which does not get a new name if both identifiers or operators represent values and they have distinguishable maximal types.

Attributes If a fitting_object_expr represents an array the maximal index type is the maximal index type of the constituent object_expr. The maximal class is that represented by *class_expr*$_2$ as defined above.

Meaning If the meaning of the constituent object expression is a model or an array of models in the class represented by:

  use rename_pair-list **in** class_expr

then the meaning of the fitted object expression is a model or an array of models in the class represented by *class_expr*.

CHAPTER 41

Type Expressions

41.1 General

Syntax

type_expr ::=
  type_literal |
  type-name |
  product_type_expr |
  set_type_expr |
  list_type_expr |
  map_type_expr |
  function_type_expr |
  subtype_expr |
  bracketed_type_expr

Terminology A *type* is a collection of values. There are three kinds of types:

- *predefined types* are represented by literals built into the language. These types include for example the integers and the Booleans. See section 41.2,
- *abstract types* are represented by identifiers introduced in sort definitions (section 38.4.1), variant definitions (section 38.4.2), union definitions (section 38.4.3) and short record definitions (section 38.4.4).
- *compound types* are build from other types by application of a *type operator* to one or more types.

A type is said to be a *pure function* type if it may be represented by a function type expression in which the access description string is absent.

A type t_1 is a *subtype* of a type t_2 if all the values in t_1 are in t_2.

Some types are said to be *maximal*. A type is maximal if it is representable by a maximal type expression.

A type expression is *maximal* if and only if it is one of the following:

- **Unit**, **Bool**, **Int**, **Real** or **Char** (i.e. all type literals except **Nat** and **Text**).

- A name whose corresponding definition is a sort definition, a variant definition, a union definition or a short record definition.
- A name whose corresponding definition is an abbreviation definition in which the constituent type expression is maximal.
- A type expression built from maximal type expressions and static access descriptions by application of one of the type operators $\times$, -**infset**, $^{\omega}$, $\overrightarrow{m}$ or $\xrightarrow{\sim}$ (i.e. all type operators except -**set**, * and $\rightarrow$).
- A subtype expression whose constituent type expression is maximal and whose restriction holds for all values of the (maximal) type represented by that maximal type expression (i.e. reduces to **true**).
- A bracketed type expression whose constituent type expression is maximal.

Two maximal types are *indistinguishable* if and only if there exist two type expressions representing them and at most differing in the static access descriptions of constituent function types.

Two maximal types are *distinguishable* if and only if they are not indistinguishable.

The union definitions in a specification give rise to potential type coercions, which are particular functions between maximal types.

There is a *potential coercion* from a maximal type t_1 to a maximal type t_2 if any of the following conditions hold:

1. t_1 is identical with t_2; in this case the potential coercion from t_1 to t_2 is the identity function.
2. There is a union definition:

   id$_2$ = ... | opt-qualification id$_1$ | ...

   such that id_1 has t_1 as maximal type and id_2 has t_2 as maximal type; in this case the potential coercion from t_1 to t_2 is id_2*_from_*id_1.
3. There are types t_{11}, t_{12}, t_{21} and t_{22}, and a static access description opt-access_desc-string such that there are potential coercions from t_{11} to t_{21}, from t_{12} to t_{22}, and t_1 and t_2 are constructed in one of the following manners; in this case a potential coercion from t_1 to t_2 is induced from the other potential coercions in an obvious way.

   (a) t_1 is ...$\times$ t_{11} $\times$ ...and t_2 is ...$\times$ t_{21} $\times$ ...
   (b) t_1 is t_{11}-**infset** and t_2 is t_{21}-**infset**.
   (c) t_1 is ${t_{11}}^{\omega}$ and t_2 *is* ${t_{21}}^{\omega}$.
   (d) t_1 is t_{11} $\overrightarrow{m}$ t_{12} and t_2 is t_{21} $\overrightarrow{m}$ t_{22}.
   (e) t_1 is $t_{11} \xrightarrow{\sim}$ opt-access_desc-string t_{12} and
   t_2 is $t_{11} \xrightarrow{\sim}$ opt-access_desc-string t_{22}.

4. There is a type t_3 such that there are potential coercions f_{13} from t_1 to t_3 and f_{32} from t_3 to t_2; in this case a potential coercion from t_1 to t_2 is $f_{32} \circ f_{13}$.

A maximal type, t_1, is said to be *coercible* to a maximal type, t_2, if there is one and only one potential coercion from t_1 to t_2.

A maximal type, t_1, is said to be *less than or equal to* a maximal type, t_2, if it is coercible to a subtype of t_2.

A maximal type, t, is said to be an *upper bound* of a collection of maximal types if all maximal types in the collection are less than or equal to t.

A maximal type is said to be a *least upper bound* of a collection of maximal types if it is an upper bound of this collection and it is less than or equal to all other upper bounds.

Attributes A type_expr has an associated maximal type. The type represented by the type_expr is a subtype of this associated maximal type. The specific maximal types are defined for each of the alternatives in the following sections.

Meaning A type_expr represents a type. The specific types are defined for each of the alternatives in the following sections.

41.2 Type Literals

Syntax

type_literal ::=
  Unit |
  Bool |
  Int |
  Nat |
  Real |
  Text |
  Char

Attributes
  The maximal type of **Unit** is **Unit**.
  The maximal type of **Bool** is **Bool**.
  The maximal type of **Int** is **Int**.
  The maximal type of **Nat** is **Int**.
  The maximal type of **Real** is **Real**.
  The maximal type of **Text** is $\mathbf{Char}^{\omega}$.
  The maximal type of **Char** is **Char**.

Meaning
  A type_literal represents a predefined type.
  The **Unit** type has as the single value ().
  The **Bool** type has as values the Booleans **true** and **false**.
  The **Int** type has as values the integers (...,−2,−1,0,1,2,...).
  The **Nat** literal is short for the subtype expression $\{|\ \mathrm{i} : \mathbf{Int} \bullet \mathrm{i} \geq 0\ |\}$.
  The **Real** type has as values the reals (...,−4.3,...,12.23,...).
  The **Char** type has as values, the ASCII characters in single quotes ($'$a$'$,$'$b$'$,...).
  The **Text** literal is short for the type expression $\mathbf{Char}^{*}$.

41.3 Names

Context Conditions For a type_expr which is a name, the name must represent a type.

Attributes See chapter 46.

Meaning See chapter 46.

41.4 Product Type Expressions

Syntax

product_type_expr ::=
  type_expr-product2

Terminology A *product* is a value of the form (v_1,...,v_n).
The *length* of a product_type_expr is the number of constituent type_exprs.

Attributes The maximal type of a product_type_expr of the form

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

is $t_1 \times ... \times t_n$, where t_1, ..., t_n are the maximal types of *type_expr*$_1$, ..., *type_expr*$_n$.

Meaning A product_type_expr of the form *type_expr*$_1 \times ... \times$ *type_expr*$_n$ represents the type of all products of the form (v_1,...,v_n) where each v_i has the type represented by *type_expr*$_i$.

41.5 Set Type Expressions

Syntax

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**

Terminology A *set* is a possibly empty unordered collection of distinct values of the same type.

Attributes The maximal type of a set_type_expr of the form *type_expr*-**set** or *type_expr*-**infset** is t-**infset**, where t is the maximal type of *type_expr*.

Meaning A set_type_expr represents a type of subsets of the set of values of the type represented by the constituent type_expr. If the type operator is **-set**, the type contains all finite subsets. If the type operator is **-infset**, the type contains all (infinite as well as finite) subsets.

A set is characterized by its members.

41.6 List Type Expressions

Syntax

list_type_expr ::=
  finite_list_type_expr |
  infinite_list_type_expr

finite_list_type_expr ::=
  type_expr*

infinite_list_type_expr ::=
  type_expr$^\omega$

Terminology A *list* is a possibly empty sequence of values of the same type, possibly including duplicates.

Attributes The maximal type of a list_type_expr of the form *type_expr** or *type_expr*$^\omega$ is t^ω, where t is the maximal type of *type_expr*.

Meaning A list_type_expr represents a type of lists of values of the type represented by the constituent type_expr. If the type operator is *, the type contains all finite lists. If the type operator is $^\omega$, the type contains all (infinite as well as finite) lists.

A list can be applied to a value in its index set to find a corresponding element of the list.

A list is characterized by its index set and the effect of applying it to members of its index set.

41.7 Map Type Expressions

Syntax

map_type_expr ::=
  type_expr $\overrightarrow{m}$ type_expr

Terminology A *map* can be conceived of as a (possibly infinite) collection of pairs (v_1, v_2) where v_1 is a domain value, v_2 is a range value and v_1 is mapped to v_2. The *domain* of a map is the set of values, v_1, for which there exists a value, v_2, such that (v_1, v_2) is in the map. The *range* of a map is the set of values, v_2, for which there exists a value, v_1, such that (v_1, v_2) is in the map.

Attributes The maximal type of a map_type_expr of the form:

type_expr$_1$ $\overrightarrow{m}$ type_expr$_2$

is $t_1 \overrightarrow{m} t_2$, where t_1 and t_2 are the maximal types of *type_expr*$_1$ and *type_expr*$_2$.

Meaning A map_type_expr represents the type of all maps, each of which has a subset of the set of values of the type represented by the first type_expr as domain and a subset of the set of values of the type represented by the second type_expr as range.

A map can be applied to a value in its domain to find a corresponding value in the range.

A map is characterized by its domain and the effect of applying it to members of its domain.

41.8 Function Type Expressions

Syntax

function_type_expr ::=
  type_expr function_arrow result_desc

function_arrow ::=
  $\tilde{\rightarrow}$ |
  $\rightarrow$

result_desc ::=
  opt-access_desc-string type_expr

Terminology A *function* is *applied* in a state to a value of one type (the *parameter type*), whereupon it can:

- Return a value of another type (the *result type*).
- Access variables by reading from them or writing to them.
- Access channels through input from them or output to them.

Attributes The maximal type of a function_type_expr of the form:

type_expr$_1$ $\tilde{\rightarrow}$ opt-access_desc-string type_expr$_2$

or:

type_expr$_1$ $\rightarrow$ opt-access_desc-string type_expr$_2$

is $t_1 \tilde{\rightarrow}$ *oads* t_2, where t_1 and t_2 are the maximal types of type_expr$_1$ and type_expr$_2$ and *oads* is the static access description of opt-access_desc-string.

Meaning A function_type_expr represents a type of functions from the parameter type represented by the type_expr to the result type represented by the type_expr of the result_desc. The access_descs of the result_desc specify which variables and channels can be accessed when the functions are applied. Depending on the function_arrow the functions are either partial or total, as described below.

- *Partial functions*
  A function_type_expr of the form:

  type_expr$_1$ $\tilde{\rightarrow}$ opt-access_desc-string type_expr$_2$

  defines the type of partial functions from the first *type_expr* to the second *type_expr*. This type contains those functions which satisfy the following constraint: for any such function f and for any x belonging to the maximal type of the first *type_expr*, the effect of the application value expression $f(x)$ in any state is such that:

  – It only accesses the variables and channels described in the *opt-access_desc-string*.
  – If it terminates (possibly after performing sequences of communications) then the resulting value is in the maximal type of the second *type_expr*.
  – If it is convergent and x is in the type represented by the first *type_expr* then the resulting value is in the type represented by the second *type_expr*.

- *Total functions*
  A function_type_expr of the form:

  type_expr $\rightarrow$ result_desc

  is short for:

  {| f : type_expr $\tilde{\rightarrow}$ result_desc • $\forall$ x : type_expr • f(x) **post true** |}

  provided that f and x do not occur free in *type_expr* and *result_desc*.
  That is, a function_type_expr of the form:

  type_expr$_1$ $\rightarrow$ opt-access_desc-string type_expr$_2$

  defines the type of total functions from the first *type_expr* to the second *type_expr*. This type contains those functions belonging to the type represented by:

  type_expr$_1$ $\tilde{\rightarrow}$ opt-access_desc-string type_expr$_2$

  which satisfy the following constraint: for any such function f and for any x belonging to the type represented by the first *type_expr*, the effect of the application value expression $f(x)$ in any state is convergent.

41.9 Subtype Expressions

Syntax

subtype_expr ::=
  {| single_typing *pure*-restriction |}

Scope and Visibility Rules The scope of the single_typing is the restriction.

Context Conditions The restriction must be pure.

Attributes The maximal type of a subtype_expr is the maximal type of the constituent single_typing.

Meaning A subtype_expr represents a subtype of the type represented by the single_typing. The subtype contains any value that makes the restriction hold in all states — evaluated in scope of the definitions given by matching the value against the binding also represented by the single_typing.

41.10 Bracketed Type Expressions

Syntax

bracketed_type_expr ::=
  (type_expr)

Attributes The maximal type of a bracketed_type_expr is the maximal type of the constituent type_expr.

Meaning A bracketed_type_expr represents the same type as represented by the type_expr.

41.11 Access Descriptions

Syntax

access_desc ::=
  access_mode access-list

access_mode ::=
  read |
  write |
  in |
  out

access ::=
  variable_or_channel-name |
  enumerated_access |
  completed_access |
  comprehended_access

enumerated_access ::=
  { opt-access-list }

completed_access ::=
  opt-qualification **any**

comprehended_access ::=

{ access | *pure*-set_limitation }

Terminology A *static read access description* is a set of variables. A *static write access description* is a set of variables. A *static in access description* is a set of channels. A *static out access description* is a set of channels.

A *static access* is a set of variables or a set of channels.

Scope and Visibility Rules In a comprehended_access the scope of the set_limitation extends to the access.

Context Conditions For an access which is a name, the name must represent:

- A variable if it occurs in the access-list of an access_desc having **read** or **write** as access_mode.
- A channel if it occurs in the access-list of an access_desc having **in** or **out** as access_mode.

In a comprehended_access the set_limitation must be pure.

Attributes An opt-access_desc-string and an access_desc have four associated static access descriptions: a static read access description, a static write access description, a static in access description and a static out access description.

For a nil-access_desc-string the static read, write, in and out access descriptions are empty.

For an access_desc-string the static read access description is the union of the static read access descriptions of all its constituent access_descs. Similarly, for write, in and out access descriptions.

For an access_desc having access mode **read** its static read access description is equal to the union of the static accesses of its constituent accesses, while its static write, in and out access descriptions are empty. Similarly for an access_desc having access mode **in** or **out**. For an access_desc having access mode **write** its static read and write access descriptions are both equal to the union of the static accesses of its constituent accesses, while its static in and out access descriptions are empty.

An access has an associated static access. A static access is obtained from an access basically by comprehending over the maximal array types of any object arrays that are mentioned in qualifications. This is necessary because it is not possible statically to distinguish between $O[e].v$ and $O[e'].v$ for arbitrary expressions e and e'. Hence the static accesses of both these accesses is (if O is an object identifier with maximal array type T):

{ O[i].v | i : T }

The static access of a name that is an unqualified identifier is the singleton set consisting of the variable or channel the identifier represents.

The static access of a name of the form qualification id is the set of variables or channels represented by id obtained by a comprehension over the maximal index types of any object arrays mentioned in the qualification.

The static access of an enumerated_access is the empty set if it has no constituent accesses, otherwise it is the union of the static accesses of its constituent accesses.

The static access of a completed_access of the form **any** is the set of all variables or channels that are defined in the innermost enclosing class expression of the completed_access, or that could be defined in any extension of that class expression.

The static access of a completed_access of the form qualification **any** is the union of the sets of all variables or channels that are defined in the set of maximal classes of the set of objects obtained by a comprehension over the maximal index types of any of any object arrays mentioned in the qualification.

The static access of a comprehended_access is the static access of its constituent access.

Meaning The access_descs in a function type expression restrict the set of functions represented by the function type expression, by stating what variables and channels can be accessed and how they can be accessed.

An opt-access_desc-string and an access_desc represent four sets of variables and channels:

- A set of variables having **read** access_mode, which may be read from.
- A set of variables having **write** access_mode, which may be written to (that is, changed by an assignment). Variables with **write** access_mode automatically have **read** access_mode.
- A set of channels having **in** access_mode, which may be input from.
- A set of channels having **out** access_mode, which may be output to.

For a nil-access_desc-string the read, write, in and out sets are empty.

For an access_desc-string the read set is the union of the read sets represented by all its constituent access_descs. Similarly for write, in and out sets.

For an access_desc having access mode **read** its read set is equal to the union of the sets represented by its constituent accesses, while its write, in and out sets are empty. Similarly for in and out.

For an access_desc having access mode **write** its read set and write set are equal to the union of the sets represented by its constituent accesses, while its in and out sets are empty.

An access represents a set of variables or channels.

The set represented by a name is the singleton set consisting of the variable or channel it represents.

The set represented by an enumerated_access is the empty set if it has no constituent accesses, otherwise it is the union of the sets represented by its constituent accesses.

The set represented by a completed_access of the form **any** is the set of all variables or channels that are defined in the innermost enclosing class expression of the completed_access, or that could be defined in any extension of that class expression.

The set represented by a completed_access of the form qualification **any** is the set of all variables or channels provided by the model represented by the qualification.

The set represented by a comprehended_access is the following. For each model in the set of models represented by the set_limitation, the constituent access represents a particular set. The result is the union of all these sets.

CHAPTER 42

Value Expressions

42.1 General

Syntax

value_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 |
  bracketed_expr |
  infix_expr |
  prefix_expr |
  comprehended_expr |
  initialise_expr |
  assignment_expr |
  input_expr |
  output_expr |
  structured_expr

Terminology A value_expr is *evaluated* (or, synonymously, *executed*) in the scope of a collection of definitions and in a state.

Occasionally we use the terminology that a value_expr is evaluated 'in a model' (meaning a model satisfying the definitions) and in a state.

Often a value_expr is said to be evaluated without mentioning a particular collection of definitions or without mentioning a particular state. Such abbreviations only occur where they cause no confusion.

The *effect* of a value_expr on a state is obtained by evaluating the value_expr in the state, whereupon it can:

- Return a value.
- Access variables by reading from them or writing to them.
- Offer to access channels through input from them or output to them.

More formally, the effect of a value_expr on a state is one of the following:

1. Possibly after changing the state, to *terminate* by returning a value.
2. Possibly after changing the state, to *offer to communicate* by waiting for input from or output to a channel and to proceed with a further effect.
3. To allow an external choice between effects which fall into categories 1 or 2.
4. To *deadlock* by stopping (in which case any external choice between it and another effect reduces to the other effect).
5. To allow an internal choice between effects which fall into categories 3 or 4.
6. To *diverge* by continuing without terminating, without offering to communicate, and without deadlocking (in which case any external choice or internal choice between it and another effect also diverges).

An effect is *non-deterministic* if it allows an internal choice between other effects. In these circumstances, in particular, there may be more than one value that it may be said to return and more than one change to the state that it may be said to achieve.

The effect of a value_expr on a state is said to *converge* if the following conditions hold:

- The effect does not allow an internal choice between an effect and deadlock (so that, in particular, it does not deadlock or diverge).
- The effect does not allow an internal choice between an effect and a different effect which terminates without offering a communication (so, in particular, if it can terminate then it does terminate and there is only one value that it may be said to return and only one change to the state that it may be said to achieve).

If there are several constituent value_exprs in a value_expr then the order of evaluation of these will be stated; this is usually from *left to right*. This has importance when the constituent value_exprs write to variables, communicate along channels or deadlock.

A value_expr is said to *statically read (from)* the variables in its static read access description, to *statically write to* the variables in its static write access description, to *statically input from* the channels in its static in access description and to *statically output to* the channels in its static out access description.

A value_expr is said to *statically access a variable* if it statically reads from or statically writes to it.

A value_expr is said to *statically access a channel* if it statically inputs from or statically outputs to it.

A value_expr is said to be *pure* if it does not statically access any variable or channel.

A value_expr is said to be *read-only* if it does not statically write to any variable and it does not statically access any channel.

Suppose there is a unique potential coercion (see section 41.1) from a type t_1 to a type t_2, i.e. t_1 is coercible to t_2. Then if a value expression has maximal type t_1 and occurs in a context requiring it to have a maximal type t, of which t_2 is a subtype, then there is said to be an *implicit coercion* from t_1 to t_2 which is an application of the potential coercion (from t_1 to t_2) to the value expression. Note that in these circumstances t_1 is less than or equal to t, so there is an implicit coercion (which may be an application of the identity function) whenever a context condition says that the maximal type of a value expression must be less than or equal to another type. There are similar implicit coercions for patterns — see section 45.1.

Similarly, if:

- Maximal type t_1 is coercible to maximal type t_2.
- Maximal type t_3 is coercible to maximal type t_4.
- t_1 and t_3 have no least upper bound,
- t_2 and t_4 have a least upper bound t,
- There are two expressions of types t_1 and t_3 respectively in a context where they are required to have a least upper bound.

Then there are said to be implicit coercions of the two expressions from t_1 to t_2 and t_3 to t_4 respectively. This may be generalized to a collection of an arbitrary number of value expressions.

Context Conditions Implicit coercions must be unique.

Context-dependent Expansions A value expression for which there is an implicit coercion which is not the identity function is short for the value expression obtained by application of the coercion.

Attributes A value_expr has an associated maximal type (such that if the value_expr terminates then its value belongs to its maximal type).

A value_expr has also four associated static access descriptions: a static read access description, a static write access description, a static in access description and a static out access description. These access descriptions are such that the value_expr statically reads a variable if the effect of the expression potentially reads that variable, and similarly for write, input and output.

The specific maximal types and static access descriptions are defined for each of the alternatives in the following sections.

42.2 Value Literals

Syntax

value_literal ::=
  unit_literal |
  bool_literal |
  int_literal |
  real_literal |
  text_literal |
  char_literal

unit_literal ::=
  ( )

bool_literal ::=
  true |
  false

Appendix C describes the syntax for int_literal, real_literal, text_literal and char_literal.

Attributes
The maximal type of a unit_literal is **Unit**.
The maximal type of a bool_literal is **Bool**.
The maximal type of a int_literal is **Int**.
The maximal type of a real_literal is **Real**.
The maximal type of a text_literal is $\mathbf{Char}^{\omega}$.
The maximal type of a char_literal is **Char**.
A value_literal does not statically access any variables or channels.

Meaning The effect of a value_literal is to return the value represented by the literal.

A text value literal of the form $''c_1 \ldots c_n''$ is short for $\langle'c_1',\ldots,'c_n'\rangle$.

42.3 Names

Context Conditions For a value_expr which is a name, the name must represent a value or a variable.

Attributes The maximal type of a name is stated in chapter 46.

A name representing a value does not statically access any variables or channels. A name representing a variable statically reads that variable; it has no other static accesses.

Meaning See chapter 46.

42.4 Pre-names

Syntax

pre_name ::=
  variable-name`

Scope and Visibility Rules If there are any local variable definitions in the innermost enclosing post-condition then they are not visible in the name.

Context Conditions A pre_name must occur within a post-condition.
The name must represent a variable.

Attributes The maximal type of a pre_name is the maximal type of the constituent name.
A pre_name statically reads the variable that it represents; it has no other static accesses.

Meaning A pre_name occurs within a post-condition, see section 42.14. The effect of a pre_name is to return the contents in the pre-state of the variable represented by name.

42.5 Basic Expressions

Syntax

basic_expr ::=
  chaos |
  skip |
  stop |
  swap

Attributes The maximal type of **skip** is **Unit**. The maximal type of **chaos**, **stop** and **swap** can be any maximal type.
A basic_expr does not statically access any variables or channels.

Meaning

- The effect of **chaos** is to diverge.
- The effect of **skip** is to return the unit value of type **Unit**.
- The effect of **stop** is to deadlock.
- The effect of **swap** is not specified; it may be to terminate, to deadlock, to allow an internal choice or to diverge.

42.6 Product Expressions

Syntax

product_expr ::=
  (value_expr-list2)

Attributes The maximal type of a product_expr of the form:

(value_expr$_1$,...,value_expr$_n$)

is $t_1 \times ... \times t_n$, where t_1, ..., t_n are the maximal types of *value_expr*$_1$, ..., *value_expr*$_n$.

A product_expr statically accesses the variables and channels which the constituent value_exprs statically access.

Meaning The effect of a product_expr of the form (*value_expr*$_1$,...,*value_expr*$_n$) is obtained by evaluating, from left to right, each *value_expr*$_i$ to give a value v_i, and then to return the product value (v_1,...,v_n).

42.7 Set Expressions

Syntax

set_expr ::=
  ranged_set_expr |
  enumerated_set_expr |
  comprehended_set_expr

42.7.1 Ranged Set Expressions

Syntax

ranged_set_expr ::=
  { *readonly_integer*-value_expr .. *readonly_integer*-value_expr }

Context Conditions The constituent value_exprs must be read-only and must have maximal type **Int**.

Attributes The maximal type of a ranged_set_expr is **Int-infset**.

A ranged_set_expr statically reads the variables which the constituent value_exprs statically read; it has no other static accesses.

Meaning The effect of a ranged_set_expr is to return a set of integers in a range delimited by a lower bound and an upper bound.

The first value_expr is evaluated to return the lower bound i_1 and the second value_expr is then evaluated to return the upper bound i_2. The set contains all integers i such that $i_1 \leq i \leq i_2$. If $i_1 > i_2$ then the set is empty.

42.7.2 Enumerated Set Expressions

Syntax

enumerated_set_expr ::=
  { *readonly*-opt-value_expr-list }

Context Conditions The constituent value_exprs must be read-only.

The maximal types of the constituent value_exprs must have a least upper bound.

Attributes The maximal type of an enumerated_set_expr having one or more constituent value_exprs is *t*-**infset**, where t is the least upper bound of the maximal types of the constituent value_exprs. The maximal type of an enumerated_set_expr having no constituent value_exprs (an empty set) is *t*-**infset**, where t is a type variable representing an arbitrary maximal type.

An enumerated_set_expr statically reads the variables which the constituent value_exprs statically read; it has no other static accesses.

Meaning The effect of an enumerated_set_expr is to return a set of explicitly specified values.

The effect of an enumerated_set_expr of the form {*value_expr*$_1$,...,*value_expr*$_n$} is obtained by evaluating, from left to right, each *value_expr*$_i$ to give a value v_i, and then to return the set value $\{v_1,...,v_n\}$.

If value_expr-list is absent, the empty set is returned.

42.7.3 Comprehended Set Expressions

Syntax

comprehended_set_expr ::=
{ *readonly*-value_expr | set_limitation }

set_limitation ::=
typing-list opt-restriction

restriction ::=
• *readonly_logical*-value_expr

Context-independent Expansions A nil-restriction is short for the restriction '• **true**'.

A restriction '• value_expr' is short for the restriction '• value_expr ≡ **true**'.

Scope and Visibility Rules In a comprehended_set_expr the scope of the set_limitation extends to the constituent value_expr.

In a set_limitation the immediate scope of the typings is the constituent restriction.

Context Conditions In a comprehended_set_expr the constituent value_expr must be read-only.

In a restriction the constituent value_expr must be read-only and must have the maximal type **Bool**.

Attributes The maximal type of a comprehended_set_expr is *t*-**infset**, where t is the maximal type of the constituent value_expr.

A comprehended_set_expr statically accesses the variables and channels which the constituent value_expr and set_limitation statically access.

A set_limitation in which a restriction is present statically accesses the variables and channels which the constituent restriction statically accesses.

A restriction statically reads the variables which the constituent value_expr statically reads; it has no other static accesses.

Meaning The effect of a comprehended_set_expr is to return a set, the elements of which are obtained by evaluating the value_expr in all those models that satisfy a certain restriction.

For each model in the set of models represented by the set_limitation (see below), the value_expr is evaluated. A comprehended_set_expr is convergent.

A set_limitation represents a subset of the models that satisfy the definitions represented by the typing-list: those that make the restriction hold.

A restriction holds if the constituent value-expr is convergent and returns the value **true**.

42.8 List Expressions

Syntax

list_expr ::=
  ranged_list_expr |
  enumerated_list_expr |
  comprehended_list_expr

42.8.1 Ranged List Expressions

Syntax

ranged_list_expr ::=
  ⟨ *integer*-value_expr .. *integer*-value_expr ⟩

Context Conditions The constituent value_exprs must have maximal type **Int**.

Attributes The maximal type of a ranged_list_expr is $\mathbf{Int}^{\omega}$.

A ranged_list_expr statically accesses the variables and channels which the constituent value_exprs statically access.

Meaning The effect of a ranged_list_expr is to return a list of integers in a range delimited by a lower bound and an upper bound.

The first value_expr is evaluated to return the lower bound i_1 and the second value_expr is then evaluated to return the upper bound i_2. The list contains all integers i such that $i_1 \leq i \leq i_2$, in increasing order. If $i_1 > i_2$ then the list is empty.

42.8.2 Enumerated List Expressions

Syntax

enumerated_list_expr ::=
  ⟨ opt-value_expr-list ⟩

Context Conditions The maximal types of the constituent value_exprs must have a least upper bound.

Attributes The maximal type of an enumerated_list_expr having one or more constituent value_exprs is t^{ω}, where t is the least upper bound of the maximal types of the constituent value_exprs.

The maximal type of an enumerated_list_expr having no constituent value_exprs (an empty list) is t^{ω}, where t is a type variable representing an arbitrary maximal type.

An enumerated_list_expr statically accesses the variables and channels which the constituent value_exprs statically access.

Meaning The effect of an enumerated_list_expr is to return a list of explicitly specified values.

The effect of an enumerated_list_expr of the form $\langle value\_expr_1,...,value\_expr_n \rangle$ is obtained by evaluating, from left to right, each $value\_expr_i$ to give a value v_i, and then to return the list value $\langle v_1,...,v_n \rangle$.

If value_expr-list is absent, the empty list is returned.

42.8.3 Comprehended List Expressions

Syntax

comprehended_list_expr ::=
⟨ value_expr | list_limitation ⟩

list_limitation ::=
binding **in** *readonly_list*-value_expr opt-restriction

Scope and Visibility Rules In a comprehended_list_expr the scope of the list_limitation extends to the constituent value_expr.

In a list_limitation the immediate scope of the binding is the constituent restriction.

Context Conditions In a list_limitation the constituent value_expr must be read-only and must have a maximal type which is a list type.

Attributes The maximal type of a comprehended_list_expr is t^{ω}, where t is the maximal type of the constituent value_expr.

In a list_limitation the maximal context type of the constituent binding is t, where t^{ω} is the maximal type of the constituent value_expr.

A comprehended_list_expr statically accesses the variables and channels which the constituent value_expr and list_limitation statically access.

A list_limitation statically reads the variables which the constituent value_expr statically reads. If a restriction is present it also accesses the variables and channels that the restriction statically accesses. It has no other static accesses.

Meaning The effect of a comprehended_list_expr is to return a list generated on the basis of another list.

For each model in the list of models represented by the list_limitation (see below) the value_expr is evaluated, and the returned value is included in the result list at the corresponding position. The list of models represented by the list_limitation is processed from left to right.

A list_limitation evaluates to a list of models as follows. The value_expr in the list_limitation returns a list. Each element in the list, processed from left to right, is then matched against the binding to give a collection of definitions. If the model (there is precisely one) satisfying these definitions makes the restriction hold, the model is included in the resulting list of models at the corresponding position.

42.9 Map Expressions

Syntax

map_expr ::=
  enumerated_map_expr |
  comprehended_map_expr

42.9.1 Enumerated Map Expression

Syntax

enumerated_map_expr ::=
  [opt-value_expr_pair-list]

value_expr_pair ::=
  readonly-value_expr $\mapsto$ *readonly*-value_expr

Context Conditions In an enumerated_map_expr the maximal domain types of the the constituent value_expr_pairs must have a least upper bound and the maximal range types of the the constituent value_expr_pairs must have a least upper bound.

In a value_expr_pair the value_exprs must be read-only.

Attributes The maximal type of an enumerated_map_expr having one or more constituent value_expr_pairs is $t_1 \xrightarrow[m]{} t_2$, where t_1 is the least upper bound of the maximal domain types and t_2 is the least upper bound of the maximal range types of the constituent value_expr_pairs.

The maximal type of an enumerated_map_expr having no constituent value_exprs (an empty map) is $t_1 \xrightarrow[m]{} t_2$, where t_1 and t_2 are type variables representing arbitrary maximal types.

The maximal domain type and the maximal range type of a value_expr_pair are the maximal types of the first and second constituent value_exprs respectively.

An enumerated_map_expr statically accesses the variables and channels which the constituent value_expr_pairs statically access.

A value_expr_pair statically reads the variables which the constituent value_exprs statically read; it has no other static accesses.

Meaning The effect of an enumerated_map_expr is to return a map of explicitly specified pairs.

The effect of an enumerated_map_expr of the form

[value_expr_pair$_1$,...,value_expr_pair$_n$]

is obtained by evaluating, from left to right, each *value_expr_pair*$_i$ to give a value pair (v_1,v_2), and then to return the map value containing all these pairs.

The effect of a value_expr_pair is obtained by evaluating the first value_expr to give a value v_1 and then evaluating the second value_expr to give a value v_2, and then to return the pair (v_1,v_2).

If the value_expr_pair-list is absent, the empty map is returned.

42.9.2 Comprehended Map Expressions

Syntax

comprehended_map_expr ::=
  [value_expr_pair | set_limitation]

Scope and Visibility Rules In a comprehended_map_expr the scope of the set_limitation extends to the constituent value_expr_pair.

Attributes The maximal type of a comprehended_map_expr is $t_1 \xrightarrow[m]{} t_2$, where t_1 is the maximal domain type and t_2 is the maximal range type of the constituent value_expr_pair.

A comprehended_map_expr statically accesses the variables and channels which the constituent value_expr_pair and set_limitation statically access.

Meaning The effect of a comprehended_map_expr is to return a map, the pairs of which are obtained by evaluating the value_expr_pair in all those models that satisfy a certain restriction.

For each model in the set of models represented by the set_limitation, the value_expr_pair is evaluated. If the value_expr_pair is convergent, the resulting value pair is included in the map. In the case of non-convergence, the particular evaluation does not contribute a pair. A comprehended_map_expr is convergent.

42.10 Function Expressions

Syntax

function_expr ::=
  λ lambda_parameter • value_expr

lambda_parameter ::=
  lambda_typing |

single_typing

lambda_typing ::=
(opt-typing-list)

Context-independent Expansions A function_expr where the lambda_parameter is a lambda_typing is short for a function expression of the form:

λ single_typing • value_expr

A function_expr of the form:

λ () • value_expr

is short for:

λ id : **Unit** • value_expr

where *id* is some identifier not already in scope.

A function_expr of the form:

λ (typing-list) • value_expr

is short for:

λ single_typing • value_expr

where *typing*-list is short for *single_typing* (see chapter 44).

Scope and Visibility Rules In a function_expr the scope of the lambda_parameter is value_expr.

Attributes The maximal type of a function_expr in which the lambda_parameter is a single_typing is $t_1 \xrightarrow{\sim} oads\ t_2$, where t_1 is the maximal type of the single_typing, t_2 is the maximal type of the value_expr and *oads* is a description of variables and channels the value_expr statically accesses.

A function_expr does not statically access any variables or channels.

Meaning The effect of a function_expr of the form:

λ single_typing • value_expr

is to return a function, say f, defined as follows.

The *single_typing* represents a (parameter) type and a (parameter) binding as described in chapter 44. The effect of applying the function f to a value v within the parameter type is the effect of evaluating the *value_expr* in scope of the definitions given by matching v against the parameter binding.

42.11 Application Expressions

Syntax

application_expr ::=
list_or_map_or_function-value_expr actual_function_parameter-string

actual_function_parameter ::=
  (opt-value_expr-list)

Context-independent Expansions Any application_expr can be expanded into an application expression of the form:

  value_expr$_1$(value_expr$_2$)

An application_expr of the form:

  value_expr actual_function_parameter$_1$ ...actual_function_parameter$_n$

where $n > 1$, is short for:

  (...(value_expr actual_function_parameter$_1$)...) actual_function_parameter$_n$

An application_expr of the form:

  value_expr()

is short for:

  value_expr(())

An application_expr of the form:

  value_expr(value_expr$_1$,...,value_expr$_n$)

where $n > 1$, is short for:

  value_expr((value_expr$_1$,...,value_expr$_n$))

Context Conditions In an application_expr having only one actual_function_parameter the maximal type of the value_expr must be a list type, a map type or a function type. Furthermore, if the maximal type of the value_expr is:

- A list type, t^ω, then the maximal type of the actual_function_parameter must be **Int**.
- A map type, $t_1 \xrightarrow[m]{} t_2$, then the maximal type of the actual_function_parameter must be less than or equal to the type t_1.
- A function type, $t_1 \xrightarrow{\sim} oads\ t_2$, then the maximal type of the actual_function_parameter must be less than or equal to the type t_1.

Attributes The maximal type of an application_expr having only one actual_function_parameter is determined by the maximal type of the constituent value_expr. If this is:

- A list type, t^ω, then it is the element type, t.
- A map type, $t_1 \xrightarrow[m]{} t_2$, then it is the range type t_2.
- A function type, $t_1 \xrightarrow{\sim} oads\ t_2$, then it is the result type, t_2.

The maximal type of an actual_function_parameter having one constituent value_expr is the maximal type of the constituent value_expr.

An application_expr having only one actual_function_parameter statically accesses the variables and channels which the constituent value_expr and actual_function_par-

ameter statically access and, if the value_expr has a maximal type which is a function type, $t_1 \xrightarrow{\sim} oads\ t_2$, then also the variables and channels which the function body statically accesses as described in *oads*.

An actual_function_parameter having one constituent value_expr statically accesses the variables and channels which the constituent value_expr accesses.

Meaning The effect of an application_expr is obtained by applying an applicable value, which is a list, map or function, to an actual parameter.

The effect of an application_expr of the form:

value_expr$_1$(value_expr$_2$)

is obtained by evaluating *value_expr*$_2$ to give an actual parameter and then evaluating the *value_expr*$_1$ to give an applicable value, and finally applying the applicable value to the actual parameter.

The application of the applicable value to the actual parameter is done as follows:

- If the applicable value is a list then the actual parameter must be in the index set of the list (the set of integers between one and the length of the list). In that case, the list element at that position becomes the value returned. Otherwise it is under-specified.
- If the applicable value is a map then the actual parameter must be in the domain of the map. In that case, the returned value is the one mapped to by the actual parameter if there is precisely one, or a non-deterministic choice between the values mapped to by the actual parameter if there are more than one. If the actual parameter is not in the map then the result of the application is under-specified.
- If the applicable value is a function then the function it represents is applied to the actual parameter.

42.12 Quantified Expressions

Syntax

quantified_expr ::=
  quantifier typing-list restriction

quantifier ::=
  ∀ |
  ∃ |
  ∃!

Scope and Visibility Rules In a quantified_expr the scope of the constituent typings is the restriction.

Attributes The maximal type of a quantified_expr is **Bool**.

A quantified_expr statically accesses the variables and channels which the constituent restriction statically accesses.

Meaning The effect of a quantified_expr is to return a Boolean value depending on the value returned by a predicate for each model in a set of models.

The models concerned are all those that satisfy the definitions represented by the typing-list.

If the quantifier is ∀, the returned value is **true** if and only if the restriction holds for all the models.

If the quantifier is ∃, the returned value is **true** if and only if the restriction holds for at least one of the models.

If the quantifier is ∃!, the returned value is **true** if and only if the restriction holds for exactly one of the models.

A quantified_expr is convergent.

42.13 Equivalence Expressions

Syntax

equivalence_expr ::=
  value_expr ≡ value_expr opt-pre_condition

pre_condition ::=
  pre *readonly_logical*-value_expr

Context Conditions The maximal types of the the constituent value_exprs must have a least upper bound.

In a pre_condition the value_expr must be read-only and must have the maximal type **Bool**.

Context-dependent Expansions An equivalence_expr of the form:

$$\text{value\_expr}_1 \equiv \text{value\_expr}_2 \textbf{ pre } \text{value\_expr}_3$$

is short for:

$$(\text{value\_expr}_3 \equiv \textbf{true}) \Rightarrow \text{value\_expr}_1 \equiv \text{value\_expr}_2$$

Attributes The maximal type of an equivalence_expr is **Bool**.

A pre_condition statically reads the variables which the constituent value_expr statically reads. It does not statically access any channels.

An equivalence_expr statically reads the variables which the constituent value_exprs statically read or write. It also statically reads the variables that the pre_condition (if any) statically reads. It does not statically access any channels.

Meaning The effect of an equivalence_expr is to return a Boolean value that depends on whether the two value_exprs give the same effect, when each is evaluated in the current state. The effects of the value_exprs are only used to determine this Boolean value and are ignored thereafter.

The value returned by an equivalence_expr of the form:

$$\text{value\_expr}_1 \equiv \text{value\_expr}_2$$

is **true** if and only if *value_expr*$_1$ evaluated in the current state represents exactly the same effect as *value_expr*$_2$ evaluated in the same state. That is, the two *value_expr*s must represent the same effect in respect of returned values, state changes, offers to communicate, external choices, deadlocks, internal choices and divergences.

An equivalence_expr is convergent.

42.14 Post-expressions

Syntax

post_expr ::=
  value_expr post_condition opt-pre_condition

post_condition ::=
  opt-result_naming **post** *readonly_logical*-value_expr

result_naming ::=
  as binding

Scope and Visibility Rules In a post_condition the scope of the opt-result_naming is the value_expr.

Context Conditions In a post_condition the value_expr must be read-only and must have the maximal type **Bool**.

Context-dependent Expansions A post_expr of the form:

value_expr$_1$ **post** value_expr$_2$

is short for:

value_expr$_1$ **as** id **post** value_expr$_2$

where *id* is some identifier not already in scope.

A post_expr of the form:

value_expr$_1$ post_condition **pre** value_expr$_2$

is short for:

(value_expr$_2$ $\equiv$ **true**) $\Rightarrow$ value_expr$_1$ post_condition

Attributes The maximal type of a post_expr is **Bool**.

The context of a post_condition determines a maximal context type for the post_condition.

In a post_expr the maximal context type for the post_condition is the maximal type of the constituent value_expr.

In a result_naming the maximal context type of the binding is the maximal context type of the innermost enclosing post condition.

A post_expr statically reads the variables which the constituent value_expr statically reads or writes. It also statically reads the variables that the pre_condition (if any) statically reads. It does not statically access any channels.

Meaning The effect of a post_expr is to return a Boolean value that depends on the effect of value_expr when evaluated in the current state, the 'pre-state'. The effect of value_expr is only used to determine this Boolean value and is ignored thereafter.

The value returned by a post_expr of the form:

value_expr$_1$ **as** binding **post** value_expr$_2$

is **true** if and only if:

- *value_expr*$_1$ is convergent.
- *value_expr*$_2$ is convergent and evaluates to **true** in the post-state resulting from the evaluation of *value_expr*$_1$ in the pre-state and in the scope of the definitions given by matching the value returned by *value_expr*$_1$ against the *binding*.

Within *value_expr*$_2$, values of variables in the pre-state can be referred to by suffixing the variable names with a hook (pre_name). Variables of the post-state are accessed through their normal (un-hooked) names.

A post_expr is convergent.

42.15 Disambiguation Expressions

Syntax

disambiguation_expr ::=
  value_expr : type_expr

Context Conditions The maximal type of the value_expr must be less than or equal to the maximal type of the type_expr.

Attributes The maximal type of a disambiguation_expr is the maximal type of the type_expr. A disambiguation_expr statically accesses the variables and channels which the constituent value_expr statically accesses.

Meaning The effect of a disambiguation_expr is the effect of the value_expr.

42.16 Bracketed Expressions

Syntax

bracketed_expr ::=
  (value_expr)

Attributes The maximal type of a bracketed_expr is the maximal type of the value_expr.

A bracketed_expr statically accesses the variables and channels which the constituent value_expr statically accesses.

Meaning The effect of a bracketed_expr is the effect of the value_expr.

42.17 Infix Expressions

Syntax

infix_expr ::=
  stmt_infix_expr |
  axiom_infix_expr |
  value_infix_expr

42.17.1 Statement Infix Expressions

Syntax

stmt_infix_expr ::=
  value_expr infix_combinator value_expr

Context Conditions For the infix_combinator which is:
  [] or ⫿: the maximal types of the two value_exprs must have a least upper bound.
  ‖ or ⫴: the two value_exprs must have the maximal type **Unit**.
  ; : the first value_expr must have the maximal type **Unit**.

Attributes For the infix_combinator which is:
  [] or ⫿: the maximal type of the stmt_infix_expr is the least upper bound of the maximal types of the constituent value_exprs.
  ‖, ⫴: the maximal type of the stmt_infix_expr is **Unit**.
  ; : the maximal type of the stmt_infix_expr is the maximal type of the second constituent value_expr.
  A statement_infix_expr statically accesses the variables and channels which the two constituent value_exprs statically access.

Meaning See the definition of infix_combinators (chapter 49).

42.17.2 Axiom Infix Expressions

Syntax

axiom_infix_expr ::=
  logical-value_expr infix_connective *logical*-value_expr

Context Conditions The two value_exprs must have the maximal type **Bool**.

Context-dependent Expansions See section 48.1 on infix_connectives.

42.17.3 Value Infix Expressions

Syntax

value_infix_expr ::=
  value_expr infix_op value_expr

Context Conditions The type $t_1 \times t_2$, where t_1 and t_2 are the maximal types of the two value_exprs, must be less than or equal to the parameter type part of the maximal type of the infix_op.

Attributes The maximal type of a value_infix_expr is the result type part of the maximal type of the infix_op.

A value_infix_expr statically accesses the variables and channels which the constituent value_exprs statically access and the variables and channels which the function body statically accesses, as described in the static access descriptions of the maximal type of the infix_op.

Meaning The effect of a value_infix_expr is obtained by evaluating the first value_expr to give a value v_1, and then evaluating the second value_expr to give a value v_2, and then to return the result of applying the function determined by the infix_op to the pair (v_1, v_2).

42.18 Prefix Expressions

Syntax

prefix_expr ::=
  axiom_prefix_expr |
  universal_prefix_expr |
  value_prefix_expr

42.18.1 Axiom Prefix Expressions

Syntax

axiom_prefix_expr ::=
  prefix_connective *logical*-value_expr

Context-independent Expansions See section 48.2 on prefix_connectives.

42.18.2 Universal Prefix Expressions

Syntax

universal_prefix_expr ::=
  □ *readonly_logical*-value_expr

Context-independent Expansions A universal_prefix_expr:

  □ value_expr

is equivalent to:

  □ (value_expr ≡ **true**)

Context Conditions The value_expr must have the maximal type **Bool**.

The constituent value_expr must be read-only.

Attributes The maximal type of a universal_prefix_expr is **Bool**.

A universal_prefix_expr does not statically access any variables or channels.

Meaning A universal_prefix_expr of the form:

□ value_expr

gives **true** if and only if for all states in which the values of variables are within the types of the variables, the value_expr is convergent and gives the value **true**.

The universal_prefix_expr itself is convergent.

42.18.3 Value Prefix Expressions

Syntax

value_prefix_expr ::=
  prefix_op value_expr

Context Conditions The maximal type of the value_expr must be less than or equal to the parameter part of the maximal type of the prefix_op.

Attributes The maximal type of a value_prefix_expr is the result type part of the maximal type of the prefix_op.

A value_prefix_expr statically accesses the variables and channels which the constituent value_expr statically accesses and the variables and channels which the function body statically accesses as described in the static access descriptions of the maximal type of the prefix_op.

Meaning The effect of a value_prefix_expr is to return the value obtained by applying the function determined by the prefix_op to the value returned by the value_expr.

42.19 Comprehended Expressions

Syntax

comprehended_expr ::=
  associative_commutative-infix_combinator { value_expr | set_limitation }

Scope and Visibility Rules In a comprehended_expr the scope of the set_limitation extends to the value_expr.

Context Conditions The infix_combinator must be associative and commutative, that is, it must be one of the following: ||, [], ⌈⌉.

For the infix_combinator || the value_expr must have the maximal type **Unit**.

Attributes The maximal type of a comprehended_expr is the maximal type of the value_expr.

A comprehended_expr statically accesses the variables and channels which the constituent value_expr and set_limitation statically access.

Meaning The effect of a comprehended_expr is obtained by applying an infix_combinator to a set of value_exprs instead of to just two value_exprs. This has a straightforward explanation, since the infix_combinators possible here are all commutative and associative.

The set contains a value_expr for each model in the set of models represented by the set_limitation. The value_expr is evaluated in that model and in the current state. The effect of evaluating the comprehended_expr is the effect of allowing an external choice between the effects of evaluating the value_expr in these models (in the case of $[]$), allowing an internal choice between the effects of evaluating the value_expr in these models (in the case of $\sqcap$) and evaluating value_expr concurrently in these models (in the case of $\|$).

If the set contains a single value_expr, the comprehended_expr represents the same effect as the value_expr. If the set is empty:

$[]\ \{\} \equiv$ **stop**
$\sqcap\ \{\} \equiv$ **swap**
$\|\ \{\} \equiv$ **skip**

42.20 Initialise Expressions

Syntax

initialise_expr ::=
  opt-qualification **initialise**

Attributes The maximal type of an initialise_expr is **Unit**.

An initialise_expr statically writes to all the variables initialised by it. It does not statically access any channels.

Meaning The effect of an initialise_expr is to assign to variables their initial values. The value returned by the initialise_expr is the unit value. The initial value of a variable may be given explicitly in the definition of the variable; if it is not given explicitly there, none the less there is a particular value which is always assigned to the variable by an initialise_expr.

If the qualification is absent, all variables determined by the access '**any**' are initialised (see section 41.11).

If the qualification is present, it represents a model. All the variables in that model are then initialised. That is, all the variables determined by the access 'qualification **any**' are initialised (see section 41.11).

42.21 Assignment Expressions

Syntax

assignment_expr ::=
  variable-name := value_expr

Context Conditions The name must represent a variable.

The maximal type of the value_expr must be less than or equal to the maximal type of the name.

Attributes The maximal type of an assignment_expr is **Unit**.

An assignment_expr statically writes to the variable represented by the constituent name and also statically accesses the variables or channels which the constituent value_expr statically accesses.

Meaning The effect of an assignment_expr is to write (or, synonymously, assign) the value of the value_expr to the variable represented by the name. The value returned by the assignment_expr is the unit value.

42.22 Input Expressions

Syntax

input_expr ::=
channel-name ?

Context Conditions The name must represent a channel.

Attributes The maximal type of an input_expr is the maximal type of the constituent name.

An input_expr statically inputs from the channel represented by the constituent name. It does not statically access any variables.

Meaning The effect of an input_expr is to offer to input from the channel represented by the name. The value returned by the input_expr is the value input from the channel, in the event of the offer to input being matched by a concurrent offer to output to the same channel.

42.23 Output Expressions

Syntax

output_expr ::=
channel-name ! value_expr

Context Conditions The name must represent a channel.

The maximal type of the value_expr must be less than or equal to the maximal type of the name.

Attributes The maximal type of an output_expr is **Unit**.

An output_expr statically outputs to the channel represented by the constituent name and also statically accesses the variables or channels which the constituent value_expr statically accesses.

Meaning The effect of an output_expr is to offer to output to the channel represented by the name. The value offered is the value returned by the value_expr. The value returned by the output_expr is the unit value, in the event of the offer to output being matched by a concurrent offer to input from the same channel.

42.24 Structured Expressions

Syntax

structured_expr ::=
  local_expr |
  let_expr |
  if_expr |
  case_expr |
  while_expr |
  until_expr |
  for_expr

42.24.1 Local Expressions

Syntax

local_expr ::=
  local opt-decl-string **in** value_expr **end**

Scope and Visibility Rules The scope of the opt-decl-string is opt-decl-string itself and the value_expr. Note that this means that the order of the definitions in the opt-decl-string is immaterial.

Context Conditions The constituent decls must be compatible.

The maximal type of the value_expr must not involve sorts, variables or channels defined in the constituent decls.

Attributes The maximal type of a local_expr is the maximal type of the value_expr.

A local_expr statically accesses the non-local variables and channels (i.e. variables and channels not defined in the opt-decl-string) which the value_expr statically accesses.

Meaning The effect of a local_expr is the effect of the value_expr evaluated in scope of the definitions that the decls stand for. The value_expr is evaluated in each of the models satisfying the definitions and a non-deterministic choice is made between the resulting effects.

At the end of the evaluation of the value_expr any outstanding communication offered along a channel declared in the opt-decl-string is concealed by being replaced by **stop**.

42.24.2 Let Expressions

Syntax

let_expr ::=
  let let_def-list **in** value_expr **end**

let_def ::=
  typing |
  explicit_let |
  implicit_let

explicit_let ::=
  let_binding = value_expr

implicit_let ::=
  single_typing restriction

let_binding ::=
  binding |
  record_pattern |
  list_pattern

Context-independent Expansions A let_expr involving more than one let_def is short for a number of nested let expressions with single let definitions. That is, a let_expr of the form:

let let_def$_1$, ... ,let_def$_n$ **in** value_expr **end**

is short for:

let let_def$_1$ **in**
  ⋮
    let let_def$_n$ **in** value_expr **end**
  ⋮
end

Scope and Visibility Rules In a let_expr of the form:

let let_def **in** value_expr **end**

the scope of *let_def* is *value_expr*.

Attributes The maximal type of a let_expr is the maximal type of the value_expr.

In an explicit_let the maximal context type of the let_binding is the maximal type of the value_expr.

A let_expr statically accesses the variables and channels which the constituent let_defs and value_expr statically access.

A typing does not statically access any variables or channels.

An explicit_let statically accesses the variables and channels which the constituent value_expr statically accesses.

An implicit_let statically accesses the variables and channels which the constituent restriction statically accesses.

Meaning A let_expr — with only a single let_def — of the form:

let let_def **in** value_expr **end**

is evaluated as follows. The *let_def* represents a set of models as described below. The *value_expr* is evaluated in each of these models and a non-deterministic choice is made between the resulting effects.
There are three kinds of let_defs:

- A let_def of the form of a typing represents the set of models that satisfy the definitions represented by the typing.
- A let_def of the form of an implicit_let represents a subset of the models that satisfy the definitions represented by the single_typing: those in which the restriction holds.
- A let_def of the form of an explicit_let represents a set of models obtained as follows. The value_expr is evaluated to return a value which is then matched against the let_binding. If the value matches the let_binding, the result is a collection of definitions and the model set contains the single model that satisfies these. If, on the other hand, the value does not match the let_binding, the model set is empty.

42.24.3 If Expressions

Syntax

if_expr ::=
  if *logical*-value_expr **then**
    value_expr
  opt-elsif_branch-string
  opt-else_branch
  end

elsif_branch ::=
  elsif *logical*-value_expr **then** value_expr

else_branch ::=
  else value_expr

Context-independent Expansions An if_expr involving elsif_branches is short for a number of nested if_exprs without elsif_branches. An if_expr of the form:

if value_expr_1 **then** $\text{value\_expr}_1'$
elsif value_expr_2 **then** $\text{value\_expr}_2'$

⋮
elsif value_expr$_n$ **then** value_expr$_n$′
opt-else_branch
end

is short for:

if value_expr$_1$ **then** value_expr$_1$′ **else**
if value_expr$_2$ **then** value_expr$_2$′ **else**
⋮
if value_expr$_n$ **then** value_expr$_n$′ opt-else_branch **end**
⋮
end
end

An if_expr of the form:

if value_expr$_1$ **then** value_expr$_2$ **end**

is short for:

if value_expr$_1$ **then** value_expr$_2$ **else skip end**

Context Conditions In an if_expr of the form:

if value_expr$_1$ **then** value_expr$_2$ **else** value_expr$_3$ **end**

value_expr$_1$ must have the maximal type **Bool** and the maximal types of *value_expr*$_2$ and *value_expr*$_3$ must have a least upper bound.

Attributes The attributes for an if_expr of the form:

if value_expr$_1$ **then** value_expr$_2$ **else** value_expr$_3$ **end**

are as follows.

The maximal type is the least upper bound of the maximal types of *value_expr*$_2$ and *value_expr*$_3$.

The if_expr statically accesses the variables and channels which *value_expr*$_1$, *value_expr*$_2$ and *value_expr*$_3$ statically access.

Meaning The effect of an if_expr is to determine the applicable alternative followed by the effect of that alternative. An if_expr of the form:

if value_expr$_1$ **then** value_expr$_2$ **else** value_expr$_3$ **end**

is evaluated by evaluating *value_expr*$_1$ to return a Boolean value — the test value. If the test value is equal to **true**, *value_expr*$_2$ is evaluated. Alternatively, if the test value is equal to **false**, *value_expr*$_3$ is evaluated.

42.24.4 Case Expressions

Syntax

case_expr ::=
  case value_expr **of** case_branch-list **end**

case_branch ::=
  pattern → value_expr

Scope and Visibility Rules In a case_branch the scope of the pattern is the value_expr.

Context Conditions In a case_expr the maximal types of the value_exprs in the constituent case_branches must have a least upper bound.

Attributes The maximal type of a case_expr is the least upper bound of the maximal types of the value_exprs in the constituent case_branches.

In a case_expr the maximal context type of the patterns in the case_branches is the maximal type of the value_expr.

A case_expr statically accesses the variables and channels which the constituent value_expr and case_branches statically access.

A case_branch statically accesses the variables and channels which the constituent value_expr statically accesses.

Meaning The effect of a case_expr is to evaluate the value_expr, determine the matching case_branch and then to evaluate the value_expr part of that case_branch.

The value_expr is evaluated to return a value — the test value. Then the case_branches are processed from left to right until the test value succeeds in matching a pattern. The successful pattern matching then results in a collection of definitions. The corresponding value_expr in the matching case_branch is then evaluated in the scope of these definitions.

If there is no matching case_branch, the effect of the case_expr is that of a non-deterministic choice between the effects in an empty set, i.e. **swap**.

42.24.5 While Expressions

Syntax

while_expr ::=
  while *logical*-value_expr **do** *unit*-value_expr **end**

Context Conditions The first value_expr must have the maximal type **Bool**. The second value_expr must have the maximal type **Unit**.

Attributes The maximal type of a while_expr is **Unit**.

A while_expr statically accesses the variables and channels which the constituent value_exprs statically access.

Meaning The effect of a while_expr is to repeat the evaluation of the second value_expr while the first Boolean value_expr evaluates to **true**. The value returned by the while_expr is the unit value.

The following equivalence holds:

while value_expr$_1$ **do** value_expr$_2$ **end**
≡
if value_expr$_1$ **then**
value_expr$_2$; **while** value_expr$_1$ **do** value_expr$_2$ **end**
else skip end

42.24.6 Until Expressions

Syntax

until_expr ::=
do *unit*-value_expr **until** *logical*-value_expr **end**

Context-independent Expansions An until_expr of the form:

do value_expr$_1$ **until** value_expr$_2$ **end**

is short for:

value_expr$_1$; **while** ∼value_expr$_2$ **do** value_expr$_1$ **end**

42.24.7 For Expressions

Syntax

for_expr ::=
for list_limitation **do** *unit*-value_expr **end**

Context Conditions The value_expr must have the maximal type **Unit**.

Context-dependent Expansions A for_expr of the form:

for list_limitation **do** value_expr **end**

is short for:

let id = ⟨ value_expr | list_limitation ⟩ **in skip end**

CHAPTER 43

Bindings

Syntax

binding ::=
  id_or_op |
  product_binding

product_binding ::=
  (binding-list2)

Terminology A value can be *matched* against a binding to give a collection of value definitions.

Context Conditions The maximal context type for a binding which is an op must be a function type which is distinguisable from the maximal type(s) of the predefined meaning(s) of op. If the op is an infix_op then the function type must have a parameter type which is a product type of length 2.

The maximal context type of a product_binding must be a product type of the same length as the binding-list2.

In a product_binding the constituent bindings must be compatible.

Attributes The context of a binding determines a *maximal context type* for the binding. For constructs containing bindings, this maximal context type is stated in the relevant sections.

An id_or_op which is a binding has as maximal type the maximal context type of the binding.

In a product_binding of the form ($binding_1$, ..., $binding_n$) having a context type of the form $t_1 \times ... \times t_n$, the maximal context types of $binding_1$, ..., $binding_n$ are t_1, ..., t_n respectively.

Meaning Matching a value, say v, of maximal context type t against a binding of the form id_or_op gives the value definition:

  id_or_op : t = v

Matching a product value, say $(v_1, ..., v_n)$, of maximal context type $t_1 \times ... \times t_n$ against a product_binding of the form $(binding_1, ..., binding_n)$ gives the collection of definitions given by matching each value v_i of maximal context type t_i against $binding_i$.

CHAPTER 44

Typings

Syntax

typing ::=
  single_typing |
  multiple_typing

single_typing ::=
  binding : type_expr

multiple_typing ::=
  binding-list2 : type_expr

commented_typing ::=
  opt-comment-string typing

Context-independent Expansions All multiple_typings and typing-lists are short for single typings.

A multiple_typing of the form:

binding_1, ..., binding_n : type_expr

is short for:

(binding_1, ..., binding_n) : type_expr $\times$ ... $\times$ type_expr

where the product type expression has length n.

A typing-list of the form:

binding_1 : type_expr_1, ..., binding_n : type_expr_n

is short for:

(binding_1, ..., binding_n) : type_expr_1 $\times$ ... $\times$ type_expr_n

A typing-list involving multiple_typings is expanded by first expanding the multiple_typings into single typings.

Attributes The maximal type of a single_typing is the maximal type of the type_expr.

In a single_typing the maximal context type of the constituent binding is the maximal type of the constituent type_expr.

The maximal definition of a value_def that is a commented_typing is obtained by replacing the type_expr of the constituent typing by the corresponding maximal type expression.

Meaning A single_typing of the form:

id_or_op : type_expr

represents a value definition — *id_or_op* is introduced for a value of the type represented by *type_expr*. A single_typing of the form:

$(\text{binding}_1,...,\text{binding}_n) : \text{type\_expr}_1 \times ... \times \text{type\_expr}_n$

represents the collection of definitions represented by each of the single typings:

$\text{binding}_1 : \text{type\_expr}_1$

$\vdots$

$\text{binding}_n : \text{type\_expr}_n$

CHAPTER 45

Patterns

45.1 General

Syntax

pattern ::=
  value_literal |
  pure_value-name |
  wildcard_pattern |
  product_pattern |
  record_pattern |
  list_pattern

Terminology A value can be *matched* against a pattern or an inner_pattern (see section 45.8) to give either *failure* or *success*. In the case of success the result of the matching is a collection of definitions.

For each kind of pattern and inner_pattern, the criteria for match success is given together with the resulting collection of definitions in the case of match success. The value matched against the pattern or inner_pattern is referred to as the 'test value'.

Attributes The context of a pattern or an inner_pattern determines a *maximal context type* for the pattern or inner_pattern. For each construct containing patterns or inner_patterns, this maximal context type is stated in the relevant section.

Context-dependent Expansions A pattern or innner_pattern that is a value_-literal, name, record_pattern, or equality_pattern has a context condition that a maximal type associated with the pattern or innner pattern must be less than or equal to its maximal context type. For these patterns an *implicit coercion* is applied to the pattern. The implicit coercion is the potential coercion (see section 41.1) from the maximal type to (a subtype of) the maximal context type. If the implicit coercion is not the identity function, the pattern or inner_pattern is short for a record pattern obtained by application of the implicit coercion:

- If the implicit coercion is a single function f then a pattern or innner_pattern p is short for the record_pattern $f(= p)$ if p is a name and $f(p)$ otherwise.
- If the implicit coercion is a functional composition $f_1 \circ ... \circ f_n$ (n $\geq$ 2) then a pattern or innner_pattern p is short for the record_pattern $f_1(...(f_n(= p))...)$ if p is a name and $f_1(...(f_n(p))...)$ otherwise.

There are similar implicit coercions in value expressions — see section 42.1.

45.2 Value Literals

Context Conditions For a pattern or an inner_pattern which is a value_literal (see section 42.2) the maximal type of the value_literal must be less than or equal to the maximal context type of the pattern or inner_pattern.

Attributes The maximal type of a value_literal that is a pattern is the maximal type of the value_literal.

Meaning

- *match success*: The value_literal must be equal to the test value.
- *resulting definitions*: None.

45.3 Names

Context Conditions For a pattern which is a name, the name must represent a value.

The maximal type of the name (see chapter 46) must be less than or equal to the maximal context type of the pattern.

Meaning

- *match success*: The value represented by the name must be equal to the test value.
- *resulting definitions*: None.

45.4 Wildcard Patterns

Syntax

wildcard_pattern ::=

—

Meaning

- *match success*: All values match a wildcard_pattern.
- *resulting definitions*: None.

45.5 Product Patterns

Syntax

product_pattern ::=
(inner_pattern-list2)

Context Conditions The maximal context type of a product_pattern must be a product type of the same length as the inner_pattern-list2.

The constituent inner_patterns must be compatible.

Attributes In a product_pattern of the form (*inner_pattern*$_1$, ..., *inner_pattern*$_n$) having a maximal context type of the form $t_1 \times ... \times t_n$, the maximal context types of *inner_pattern*$_1$, ..., *inner_pattern*$_n$ are t_1, ..., t_n, respectively.

Meaning

- *match success*: Let the product_pattern be of the form:

  (inner_pattern$_1$, ..., inner_pattern$_n$)

  Then the test value must be a product value of the form:

  (v_1, ..., v_n)

  and each value v_i must additionally match the corresponding *inner_pattern*$_i$.
- *resulting definitions*: The collection of the definitions given by matching each value v_i against *inner_pattern*$_i$.

45.6 Record Patterns

Syntax

record_pattern ::=
pure_value-name (inner_pattern-list)

Context Conditions In a record_pattern the name must represent a value and have a maximal type which is a pure function type. The result type part of this type must be less than or equal to the maximal context type of the record_pattern. Furthermore, if the record_pattern is of the form

name(inner_pattern$_1$, ..., inner_pattern$_n$) (n > 1)

then the parameter part of the function type must be of the form $t_1 \times ... \times t_n$.

In a record_pattern the constituent inner_patterns must be compatible.

Attributes In a record_pattern of the form *name*(*inner_pattern*) the maximal context type of *inner_pattern* is the parameter part of the maximal type of the name.

In a record_pattern of the form *name*(*inner_pattern*$_1$,...,*inner_pattern*$_n$) the maximal context types of *inner_pattern*$_1$,...,*inner_pattern*$_n$ are t_1,...,t_n, respectively, where the parameter type part of the maximal type of the *name* is $t_1 \times ... \times t_n$.

Meaning

- *match success*: Let the record_pattern be of the form:

  $\text{name}(\text{inner\_pattern}_1,...,\text{inner\_pattern}_n)$

  (with $n = 1$ as a special case) and let v be the test value. Then there must exist values $v_1,...,v_n$, such that:

  $\text{v} = \text{name}(\text{v}_1,...,\text{v}_n)$

  and such that each value v_i additionally matches the corresponding *inner_pattern*$_i$.
- *resulting definitions*: Values $v_1,...,v_n$ are non-deterministically chosen as indicated above such that:

  $\text{v} = \text{name}(\text{v}_1,...,\text{v}_n)$

  and such that each value v_i matches *inner_pattern*$_i$. The resulting definitions are then the collection of the definitions given by matching each value v_i against *inner_pattern*$_i$.

45.7 List Patterns

Syntax

list_pattern ::=
  enumerated_list_pattern |
  concatenated_list_pattern

Context Conditions The maximal context type of a list_pattern must be a list type.

45.7.1 Enumerated List Patterns

Syntax

enumerated_list_pattern ::=
  ⟨ opt-inner_pattern-list ⟩

Context Conditions The constituent inner_patterns must be compatible.

Attributes The maximal context type of each of the constituent inner_patterns is the element part of the maximal context type of the list_pattern.

Meaning

- *match success*: Let the enumerated_list_pattern be of the form:

  $\langle \text{inner\_pattern}_1, ..., \text{inner\_pattern}_n \rangle$

  (with $n = 0$ as a special case). Then the test value must be a list of the form:

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

and each value v_i must additionally match *inner_pattern*$_i$.
- *resulting definitions*: The collection of the definitions given by matching each value v_i against *inner_pattern*$_i$.

45.7.2 Concatenated List Patterns

Syntax

concatenated_list_pattern ::=
  enumerated_list_pattern ^ inner_pattern

Context Conditions The constituent enumerated_list_pattern and inner_pattern must be compatible.

Attributes The maximal context type of the constituent enumerated_list_pattern and inner_pattern is the maximal context type of the concatenated_list_pattern.

Meaning

- *match success*: Let the concatenated_list_pattern be of the form:

  enumerated_list_pattern ^ inner_pattern

  Then the test value must be a list of the form:

  l_1 ^ l_2

  where l_1 matches *enumerated_list_pattern* and where l_2 matches *inner_pattern*.
- *resulting definitions*: The collection of the definitions given by matching the list l_1 against *enumerated_list_pattern* and by matching the list l_2 against *inner_pattern*.

45.8 Inner Patterns

Syntax

inner_pattern ::=
  value_literal |
  id_or_op |
  wildcard_pattern |
  product_pattern |
  record_pattern |
  list_pattern |
  equality_pattern

Terminology See section 45.1.

Attributes See section 45.1.

Context-dependent Expansions See section 45.1.

45.8.1 Value Literals

See section 45.2.

45.8.2 Identifiers or Operators

Context Conditions The maximal context type for an inner_pattern which is an op must be a function type which is distinguishable from the maximal type(s) of the predefined meaning(s) of the op. If the op is an infix_op then the function type must have a parameter type which is a product type of length 2.

Attributes An id_or_op which is an inner_pattern has as maximal type the maximal context type of the inner_pattern.

Meaning

- *match success*: All values of the maximal context type match the id_or_op.
- *resulting definitions*: Let v be the test value and let t be the maximal context type of the id_or_op. Then the following definition results:

  id_or_op : t = v

45.8.3 Wildcard Patterns

See section 45.4.

45.8.4 Product Patterns

See section 45.5.

45.8.5 Record Patterns

See section 45.6.

45.8.6 List Patterns

See section 45.7.

45.8.7 Equality Patterns

Syntax

equality_pattern ::=
  = *pure_value*-name

Context Conditions In an equality_pattern the name must represent a value.

In an equality_pattern the maximal type of the name (see chapter 46) must be less than or equal to the maximal context type of the equality_pattern.

Meaning

- *match success*: The value represented by the name must be equal to the test value.
- *resulting definitions*: None.

CHAPTER 46

Names

46.1 General

Syntax

name ::=
  qualified_id |
  qualified_op

Attributes If a name represents a scheme then it has a *maximal class*, the maximal class of the associated class expression. When the scheme is parameterized, then it also has a formal scheme parameter. If a name represents an object then it has a *maximal class*, the maximal class of the associated class expression. When the object is an array, then it also has a maximal index type. If a name represents a type, a value, a variable or a channel then it has an associated maximal type. The specific attributes are defined for each of the alternatives in the following sections.

Meaning A name represents an entity which is a scheme, object, type, value, variable or channel.

46.2 Qualified Identifiers

Syntax

qualified_id ::=
  opt-qualification id

qualification ::=
  element-object_expr .

Scope and Visibility Rules In a qualified_id the scope of the qualification is extended to the id, while no other definitions are visible there.

Context Conditions In a qualification the object_expr must represent a model.

Attributes The attributes of a qualified_id are the attributes of the constituent id.

Meaning If the qualification is absent, the id represents the entity to which it has been bound by its corresponding definition.

If the qualification is present, the id represents the entity obtained by looking up the id in the model represented by the qualification.

A qualification represents the model represented by the object_expr.

46.3 Qualified Operators

Syntax

qualified_op ::=
  opt-qualification (op)

Scope and Visibility Rules In a qualified_op in which a qualification is present the scope of this qualification is extended to the op, while no other definitions (including predefined meanings of operators) are visible there.

In a qualified_op in which no qualification is present all predefined meanings of operators are visible.

Attributes The maximal type of a qualified_op is the maximal type of the constituent op.

Meaning If the qualification is absent, the op represents the entity to which it has been bound by its corresponding definition.

If the qualification is present, the op represents the entity obtained by looking up the op in the model represented by the qualification.

The brackets turn the op into a function that must be applied with prefix notation via an application expression (section 42.11). That is, noting that an op can either be a prefix_op or an infix_op, the following equivalences hold:

prefix_op value_expr ≡ (prefix_op)(value_expr)
value_expr$_1$ infix_op value_expr$_2$ ≡ (infix_op)(value_expr$_1$,value_expr$_2$)

CHAPTER 47

Identifiers and Operators

47.1 General

Syntax

id_or_op ::=
  id |
  op

op ::=
  infix_op |
  prefix_op

Terminology Each occurrence of an identifier or operator (id, op or id_or_op) is either a *defining* or an *applied* occurrence.

The following occurrences are defining occurrences:

- The id (or ids) occurring immediately within a scheme_def, object_def, sort_def, variant_def, union_def, short_record_def, abbreviation_def, prefix_application, infix_application, single_variable_def, multiple_variable_def, single_channel_def, multiple_channel_def, axiom_naming or id_or_wildcard.
- The new id_or_op in a rename_pair.
- The id_or_op occurring immediately within a constructor, destructor, reconstructor or binding.
- The id_or_op which is an inner_pattern.

All other occurrences are applied occurrences.

Each defining occurrence of an identifier or operator is part of a declarative construct that represents at least a definition introducing this identifier or operator.

An applied occurrence of an identifier or operator is said to be *visible* if there is a visible definition introducing it.

A legal applied occurrence of an identifier or operator has a *corresponding definition* (or *interpretation*). There are three cases for an applied occurrence of an identifier or operator:

1. There is no visible definition introducing it, i.e. it is not visible. In that case the occurrence is illegal and hence the identifier or operator has no corresponding definition.
2. There is exactly one visible definition introducing it. This definition is the corresponding definition of the identifier or operator.
3. There are two or more visible definitions introducing it. According to the visibility rules and context conditions for declarative constructs this can only be the case for identifiers and operators representing values. Chapter 36 on overloading explains how to find the single corresponding definition in this case, if possible. If it is not possible to find a unique corresponding definition then the occurrence is illegal.

An applied occurrence of an identifier or operator *represents* the entity of its corresponding definition.

Scope and Visibility Rules Note that all operators have one or more predefined meanings which have the entire specification as scope and cannot be hidden, except within the operator part of qualified operators (see section 46.3). So with this exception operators are always visible.

Context Conditions An applied occurrence of an identifier or operator must be visible with a unique corresponding definition. This implies that a defining occurrence of an operator must have a maximal type which is distinguishable from the maximal type(s) of the predefined meanings of the operator.

Attributes For an identifier representing a scheme its maximal class and possible formal scheme parameter are determined by its corresponding definition (see section 38.2). For an identifier representing an object the maximal class and possible maximal index type are determined by its corresponding definition (see section 38.3). For an identifier or operator representing a type, a value, a variable or a channel its maximal type is determined by its corresponding definition.

47.2 Infix Operators

Syntax

infix_op ::=
= |
≠ |
> |
< |
≥ |
≤ |
∪ |
∩ |
⊇ |
⊆ |

$\in$ |
$\notin$ |
$+$ |
$-$ |
$\setminus$ |
$\frown$ |
$\cup$ |
$\dagger$ |
$*$ |
$/$ |
$\circ$ |
$\cap$ |
$\uparrow$

Context Conditions See section 47.1.

Attributes The maximal type of an infix_op is determined by its corresponding definition — see section 47.1. The maximal type of an infix_op representing its predefined meaning is the maximal type of the type expression stated below.

Meaning The meaning of an infix_op is determined by its corresponding definition. The predefined meanings of applications of the infix_ops are listed below.

The infix_ops operate on pairs of values referred to as arguments. Some infix_ops may have pre-conditions that must hold for the arguments. When a pre-condition is violated, the effect of the value_infix_expr in which the infix_op occurs is underspecified.

The type T and subscripted versions of T occurring in the infix_op signatures are type variables representing arbitrary types.

- **Equality:**

  $= : \mathrm{T} \times \mathrm{T} \to \mathbf{Bool}$

  The result is **true** if and only if the values of the two arguments are equal.
- **Inequality:**

  $\neq : \mathrm{T} \times \mathrm{T} \to \mathbf{Bool}$

  The result is **true** if and only if the values of the two arguments are not equal.
- **Integer addition:**

  $+ : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}$

  The result is the sum of the two integers.
- **Real addition:**

  $+ : \mathbf{Real} \times \mathbf{Real} \to \mathbf{Real}$

  The result is the sum of the two reals.
- **Integer subtraction:**

  $- : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}$

The result is the difference between the first integer and the second integer.

- **Real subtraction:**

  $- : \mathbf{Real} \times \mathbf{Real} \to \mathbf{Real}$

  The result is the difference between the first real and the second real.
- **Integer multiplication:**

  $* : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Int}$

  The result is the product of the two integers.
- **Real multiplication:**

  $* : \mathbf{Real} \times \mathbf{Real} \to \mathbf{Real}$

  The result is the product of the two reals.
- **Integer exponentiation:**

  $\uparrow : \mathbf{Int} \times \mathbf{Int} \xrightarrow{\sim} \mathbf{Int}$

  Pre-condition: The second integer must not be negative, and the second integer must not be zero if the first is zero.

  The result is the first integer raised to the power of the second integer.
- **Real exponentiation:**

  $\uparrow : \mathbf{Real} \times \mathbf{Real} \xrightarrow{\sim} \mathbf{Real}$

  Pre-condition: If the second real is not positive then the first real must be different from zero (0.0). If the second real is not a whole number then the first real must be non-negative.

  The result is the first real raised to the power of the second real.
- **Function composition:**

  $\circ : (\mathrm{T}_2 \xrightarrow{\sim} \mathrm{a}\ \mathrm{T}_3) \times (\mathrm{T}_1 \xrightarrow{\sim} \mathrm{a}'\ \mathrm{T}_2) \to (\mathrm{T}_1 \xrightarrow{\sim} \mathrm{a}''\ \mathrm{T}_3)$

  where a'' is the union of a and a'.

  The result is the composition of the two functions defined as follows:

  $\mathrm{f}_1 \circ \mathrm{f}_2 \equiv \lambda \mathrm{x} : \mathrm{T}_1 \bullet \mathrm{f}_1(\mathrm{f}_2(\mathrm{x}))$
- **Map composition:**

  $\circ : (\mathrm{T}_2 \xrightarrow[m]{} \mathrm{T}_3) \times (\mathrm{T}_1 \xrightarrow[m]{} \mathrm{T}_2) \to (\mathrm{T}_1 \xrightarrow[m]{} \mathrm{T}_3)$

  The result is the composition of the two maps defined as follows:

  $\mathrm{m}_1 \circ \mathrm{m}_2 \equiv [\mathrm{x} \mapsto \mathrm{m}_1(\mathrm{m}_2(\mathrm{x})) \mid \mathrm{x} : \mathrm{T}_1 \bullet \mathrm{x} \in \mathbf{dom}\ \mathrm{m}_2 \wedge \mathrm{m}_2(\mathrm{x}) \in \mathbf{dom}\ \mathrm{m}_1]$
- **Integer division:**

  $/ : \mathbf{Int} \times \mathbf{Int} \xrightarrow{\sim} \mathbf{Int}$

  Pre-condition: The second integer must not be zero (0).

  The absolute value (without sign) of the result is the number of times that the absolute value of the second integer is within the absolute value of the first integer. The sign of the result is the product of the signs of the arguments.
- **Real division:**

$/ : \mathbf{Real} \times \mathbf{Real} \xrightarrow{\sim} \mathbf{Real}$

Pre-condition: The second real must not be zero (0.0).

The result is obtained by dividing the first real by the second real.

- **Map restriction to:**

$/ : (\mathrm{T}_1 \xrightarrow[m]{} \mathrm{T}_2) \times \mathrm{T}_1\textbf{-infset} \to (\mathrm{T}_1 \xrightarrow[m]{} \mathrm{T}_2)$

The result is the map with its domain limited to the elements of the set.

- **Integer remainder:**

$\backslash : \mathbf{Int} \times \mathbf{Int} \xrightarrow{\sim} \mathbf{Int}$

Pre-condition: The second integer must not be zero (0).

The absolute value of the result is the remainder after having divided the absolute value of the second integer into the absolute value of the first integer. The sign of the result is the sign of the first integer. This implies the following relationship between integer division and integer remainder. Let a and b be integers, b not zero:

$$\mathrm{a} = (\mathrm{a}/\mathrm{b}) * \mathrm{b} + (\mathrm{a} \backslash \mathrm{b})$$

- **Set difference:**

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

The result is the set of all elements which appear in the first set and not in the second.

- **Map restriction by:**

$\backslash : (\mathrm{T}_1 \xrightarrow[m]{} \mathrm{T}_2) \times \mathrm{T}_1\textbf{-infset} \to (\mathrm{T}_1 \xrightarrow[m]{} \mathrm{T}_2)$

The result is the map with the elements of the set removed from its domain.

- **Integer greater than:**

$> : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Bool}$

The result is **true** if and only if the first integer is greater than the second integer.

- **Real greater than:**

$> : \mathbf{Real} \times \mathbf{Real} \to \mathbf{Bool}$

The result is **true** if and only if the first real is greater than the second real.

- **Integer less than:**

$< : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Bool}$

The result is **true** if and only if the first integer is less than the second integer.

- **Real less than:**

$< : \mathbf{Real} \times \mathbf{Real} \to \mathbf{Bool}$

The result is **true** if and only if the first real is less than the second real.

- **Integer greater than or equal to:**

$\geq : \mathbf{Int} \times \mathbf{Int} \to \mathbf{Bool}$

The result is **true** if and only if the first integer is greater than or equal to the second integer.

- **Real greater than or equal to:**

  $\geq$: **Real** $\times$ **Real** $\rightarrow$ **Bool**

  The result is **true** if and only if the first real is greater than or equal to the second real.
- **Integer less than or equal to:**

  $\leq$: **Int** $\times$ **Int** $\rightarrow$ **Bool**

  The result is **true** if and only if the first integer is less than or equal to the second integer.
- **Real less than or equal to:**

  $\leq$: **Real** $\times$ **Real** $\rightarrow$ **Bool**

  The result is **true** if and only if the first real is less than or equal to the second real.
- **Set superset (proper):**

  $\supset$: T**-infset** $\times$ T**-infset** $\rightarrow$ **Bool**

  The result is **true** if and only if the second set is a proper subset of the first set. That is, it is a subset of the first set but not equal to it.
- **Set subset (proper):**

  $\subset$: T**-infset** $\times$ T**-infset** $\rightarrow$ **Bool**

  The result is **true** if and only if the first set is a proper subset of the second set. That is, it is a subset of the second set but not equal to it.
- **Set superset:**

  $\supseteq$: T**-infset** $\times$ T**-infset** $\rightarrow$ **Bool**

  The result is **true** if and only if the second set is a subset of the first set.
- **Set subset:**

  $\subseteq$: T**-infset** $\times$ T**-infset** $\rightarrow$ **Bool**

  The result is **true** if and only if the first set is a subset of the second set.
- **Set membership:**

  $\in$: T $\times$ T**-infset** $\rightarrow$ **Bool**

  The result is **true** if and only if the first argument is a member of the set.
- **Set membership (negation):**

  $\notin$: T $\times$ T**-infset** $\rightarrow$ **Bool**

  The result is **true** if and only if the first argument is not a member of the set.
- **Set intersection:**

  $\cap$: T**-infset** $\times$ T**-infset** $\rightarrow$ T**-infset**

  The result is the set containing all elements which appear in both of the two

sets.

- **Set union:**

  $\cup$: T-**infset** $\times$ T-**infset** $\to$ T-**infset**

  The result is the set containing all elements which appear in one or both of the two sets.
- **Map union:**

  $\cup : (T_1 \xrightarrow[m]{} T_2) \times (T_1 \xrightarrow[m]{} T_2) \to (T_1 \xrightarrow[m]{} T_2)$

  The result is the map containing all the pairs of the first map and all the pairs of the second map.
- **List concatenation:**

  $\frown : T^\omega \times T^\omega \xrightarrow{\sim} T^\omega$

  Pre-condition: The first list must be finite.

  The result is the concatenation of the two lists. That is, the list containing all the elements of the two lists, ordered as in the two lists and with all the elements of the first list appearing first.
- **Map override:**

  $\dagger : (T_1 \xrightarrow[m]{} T_2) \times (T_1 \xrightarrow[m]{} T_2) \to (T_1 \xrightarrow[m]{} T_2)$

  The result is the first map overridden with the second map. Where the two maps have common domain elements, the second map overrides the first.

47.3 Prefix Operators

Syntax

prefix_op ::=
  abs |
  int |
  real |
  card |
  len |
  inds |
  elems |
  hd |
  tl |
  dom |
  rng

Context Conditions See section 47.1.

Attributes The maximal type of a prefix_op is determined by its corresponding definition — see section 47.1.

The maximal type of a prefix_op representing its predefined meaning is the maximal type of the type expression stated below.

Meaning The meaning of a prefix_op is determined by its corresponding definition. The predefined meanings of applications of the prefix_ops are listed below.

The prefix_ops operate on values (arguments). Some prefix_ops may have pre-conditions that must hold for the argument. When a pre-condition is violated, the effect of the value_prefix_expr in which the prefix_op occurs is under-specified unless otherwise stated.

The type T occurring in the prefix_op signatures is a type variable representing an arbitrary type.

- **Integer absolute value:**

  abs : **Int** $\rightarrow$ **Int**

  The result is the absolute value of the integer. That is, if the integer is negative, the negated value is returned. The operator is the identity on non-negative integers.
- **Real absolute value:**

  abs : **Real** $\rightarrow$ **Real**

  The result is the absolute value of the real. That is, if the real is negative, the negated value is returned. The operator is the identity on non-negative reals.
- **Real to integer conversion:**

  int : **Real** $\rightarrow$ **Int**

  The absolute value (without sign) of the result is the greatest integer that is smaller than or equal to the absolute value of the real. The sign is the sign of the real.
- **Integer to real conversion:**

  real : **Int** $\rightarrow$ **Real**

  The result is the identity on the argument, just changing its type. This implies the following relationship between the conversion functions. Let a be an integer:

  int real a = a
- **Set cardinality:**

  card : T-**infset** $\xrightarrow{\sim}$ **Int**

  The result is the number of elements in the set. The effect of applying **card** to an infinite set is to diverge.
- **List length:**

  len : T^{ω} $\xrightarrow{\sim}$ **Int**

  The result is the length of the list. The effect of applying **len** to an infinite list is to diverge.
- **List indices:**

  inds : T^{ω} $\rightarrow$ **Int-infset**

The result is the index set for the list. Let *f_list* be a finite list and let *i_list* be an infinite list, then:

inds f_list = $\{i \mid i : \textbf{Int} \bullet i \geq 1 \wedge i \leq \textbf{len}\ f\_list\}$
inds i_list = $\{i \mid i : \textbf{Int} \bullet i \geq 1\}$

- **List elements:**

  elems : $T^\omega \to T$**-infset**

  The result is the set of elements of the list.
- **List head:**

  hd : $T^\omega \xrightarrow{\sim} T$

  Pre-condition: The list must be non-empty.
  The result is the first element in the list.
- **List tail:**

  tl : $T^\omega \xrightarrow{\sim} T^\omega$

  Pre-condition: The list must be non-empty.
  The result is the list which remains after removing the first element.
- **Map domain:**

  dom : $(T_1 \xrightarrow[m]{} T_2) \to T_1$**-infset**

  The result is the domain of the map: the values for which it is defined.
- **Map range:**

  rng : $(T_1 \xrightarrow[m]{} T_2) \to T_2$**-infset**

  The result is the range of the map: the values that can be obtained by applying the map to the values in its domain.

CHAPTER 48

Connectives

48.1 Infix Connectives

Syntax

infix_connective ::=
  ⇒ |
  ∨ |
  ∧

Context-independent Expansions The infix_connectives are intended to compose Boolean value expressions into new Boolean value expressions.

The effect of an axiom_infix_expr, in which an infix_connective occurs, follows a conditional logic where in general the second constituent value expression is evaluated only if the value of the first constituent value expression does not uniquely determine the value of the axiom_infix_expr. In this way the eventual divergence or deadlock in the second constituent value expression can be avoided when possible. The meaning of the axiom_infix_exprs is given in terms of the if expressions for which they are short.

- **Boolean and:**

  $\text{value\_expr}_1 \wedge \text{value\_expr}_2$

  is short for:

  if value_expr_1 **then** value_expr_2 **else false end**

- **Boolean or:**

  $\text{value\_expr}_1 \vee \text{value\_expr}_2$

  is short for:

  if value_expr_1 **then true else** value_expr_2 **end**

- **Boolean implication:**

  $\text{value\_expr}_1 \Rightarrow \text{value\_expr}_2$

is short for:

if value_expr$_1$ **then** value_expr$_2$ **else true end**

48.2 Prefix Connectives

Syntax

prefix_connective ::=
  ∼

Context-independent Expansions The prefix_connective composes Boolean value expressions into new Boolean value expressions.

- **Boolean not:**
  An axiom_prefix_expr of the form:

  ∼value_expr

  is short for:

  if value_expr **then false else true end**

CHAPTER 49

Infix Combinators

Syntax

infix_combinator ::=
  [] |
  ⫏ |
  ||| |
  ⫴ |
  ;

Meaning The infix_combinators are intended to compose value expressions that either communicate or have effects on variables. Some simple proof rules are associated with each infix_combinator in order to clarify its semantics.

- **External choice:**

  $\text{value\_expr}_1 \mathbin{[]} \text{value\_expr}_2$

  An external choice is made between the effects of the two *value_expr*s. The possible effect of a concurrently executing third value expression can influence the choice.

  External choice has unit **stop**, has zero **chaos**, is idempotent, is commutative, is associative, and is distributive through internal choice:

  $\text{value\_expr} \mathbin{[]} \textbf{stop} \equiv \text{value\_expr}$

  $\text{value\_expr} \mathbin{[]} \textbf{chaos} \equiv \textbf{chaos}$

  $\text{value\_expr} \mathbin{[]} \text{value\_expr} \equiv \text{value\_expr}$

  $\text{value\_expr}_1 \mathbin{[]} \text{value\_expr}_2 \equiv \text{value\_expr}_2 \mathbin{[]} \text{value\_expr}_1$

  $\text{value\_expr}_1 \mathbin{[]} (\text{value\_expr}_2 \mathbin{[]} \text{value\_expr}_3) \equiv$
  $(\text{value\_expr}_1 \mathbin{[]} \text{value\_expr}_2) \mathbin{[]} \text{value\_expr}_3$

value_expr$_1$ [] (value_expr$_2$ ⊓ value_expr$_3$) ≡
  (value_expr$_1$ [] value_expr$_2$) ⊓ (value_expr$_1$ [] value_expr$_3$)

- **Internal choice:**

  value_expr$_1$ ⊓ value_expr$_2$

  An internal — non-deterministic — choice is made between the effects of the two *value_expr*s. The possible effect of a concurrently executing third value expression cannot influence the choice.

  Internal choice has zero **chaos**, is idempotent, is commutative, and is associative:

  value_expr ⊓ **chaos** ≡ **chaos**

  value_expr ⊓ value_expr ≡ value_expr

  value_expr$_1$ ⊓ value_expr$_2$ ≡ value_expr$_2$ ⊓ value_expr$_1$

  value_expr$_1$ ⊓ (value_expr$_2$ ⊓ value_expr$_3$) ≡
    (value_expr$_1$ ⊓ value_expr$_2$) ⊓ value_expr$_3$

- **Concurrent composition:**

  value_expr$_1$ || value_expr$_2$

  The two *value_expr*s are executed concurrently with another until one of them terminates, when the other continues. The two *value_expr*s may offer to communicate along channels, so they may be able to communicate with one another (by having one of them input from a channel and the other of them output to the same channel). If they are able to communicate with one another, they may make an internal choice which dictates that they will do so; alternatively they may make an internal choice which leaves them free to do so or to communicate with other concurrently executing value expressions. If they are able to communicate with other concurrently executing value expressions but not with one another, they are left free to do so.

  Concurrent composition has unit **skip**, has zero **chaos**, is commutative, is associative, and is distributive through internal choice:

  value_expr || **skip** ≡ value_expr

  value_expr || **chaos** ≡ **chaos**

  value_expr$_1$ || value_expr$_2$ ≡ value_expr$_2$ || value_expr$_1$

  value_expr$_1$ || (value_expr$_2$ || value_expr$_3$) ≡
    (value_expr$_1$ || value_expr$_2$) || value_expr$_3$

  value_expr$_1$ || (value_expr$_2$ ⊓ value_expr$_3$) ≡

(value_expr$_1$ ‖ value_expr$_2$) ⨅ (value_expr$_1$ ‖ value_expr$_3$)

The following two equivalences hold if the value expression *value_expr* is convergent and does not involve communication and if $c_1 \neq c_2$:

x := c$_1$? ‖ c$_2$!value_expr ≡
(x := c$_1$? ; c$_2$!value_expr) [] (c$_2$!value_expr ; x := c$_1$?)

x := c? ‖ c!value_expr ≡
(((x := c? ; c!value_expr) [] (c!value_expr ; x := c?))
[] (x := value_expr))
⨅ (x := value_expr)

These are special cases of general laws given in [37].

- **Interlocked composition:**

value_expr$_1$ ⫴ value_expr$_2$

The *value_expr*s are executed interlocked with one another until one of them terminates, when the other continues. The two *value_expr*s may offer to communicate along channels, so they may be able to communicate with one another (by having one of them input from a channel and the other of them output to the same channel). If they are able to communicate with one another, they will do so. If they are able to communicate with other concurrently executing value expressions but not with one another, they deadlock unless one of them can terminate.

Interlocked composition has unit **skip**, has zero **chaos**, is commutative, and is distributive through internal choice:

value_expr ⫴ **skip** ≡ value_expr

value_expr ⫴ **chaos** ≡ **chaos**

value_expr$_1$ ⫴ value_expr$_2$ ≡ value_expr$_2$ ⫴ value_expr$_1$

value_expr$_1$ ⫴ (value_expr$_2$ ⨅ value_expr$_3$) ≡
(value_expr$_1$ ⫴ value_expr$_2$) ⨅ (value_expr$_1$ ⫴ value_expr$_3$)

Note that, unlike ‖, ⫴ is not associative.

The following two equivalences hold if the value expression *value_expr* is convergent and does not involve communication and if $c_1 \neq c_2$:

x := c$_1$? ⫴ c$_2$!value_expr ≡ **stop**

x := c? ⫴ c!value_expr ≡ x := value_expr

The interlocking combinator illustrates the distinction between external choice and internal choice. The following equivalences hold if *value_expr*$_1$ and *value_expr*$_2$ are convergent and do not involve communication and if $c_1 \neq c_2$:

$(x := c_1? \ [] \ c_2!value\_expr_2) \parallel c_1!value\_expr_1 \equiv x := value\_expr_1$

$(x := c_1? \ [\!] \ c_2!value\_expr_2) \parallel c_1!value\_expr_1 \equiv (x := value\_expr_1) \ [\!] \ \mathbf{stop}$

These are special cases of general laws given in [37].

- **Sequential composition:**

  $value\_expr_1 \ ; \ value\_expr_2$

  The second *value_expr* is made to execute sequentially after the first *value_expr*. The value returned is the value returned by the second *value_expr*.

  Sequential composition has unit **skip**, is associative, and is distributive in its first argument through internal choice:

  $value\_expr \ ; \ \mathbf{skip} \equiv value\_expr$

  $\mathbf{skip} \ ; \ value\_expr \equiv value\_expr$

  $value\_expr_1 \ ; \ (value\_expr_2 \ ; \ value\_expr_3) \equiv$
  $(value\_expr_1 \ ; \ value\_expr_2) \ ; \ value\_expr_3$

  $(value\_expr_1 \ [\!] \ value\_expr_2) \ ; \ value\_expr_3 \equiv$
  $(value\_expr_1 \ ; \ value\_expr_3) \ [\!] \ (value\_expr_2 \ ; \ value\_expr_3)$

  These laws for sequential composition are derived from laws for let expressions in [37], arising from the fact that:

  $value\_expr_1 \ ; \ value\_expr_2$

  is short for:

  $\mathbf{let}$ id $= value\_expr_1 : \mathbf{Unit\ in}\ value\_expr_2\ \mathbf{end}$

  provided *id* is not already in scope.

Part III

Appendices

APPENDIX A

Syntax Summary

Specifications

specification ::=
  module_decl-string

module_decl ::=
  scheme_decl |
  object_decl

Declarations

decl ::=
  scheme_decl |
  object_decl |
  type_decl |
  value_decl |
  variable_decl |
  channel_decl |
  axiom_decl

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

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]

Type Declarations

type_decl ::=
  type type_def-list

type_def ::=
  sort_def |
  variant_def |
  union_def |
  short_record_def |
  abbreviation_def

Sort Definitions

sort_def ::=
  opt-comment-string id

Variant Definitions

variant_def ::=
  opt-comment-string id == variant-choice

variant ::=
  constructor |

record_variant

record_variant ::=
  constructor (component_kind-list)

component_kind ::=
  opt-destructor type_expr opt-reconstructor

constructor ::=
  id_or_op |
  —

destructor ::=
  id_or_op :

reconstructor ::=
  ↔ id_or_op

Union Definitions

union_def ::=
  opt-comment-string
  id = name_or_wildcard-choice2

name_or_wildcard ::=
  type-name |
  —

Short Record Definitions

short_record_def ::=
  opt-comment-string
  id :: component_kind-string

Abbreviation Definitions

abbreviation_def ::=
  opt-comment-string id = type_expr

Value Declarations

value_decl ::=
  value value_def-list

value_def ::=
  commented_typing |
  explicit_value_def |
  implicit_value_def |
  explicit_function_def |
  implicit_function_def

Explicit Value Definitions

explicit_value_def ::=
  opt-comment-string
  single_typing = *pure*-value_expr

Implicit Value Definitions

implicit_value_def ::=
  opt-comment-string
  single_typing *pure*-restriction

Explicit Function Definitions

explicit_function_def ::=
  opt-comment-string single_typing
  formal_function_application ≡ value_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

Implicit Function Definitions

implicit_function_def ::=
  opt-comment-string
  single_typing formal_function_application
  post_condition opt-pre_condition

Variable Declarations

variable_decl ::=
  variable variable_def-list

variable_def ::=
  single_variable_def |
  multiple_variable_def

single_variable_def ::=

opt-comment-string
id : type_expr opt-initialisation

initialisation ::=
:= *pure*-value_expr

multiple_variable_def ::=
opt-comment-string id-list2 : type_expr

Channel Declarations

channel_decl ::=
channel channel_def-list

channel_def ::=
single_channel_def |
multiple_channel_def

single_channel_def ::=
opt-comment-string id : type_expr

multiple_channel_def ::=
opt-comment-string 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
readonly_logical-value_expr

axiom_naming ::=
[id]

Class Expressions

class_expr ::=
basic_class_expr |
extending_class_expr |
hiding_class_expr |
renaming_class_expr |
scheme_instantiation

Basic Class Expressions

basic_class_expr ::=
class opt-decl-string **end**

Extending Class Expressions

extending_class_expr ::=
extend class_expr **with** class_expr

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)

Rename Pairs

rename_pair ::=
defined_item **for** defined_item

Defined Items

defined_item ::=
id_or_op |
disambiguated_item

disambiguated_item ::=
id_or_op : type_expr

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*-value_expr-list]

Array Object Expressions

array_object_expr ::=
  [| typing-list • *element*-object_expr |]

Fitting Object Expressions

fitting_object_expr ::=
  object_expr { rename_pair-list }

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 |
  bracketed_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 $\overset{m}{\Rightarrow}$ 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

Subtype Expressions

subtype_expr ::=
  {| single_typing *pure*-restriction |}

Bracketed Type Expressions

bracketed_type_expr ::=
  (type_expr)

Access Descriptions

access_desc ::=
  access_mode access-list

access_mode ::=
  read |
  write |
  in |
  out

access ::=
  variable_or_channel-name |
  enumerated_access |
  completed_access |
  comprehended_access

enumerated_access ::=
  { opt-access-list }

completed_access ::=
  opt-qualification **any**

comprehended_access ::=
  { access | *pure*-set_limitation }

Value Expressions

value_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 |
  bracketed_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 ::=
  (value_expr-list2)

Set Expressions

set_expr ::=
  ranged_set_expr |
  enumerated_set_expr |
  comprehended_set_expr

Ranged Set Expressions

ranged_set_expr ::=
  { *readonly_integer*-value_expr ..
    readonly_integer-value_expr }

Enumerated Set Expressions

enumerated_set_expr ::=
  { *readonly*-opt-value_expr-list }

Comprehended Set Expressions

comprehended_set_expr ::=
  { *readonly*-value_expr | set_limitation }

set_limitation ::=
  typing-list opt-restriction

restriction ::=
  • *readonly_logical*-value_expr

List Expressions

list_expr ::=
  ranged_list_expr |
  enumerated_list_expr |
  comprehended_list_expr

Ranged List Expressions

ranged_list_expr ::=
  ⟨ *integer*-value_expr .. *integer*-value_expr ⟩

Enumerated List Expressions

enumerated_list_expr ::=
  ⟨ opt-value_expr-list ⟩

Comprehended List Expressions

comprehended_list_expr ::=
  ⟨ value_expr | list_limitation ⟩

list_limitation ::=
  binding **in** *readonly_list*-value_expr
  opt-restriction

Map Expressions

map_expr ::=
  enumerated_map_expr |
  comprehended_map_expr

Enumerated Map Expressions

enumerated_map_expr ::=
  [opt-value_expr_pair-list]

value_expr_pair ::=
  readonly-value_expr ↦ *readonly*-value_expr

Comprehended Map Expressions

comprehended_map_expr ::=
  [value_expr_pair | set_limitation]

Function Expressions

function_expr ::=
  λ lambda_parameter • value_expr

lambda_parameter ::=
  lambda_typing |
  single_typing

lambda_typing ::=
  (opt-typing-list)

Application Expressions

application_expr ::=
  list_or_map_or_function-value_expr
  actual_function_parameter-string

actual_function_parameter ::=
  (opt-value_expr-list)

Quantified Expressions

quantified_expr ::=
  quantifier typing-list restriction

quantifier ::=
  ∀ |
  ∃ |
  ∃!

Equivalence Expressions

equivalence_expr ::=
  value_expr ≡ value_expr opt-pre_condition

pre_condition ::=
  pre *readonly_logical*-value_expr

Post Expressions

post_expr ::=

value_expr
post_condition opt-pre_condition

post_condition ::=
opt-result_naming
post *readonly_logical*-value_expr

result_naming ::=
as binding

Disambiguation Expressions

disambiguation_expr ::=
value_expr : type_expr

Bracketed Expressions

bracketed_expr ::=
(value_expr)

Infix Expressions

infix_expr ::=
stmt_infix_expr |
axiom_infix_expr |
value_infix_expr

Statement Infix Expressions

stmt_infix_expr ::=
value_expr infix_combinator value_expr

Axiom Infix Expressions

axiom_infix_expr ::=
logical-value_expr infix_connective
logical-value_expr

Value Infix Expressions

value_infix_expr ::=
value_expr infix_op value_expr

Prefix Expressions

prefix_expr ::=
axiom_prefix_expr |
universal_prefix_expr |
value_prefix_expr

Axiom Prefix Expressions

axiom_prefix_expr ::=
prefix_connective *logical*-value_expr

Universal Prefix Expressions

universal_prefix_expr ::=
□ *readonly_logical*-value_expr

Value Prefix Expressions

value_prefix_expr ::=
prefix_op value_expr

Comprehended Expressions

comprehended_expr ::=
associative_commutative-infix_combinator
{ value_expr | set_limitation }

Initialise Expressions

initialise_expr ::=
opt-qualification **initialise**

Assignment Expressions

assignment_expr ::=
variable-name := value_expr

Input Expressions

input_expr ::=
channel-name ?

Output Expressions

output_expr ::=
channel-name ! value_expr

Structured Expressions

structured_expr ::=
local_expr |
let_expr |
if_expr |
case_expr |
while_expr |
until_expr |
for_expr

Local Expressions

local_expr ::=
  local opt-decl-string **in** value_expr **end**

Let Expressions

let_expr ::=
  let let_def-list **in** value_expr **end**

let_def ::=
  typing |
  explicit_let |
  implicit_let

explicit_let ::=
  let_binding = value_expr

implicit_let ::=
  single_typing restriction

let_binding ::=
  binding |
  record_pattern |
  list_pattern

If Expressions

if_expr ::=
  if *logical*-value_expr **then**
  value_expr
  opt-elsif_branch-string
  opt-else_branch
  end

elsif_branch ::=
  elsif *logical*-value_expr **then** value_expr

else_branch ::=
  else value_expr

Case Expressions

case_expr ::=
  case value_expr **of** case_branch-list **end**

case_branch ::=
  pattern → value_expr

While Expressions

while_expr ::=
  while *logical*-value_expr
  do *unit*-value_expr **end**

Until Expressions

until_expr ::=
  do *unit*-value_expr
  until *logical*-value_expr **end**

For Expressions

for_expr ::=
  for list_limitation **do** *unit*-value_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

commented_typing ::=
  opt-comment-string typing

Patterns

pattern ::=
  value_literal |
  pure_value-name |
  wildcard_pattern |
  product_pattern |
  record_pattern |
  list_pattern

Wildcard Patterns

wildcard_pattern ::=
  _

Product Patterns

product_pattern ::=
  (inner_pattern-list2)

Record Patterns

record_pattern ::=
  pure_value-name (inner_pattern-list)

List Patterns

list_pattern ::=
  enumerated_list_pattern |
  concatenated_list_pattern

Enumerated List Patterns

enumerated_list_pattern ::=
  ⟨ opt-inner_pattern-list ⟩

Concatenated List Patterns

concatenated_list_pattern ::=
  enumerated_list_pattern ⌢ inner_pattern

Inner Patterns

inner_pattern ::=
  value_literal |
  id_or_op |
  wildcard_pattern |
  product_pattern |
  record_pattern |
  list_pattern |
  equality_pattern

Equality Patterns

equality_pattern ::=
  = *pure_value*-name

Names

name ::=
  qualified_id |
  qualified_op

Qualified Identifiers

qualified_id ::=
  opt-qualification id

qualification ::=
  element-object_expr .

Qualified Operators

qualified_op ::=
  opt-qualification (op)

Identifiers and Operators

id_or_op ::=
  id |
  op

op ::=
  infix_op |
  prefix_op

Infix Operators

infix_op ::=
  = |
  ≠ |
  > |
  < |
  ≥ |
  ≤ |
  ⊃ |
  ⊂ |
  ⊇ |
  ⊆ |
  ∈ |
  ∉ |
  + |
  − |
  \ |
  ⌢ |
  ∪ |
  † |
  * |
  / |
  ° |
  ∩ |
  ↑

Prefix Operators

prefix_op ::=
  abs |
  int |
  real |
  card |
  len |
  inds |
  elems |
  hd |
  tl |
  dom |
  rng

Connectives

connective ::=
  infix_connective |
  prefix_connective

Infix Connectives

infix_connective ::=
  ⇒ |
  ∨ |
  ∧

Prefix Connectives

prefix_connective ::=
  ∼

Infix Combinators

infix_combinator ::=
  ⫿ |
  ⫍ |
  ‖ |
  ⫴ |
  ;

Precedence and Associativity of Operators

Value operator precedence – increasing		
Prec	Operator(s)	Associativity
14	$\Box\ \lambda\ \forall\ \exists\ \exists!$	Right
13	$\equiv$ **post**	
12	$[]\ \sqcap\ \|\|\ \#$	Right
11	;	Right
10	:=	
9	$\Rightarrow$	Right
8	$\vee$	Right
7	$\wedge$	Right
6	$= \neq > < \geq \leq \subset \subseteq \supset \supseteq \in \notin$	
5	$+\ -\ \setminus\ ^\frown\ \cup\ \dagger$	Left
4	$*\ /\ \circ\ \cap$	Left
3	$\uparrow$	
2	:	
1	$\sim$ prefix_op	

Type operator precedence – increasing		
Prec	Operator(s)	Associativity
3	$\overrightarrow{m}\ \xrightarrow{\sim}\ \rightarrow$	Right
2	$\times$	
1	**-set -infset** $^*\ ^\omega$	

APPENDIX C

Lexical Matters

This chapter describes lexical matters, i.e. the microsyntax for RSL.

Basically, RSL follows the rules now in current practice for most programming languages: a text (i.e. an RSL specification) is represented as a string of characters, which is interpreted left-to-right and broken into a string of tokens. The characters are drawn from a superset of the ASCII characters called the *full RSL character set*. Tokens may be separated by 'whitespace', which is strings of one or more of the following characters: line-feed, carriage-return, space and tab. (Note that *comments* are part of the RSL syntax and thus cannot be used freely as whitespace. Also note that comments may not be nested.)

There are two types of tokens in RSL: varying and fixed.

Varying Tokens

The microsyntax for varying tokens is defined by the syntax rules below, where the characters used in forming tokens are shown in quotes, as in '$'. Furthermore, LF, CR and TAB are used to denote the ASCII characters line-feed, carriage-return and tab.

```
id ::=
  letter opt-letter_or_digit_or_underline_or_prime-string

letter_or_digit_or_underline_or_prime ::=
  letter | digit | underline | prime

letter ::=
  ascii_letter | greek_letter

comment ::=
  '/' '*' comment_item-string '*' '/'

comment_item ::=
  comment_char

comment_char ::=
  LF | CR | TAB | ascii_letter | digit | graphic | prime | quote | backslash

int_literal ::=
```

digit-string

real_literal ::=
digit-string '.' digit-string

text_literal ::=
'"', opt-text_character-string '"',

char_literal ::=
"'", char_character "'",

text_character ::=
character | prime

char_character ::=
character | quote

character ::=
ascii_letter | digit | graphic | escape

digit ::=
'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

ascii_letter ::=
'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' |
'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' |
'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' |
'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'

greek_letter ::=
'`alpha' | '`beta' | '`gamma' | '`delta' | '`epsilon' | '`zeta' | '`eta' | '`theta' | '`iota' |
'`kappa' | '`mu' | '`nu' | '`xi' | '`pi' | '`rho' | '`sigma' | '`tau' | '`upsilon' | '`phi' | '`chi' |
'`psi' | '`omega' | '`Gamma' | '`Delta' | '`Theta' | '`Lambda' | '`Xi' | '`Pi' | '`Sigma' |
'`Upsilon' | '`Phi' | '`Psi' | '`Omega'

underline ::=
'_'

prime ::=
"'",

quote ::=
'"',

backslash ::=
'\'

graphic ::=
' ' | '!' | '#' | '$' | '%' | '&' | '(' | ')' | '*' | '+' | ',' | '-' | '.' | '/' | ':' | ';' |
'<' | '=' | '>' | '?' | '@' | '[' | ']' | '^' | '_' | '`' | '{' | '|' | '}' | '~'

escape ::=
‘\’‘r’ | ‘\’‘n’ | ‘\’‘t’ | ‘\’‘a’ | ‘\’‘b’ | ‘\’‘f’ | ‘\’‘v’ | ‘\’‘?’ |
‘\’‘\’ | ‘\’‘′’ | ‘\’‘″’ | ‘\’ oct_constant | ‘\’‘x’ hex_constant

oct_constant ::= oct_digit | oct_digit oct_digit | oct_digit oct_digit oct_digit

hex_constant ::=
hex_digit-string

oct_digit ::=
‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’

hex_digit ::=
digit | ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ‘A’ | ‘B’ | ‘C’ | ‘D’ | ‘E’ | ‘F’

ASCII Forms of Greek Letters

Greek letters, which may be used in identifiers, have ASCII forms as follows:

ASCII	Full	ASCII	Full
`alpha	α		
`beta	β		
`gamma	γ	`Gamma	Γ
`delta	δ	`Delta	Δ
`epsilon	ϵ		
`zeta	ζ		
`eta	η		
`theta	θ	`Theta	Θ
`iota	ι		
`kappa	κ		
		`Lambda	Λ
`mu	μ		
`nu	ν		
`xi	ξ	`Xi	Ξ
`pi	π	`Pi	Π
`rho	ρ		
`sigma	σ	`Sigma	Σ
`tau	τ		
`upsilon	υ	`Upsilon	Υ
`phi	ϕ	`Phi	Φ
`chi	χ		
`psi	ψ	`Psi	Ψ
`omega	ω	`Omega	Ω

Fixed Tokens

The representation of individual fixed tokens is given directly in the syntax rules for RSL. However, a representation using only ASCII characters is possible, as defined in the following table:

ASCII	Full	ASCII	Full	ASCII	Full
`><`	$\times$	`isin`	$\in$	`~isin`	$\notin$
`||`	$\parallel$	`++`	$+\!\!+$	`-\`	λ
`|=|`	$\square$	`|^|`	$\amalg$	`-list`	*
`**`	$\uparrow$	`-inflist`	$^\omega$	`~=`	$\neq$
`/\`	$\wedge$	`\/`	$\vee$	`+>`	$\mapsto$
`>=`	$\geq$	`exists`	$\exists$	`all`	$\forall$
`<=`	$\leq$	`union`	$\cup$	`!!`	$\dagger$
`inter`	$\cap$	`<<`	$\subset$	`always`	$\Box$
`-m->`	$\overrightarrow{m}$	`<<=`	$\subseteq$	`=>`	$\Rightarrow$
`-~->`	$\xrightarrow{\sim}$	`>>`	$\supset$	`is`	$\equiv$
`->`	$\rightarrow$	`>>=`	$\supseteq$	`<->`	$\leftrightarrow$
`#`	$\circ$	`<.`	$\langle$	`.>`	$\rangle$
`:-`	$\bullet$				

The word equivalents of certain symbols: `all, exists, union, inter, isin, always` are reserved, and cannot be used as identifiers.

RSL Keywords

The RSL keywords are listed below. They cannot be used as identifiers.

Keywords for RSL			
Bool	**class**	**inds**	**skip**
Char	**do**	**initialise**	**stop**
Int	**dom**	**int**	**swap**
Nat	**elems**	**len**	**then**
Real	**else**	**let**	**tl**
Text	**elsif**	**local**	**true**
Unit	**end**	**object**	**type**
abs	**extend**	**of**	**until**
any	**false**	**out**	**use**
as	**for**	**post**	**value**
axiom	**forall**	**pre**	**variable**
card	**hd**	**read**	**while**
case	**hide**	**real**	**with**
channel	**if**	**rng**	**write**
chaos	**in**	**scheme**	

APPENDIX D

Bibliography

[1] P. Behm, P. Lucas, S. Prehn, and H. Toetenel, editors. *VDM'91: Software Development, Tutorials and Project Reports; Proc. of Formal Methods Europe (formerly VDM-Europe) Symposium '91*, volume 2 of 2 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, Germany, 1991.

[2] D. Bjørner. *Abstract Software Specifications*, volume 86 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, Germany, 1980.

[3] D. Bjørner. *Software Architectures and Programming Systems Design; volume I: Specification Principles — the VDM Approach*. Addison-Wesley/ACM Press, 1991.

[4] D. Bjørner. *Software Architectures and Programming Systems Design; volume II: Implementation Principles — the VDM Approach*. Addison-Wesley/ACM Press, 1991.

[5] D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors. *VDM & Z — Formal Methods in Software Development, Proc. of VDM-Europe Symposium '90*, volume 428 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, Germany, 1990.

[6] D. Bjørner and C.B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, Germany, 1978.

[7] D. Bjørner and C.B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall International, 1982.

[8] D. Bjørner, M. Mac an Airchinnigh, E. Neuhold, and C.B. Jones, editors. *VDM – A Formal Method at Work, Proc. of VDM-Europe Symposium '87*. Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, Germany, 1987.

[9] D. Bjørner and O. Oest. *Towards a Formal Description of Ada*, volume 98 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, Germany, 1980.

[10] A. Blikle, H. Langmaack, H. Toetenel, and J. Woodcock, editors. *VDM'91: Software Development, Contributed and Invited Papers; Proc. of Formal Methods Europe (formerly VDM-Europe) Symposium '91*, volume 1 of 2 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, Germany, 1991.

[11] R. Bloomfield, L. Marshall, and R. Jones, editors. *VDM – The Way Ahead, Proc. of VDM-Europe Symposium'88*, volume 328 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, Germany, 1988.

[12] S. Brock and C.W. George. The RAISE Method Manual. Technical Report LACOS/CRI/-DOC/3, CRI: Computer Resources International, 1990.

[13] R.M. Burstall and J.A. Goguen. Putting theories together to make specifications. In *Proc. of (IJCAI) Int'l. Joint Conf. on AI*. Boston, Aug. 1977.

[14] R.M. Burstall and J.A. Goguen. The semantics of Clear: A specification language. *[2]*,

pages 292–332, 1980.

[15] R. de Nicola and M. Hennessy. CCS without τs. In *TAPSOFT '87. Volume 1. Proceedings*, pages 138–152. Springer-Verlag, Heidelberg, Germany, Lecture Notes in Computer Science, Vol. 249, 1987.

[16] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1, Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science, vol. 6, Springer-Verlag, 1985.

[17] L.M.G. Feijs. Norman's Database Modularised in COLD-K. In *Algebraic Methods II: Theory, Tools and Applications*, pages 205–231. Springer-Verlag, Heidelberg, Germany, Lecture Notes in Computer Science, Vol. 490, 1991.

[18] C.W. George and S. Prehn. The RAISE Justification Handbook. Technical Report LACOS/-CRI/DOC/7, CRI: Computer Resources International, 1991.

[19] J.A. Goguen. Some design principles and theory for OBJ-0. In *Lecture Notes in Computer Science, Vol. 75*, pages 425–471. Springer-Verlag, Heidelberg, Germany, 1979.

[20] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright. Abstract data types as initial algebras and correctness of data representations. In *ACM Conf. on Computer Graphics*, pages 89–93, May 1975.

[21] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology*. Prentice Hall, 1978.

[22] M. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF. In *Lecture Notes in Computer Science, Vol. 78*. Springer-Verlag, Heidelberg, Germany, 1980.

[23] J. Guttag, J.J. Horning, and J.M. Wing. Larch in five easy pieces. Technical Report 5, DEC SRC, Dig. Equipm. Corp. Syst. Res. Ctr., Palo Alto, California, USA, 1985.

[24] J.V. Guttag and J.J. Horning. The algebraic specification of data types. *Acta Informatica*, 10:27–52, 1978.

[25] I. Ørding Hansen and Jesper Jørgensen. Consolidated Meta-IV — abstract syntax. Technical Report RAISE/DDC/JJ/4/V2, Dansk Datamatik Center, 16. Dec. 1985.

[26] K. Havelund and R.E. Milne. The Semantics of RSL. Technical Report RAISE/DDC/KH/-43, CRI: Computer Resources International, 13. Sept. 1989.

[27] K. Havelund and K. Ritter Wagner. Kentrikos. Technical Report RAISE/DDC/KH/27, Dansk Datamatik Center, 1987.

[28] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[29] C.B. Jones. *Software Development: A Rigorous Approach*. Prentice-Hall International, 1980.

[30] C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.

[31] C.B. Jones. *Systematic Software Development — Using VDM, 2nd Edition*. Prentice-Hall International, 1989.

[32] C.B. Jones and R.C. Shaw. *Case Studies in Systematic Software Development*. Prentice-Hall International, 1990.

[33] H.B.M. Jonkers. Introduction to COLD-K. In *Algebraic Methods: Theory, Tools and Applications*, pages 139–205. Lecture Notes in Computer Science, Vol. 394, Springer-Verlag, Heidelberg, Germany, 1989.

[34] R.E. Milne. The implementation relation for the RAISE specification language. Technical Report RAISE/STC/REM/1, STC/STL, Harlow, UK, 1987.

[35] R.E. Milne. The sequential imperative aspects of the RAISE specification language. Technical Report RAISE/STC/REM/2, STC/STL, Harlow, UK, 1987.

[36] R.E. Milne. The concurrent imperative aspects of the RAISE specification language. Technical Report RAISE/STC/REM/7, STC/STL, Harlow, UK, 1988.

[37] R.E. Milne. The proof theory for the RAISE specification language. Technical Report RAISE/STC/REM/12, STC/STL, Harlow, UK, 1990.

[38] R.E. Milne. The semantic foundations of the RAISE specification language. Technical Report RAISE/STC/REM/11, STC/STL, Harlow, UK, 1990.

[39] R. Milner. *Calculus of Communication Systems*, volume 94 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, Germany, 1980.

[40] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[41] R. Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

[42] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Mass. and London, England, 1990.

[43] M. Nielsen, K. Havelund, K. Ritter Wagner, and C.W. George. The RAISE Language, Method and Tools. *Fomal Aspects of Computing*, 1:85–114, 1989.

[44] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.

[45] D.L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 14(5), May 1972.

[46] S. Prehn. From VDM to RAISE. In *[8]*, pages 141–150. Springer-Verlag, Heidelberg, Germany, 1987.

[47] S. Prehn and I. Ø. Hansen. Formal methods appraisal. Technical Report FMA/DDC/SP, Dansk Datamatik Center, 1983.

[48] D. Sannella and A. Tarlecki. Extended ML: an institution-independent framework for formal program development. In *Category Theory and Computer Programming*, volume 240 of *Lecture Notes in Computer Science*, pages 364–389. Springer-Verlag, Heidelberg, Germany, 1986.

[49] D. Sannella and A. Tarlecki. Toward formal development of ML programs: foundations and methodology. Technical Report ECS-LFCS-89-71, Department of Computer Science, Edinburgh Univ., Scotland, 1989.

[50] W. Turski and T.S.E. Maibaum. *The Specification of Computer Programs*. Intl. Comp. Sci. Series. Addison-Wesley Publishing Company, 1987.

[51] M. Wirsing. *A Specification Language*. PhD thesis, Techn. Univ. of Munich, FRG, 1983.

APPENDIX E

Index

The index is in four parts:

Symbols The symbols used to construct RSL value and type expressions.

Literals The RSL value and type literals.

Terms The main concepts used in the book, including the names of all the non-terminal symbols from the RSL grammar.

Examples The examples from the tutorial part that are named modules.

In each index page references in normal, roman type are to the tutorial part. Page references in bold type are to definitions, usually to be found in the reference part. (For the examples the roman/bold distinction indicates usages/definitions of the example modules.) Page references in sans serif type occur in the terms section, and refer to the syntactic definitions of non-terminals in the syntax summary in Appendix A.

Symbols

Literals

Terms

Examples