ANOTHER FORMAL SPECIFICATION LANGUAGE

Copyright ©2000 Erik Saaman
ISBN 90-367-1308-0
IPA Dissertation Series 2000-10

Typeset with LaTeX
Cover by Karen Saaman

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

RIJKSUNIVERSITEIT GRONINGEN

ANOTHER FORMAL SPECIFICATION LANGUAGE

PROEFSCHRIFT

ter verkrijging van het doctoraat in de
Wiskunde en Natuurwetenschappen
aan de Rijksuniversiteit Groningen
op gezag van de
Rector Magnificus, dr. D.F.J. Bosscher,
in het openbaar te verdedigen op
vrijdag 24 november 2000
om 14.15 uur

door

Erik Harald Saaman

geboren op 18 september 1964
te Dokkum

Promotores:	Prof.dr. G.R. Renardel de Lavalette
	Prof.dr. P. Klint
Beoordelingscommissie:	Prof.dr. P.D. Mosses
	Prof.dr.ir. L.M.G. Feijs
	Prof.dr. W.H. Hesselink

ACKNOWLEDGMENTS

I would like to express my thanks to all those who contributed in some way to this thesis. In particular I thank Gerard Renardel de Lavalette for his unconditional support and for creating the opportunity to do research and write this thesis; Paul Klint for his pragmatic, always cheerful, but competent supervision; Rix Groenboom for being partner in research and the main contributor to the design of AFSL; Metinna Veenstra for being a patient victim of my ideas about formal specification and innumerable different versions of AFSL; Jan Jongejan for being there from day one, giving valuable input about language design and tool support; Pietro Cenciarelli for making me aware of the existence of monads and their applications, and from then on disagreeing with every idea I had (which really helped); and the members of the reading committee who approved of the manuscript: Wim Hesselink, Lou Feijs, and Peter Mosses. I am also grateful to Louwarnoud van der Duim and Wim Liebrand for giving me the opportunity to finish this thesis while I have been working for ECCOO; Joost Visser for implementing the AFSL parser; Victor Bos, André Engels and Koen Hendriks who contributed as a student to the design of AFSL; and Karen Saaman for designing the cover. Indispensable were those who proof-read all or parts of this thesis: Gerard Renardel de Lavalette, Paul Klint, Rix Groenboom, Greetje Koers, Jan Jongejan, Jeroen Wouda and Bert Wiersema. Last but not least there is a large group of people whom I have to thank for their moral support and valuable comments: Corry de Groot, Tom Holkenborg, Hedda Werkman, Manon Breumelhof, Saskia Noordewier, Sanne Meeder, Daniël Lechner, Paul Swennenhuis, Bert Jan Bakker, Peter Fokkinga, Denise Walraven, Eelco Dijkstra, Eelco Visser, Arie van Deursen, Mark van den Brand, Merijn de Jonge, Tobias Kuipers, Jeroen Scheerder, Leon Moonen, Hans van Ditmarsch, Rudger Dijkstra, Willem Mallon, and Sietse Achterop.

Thanks!

INHOUDSOPGAVE

1. INTRODUCTION

Abstract

There are a number of reasons why software construction is an inherently hard process to master. Specification plays a central role here; therefore, better means of specification improve productivity. One way of achieving this may be the use of formal specification languages, which have the advantage of being unambiguous. This thesis proposes and motivates a new formal specification language called AFSL (Almost Formal Specification Language or Another Formal Specification Language). AFSL is not a full-grown tool that can be used productively in the software industry right away. Rather, its development has been a quest for creative ideas that may enhance the applicability of formal specification in the future. This introduction explains what formal specification is and argues why it is useful, discusses the historical background of AFSL, and gives an overview of the rest of the thesis.

1.1 Software Crisis

Poorly functioning computer software is nowadays probably the largest source of annoyance after traffic jams and bad weather. The most often heard complaints about software are that it is buggy, that it does not function adequately, that it is too expensive, and that it is delivered late. Of course, one can wonder whether these grievances are really very consequential; judging from the large amount of money spent on software, apparently it is worth it. However, it is clear that the public expects better achievement from the software industry. Many software engineering experts believe the development of software is a hard to control process for which there are no methods and techniques available (yet) (Brooks 1987). This state of affairs is often referred to as the software crisis.

It is very hard to analyze scientifically why it is so difficult to produce adequate software. The underlying mechanisms are hard to observe and it is not feasible to study the process in a laboratory. The reasons given in the literature are, at best, educated guesses. McDermid (1991) identifies five problems inherent in software development which I think many experts will agree with:

- Software is often too *complex* to be entirely understood by a single individual. We can try to manage complexity by dividing the system into subsystems, but, as systems grow, the interaction between subsystems increases non-linearly.
- It is notoriously difficult to establish an adequate and stable set of *requirements* for a software system. Often there are hidden assumptions, there is no analytic procedure for determining when the users have told the developers everything they need to know, and developers and users do not have a common understanding of terms used.
- The interaction between the different parts of a system makes *change* difficult.
- Software is essentially *thought stuff* (that is, the result of a thought process) and much of what is important about software is not manifest in the programs themselves (such as the reasons for making design decisions).
- A requirements specification for a system contains, perhaps implicitly, an *application domain model* (for example, describing the rules of air traffic). Development of application domain theories is very difficult.

All these aspects are directly related to written communication. Managing complexity depends on an ability to document the interfaces (parameters and functionality) of the modules involved. Requirements are an important reference for the whole process and should, therefore, be unambiguously and accessibly described for everyone. To keep track of changes it is important to document what exactly has been changed. Software can be made more visible by describing non-material artifacts, such as the overall design of a program. Domain models should, just like the requirements, be well documented. Thus, software engineering can benefit from good techniques to describe systems (programs, subsystems, etc.).

1.2 Specifications

For communication purposes, one is usually not interested in the internal details of a system. A *specification* is an abstract description of a system which focuses on what it does (or should do) rather than how it does it. The notion of "specification" is a relative one; it only indicates that it is used as an abstract description of something. A system can be specified as one black box (that is, only describing the relationship between the in- and outputs), but, during the realization of the system it may be divided into many subsystems which can, in turn, be specified as little black boxes. The specification of the little boxes is, in a way, an implementation of the big box, which is much more abstract. For example, a black box specification of a bridge is the statement "an artifact that enables people to cross water" (although that statement would

include boats). A specification at a more detailed level is a construction drawing of a bridge which describes the parts the bridge is built of (which would exclude boats). The drawing is an implementation of the black box, but is still an abstract description of a real bridge.

Thus, specifications can play many different roles as a means of communication in the software construction process: as contract between customer and builder, as documentation of design or implementation, as working paper during requirements analysis or design, as defining document for evaluation or standardization, as reference for testing or verification, etc. Writing specifications has the additional advantage that it forces one to think very carefully about the functionality of the system, which leads to a better definition of the problem.

There is general agreement among software engineers that specifications should be made during software development (which does not mean that it is actually being done). The alternatives are either using no documentation at all (using only verbal communication) or using program sources as non-abstract descriptions. The first option makes the process uncontrollable; the second option is not practical since sources contain too many irrelevant details and are hard to read by humans.

1.3 Specification Languages

Specifications are often written in natural language. This may not be the most suitable form of notation due to its extreme ambiguity in syntax and semantics, in which people can add their own terminology without precise definition. One has to be initiated into the culture and terminology of a certain domain before one can join in. Also, natural language allows only limited automated tool support and limited rigorous (mathematical) analysis. The advantage of natural language is that people already know it, which makes it the most natural language to use. The disadvantages can be eased by using a structured form of natural language which is limited to less ambiguous constructs. However, this sacrifices the naturalness of the language.

Another alternative is the use of mathematical notation, which is less ambiguous and allows rigorous analysis. The disadvantages of natural language also hold for the language of mathematics, although somewhat less because it is more precise. On the other hand, mathematical notation is harder to use because of the many exotic symbols, the often implicit notational assumptions, and the private enhancements many authors make.

Software engineering is using more and more graphical notations, such as flow diagrams, entity-relationship diagrams, and flow charts (see, for example, the Unified Modeling Language (Si Alhir 1998)). In principle, graphical languages can have a clear and unambiguous definition (although they often do not). Moreover, graphics seem to appeal to many people; one picture can often say more than a thousand words.

However, graphical languages usually have limited expressiveness and are aimed at particular aspects.

1.4 Formal Specification

In this thesis another type of language is investigated: a *formal specification language* is a formal language specially designed to be used for the specification of software. A *formal language* is an artificial language with well-defined syntax (defining what sentences are allowed) and semantics (defining what the meaning is of these sentences).

The advantages of formal languages are that they are unambiguous and explicit and that they can be supported by computer and by mathematical analysis. But all this has its price: staff has to learn the language (which most people find difficult) and formal specifications take more time to make (precisely because they are unambiguous and explicit). Therefore, industrial application of formal specification could turn out to be too expensive, but that is hard to tell. Case studies are often not representative: they are usually performed by personnel that is academically trained in formal methods, which is not the profile of the average software engineer. Glass (1999) shows that there is as yet no hard evidence that formal methods (or, in fact, any other method) are cost effective. No wonder that at this moment there is no consensus on the level of formality needed for specifications. Whether currently economically useful or not, formal specification does hold the promise of more effective communication which makes it worthwhile to investigate it.

The use of formal specification languages is part of a discipline called *formal methods*, which uses mathematical techniques to construct correct software. Bowens web site at `archive.comlab.ox.ac.uk/formal-methods.html` gives an extensive entry into the world of formal methods. Not all formal methods use formal languages; some use mathematical notation (which is not a formally defined language). I distinguish between two types of formal methods: *straight edge*, which aim at complete correctness proofs of software, and *light*, which only use formal languages as an unambiguous form of notation. The first requires mathematical manipulation by the user; the second does not. Of course, there can be mixtures between the two types. The work presented in this thesis belongs to the category of light formal methods. However, as mathematical analysis of software depends on sound specifications, straight edge formal methods may also benefit from the research presented here.

1.5 The FSA project

AFSL was developed as part of the Formal System Analysis (FSA) project of the department of Computing Science of the University of Groningen. FSA aimed at the

use of formal methods for the analysis of *open requirements* (defined by Blum (1993) as poorly understood and dynamic requirements). The project was inspired by experiences with the use of formal methods in the DETER project (Brookhuis & Oude Egbrink 1992). DETER was concerned with the development and testing of systems that influence driver behavior by giving feedback on traffic violations. The formal specification language COLD (Feijs, Jonkers & Middelburg 1994) was used for analyzing the requirements and implementation of the prototype system InDETER-1 (Saaman, Politiek & Brookhuis 1994, de Waard, Brookhuis, van der Hulst & van der Laan 1994). One of the motivations for the use of formal specification was the legal aspect which forced us to be very precise.

It turned out that InDETER-1 had open requirements (although we were not aware of that at the time). Even for relatively simple rules (such as traffic light violations) legislation is incomplete or ambiguous. Attempts were made to produce a formal specification; these did not succeed (they did, however, improve understanding of the requirements and a working system was delivered in time). There were two main reasons for this failure. First, COLD was a difficult language to master for which not much documentation or tools were available. Secondly, there was a lack of (public) guidelines on how a specification must be made from scratch, in particular if the requirements are open.

Based on these experiences we decided to develop a *specification method* based on principles of object-oriented analysis and looked for a formal specification language that would suit this method. During the DETER case I started experimenting with my own specification notation (which was not a real language yet). When no language was found that met the requirements of the FSA project (see Section 2.3), it was decided to expand the experimental notation to a formal specification language, which became AFSL. Although the original ambition of the project was not realized, it led to several valuable insights, which are discussed in this thesis.

1.6 Case Studies

During its development AFSL was used in two major specification case studies. The first project, *Formalization of ANesthesia* (FAN), was a cooperation between the Department of Anesthesia of the University Hospital in Groningen and the Department of Computing Science of the University of Groningen. The second project, *Minimalist Parser* (MP), was a joint activity with the Department of Computational Linguistics of the University of Groningen.

Prior to FAN, the feasibility of developing decision support systems had been investigated and some prototypes were built (de Geus & Rotterdam 1992, de Geus, Rotterdam, van Denneheuvel & van Emde Boas 1991). This led to the insight that both the limited degree of exactness and the complexity of the related knowledge

domain were major bottlenecks, which led to this project. FAN produced a formal specification in AFSL of the domain knowledge that is needed to construct a decision support system for anesthesiology (Groenboom 1997, Groenboom & Renardel de Lavalette 1996).

The *Minimalist Program* proposed by Chomsky (Chomsky 1992) aims at the development of a theory which explains how humans construct natural language sentences. A Minimalist Parser is a natural language parser based on the Minimalist Program. A formal specification of such a parser in AFSL was produced (Veenstra 1998), which in fact included the formalization of the underlying Minimalist theory. Although this theory is rather structured and incorporates some mathematical notions (such as trees), it is nowhere described formally (although Stabler (1992) gives a formal specification of an earlier theory of Chomsky) and many details are left implicit or still open.

These two cases provided useful feedback and new ideas for the final version of AFSL. Some features were added only after the cases were finished and, therefore, have unfortunately not been tested in any sizable application. It has been noted by van Deursen (1994) that language and tools should be developed simultaneously in order to benefit from the resulting insights into the nature of the language. In parallel with the development of AFSL, a parser, type checker, and signature diagram generator were implemented. As expected, the design of AFSL benefited from this; the syntax, type system, and module mechanism especially improved.

1.7 Results

AFSL is a case study in language design which combines a number of features that are essential for a broad application of formal specification in a simple language. It is the combination of features which is important here. Too often language features are studied in isolation without the guarantee that they can be combined. The design of AFSL is based on experiences gained in the two major case studies in which it has been used (see previous section). Important qualities of AFSL are that it features a simple first-order algebraic foundation (all other features are defined as an extension of this kernel), that natural language is allowed in informal definitions, and that implicit functions model subsorts and inheritance. But the main innovation of AFSL is the way it models non-functional features of operations. These are features that go beyond the functional input/output behavior of operations, such as exceptions, state changes, nondeterminacy, and communication with the environment. Non-functional features are modeled within AFSL in a way similar to the use of Monads in Haskell (Wadler 1993), although AFSL is a first-order language. At first, this results in considerable notational overhead, which, however, can be reduced by the use of a mechanism called application redirection which is introduced in this thesis.

1.8 Thesis Overview

Chapter 2 discusses the general principles of formal specification languages (which includes related work) and the FSA method. Based on the needs for the FSA project, requirements are formulated for a specification language and it is discussed why it was decided to design a new language. An overview is given of the features of AFSL. The subsequent chapters give the detailed definition of AFSL, demonstrated by many examples. These chapters can be read without Chapter 2. Chapter 3 describes the basic language, Chapter 4 describes generic modules (that is, parameterized modules), Chapter 5 describes implicit functions (that is, subsorts and inheritance), Chapter 6 describes the modeling of non-functional features (such as partial functions and state changes), and Chapter 7 describes lifting and application redirection (the technique used to reduce the notational overhead of non-functional features). Finally, Chapter 8 discusses the design choices made in AFSL, how AFSL meets the initial language requirements of Chapter 2, and gives possible directions for future research.

All AFSL specifications included in this thesis are online at:

```
www.eccoo.rug/~Erik/AFSL/Specs
www.eccoo.rug/~Erik/AFSL/Expanded
```

The first directory contains the complete specifications, including parts that are left out in the thesis. All these modules have been checked by a prototype type checker (see Section 8.9). The second directory contains the expanded versions of the specifications (although, the origins of names have been omitted in order to make the expansions better readable).

2. AFSL RATIONALE

Abstract

This chapter motivates the design of AFSL. First, relevant properties of formal specification languages in general are discussed. Then an outline is given of the Formal System Analysis (FSA) specification method. Based on the discussion of language properties and specification method, criteria for a formal specification language are given. When the FSA project started there was no language that satisfied these criteria; therefore, a new one was designed, which became AFSL. An overview is given of its features.

2.1 Aspects of Formal Specification

In principle any formal language, provided it is expressive enough, can be used for specification. For example, a simple logical language like first-order predicate logic or a declarative programming language like Prolog are possible candidates. However, specifying in such non-specific languages will often be needlessly complicated and may not fit well into the overall development process. For example, predicate logic is so rudimentary that specifying in it is a bit like writing programs in assembler. Therefore, tailored languages should be used instead. Some aspects that can be dealt with in a specialized language are data structures, typical software aspects (such as input/output), prototyping, modularity, software development method, tool support, and correct implementation.

Although formal specification languages vary widely in detail, their general structure is similar. In this section a general description is given of these languages in order to explain why it was decided to design a new language. This discussion is limited in scope and does not fully appreciate the individual qualities of the different languages. However, it is useful to view all specification languages roughly as an instance of the same underlying concept. Research has been done in formulating a formal framework of so-called *institutions* that captures the general properties of formal languages (Goguen & Burstall 1992, Tarlecki 1999), but institutions are too theoretical for the current context.

A formal specification language can roughly be seen as a programming language

```
module Booleans

imports Layout

exports
  sorts BOOL
  context-free syntax
    true          -> BOOL {constructor}
    false         -> BOOL {constructor}
    BOOL and BOOL -> BOOL {left}
    BOOL or BOOL  -> BOOL {left}
    not BOOL      -> BOOL
    "(" BOOL ")"  -> BOOL {bracket}
  priorities
    not > and > or
  variables
     P -> BOOL
equations
  [1]  true  or  P = true
  [2]  false or  P = P
  [3]  true  and P = P
  [4]  false and P = false
  [5]  not true    = false
  [6]  not false   = true
```

Fig. 2.1: Specification of booleans in ASF+SDF.

```
COMPONENT ARRAY SPECIFICATION
CLASS
  SORT Array  VAR

  FUNC value : Array # Nat -> String  VAR

  PROC create : -> Array
  OUT  a
  PRE  True
  SAT  NEW Array
  POST FORALL n:Nat (value(a,n)^)

  PROC update : Array # Nat # String ->
  IN   a,n,s
  PRE  True
  SAT  MOD value(a,n)
  POST value(a,n) = s
END
```

Fig. 2.2: Specification of arrays of strings in COLD (assuming `Nat` is the sort of natural numbers and `String` of strings). (The operation ^ denotes undefinedness of a term.)

in which definitions of operations may not be executable (that is, cannot be compiled or interpreted). Each formal specification consists of a collection of *declarations*, which determine which names (that is, identifiers) and variables (that is, wild cards that can be used as placeholders in definitions) can be used, and a collection of *definitions*, which determine the properties of the names. Example specifications are given in Figure 2.1, which specifies booleans in ASF+SDF (Bergstra, Heering & Klint 1989, Klint 1993), and Figure 2.2, which specifies mutable arrays of strings in COLD (Feijs et al. 1994, de Bunje 1992).

The rest of the current section discusses different aspects of formal specification languages: the basic concepts of formal specification that can be found in (almost) any language; variations that apply to some languages; semantics; structuring; and application domain.

2.1.1 Basic Concepts

Sorts and Operations

The collection of all declarations of names in a specification is called its *signature*. There are at least two kinds of names: *sort names* (`BOOL` and `Array` in the examples) and *operation names* (`true`, `and`, `value`, `create`, etc. in the examples).

Names have a *value*. The value of a sort name is a *sort*, which is a collection of

objects. Here the term "object" is used without any specific object-oriented meaning. The value of an operation name is an *operation*, which, given a number of objects (the *arguments*), returns a *result* (which is an object) and does "something" (for example, `value` *fails* if an array does not have a value for the given index, `update` *changes* the contents of an array).

Sorts are sometimes called types. Some languages do not have sorts, which is equivalent to having just one sort of all objects (such languages are called *single-sorted*, as opposed to *many-sorted*).

The notion of operation as used here includes the concepts of functions, attributes, methods, etc. that some languages have. These can all be seen as operations of some kind. Some languages allow operations that do not have a result, but this can be mimicked by returning any arbitrary result which is subsequently ignored (for example, a special "void" object which cannot be used for anything). Many languages have *constants* (that is, object names), but these can be mimicked by operation names with zero arguments. For example, in Figure 2.1 `true` and `false` are nullary operations that return a boolean value.

In this thesis the aspects of an operation which go beyond simply returning a result are called *non-functional features* (such as state dependency, side-effects, and raising error conditions). Our definition of "non-functional" is different from the usual one which refers to properties that are not directly implemented in a system, such as reliability, efficiency, and price. In this thesis non-functional features are assumed to be formal properties of operations. Usually specification languages allow a fixed number of non-functional features, such as partial functions (Barringer, Cheng & Jones 1984, Cheng 1986, Cheng 1990, Jones 1986) and state-changing operations in COLD (Feijs & Jonkers 1992, Feijs et al. 1994). As far as I know, there are no specification languages (except AFSL) that allow the user to specify non-functional features.

Variables

Variables have an arbitrary value; they are used in specifications at positions where any value may be substituted (like variable `n` in `n+0=n`). This kind of variables is sometimes called *logical variables* in order to distinguish them from *program variables* (found in imperative programming languages) whose values are stored somewhere and can be read and updated.

Terms

From operations and variables, *terms* can be built, which are either variables or operation calls (with terms as arguments). Given a value for the operations and variables,

a term has a value and does have some non-functional features (the accumulation of the non-functional features of the operations in the term).

Typing

Many-sorted languages are usually *typed*, similar to programming languages. That is, every operation has a so-called *type* (or *arity*) which determines the number of arguments, the sorts to which the arguments must belong, and the sort to which the result belongs. Also, in a typed language every variable has a type, which is a sort that determines the domain the value of the variable ranges over. The two example specifications in Figures 2.1 and 2.2 are typed. For example, `and` has two arguments of sort `BOOL` and `update` has three arguments of sorts `Array`, `Nat`, and `String`. An operation call is *well-typed* if it has the correct number of well-typed arguments. In a typed language it is required that all operation calls are well typed.

Usually well-typedness can be checked automatically by a *type checker* so that some semantic errors (such as adding strings to numbers) are detected easily. Another advantage of a typed language is that the type of an operation gives a clue about its meaning. For example, what can we expect from an operation `+` that takes two numbers as arguments and returns a number? A disadvantage of a typed language can be that it is too restrictive. For example, the lack of polymorphism in AFSL turned out to be a limitation (see Section 8.5). Most specification languages are typed, although there is some discussion about the necessity of this (Lamport & Paulson 1999).

Definitions

Definitions determine the meaning of the names declared in the signature. Here the term "definition" is used liberally and includes ambiguous definitions (*under-specification*) and inconsistent definitions (*over-specification*). That is, definitions restrict the possible meanings of the names rather than determine them. There are different ways in which definitions can be formulated.

The most basic form of definition is by using *axioms*, which are logical assertions that have to be satisfied. For example, in Figure 2.1 the boolean operators `or`, `and`, and `not` are defined by listing a number of equations that must be satisfied. COLD also allows axioms, but these are not used in the example given here. In principle the use of axioms provides enough definitional power, provided the language has the appropriate logical primitives. Most languages allow axioms in some form, but many also have other definition constructs.

One frequently occurring form of definition is that of *attributed declarations*, which is a declaration that also contains some information about the meaning of the declared name. In Figure 2.1 the attribute `constructor` denotes that `true` and `false` are the constructors of the sort `BOOL` (that is, the only elements of `BOOL` are

`true` and `false`). In Figure 2.2 the attribute `VAR` denotes that `Array` and `value` are variables (that is, they are state dependent). These are examples of simple attributes that consist of only one singular statement. More complicated forms can be found in the declarations of `create` and `update` in Figure 2.2 where the `PRE` parts are the preconditions for successful execution, the `SAT` parts indicate the modification rights, and the `POST` parts restrict the allowed states after execution.

2.1.2 Variations

Functions

An operation is called a *function* if the only thing it does is mapping arguments to a result (according to some input/output relation) without any additional non-functional features. For example, the operations in the ASF+SDF specification of Figure 2.1 are all functions. A language in which all operations are functions is called *functional* (such as a functional programming language).

Predicates

In addition to sort and operation names, many specification languages distinguish *predicate names*. For example, $\leq$ and `ordered` in the specification of Figure 2.5 on page 25 are predicate names. Like functions, a predicate can be applied to a number of arguments and does not have non-functional features. But predicates do not return an object as result; instead, for given arguments, a predicate is either valid or not. For example, $1 \leq 2$ is valid and $2 \leq 1$ is not. Being valid or not can be seen as a *logical value*, but that is something different than an object. Logical values are not elements of a sort and cannot be arguments of operations or other predicates.

Having logical values also introduces the need for *logical operations*, which are similar to predicates but have logical values as arguments (or a mixture of logical values and objects). For example, there can be logical operations for "or", "implies", and "if ... then ... else ... ".

Predicates model relations between objects. Those arguments for which a predicate is valid are related; the rest is not. For example, 1 and 2 are $\leq$-related, 2 and 1 are not. Predicates are often called relations, but here we distinguish the concept "relation" from the mathematical notion "relation". Relations are an important concept for modeling software. We cannot do without relations, but we can do without predicates. Relations can also be modeled by functions that return boolean results, where the booleans are the *objects* "true" and "false". For example, if $\leq$ is a boolean-valued function, then $1 \leq 2$ has value "true" and $2 \leq 1$ has value "false". Contrary to logical values, booleans are objects and can be arguments of operations. The distinction between predicates and boolean-valued functions may seem artificial and irrelevant.

However, it allows functions and predicates to be treated differently in some respects, two important examples of which are given here.

First, one can allow partial functions but forbid partial predicates. The result of a partial boolean-valued function may be undefined, which introduces a kind of three-valued logic ("true", "false", and "undefined") or even four-valued (Bergstra, Bethke & Rodenburg 1994). A three-valued logic is not as straightforward as a two-valued logic. For example, should the value of "true or undefined" be "true" or "undefined" (see Cheng (1986) and Astesiano, Kreowski & Krieg-Brückner (1999) for different approaches to three-valued logic). Even if boolean-valued functions are forbidden to be partial (which would introduce a special class of functions), there is a problem. Most languages with partial functions are strict in the sense that the result of a function is undefined if one of its arguments is undefined. Thus, even a total boolean-valued function may have an undefined result.

Second, one can define a language so that by default a predicate is not valid unless defined otherwise (this is similar to the closed world assumption of Prolog (Sterling & Shapiro 1994)). This allows concise and inductive definition of predicates (CoFI Task Group on Language Design 1997). For example, the less-than-equal operation `<=` on natural numbers can then be defined by:

```
n <= n
n <= m+1 <== n < m
```

Without a default value for predicates it must be stated explicitly for which arguments `<=` is not valid. This is not only less concise, but some languages (for example, Prolog) do not even allow such an explicit definition.

Default predicate values is based on initial semantics (see Section 2.1.3) where a predicate p_1 is smaller then p_2 if and only if p_1 is valid for fewer arguments then p_2. Initial semantics selects the smallest possible interpretation of the declared names (the initial model) as *the* interpretation. In order to guarantee that such an initial model exists, the kind of definitions the language allows must be restricted. To illustrate this, consider the example definition of predicate `P`:

```
P(1) <== Not P(2)
P(2) <== Not P(1)
```

Here it is not defined whether `P(1)` and `P(2)` are valid or not, but they cannot both be invalid at the same time. This makes the default definition of `P` (not valid for all arguments) impossible. Therefore, a language that uses default values should forbid definitions like the above, which could be too much of a compromise.

Using boolean-valued functions instead of predicates keeps a language simpler. The syntax and semantics do not have to consider three different forms of operation names (normal operations, predicates, and logical operations) and two forms of expressions (object valued and logical valued). Moreover, distinguishing logical values

usually makes a language less flexible because all logical operations are builtin and cannot be declared by the specifier. This severely limits the adaptability of the language and puts a burden on its definition, since sufficient logical operations should be builtin.

Advanced Type Systems

There are different typing mechanisms. The most common special features are higher-order typing, polymorphism, and sub-sorting.

Higher-order typing allows arguments of operations to be themselves operations. This means that operations are considered to be objects, accompanied by special sorts (for example, $s_1 \rightarrow s_2$ as the sort of operations from s_1 to s_2). A language that does not allow higher-order typing is called *first-order*.

Polymorphism allows an operation type to contain sort variables that can be instantiated by any arbitrary sort. The term "polymorphism" is sometimes used in a broader sense for "having more than one type", which includes overloading and inheritance. In this thesis it only refers to the possibility to instantiate sort variables.

Sub-sorting allows one sort s_1 to be declared as a *subsort* of another sort s_2. If an operation expects an argument of s_2 it is allowed to give an argument x of the subsort s_1. This mechanism is called *inheritance*, since s_1 inherits the possibilities of s_2. An *implicit conversion* from s_1 to s_2 is applied to x to make it fit. For example, lists can be seen as a subsort of sets, where the conversion function transforms a list to a set by putting all the elements of the list in the set. It is possible that the conversion function is the identity function (that is, no conversion is necessary), which amounts to s_1 being a subset of s_2. Depending on the language the specifier may or may not define the conversion function explicitly. For example, ASF+SDF has the possibility to define a special kind of function from s_1 to s_2 called an *injection* which is an "invisible" unary function (that is, it is denoted by an empty name). Some languages may put restrictions on the definition of the conversion functions: for example, that they should be injective. A potential problem of inheritance is that it causes ambiguities if there are multiple ways to convert an argument from s_1 to s_2 (multiple inheritance).

Syntax

Some languages use only ASCII symbols; others also allow mathematical symbols like $\forall$ and $\neg$ (for example, in Figures 2.4 and 2.5). Mathematical symbols make specifications more readable for those who are used to these symbols. However, specifications should be readable by people who are not mathematically trained (such as programmers); therefore, it may be better to use the more verbose `Forall` and `Not` instead. Moreover, since most text editors do not support mathematical symbols they have to be entered using special symbols like `\forall` and `\neg`. This leads to two

different presentations of the same specification: the ASCII version (the file being edited) and the formatted version, which can cause confusion.

The standard notation for operation calls consists of the operation followed by a list of arguments (that is, $f(x_1, \ldots, x_n)$, for operation f and arguments x_1, ..., x_n). Other common notations, which are not supported by all languages, are the *infix notation* ($x_1\ f\ x_2$), *prefix notation* ($f\ x_1$), and *postfix notation* ($x_1\ f$). A very liberal syntax for operation calls that is allowed by some languages is the *mixfix notation* where the name of the operation consists of a number of separate components, and the arguments are written between these components. An example of mixfix notation is:

```
if n>0 then n-1 else 0
```

in which `if␣then␣else␣` is the operation and `n>0`, `n-1`, and `0` are the arguments.

The "fix" notations (in particular the mixfix notation) allow specifications which are closer to the texts one is used to (either in mathematics or some other language). However, they can easily introduce syntactic ambiguities. That is, some expressions can be parsed in more than one way. For example, `9+5/2` can be read as both `(9+5)/2` and `9+(5/2)`. These ambiguities can, in many cases, be solved by user-declared priorities and associativities. For example, in Figure 2.1 `and` has a higher priority than `or` (thus, `P or Q and R` is parsed as `P or (Q and R)`) and both are left associative (thus, `P or Q or R` is parsed as `(P or Q) or R`).

Overloading

Some languages allow name *overloading*: that is, the same name may denote more than one value. Overloading may lead to ambiguities which are usually resolved by using context information, such as the number and type of arguments of an operation. The advantages of overloading are that it allows one to give similar things the same name (intended overloading) and that it is easier to combine specifications that were written independently (accidental overloading). The disadvantage is that it can be confusing, even if there is some procedure to resolve ambiguities.

Lean languages

A *lean language* is a small language that only has a minimal set of base constructions; more complicated concepts have to be defined within specifications themselves. Most algebraic languages are lean, most model-oriented languages (see Section 2.1.3) are not (I do not know why). The main advantage of a lean language is that its definition is simpler. Also, tools can be simpler because they have to consider fewer language constructions. Moreover, keeping a language lean is a good test to see whether the

language is expressive enough. A disadvantage of a lean language is that some things have to be formulated more elaborately.

There is a trade-off between economy of language design and economy of notation. In order to avoid verbose, unreadable specifications, a lean language should have appropriate facilities for extending the base constructions, such as parameterized modules, higher-order operations, type polymorphism, and mixfix notation. For example, higher-order operations allow the definition of quantifiers and mixfix notion makes it possible to have an `if then else` construction.

Kernel Languages

Some languages are defined by giving a syntactical translation to a smaller *kernel language*. For example, COLD-1 is defined by translating it to the kernel COLD-K (Feijs, Jonkers, Koymans & Renardel de Lavalette 1987). The kernel can be a sublanguage, but can also be a completely independent language (similar to translating a higher programming language to assembly language). A kernel language is not the same as a lean language. The derived concepts are translated to the base constructs of a kernel language rather than defined in terms of the basic constructs of a lean language. The translations are part of the language definition (tools, for example, need to be aware of that), whereas the definitions are part of a specification. Still, the advantages are similar as long as the translation process is kept simple.

2.1.3 Semantics

Models

A possible interpretation of all names declared in a signature can be represented by an *algebra*, which is a mapping from each declared name x to a mathematical representation of the entity y denoted by x. What exactly the structure of y is depends on the kind of specification language we are dealing with. If x is a sort then y is usually a set. If x is an operation then y contains a mapping from arguments to results together with some representation of the non-functional features of the operation. For example, text output can be modeled by linking an output string to the result and non-determinism can be modeled by mapping arguments to a set of possible results instead of one single result.

A *model* is an algebra that satisfies all definitions of a specification. Thus, a specification defines a class of models, which is what is actually being specified by a formal specification: a number of possible interpretations of declared names. The notions “algebra” and “model” are not always explicitly used in the definition of a specification language; still, they are useful concepts to explain formal specification in general.

Formal specification can be seen as a means of defining terminology. As such it can be very useful to support communication during software development. This is the way AFSL was used in the Minimalist Program case study discussed in Section 1.6. But, a formal specification can also specify software and hardware more directly. An algebra can be seen as an abstraction of an implementation. Thus, a specification defines a class of correct implementations (those corresponding to the models of the specification). There is no general definition of what it exactly means for a system to be an implementation of an algebra. One has to agree on that before a specification can be used as a reference for implementation. Here two common ways are discussed in which algebras can represent systems.

The simplest way to view an algebra as an abstract implementation is using functions as black box descriptions of a system. For example, a digital switch with n inputs can be specified by a function name `Switch` with n boolean arguments (representing the inputs) and a boolean output (representing the output). The models of the specification of `Switch` determine the allowed input/output behavior of the switch. A more complicated system that processes n streams of digital information (that is, the output depends on the complete history of received bits, not only the most recent inputs) can be specified by a function `Process` with n lists of booleans as argument (representing the received bits up to some moment t) and returns as a result a list of bits (representing the transmitted bits up to t).

Strictly speaking we have to be more precise about the way the abstractions model physical reality. For example, how do booleans relate to the input/output voltages of the switch/processor? In many cases this is either implicitly understood (engineers already use the boolean abstraction for digital signals) or not relevant (for example, when building the switch/processor from basic digital ports). However, this is a point that can easily be overlooked. If the relationship between abstraction and reality is not clear a formal specification will not be interpreted correctly, no matter how formal the language used is.

A black box specification does not describe the internal structure of an implementation, only its observable behavior. Often it is necessary to give a more direct description of the program code that implements a system: for example, to document the design of the implementation. In that case an implementation is seen as a *refinement* of an algebra. Detailed treatment of refinement can be found in van Leeuwen (1990), McDermid (1991), and Astesiano et al. (1999); here only a rough sketch is given.

A program $\mathcal{P}$ is a refinement of an algebra $\mathcal{A}$ if for each object of $\mathcal{A}$ there is a data structure in $\mathcal{P}$ representing that object (for example, a number can be represented by a sequence of bits) and if for each operation f of $\mathcal{A}$ there is a procedure (or function) p in $\mathcal{P}$ that for inputs representing objects $x_1, \ldots, x_n$ computes a data structure that represents the result of f for arguments $x_1, \ldots, x_n$. Moreover, the non-functional

features of p should be in accordance with those of f. It fully depends on the nature of the non-functional features what that means. For example, the implementation of string output of f could mean that p should output a string representation to some buffer.

A notion of "refinement" can be formally defined if the programming language is itself a formal language. However, such a definition is not part of the definition of the specification language (unless the programming language is part of the specification language), but belongs to the agreement about the use of the specification language.

Operational vs Declarative Semantics

A specification language with *operational semantics* is in fact a programming language (although it may still be designed especially for specification purposes). Operational specifications define data structures that represent objects; its definitions of operations are prescriptions for computing the result and non-functional features for given arguments. In specifications written in a language with *declarative semantics*, definitions somehow describe the values of names without determining how objects are represented and operations are computed. A specification language can have both operational and declarative semantics (similar to the programming language Prolog (Sterling & Shapiro 1994)), but most specification languages have only one.

The advantage of a language with operational semantics is that specifications can be used as prototypes in order to validate definitions (do they define what they ought to). There is discussion about the necessity of operational semantics (see, for example, Hayes & Jones (1989)). Executability limits the expressiveness of a language, since not all definitions are executable. For example, the definition of $-$ by the equation $(x - y) + y = x$ will generally not be executable (although that may depend on the cleverness of the interpreter/compiler). In general, executable specifications will contain details that are redundant from a pure specification point of view.

Model-Oriented vs. Property-Oriented Languages

There are two types of declarative languages: model-oriented and property-oriented (Monahan & Shaw 1991, Barwise 1989). In a *model-oriented language* objects are defined as structures built from a number of given basic mathematical structures (such as sets, tuples, and sequences) and the result (and non-functional features) of operations are explicitly defined in terms of these structures. In a *property-oriented* language the internal structure of objects is not described (they are black boxes) and operations are implicitly defined by listing a number of properties (axioms). An example of a model-oriented specification is the definition of a database table as a list of key/value pairs. A property-oriented specification would characterize a table in

terms of its update and lookup operations. A model-oriented specification is like a scale model, whereas a property-oriented specification is like a construction drawing. Examples of model-oriented specification languages are VDM (Jones 1986) and Z (Diller 1994); property-oriented specification languages are OBJ (Goguen & Winkler 1988) and CASL (CoFI Task Group on Language Design 1999).

Model-oriented specifications contain unnecessary representation details that blur the view of the properties that are actually being specified. Moreover, there is the danger of implementation bias, since the model can be used as a starting point for the implementation. On the other hand, model-oriented definition gets properties for free from the basic structures being used. For example, in a model-oriented specification it is clear that adding the same value twice to a table results in the same table, a property-oriented specification needs extra information for that. Also, model-oriented specifications are more concrete and, therefore, may for some be easier to understand. I am not aware of any research about the cognitive pros and cons of the two types of specification (that would indeed be interesting research).

Initial Semantics

In order to avoid a multitude of axioms some property-oriented languages only allow *initial algebras* as models. These are algebras in which two objects are equal if-and-only-if this can be derived from the axioms, and sorts may only contain objects that can be constructed by the operations of the signature. That is, there never can be confusion about the equality of objects and there are no ghost objects that cannot be named.

The advantage of initial semantics is that the properties satisfied by initial algebras do not have to be axiomatized (they hold by default). A complication of initial semantics is that axioms have to conform to some restrictions in order to guarantee that there exist initial algebras that satisfy them. For an extended treatment of initial semantics see van Leeuwen (1990) and Astesiano et al. (1999). A language that allows any algebra as a model is said to have *loose semantics*.

2.1.4 Structuring Concepts

Modules

Just like computer programs, specifications can become very large (thus, hard to handle) and share common components between projects (which calls for reuse). Therefore, it is worthwhile to split specifications into smaller modules which are easier to manage and reuse. Abstract modules that can be instantiated in different ways should also be possible. The advantages of modularity and abstraction for formal specification are not discussed here; they are the same as for programming (see, for example, McDermid (1991)). Virtually all specification languages have a module mechanism

```
module Sequences
begin
 parameters Items
  begin
   sorts     ITEM
   functions eq   : ITEM # ITEM -> BOOL
  end Items
 exports
  begin
   sorts     SEQ
   functions null :             -> SEQ
             seq  : ITEM # SEQ -> SEQ
... etc ...
end Sequences

module Strings
begin
  imports Sequences { renamed by [SEQ -> STRING, nul -> null-string]
                      Items bound by [ITEM -> CHAR, eq -> eq]
                      to Characters
                    }
... etc ...
end Strings
```

Fig. 2.3: Example of a parameterized specification of sequences in ASF, where formal parameters are specifications, together with an example import that defines strings as sequences of characters. It is assumed `Characters` is some module that specifies sort `CHAR` and function `eq`.

and most of them have a form of parameter abstraction. There are two main types of parameter mechanisms: the formal parameters are either specifications or names (that is, sorts and operations).

An example in which the module parameters are specifications is the specification of strings as sequences of characters given in Figure 2.3 in ASF, which is taken from Bergstra et al. (1989) where ASF is introduced. Note that ASF+SDF, the successor of ASF, does not support this parameter mechanism. The formal parameter of the module `Sequences` is the abstract specification `Items` which is (within `Sequences`) being extended with `SEQ`, `null`, and `seq`.

For some languages (for example, ASF) parameters like `Items` may only contain declarations; others (for example, CASL used in Figure 2.5) also allow axioms in parameters which restrict the actual parameters that can be substituted for the formal parameter. Some languages allow formal parameters to be specified within the module (like `Items` in the example); others use a reference to an external module (like `PO`

```
Bag(E): trait
  introduces
    { }              :       → B
    insert, delete : E, B → B
    __ ∈ __          : E, B → Bool
  asserts
    B generated by { }, insert
    B partitioned by delete, ∈
    ∀ b: B, e, e': E
      ¬ (e ∈ { })
      e ∈ insert(e',b)        == e = e' ∨ e ∈ b
      delete(e,{ })           == { }
      delete(e',insert(e,b)) == if e = e'
                                  then b
                                  else insert(e,delete(e',b))
```

Fig. 2.4: Example of a specification parameterized by names in LSL. `Bag` specifies the sort `B` of bags with elements of module parameter `E`. Note that here sorts `E` and `B` are declared implicitly by their occurrence in other declarations.

in Figure 2.5).

Within the module `Strings` the import of `Sequences` supplies `Characters` as actual parameter `bound to` formal parameter `Items`. Because the names in the signature of the formal parameter are often different from the corresponding names in the actual parameter they are being renamed. Here `ITEM` is renamed to `CHAR` and `eq` to (a different instance of) `eq`. Further explanation about this type of parameter mechanism can be found in Orejas (1999) and Wirsing (1990).

Although the specifications-as-parameters mechanism is the most common (in particular in theory about formal languages), there is yet another form, where the formal parameters of a module are names (sorts, operations, etc.). An example is the specification of bags in the Larch Shared Language (LSL) given in Figure 2.4, which is taken from Guttag, Horning & Modet (1990). When `Bag` is imported names have to be supplied as actual parameters. For example, a specification of bags of integers can be (assuming module `Integer` declares the sort `Int`):

```
IntegerBag: trait
  includes
    Integer, Bag(Int)
```

The net effect of the import of `Bag` is that its declarations and axioms are included in `IntegerBag` *after* `E` is replaced by `Int`. In LSL `Bag(Int)` is just shorthand for the renaming `Bag(Int for E)`.

The names-as-parameters mechanism thus is a convenient alternative to renaming. The advantages are that the imports are less verbose (only the new names need

to be listed) and renaming is done consistently (the same names are always renamed and in the same order).

Compound Names

An overloading problem occurs if the same parameterized module is imported more than once (independently of the type of parameter mechanism). For example, if the module `Sequences` of Figure 2.3 is imported twice, once with `Characters` and once with `Integers` as actual parameter, there will be two conflicting declarations of `SEQ`. Apart from that, the name `SEQ` is not a very descriptive one for either lists of characters or lists of integers. Therefore, imports are often combined with renamings of imported names. For example, in the import of `Sequences` the sort `SEQ` is renamed to `STRING`. An import of `Sequences` with actual parameter `Integers` could rename `SEQ` to `INTLIST`, thus avoiding overloading of `SEQ` and introducing more descriptive names.

Having many renamings can be laborious. This can partly be avoided by allowing overloading of operations (using the type of their arguments to resolve overloading). Such a mechanism cannot, however, be used for sorts (such as `SEQ`). Another way to avoid renaming of an overloaded name x is to add the actual parameter of the import that declared x. For example, distinguish two instances of `SEQ` by writing `SEQ[Character]` and `SEQ[Integer]`. ASF does not actually support this, but, for example, COLD (Feijs et al. 1994) does. A third way to avoid renamings of imported names is the use of compound names.

An example of the use of compound names is the specification of ordered lists in CASL given in Figure 2.5 taken from Sannella & Wirsing (1999). Here sort name `Elem` (declared in the parameter specification `OP`) is an *index* of *compound names* `List[Elem]` and `OrdList[Elem]`. The import:

```
ORDLIST[NAT fit Elem |-> Nat, __ ≤ __]
```

causes the declaration of sorts `List[Nat]` and `OrdList[Nat]`, because the index `Elem` has been replaced by the corresponding name `Nat` of the actual parameter. Another import of `ORDLIST` which, for example, binds `Elem` to `Char` would declare distinct sorts `List[Char]` and `OrdList[Char]`. The compound name mechanism is very similar to adding the formal parameters to imported names. But it is somewhat more selective, because one can freely determine which (names declared in) parameters are used as indices.

Hiding

Many languages with modules have the option to selectively export name declarations (hiding the other names for importing modules). This is motivated by the idea that

```
spec PO =
  type Elem
  pred __ ≤ __ : Elem * Elem
  vars x,y,z : Elem
  axioms x≤x ;
         x≤y ∧ y≤x ⇒ x=y ;
         x≤y ∧ y≤z ⇒ x≤z
  end

spec ORDLIST[PO] =
  free type List[Elem] ::= nil | __::__ (hd:?Elem; tl:?List[Elem])
  pred ordered : List[Elem]
  vars a,b : Elem; l : List[Elem]
  axioms ordered(nil) ;
         ordered(a::nil) ;
         a≤b ∧ ordered(b::l) ⇒ ordered(a::(b::l))
  type OrdList[Elem] =  l : List[Elem] . ordered(l)
  end
```

Fig. 2.5: Example of a parameterized specification in CASL with compound name `List[Elem]` (lists) and `OrdList[Elem]` (ordered lists). `PO` specifies partial order $\leq$ and `ORDLIST` ordered lists.

modules that import module m do not need to know anything about auxiliary sorts and operations of m. Hiding auxiliary names makes it easier to change m without affecting those modules that import m. Hiding is a generally accepted principle in programming because the user of a module only needs to know the specification of the main operations (which usually only specify the interface), not the auxiliary operations used to implement them. For specification languages the need for hiding is less obvious, since without knowledge of the auxiliary names it is unknown what the meaning of the exported names is. On the other hand, hiding can be useful because having fewer declared names reduces the chance of name overloading.

2.1.5 Application Domain

Wide-Spectrum Languages

Although a language used for specification must fit its purpose, its applicability need not be limited to requirements specification only and may extend to all aspects of software development, including requirements, prototyping, design, coding, and testing. Such a language is called *wide-spectrum*.

Using different languages for different texts within the same project causes *syntactic and semantic gaps*. That is, similar concepts have different notation or mean-

ing; therefore, translations have to be made. For example, if one wants to validate a program with respect to its specification, the program code and specification have to be translated to one common language that allows correctness proofs. Translation is laborious and error prone; therefore, it obscures the relationship between different documents, in particular if the differences between languages are subtle. Since the software development process is non-linear, translations have to be made more than once and possibly in both directions. For example, requirements are sometimes changed because of problems revealed during implementation.

General-Purpose Languages

General-purpose languages can be used for any type of system, as opposed to *Domain-specific* languages which aim at specific application domains, such as knowledge-based systems (for example, KARL (Fensel 1995) and $(ML)^2$ (van Harmelen & Balder 1992)), concurrent and distributed systems (for example, LOTOS (van Eijk, Vissers & Diaz 1989)), real-time systems (for example, TRIO (Ghezzi, Mandrioli & Morzenti 1990) and others (Heitmeyer & Mandrioli 1996, Abrial, Börger & Langmaack 1996)), or financial products (for example, RISLA in (van Deursen 1994)).

Domain-specific languages are optimized for a particular domain, making it easier to write and maintain specifications. However, its limited group of users makes it more expensive to develop a domain-specific language (language definition, tools, generic libraries, staff training, etc.). Moreover, it is difficult to anticipate all the possible uses of a language, even within a specific domain. A general-purpose language can be used by a much larger group of people and has the possibility to model aspects that at first sight lay outside the domain. That is the reason why I prefer general-purpose languages, provided they have facilities to configure syntax and semantics for arbitrary domains.

Note that although "wide-spectrum" is not the same as "general-purpose", facilities for configuring syntax and semantics may also be used for adapting a language to different phases of software development (rather than using different types of languages).

2.2 Specification Method

A formal language on its own does not give any clue or support for writing specifications. Therefore, a specification language should be part of a *specification method*, which should include *techniques* (language, mathematical theory, etc.), *guidelines*, and *tools*. Despite their name, most existing formal methods are not methods in this sense. Formal methods tend to be product-oriented (that is, centered on the techniques for manipulation of mathematical artifacts) and under-expose the process (that

is, how the artifacts arise and evolve) of software development (Floyd 1995, Fraser, Kumar & Vaishnavi 1994). Most formal methods give few guidelines on when and how to apply the formal techniques within the software development process (Bowen & Hinchey 1994). Moreover, important non-mathematical issues (such as the interaction between team members, planning, and the constant change of requirements) are overlooked (Floyd 1995, Floyd, Züllighoven, Budde & Keil-Slawik 1992, Jirotka & Goguen 1994, Leith 1990).

Without a supporting method, formal techniques are mainly suited for the design of systems satisfying *closed requirements*, that is, with already well-defined and stable requirements (Blum 1993, Blum 1994). The requirements for new systems (including the FSA case studies) are often open. Therefore the main goal of the FSA project was to develop a specification method. Although this method is not the subject of this thesis, some of its underlying principles are relevant for the choice of specification language for the case studies. Therefore, a short overview of the method is given here.

2.2.1 Stepwise Formalization

The FSA method is based on the idea of *stepwise formalization*. That is, formal specifications are constructed in three phases:

1. Dictionary: relevant terminology defined in natural language.
2. Signature: a conceptual model that identifies and structures relevant entities and their relations.
3. Definition: formalization of the properties of the signature elements.

These three steps represent three major aspects of a specification: *intention*, *structure*, and *detail*. Formal definitions are less flexible than natural language descriptions (the interpretation of words is easier to bend). At the beginning of a specification one should concentrate on the broad picture and express definitions in general terminology that is familiar, concise, and leaves room to delay details until a later time. Next it is important to come up with a suitable structure for a specification (which are the sorts, operations, and modules needed). Restructuring at a later stage can be quite laborious if many formal definitions have to be changed. Finally the formal details have to be filled in.

The given order of stepwise formalization does not imply a strict waterfall-like process in which each step has to be finished before one can move on to the next one. Each step may reveal shortcomings of the previous one which have to be repaired. However, it is assumed that each step is completed as much as possible before going on to the next step. This way each of the three aspects -intention, structure, and detail-

can be handled at the most appropriate level. Going to a next step too early is a waste of energy (at least, that is an assumption of the proposed method).

2.2.2 Stepwise Formalization in Practice

Stepwise formalization was used in the MP case study (see Section 1.6). It turned out to be very hard to write comprehensive dictionary definitions. In the software engineering literature it is either taken for granted that one knows how to write a definition, or a (semi) formal language is used. Many mistakes were made involving the use of a concept not in accordance with its definitions (type errors). Therefore, it was decided to add to each dictionary item restrictions on the use of the term, for example:

```
TERM grandmother
KIND attribute of a node
ISA  node
DEF  The grandmother of node N is the mother of a parent of N.
```

Later on it was decided not to adopt this idea, since it led to a large amount of effort devoted to work that should not be part of writing a dictionary: discussions about the exact use of the `KIND` and `ISA` information, and many auxiliary entries that were only needed for the definition of other entries. Moreover, the resulting dictionary was hard to read. Adding the `KIND`/`ISA` parts to a dictionary comes down to supplying type information; it seems more appropriate to delegate that to the signature part. Because of the non-sequential character of stepwise formalization, this typing activity can be done in parallel with making a dictionary.

Signature restructuring was a central activity in the specification process, especially when the domain was not very structured yet. However, the price of restructuring (in terms of the time needed to edit the specifications) was often so high that it was decided not to perform a sensible change. We think that changes are unavoidable, and the best thing one can do is to minimize them by delaying formalization as much as possible. It would help to have a theory of change (that is, what kind of change is useful in what situation) together with tools to perform the changes without too much effort.

Even if an ideal dictionary and signature are given, most people find it very hard to translate informal definitions into formal axioms. Automatic or semi-automatic translation of natural language to formal language can be a way to avoid problems of formalization (for research on this subject, see, for example, Ishihara, Seki & Kasami (1992)). However, automatic translation will probably depend on having an ideal dictionary written in a restricted form of natural language, which is already close to a formal language with many of the same problems. Automatic translation shifts some essential problems of formal specification to the informal dictionary. It seems that

any attempt to be more precise about the use of natural language is at conflict with the original intention of the use of natural language, namely freedom of expression.

2.2.3 Object-Oriented Methods

Although the simple stepwise formalization guideline does give some support to writing formal specification, it does not tell how to discover the relevant, often implicit, information needed. Therefore we had the idea to combine the FSA method with an existing method for requirements analysis. One could use a separate analysis method prior to the formal method. But using two methods with different notations has the drawback that the translation of the specifications between the two methods is error prone. It is therefore better to integrate the analysis and specification method and use one specification language.

There are two good reasons to think that concepts from object-oriented analysis can help in making formal specifications. First, it is claimed that an object-oriented method provides good guidelines for structuring requirements and software design. Second, the conceptual models (a kind of entity-relationship model) that are the result of object-oriented analysis can serve as the signature of a formal specification. Object-oriented methods also tend to be product-oriented, paying a lot of attention to notation. However, they do give guidelines for domain modeling (although methods do not agree on these guidelines) and stress the non-technical issues of systems development. Unfortunately, there is no scientific proof that object-oriented methods are indeed effective (Glass 1999).

The essence of object-oriented development is the identification and organization of application-domain concepts (Rumbaugh, Blaha, Premerlani, Eddy & Lorensen 1991). This leads to an emphasis on the description of objects rather than their use, based on the assumption that the specifications of objects are much more stable than the specifications of operations. It is also thought to be "natural" to organize the world as a collection of real-world objects grouped in classes. The resulting object models are structured by using *classification* (classes which can be specialized in sub-classes), *inheritance* (operations are shared among classes based on the sub-class relationship), *encapsulation* (internal details of objects are hidden), *polymorphism* (an operation may behave differently on different classes), and *modularity* (several object-oriented methods provide some kind of parameterized modules).

2.3 Why Yet Another Language?

Based on the goals of the FSA project (Section 1.5) formal foundation of software specification, tool development, specification of open domains, specification guidelines, education in formal specification) a number of requirements were formulated that should be satisfied by the specification language used in the project:

- *Lean, clean, simple, and unambiguous*: it should be easy to learn and use, and it should be simple to implement tools for it.
- *Stepwise formalization*: it must be possible to specify signatures with only informal definitions.
- *Wide-spectrum*: it must avoid semantic gaps.
- *General purpose*: it must allow reuse of method and specifications.
- *Object-oriented*: it must support classification, inheritance, encapsulation, and polymorphism.
- *Parameterized modules*: inheritance alone is not powerful enough for structuring and reuse.

In particular the "lean, clean, and simple" requirement was important for the FSA project, since the language was intended to be used by people with only limited mathematical training and in courses that do more than only teaching a formal language. This excludes most model-oriented languages (like Z and VDM), because, for some reason, they tend to be rather complicated and, in some cases, do not even have a formal definition. Unfortunately, most property-based (that is, algebraic) languages have limited expressiveness (not object oriented, wide spectrum, or general purpose). Moreover, algebraic languages often show their scientific roots by using difficult concepts (such as specifications-as-parameters) which may be interesting from a theoretical point of view, but not easy to use by non computer scientists (although these concepts may very well be "lean and clean"). The best way to guarantee that a language is unambiguous is to demand a formal definition (although there are different levels of formality too).

We thought stepwise formalization would benefit from the possibility of having informal definitions; that is, definitions in natural language, such as:

```
AXIOM {'BitS' is the sort of bits}
AXIOM n1+n2 = {the sum of 'n1' and 'n2'}
AXIOM {'l1++l2' is the concatenation of 'l1' and 'l2'}
```

Informal definitions are different from comment, since they replace formal definitions instead of clarifying them. Moreover, informal definitions may contain quoted formal parts which have to satisfy the well-formedness rules for formal terms (such as being syntactically correct and well-typed). Informal definitions can be used whenever full formalization is not that important: for example, as dictionary definitions or during the second step of stepwise formalization. They can also be used as a form of documentation besides the formal definitions (similar to comment). We have not found any specification language that supports informal definition. However, we could have

worked around this by using comments instead, but then we would have missed the well-formedness tests.

It is hard to say what exactly is needed to make a language truly wide spectrum and general purpose, but at least it should have flexible syntax and semantics, in order to use the notation and operations which are most suitable for a given task. Flexible syntax asks for mixfix notation for operation calls. For example, the way this is implemented in ASF+SDF (Klint 1993) allows the user to freely add notation that can be described by a context free grammar. There are more languages that allow some form of mixfix notation: for example, CASL (CoFI Task Group on Language Design 1999). With respect to flexible semantics, I think a specification language should allow one to model operations with non-functional features of any kind (as long as they can be described formally, of course). Equally important, it should allow operations on non-functional features (such as a while-do or for-to-do construction), since otherwise the user is stuck to the builtin vocabulary of the language. As far as I know there are no formal specification languages that allow user-defined non-functional features and/or operations on non-functional features. An additional benefit of flexible syntax and semantics is that it allows a leaner language definition, since most constructions can be defined within the language.

Both object-orientation and modularity are required to support systematic development and reuse (see Section 2.2.3). Parameterized modules are required because object-orientation does not supply this form of abstraction. They are, for example, needed for specifying typed lists (that is, lists in which all elements must be of a given sort).

In order to keep the language simple it does not have to fully formalize some object-oriented method. What is needed is classification (any language with sorts will do), inheritance (some form of sub-sorts, for example, by using implicit functions), encapsulation (objects are black boxes), and polymorphism (the same operation name can be used for different sorts; this can be realized by allowing arbitrary overloading). These features were considered enough to technically allow an object-oriented style of modeling. Typical object-oriented features that were not required: state-based operations (all common object-oriented methods are state-based), joining a sort and accompanying operations into a “class”, and all kinds of specific terminology like “attribute”, “method”, “link”, “association”, “static”, etc.

When the FSA project started we could not find a language that satisfied all the requirements in the way discussed here (as a matter of fact, at the moment of writing this thesis I am still not aware of any language that does satisfy all these requirements). Languages that were considered are VDM (Jones 1986), COLD (Feijs et al. 1994), EHDM (Rushby, von Henke & Owre 1991), Larch (Guttag & Horning 1993), and OBJ3 (Goguen, Kirchner, Kirchner, Mégrelis & Meseguer 1988, Goguen & Winkler 1988). It was decided to design a new language: AFSL. The challenge

was to make a language that satisfied all wishes at the same time. Languages do exist that satisfy some of the individual requirements, but that does not mean these can be combined at will. One of the main obstacles in language design is the interaction of language features.

A secondary argument for developing a new language was that formal specification could function as a connecting element between related research within our department (in the field of compiler construction, formal languages, program correctness, and software development methods). For those related interests it was important that we used a relatively simple language which could easily be fit to special needs.

2.4 Main Features of AFSL

AFSL was inspired by predicate logic, COLD (Feijs et al. 1994), EHDM (Rushby et al. 1991), and LSL (Guttag et al. 1990), but has a number of typical features in order to implement the requirements listed in the previous section (although I do not claim that these requirements have been satisfied completely, see the discussion in Chapter 8). In this section a brief overview of AFSL is given. Subsequent chapters give an in depth discussion.

Formal AFSL is a formal language. The syntax is defined in a BNF-like notation. It was harder to give a precise, and still comprehensible, definition of the semantics. Ideally the semantics of a formal language should also be formulated in a special meta-language. Unfortunately, there is no such meta-language which is generally accepted (like BNF for syntax definitions). Therefore, the semantics of AFSL is now based on a very simple kernel with obvious semantics (which is more or less mathematically defined). The advanced language features are defined as extensions of the kernel that can be eliminated by rewriting rules.

Initially I tried to define the semantics of AFSL mathematically, but, mathematics is not suitable for the definition of a full language (that is, anything bigger than idealized toy languages). Mathematics is mainly suited for *deriving* properties of relatively simple structures, instead of *defining* complex ones. Formal specifications are complex structures which are hard to handle in a language which has no structuring mechanism, type mechanism, or tool support. Moreover, mathematics is itself poorly defined (every "school" uses its own notation), which makes it hard for those who need to know the definition of a language (users and tool developers, who are often not mathematically trained) to read the definition.

Typed AFSL is typed in order to avoid semantical "mismatch" errors and to enable classification (which is needed to support object-orientation). The type system

is kept as simple as possible and, therefore, is first-order and does not support polymorphism. Initially such a type system seemed powerful enough, in particular in combination with parameterized modules which allows a kind of parameterized polymorphism. However, during the development of AFSL it turned out that higher order concepts were needed to model non-functional features (see below). At that time I did not want to change the basis of the language (with possibly far-reaching implications). Therefore, so-called function representations (which are objects that represent functions) were introduced as an ad-hoc solution.

Indexed Names AFSL has compound names in order to avoid overloading without renaming (see Section 2.1.4).

Implicit Functions Unary functions can be declared to be *implicit functions*, that can be used implicitly (that is, without actually writing them down) to convert an argument of a function. Declaring an implicit function from s_1 to s_2 has the effect of making s_1 a sub-sort of s_2 (see Section 2.1.1).

Informal Terms Terms can be descriptions in natural language. These *informal terms* do not have a formal semantics, which explains the name *Almost* Formal Specification Language. There are no restrictions on the use of informal language (in fact, almost any sequence of symbols is allowed). Informal terms may include formal parts which are marked as such. This allows tool support (for example, typechecking) for the formal parts of an informal term.

Overloading Because name overloading is virtually indispensable in some situations (for example, a + for different types of numbers or concatenation operations ++ for different types of lists), it is allowed in AFSL. Overloading of operations is resolved based on the type of arguments and result. There are no restrictions on the use of overloading in order to keep the language simple. It is, for example, possible to have multiple instances of the same operation name which all have the same type (provided these operations are declared in distinct modules). This may lead to confusing situations, but it is up to the user to make sensible use of overloading.

Syntax AFSL allows prefix, postfix, and infix notation for applications. Mixfix notation was considered too complicated to implement in the context of the FSA project. Also, it is not clear to what extent mixfix notation conflicts with other AFSL features. User-defined priorities and associativities can be specified.

Assertions A distinction is made between three kinds of assertions (that is, logical statements): axioms, requirements, and lemmas. Axioms play the same role as in any property-based language: they define possible values of the declared names. Requirements are used to put restrictions on future definitions of names. Lemmas are used for validation purposes; they indicate what properties should follow from the axioms and requirements if they are formulated correctly (with respect to the intended specification).

In languages that have a specifications-as-parameters mechanism, formal parameters of modules can usually contain axioms that restrict the possible values of the actual parameters. AFSL uses the names-as-parameters mechanism. In order to restrict these parameters, requirements can be used.

Requirements can also be used to put restrictions on functions f that are not yet defined in module m. The requirements then restrict the possible definition of f in modules that import m. Such f play a similar role as abstract (or virtual) methods in object-oriented programming. A specification in which all requirements follow from the axioms is said to be satisfied (that is, all names are defined in accordance with their requirements).

Semantics AFSL is property-oriented and has loose semantics. Encapsulation is considered incompatible with model-oriented languages because they define the internal structure of objects (unless representation details can somehow be hidden from being used in the rest of the specification). In this respect languages with initial semantics are also unwanted, since they also specify exactly one model. In a way, languages with initial semantics are model-oriented, because they model all objects as trees (in which the constructors used to make an object form the nodes of the tree). Moreover, initial semantics restricts the kind of axioms that are allowed and makes the definition of the language more complicated.

In particular, combining initial semantics and modularity causes some difficulties because some default properties are assumed. Assume module m_1 declares sort s and is imported into module m_2. If in m_2 we want to add new constructors (all functions are constructors with initial semantics!) or equality axioms for s, some of the default properties for s in m_1 are overwritten. This leads to a form of non-monotonic module import (that is, not all properties that hold within the context of m_1 still hold in m_2).

I think it is unwanted to have non-monotonic imports because it can be very confusing if different modules can overwrite properties at will (maybe even contradicting each other). Therefore, initial semantics requires that somehow it is forbidden to add new constructors or equations to imported sorts. But, this very much conflicts with the idea that in a specification non-relevant information (such as representation details) is postponed to a later stage (that is, some importing module).

Inductive Sorts Sorts can be defined inductively by specifying what its constructors are. Actually, this feature was added to the language in order to allow definition of inductive structures such as numbers, lists, and trees. This compensates for the absence of initial semantics.

Declaration of inductiveness is necessary as a special language feature because without it it is impossible to specify in a first-order language that a sort does not contain infinite data structures. AFSL is not a pure first-order language, function sorts and lambda-abstractions do make it possible to specify finiteness without using special language constructs. However, this is done by adding an induction axiom for each inductive sort; which is rather cumbersome.

Non-Executable Definitions are non-executable. Executable definitions are in form and expressiveness more restrictive than non-executable ones, which was thought to be too much of a limitation. Executable definitions, for example, do not allow implicit definitions like the definition of $-$ by $(x - y) + y = 0$. In retrospect, I think it was a wrong design decision to make AFSL non-executable (see the discussion in Section 8.4).

Modules Specifications are organized in modules which can be parameterized. Parameters are names (instead of specifications) because that is considered simpler to use and define (see Section 2.1.4).

Non-Functional Features All operations in AFSL are total functions (that is, without non-functional features). Because the language should be wide spectrum and general purpose, there are special provisions to model operations with non-functional features. The treatment of non-functional features is based on the idea that they are not properties of the operation but of the result. An object that carries information about a non-functional feature is called an *action*. Actions have to be "performed" in order to "release" the non-functional feature and return the actual value of the operation. For example, the result of a state modifying operation is a state modifying action, which will change the state and return a value when performed. A state modifying action can be modeled by a mapping that maps the initial state to the modified state plus the result value. This might seem an unusual way to look at non-functional features, however Chapters 6 and 7 show that it can be very effective.

The advantages of modeling operations with non-functional features by functions that return actions over having different types of operations builtin, are that it is generic and more flexible. The definition of the language does not have to be changed for each new type of operation. In fact we only need functions, which keeps the language simple. Because actions are ordinary objects, functions that handle non-functional features can be defined. For example, we can define if-then-else or

while-do constructions for state modifications. This is not possible in languages with non-functional features builtin to operations; those would have only a limited, prefabricated, set of builtin operations (such as an "if then else" and "while do" operation).

The use of actions in AFSL is inspired by monads, which are mathematical structures. The notion of "monad" originates from category theory and was put forward by Moggi (1991) as a structure to represent non-functional features in functional programming. Because monads are not addressed directly in this thesis, we will not go into detail. The way non-functional features are handled in AFSL is similar to the use of monads in the functional programming language Haskell (Wadler 1992, Wadler 1995, Bird 1998). However, there are two major differences, which are explained here for those readers familiar with Haskell.

First, AFSL allows bindings to be implicit. That is, instead of

```
do x <- y, f(x)
```

(which first performs action y, binds its result to x, and then performs $f(x)$), one can write simply `f(y)` (using so-called application redirection to insert the necessary monad operations, see Section 7.5). This not only saves on notation, but also realizes a notation closer to that of languages that have builtin non-functional features. For example, one could have `ReadNum+4` as an expression that first reads a number from some input and then adds four to it. This contrasts with the approach followed in Haskell, where the operations associated with a monad (such as `do`) are always explicitly used, which clutters programs.

Second, the action mechanism is treated different technically in AFSL. Haskell uses higher-order polymorphic functions to model monads, which are not available in AFSL (a decision which was made before actions were considered as a way to handle non-functional features). The lack of higher-order functions was solved by the ad-hoc solution of adding so-called function representations as builtin objects to AFSL; polymorphism has been simulated by the use of parameterized modules (which unfortunately leads to a flood of module imports). Although these solutions work, they are not very elegant and at conflict with the wish for a simple language (see the discussion in Section 8.7).

Predicates There are no predicates; boolean valued functions are used instead (see Section 2.1.2).

Tools Some tools have been implemented for AFSL. A parser/type-checker is implemented for the first version of AFSL that was used in the FSA case studies (this version did not support application redirection and had a slightly different way of resolving ambiguities). Also, there is a tool that generates diagrams from the signatures of modules of this early version of AFSL. These tools were implemented in Standard

ML (Harper, McQueen & Milner 1986, Myers, Clack & Poon 1993) (error messages were interfaced with the editor Emacs) and the diagrams are drawn by daVinci (Fröhlick & Werner 1995). For the final version of AFSL, as presented in this thesis, a prototype type-checker is implemented in Haskell (Fasel, Hudak, Peyton-Jones & Wadler 1992, Bird 1998), which is discussed in Section 8.9.

3. THE BASIC LANGUAGE

Abstract

This chapter defines the basic concepts of the specification language AFSL. This includes all language features except generic modules, implicit functions, and implicit binding, which are discussed in subsequent chapters. It is assumed that the reader is familiar with the concept of predicate logic or algebraic specification, even though this chapter is mostly self-contained.

3.1 Example: Bits and Lists of Bits

First a specification of lists of bits is given, which will serve as an example to explain the basic characteristics of AFSL throughout the rest of the current chapter.

Module `Bit1M` in Figure 3.1 defines an inductive sort `BitS` of bits with constructors `Zero` and `One` that represent the two bits zero and one. Furthermore, the operation `*` and the ordering `<=` are defined for bits. Definition of other relevant operations for bits are omitted from this example (throughout this thesis omissions are indicated by comment started with `%...`). Because there is no special syntactic category for constants, `Zero` and `One` are declared as functions without arguments. Since an empty tuple of arguments may be omitted from function applications (that is, `Zero` is shorthand for `Zero()`) functions without arguments can be used as if they are true constants.

The attribute `INDUCTIVE` indicates that `BitS` is an inductive sort, which means it contains precisely those objects that can be created by its constructors (which are functions with attribute `CONSTRUCTOR`). That is, `BitS` contains exactly `Zero` and `One`. See Section 3.13 for a detailed discussion of inductive sorts. Both sorts defined in the current section (`BitS` and `BitListS`) are inductive, however, in general sorts do not need to be inductive.

The first axiom, that states that `Zero` is not equal to `One` defines equality for bits. This axiom is necessary to exclude the possibility that `Zero` and `One` denote the same object. The lemma does not affect the actual definition of `BitS`, but gives a different view on the definition (see Section 3.12). This particular lemma is an example of an informal term, which means that it uses natural language instead of formal language

```
MODULE Bit1M

% declaration of variables b, b1, ... that range over BitS
VAR    b : BitS

% declaration of sort BitS and constructors Zero and One
SORT   BitS (INDUCTIVE)
FUNC   Zero : -> BitS (CONSTRUCTOR)
FUNC   One  : -> BitS (CONSTRUCTOR)
% equality for bits
AXIOM  Zero /= One
% alternative formulation of the definition of BitS
LEMMA  {'BitS' is the sort of bits with two distinct elements
       'Zero' and 'One'}

% declaration and definition of the operation *
FUNC   * : BitS, BitS -> BitS
AXIOM  Zero * b = Zero
AXIOM  One  * b = b

% declaration and definition of less-than-equal relation <=
FUNC   <= : BitS, BitS -> BoolS
AXIOM  Zero <= b
AXIOM  (One <= b) = (b = One)

%... more functions for bits

END MODULE
```

Fig. 3.1: A specification of bits.

(see Section 3.11). Natural language can be used either in situations where an informal description is preferred because it is easier to write or read (depending on the audience, of course), or as a temporary description if full details are not yet available or irrelevant (such an informal description can be replaced by a formal one at a later stage).

The binary function `*` is defined by two axioms that determine the result for all possible arguments. Here `*` is used as an infix operation, this is allowed for any function (see Section 3.8). The axioms use variable `b` to denote an arbitrary bit. The declaration of `b` not only declares `b` to be of type `BitS`, but also any extension of `b` with a natural number (`b0`, `b1`, `b24`, etc.).

The relations `/=` and `=` are builtin functions, which means that they can be used without being declared (see Section 3.6). The equality operations are ordinary functions with co-domain `BoolS`, they do not have a special status as a relation. The same holds for logical operations (boolean arguments and result), these are also functions. See Section 2.1.2 for a discussion of the reasons why AFSL does not distinguish predicates but uses boolean-valued functions instead.

Module `BitList1M` in Figure 3.2 defines the inductive sort `BitListS` of finite lists of bits, where constructor `Empty` is the empty list and constructor `&` is the operation that adds a bit in front of a list of bits. The definition of `BitS`, plus its operations, is made available within `BitList1M` through the import of `Bit1M`.

Module import is similar to the `input` construct in LaTeX or the module mechanism that some programming languages have. A module with imports is called a *structured module*. More information on the import mechanism can be found in Section 3.3. In Chapter 4 generic modules are discussed, which are parameterized modules.

The import of `LogicM` provides the definition of logical operations. There are no builtin logical operations. If they are needed they have to be defined within the specification. This is possible because logical operations are ordinary functions. Throughout this thesis the module `LogicM`, given in Figure 3.3, is used for the definition of logical operations. Note that `LogicM` does not define `BoolS` as an inductive sort (with constructors `True` and `False`) because `BoolS` is a builtin sort and as such cannot be declared to be inductive. However, `BoolS` has a fixed interpretation, it contains exactly the two, distinct, objects "true" and "false". `BoolS` is builtin because it has a special status: all assertions (axioms, lemma's, and requirements) must be of type `BoolS`. The interpretation of all specified names must be such that the value of all axioms is "true". This guarantees that in `LogicM` the value of `True` is "true" and of `False` is "false".

The import of module `NatM` (see Figure 5.8 on page 94) provides the definition of the sort `NatS` of natural numbers plus operations like `+`. It is assumed there is a total order `<=` on lists. The relation `<=` is declared, but not defined in `BitList1M`, it is only

```
MODULE BitList1M

% declaration of variables b, b1, b2, ... and l, l1, l2, ...
VAR    b : BitS
VAR    l : BitListS

% imports
IMPORT Bit1M  % Fig. 3.1 page 40
IMPORT NatM   % Fig. 5.8 page 94
IMPORT LogicM % Fig. 3.3 page 43

% definition of the sort of bit-lists
SORT   BitListS (INDUCTIVE)
FUNC   Empty :                -> BitListS (CONSTRUCTOR)
FUNC   &     : BitS, BitListS -> BitListS (CONSTRUCTOR)
% definition of inequality of lists
AXIOM  b&l   /= Empty
AXIOM  b1&l1 /= b2&l2  <==  b1 /= b2  Or  l1 /= l2

% declaration and requirements for a total ordering on lists
FUNC   <= : BitListS, BitListS -> BoolS
REQ    l  <= l
REQ    l1 <= l2  And  l2 <= l3  ==>  l1 <= l3
REQ    l1 <= l2  And  l2 <= l1  ==>  l1 = l2
REQ    l1 <= l2  Or   l2 <= l1

% definition of the function that returns the length of a list
FUNC   Length : BitListS -> NatS
AXIOM  Length Empty = 0
AXIOM  Length (b&l) = Length l + 1

% definition of the function that appends one list to another
FUNC   ++ : BitListS, BitListS -> BitListS
AXIOM  l1 ++ l2 = {the concatenation of 'l1' and 'l2'}
LEMMA  Length (l1 ++ l2) = Length l1 + Length l2

%... further operations for lists

END MODULE
```

Fig. 3.2: A specification of lists of bits.

```
MODULE LogicM

VAR    b : BoolS

FUNC   True : -> BoolS
AXIOM  True

FUNC   False : -> BoolS
AXIOM  False /= True

FUNC   Not : BoolS -> BoolS
AXIOM  (Not True)  = False
AXIOM  (Not False) = True

FUNC   And : BoolS, BoolS -> BoolS
AXIOM  (True  And b) = b
AXIOM  (False And b) = False

FUNC   Or : BoolS, BoolS -> BoolS
AXIOM  (True  Or b) = True
AXIOM  (False Or b) = b

FUNC   ==> : BoolS, BoolS -> BoolS
AXIOM  (b1 ==> b2) = (Not b1  Or b2)

FUNC   <== : BoolS, BoolS -> BoolS
AXIOM  (b1 <== b2) = (b2 ==> b1)

FUNC   <=> : BoolS, BoolS -> BoolS
AXIOM  (b1 <=> b2) = ((b1 ==> b2) And (b1 <== b2))

END MODULE
```

Fig. 3.3: Definition of the standard logical operations.

required (indicated by REQ) that <= will be defined as a total order in some module that imports BitList1M (see Section 3.12 for an explanation of requirements). Furthermore, list operations Length and ++ are defined. In Bit1M the lemma for BitS is completely informal, but assertions can also be partly informal. For example, in the definition of ++ only the part {the ··· 'l2'} is informal.

The module BitList1M is actually only a part of the specification of lists of bits, since it lacks information on the meaning of the imported modules LogicM, Bit1M, and NatM. Therefore, a specification consists of a module M plus a set of modules, which is the context in which M should be interpreted: for example, the context of BitList1M consists of LogicM, Bit1M, and NatM. Within a specification each module must have a unique name to ensure unambiguous identification of modules.

3.2 Lexical Syntax

The tokens used in AFSL are (excluding those sequences that are keywords):

- *Module identifiers* (the name of a module), which is a sequence of letters starting with a capital and ending with an M: for example, Bit1M.
- *Sort identifiers* (the main part of a sort name), which is a sequence of letters starting with a capital and ending with an S: for example, BitListS.
- *Function identifiers* (the main part of a function name), which can be:
  - Sequences of alpha-numeric symbols starting with a capital and not ending with an M or S: for example, Length or Bit2Nat.
  - Non-empty sequences of the non-letter symbols:

    ! # $ & * + - / . : ; < = > ? @ ^ _ ‘ | ~

    for example, +, ++, =<, or :=.
  - *Strings*, which are sequences of symbols enclosed by double quotes: for example, "Erik" or "John loves Mary".
  - *Numerals* are defined to be nonempty sequences of digits, such as 123, 7, or 007. Leading zeros are significant in numerals (which is relevant, for example, in the specification of floats in Figure 5.10 on page 96).
- *Variables*, which are nonempty sequences of lowercase characters optionally followed by a sequence of digits, the so-called *subscript*: for example, x, x10 (subscript 10), or x007 (subscript 007).
- *Natural language* (the building stones of informal terms, see Section 3.11), which can be almost any sequence of symbols, excluding space, tab, %, {, }, and '.

The formal definition of the (lexical) syntax written in SDF can be found in Appendix A. Note that strings and numerals have no special status, they are all function identifiers. That is the reason why `Zero` and `One` can be declared as functions in `Bit1M`. The tokens and keywords of a module may contain *comment*, which starts with the symbol `%` or `\` and ends with a newline. The reason that module identifiers should end with an `M` and sort identifiers with an `S` is that it helped to make the first parser for AFSL (this was a generated YACC-like parser in SML (Harper et al. 1986, Myers et al. 1993)). More advanced parsing techniques might allow a more liberal lexical syntax.

3.3 Modules

The syntax for modules is defined in Figure 3.4 (an SDF version of the AFSL syntax can be found in Appendix A). Here an extended BNF-notation is used, where $[x]$ denotes an optional occurrence of x, $x*$ denotes a sequence of zero or more occurrences of nonterminal x, and $\{x\ y\}*$ denotes a sequence of zero or more occurrences of nonterminal x separated by the terminal y (that is, $x\ y\ x \ldots y\ x$). A module consists of a heading and a body. The *heading of a module* is formed by the module identifier and optionally a list of *formal parameters*. The example modules in the current chapter have no formal parameters. See Chapter 4 for a discussion of modules with formal parameters. The *body of a module* is a sequence of three kinds of *specification items*: imports (`IMPORT` followed by a list of *actual parameters*), declarations (`SORT`, `FUNC`, and `VAR`), and assertions (`AXIOM`, `REQ`, and `LEMMA`).

3.4 Names and Variables

3.4.1 Sort and Function Names

Any name used in a module must be declared somewhere within that module or in an imported module (except for the builtin names). Names used in declarations and imports must be declared before they are used. The definition of the syntax of declarations and names is given in Figure 3.4. The optional *origin* of a name denotes the module the name belongs to (the terminology "origin" is taken from Bergstra et al. (1989)). The optional *type info* of a function name denotes the domain and codomain of the function it represents. Usually origins are omitted from declarations, in which case the name of the module that contains the declaration is assumed to be its origin. In its declaration a function name is obliged to have type info. A name declaration may be concluded by a sequence of *attributes* which specify additional properties of the name declared (the different attributes are explained later on).

Sort names play a similar role as types in programming languages, except that sorts are not defined to be equal to some type expression. Instead, sorts are im-

<module>	::=	"MODULE" <module-id> [<name-list>]
		<item> $*$
		"END" "MODULE"
<item>	::=	"IMPORT" <module-id> [<name-list>]
	|	<declaration>
	|	<assertion>
<name-list>	::=	"[" { <name> "," } $*$ "]"
<declaration>	::=	"SORT" <sort> [<attributes>]
	|	"FUNC" <function> [<attributes>]
	|	"VAR" <variable> ":" <sort>
<assertion>	::=	<qualifier> <term>
<qualifier>	::=	"AXIOM" | "LEMMA" | "REQ"
<term>	::=	... see Section 3.8
<name>	::=	<sort> | <function>
<sort>	::=	[<origin>] <sort-id> [<indexes>]
<function>	::=	[<origin>] <function-id> [<indexes>] [<type-info>]
<origin>	::=	<module-id> ":"
<indexes>	::=	"[" { <name> "," } $*$ "]"
<type-info>	::=	":" { <sort> "," } $*$ "->" <sort>
<attributes>	::=	"(" { <attribute> "," } $*$ ")"
<attribute>	::=	"INDUCTIVE"
	|	"CONSTRUCTOR"
	|	"IMPLICIT"
	|	"APPLICATION"

Fig. 3.4: Syntax of modules.

plicitly defined by the functions that are defined on them. Function names play the same role as functions in a functional programming language. There are no special names for constants, they are modeled as functions without arguments (such as `Zero : -> BitS`).

The name declarations of a module (including the name declarations of its imported modules) form its so-called *signature*. If, given a signature Σ, sort s is declared we write $\Sigma \vdash s$. If function f is declared to be of type $s_1, \ldots, s_n$`->`s, we write $\Sigma \vdash f : s_1, \ldots, s_n \rightarrow s$ ($\Sigma \vdash$ can be omitted if it is clear what Σ is).

3.4.2 Indexed Names

An *indexed name* is a name that contains a list of *indexes*. Examples of indexed names are:

```
ListS[BitS]
Sort[<=]
TableS[StringS,IntS]
ListS[ListS[BitS]]
```

If the list of indexes is omitted from a name it is assumed to be empty. Within a module an indexed name is treated as an indivisible whole: for example, `ListS[BitS]` is treated as if it were the non-indexed name `ListSBitS`. The importance of indexed names lies in the fact that an index can be a formal parameter (see Chapter 4). For example, in Figure 4.4 the sort `ListS[XS]` is declared, where `XS` is a formal parameter. When unfolding a module import, indexes that are formal parameters are replaced by their corresponding actual parameters: for example, `ListS[XS]` becomes `ListS[BitS]` if `BitS` is supplied as actual parameter to replace `XS`.

3.4.3 Name Abbreviations

A name that includes all its optional parts (origin and type info) is called a *complete name*. A name that lacks some of its optional components is called a *name abbreviation*. All the complete names declared in `Bit1M` and `BitList1M` are listed in Figure 3.5, all names used in the example modules are abbreviations. If name n is an abbreviation of complete name n', then n' is called a *completion* of n, which is denoted by $n \leadsto n'$.

Complete names can be quite long. Therefore, usually the optional parts are omitted when a name is used. This may lead to overloading. A name abbreviation is called *overloaded* if it has multiple completions: for example, `<=` is overloaded in `BitList1M`. When an overloaded name is used as a function in a function application, the context of its use may resolve this conflict (see Section 3.8). However, in all other situations name abbreviations should not be overloaded. Note that complete names cannot be overloaded. That is, a complete name always denotes the same sort or

```
Bit1M:BitS[]
Bit1M:Zero[]:->Bit1M:BitS[]
Bit1M:One[]:->Bit1M:BitS[]
Bit1M:*[]:Bit1M:BitS[],Bit1M:BitS[]->Bit1M:BitS[]
Bit1M:<=[]:Bit1M:BitS[],Bit1M:BitS[]->Builtin:BoolS[]
BitList1M:BitListS[]
BitList1M:Empty[]:->BitList1M:BitListS[]
BitList1M:&[]:Bit1M:BitS[],BitList1M:BitListS[]->BitList1M:BitListS[]
BitList1M:<=[]:BitList1M:BitListS[],BitList1M:BitListS[]
               ->Builtin:BoolS[]
BitList1M:Length[]:BitList1M:BitListS[]->NatM:NatS[]
BitList1M:++[]:BitList1M:BitListS[],BitList1M:BitListS[]
               ->BitList1M:BitListS[]
```

Fig. 3.5: The complete names declared locally in `Bit1M` and `BitList1M`.

function, no matter the context in which it is used. Because double declarations of the same name are confusing, a declaration should declare a complete name not yet declared.

3.4.4 Variables

Variables have to be declared locally before they are used, variable declarations are not imported. Together the variable declarations of a module $\mathcal{M}$ form the so-called *type context* of $\mathcal{M}$. Variables are given a sort as type in their declaration, for example, the type of `b` is `BitS`. A single variable declaration not only declares the variable mentioned but also any extension of this variable with a natural number, called its *subscript*. For example, the declaration `VAR b : BitS` declares the variables `b`, `b1`, `b2`, etc., all of type `BitS`. If, given a type context Γ, variable v is of type s we write $\Gamma \vdash v : s$ ($\Gamma \vdash$ can be omitted if it is clear what Γ is)

3.5 Value of Names and Variables

The names declared in a specification denote a *value*. There may be many possible values for the names in a signature. A particular assignment of values to the names of a given signature Σ is called a Σ*-algebra*. One can think of an algebra as a possible interpretation of the names in a signature. Formally, an algebra is a mapping from names to values. Therefore, the value of name n for algebra $\mathcal{A}$ is denoted by $\mathcal{A}(n)$. A sort s denotes a set of objects $\mathcal{A}(s)$ and a function $f : s_1, \ldots s_n \rightarrow s$ denotes a mapping $\mathcal{A}(f)$ from the tuples in $\mathcal{A}(s_1) \times \cdots \times \mathcal{A}(s_n)$ to elements of $\mathcal{A}(s)$.

A variable of type s is, given an algebra $\mathcal{A}$, a placeholder for an arbitrary object in $\mathcal{A}(s)$. There may be many possible values for the variables in a type context. A

```
BuiltinM:BoolS
BuiltinM:= : s, s -> BuiltinM:BoolS
BuiltinM:/= : s, s -> BuiltinM:BoolS
BuiltinM:FuncS[s_1,s_2]
BuiltinM:@ : BuiltinM:FuncS[s_1,s_2], s_1 -> s_2
```

$$\mathcal{A}(\texttt{BoolS}) = \{true, false\}$$

$$\mathcal{A}(\texttt{=})(x_1, x_2) = \begin{cases} true & \text{if } x_1 = x_2 \\ false & \text{otherwise} \end{cases}$$

$$\mathcal{A}(\texttt{/=})(x_1, x_2) = \begin{cases} false & \text{if } x_1 = x_2 \\ true & \text{otherwise} \end{cases}$$

$$\mathcal{A}(\texttt{FuncS[}s_1\texttt{,}s_2\texttt{]}) = \text{the set of total mappings from } \mathcal{A}(s_1) \text{ to } \mathcal{A}(s_2)$$

$$\mathcal{A}(\texttt{@})(f, y) = f(y)$$

Fig. 3.6: Builtin names and their semantics. (Given signature Σ and Σ-algebra $\mathcal{A}$. For sorts s, s_1, and s_2, objects $x_1, x_2 \in \mathcal{A}(s)$ and $y \in \mathcal{A}(s_1)$, and total mapping f from $\mathcal{A}(s_1)$ to $\mathcal{A}(s_2)$.)

particular assignment of values to variables is called a *value context* (also known as valuation or variable assignment). One can think of a value context as a context in which each placeholder is filled with a particular value. Formally, a value context is a mapping from variables to values. Therefore, the value of variable v for algebra α is denoted by $\alpha(v)$.

3.6 Builtin Names

Some sorts and functions are builtin, which means that they are always part of an AFSL signature (without declaration). Given a particular interpretation of the non builtin names, builtin names have a fixed semantics. The builtin names and their semantics are listed in Figure 3.6 (name abbreviations are used in the definition of the semantics). `BuiltinM` is a reserved module identifier that is used as the origin of builtin names. The function identifiers `=`, `/=`, and `@` are overloaded which means that there are different complete names for different instances of s, s_1, and s_2.

3.6.1 Booleans

The sort `BoolS` of truth values *true* and *false* is builtin because it is the type of formulas (a particular type of terms) and the co-domain type of the equality operations `=` and `/=`, which are the basic operations that allow the specification of new functions.

3.6.2 Function Representations

In Chapter 7 functions that have functions as arguments are used to handle so-called non-functional features in a proper way. Functions as arguments are not allowed in a first-order language like AFSL (in which arguments must be objects, functions are no first-order objects). Therefore, a trick is used: for all sorts s_1 and s_2 there is a builtin *function sort* `FuncS[`s_1`,`s_2`]` of total mappings from s_1 to s_2. Note that `FuncS[`s_1`,`s_2`]` is not the same as s_1`->`s_2, the former is a sort, the latter is a function type. However, they both represent the same collection of mappings! In order to emphasize the distinction between the function sort and the function type, an element of a function sort is called a *function representation*. For terms t_1 : `FuncS[`s_1`,`s_2`]` and t_2 : s_1 the function-application t_1`(`t_2`)` is not a legal term. Therefore, there is a builtin function `@` such that t_1 `@` t_2 denotes the application of the mapping denoted by t_1 to the object denoted by t_2.

Function sorts may seem an ad-hoc extension of an otherwise first-order language. The main reason why AFSL is not extended to a full higher-order language is that it would complicate the language too much in the current setting. Still, it is very well possible that the language would benefit from having real higher-order features, see Section 8.1 for a further discussion of this subject.

Function representations were added to AFSL in order to facilitate non-functional features, but they can also be used for other purposes. For example, some quantifiers that operate on representations of boolean valued functions (that is, elements of `FuncS[`s`,BoolS]`) are specified in the parameterized module `QuantifiersM` (Figure 3.7). The lambda abstractions used here are introduced in Section 3.8. Note that the use of the quantifiers specified in `QuantifiersM` requires a separate import for each domain that needs to be quantified over. This could lead to considerable overhead which can be avoided if polymorphism would be added to the type system (see the discussion in Section 8.5). Preliminary versions of AFSL had builtin constructs for universal and existential quantification. These were removed when lambda abstractions were added to the language.

3.7 Literals

The given builtin names are sufficient to specify the standard data structures for strings, natural numbers, integers, and floating points. However, it is not possible

```
MODULE QuantifiersM [XS]

VAR    x    : XS
VAR    pred : FuncS[XS,BoolS]

SORT   XS

IMPORT LogicM % Fig. 3.3 page 43

FUNC   Forall : FuncS[XS,BoolS] -> BoolS
AXIOM  Forall pred = (pred = (x| True))

FUNC   Exists : FuncS[XS,BoolS] -> BoolS
AXIOM  Exists pred = (pred /= (x| False))
LEMMA  Exists pred = (Not Forall (x| Not pred@x))

%... other quantifiers

END MODULE
```

Fig. 3.7: Specification of quantifiers.

to use the standard denotations for these data structures (so-called *literals*), such as `"Paul and Mary"`, `380`, `-213`, and `3.815`. Instead more complicated terms are needed to denote these values, such as `Succ(Succ(...(0)))` (*380* times `Succ`, the successor function for natural numbers, instead of `380`), which leads to unreadable specifications.

Therefore, two types of literals are treated as builtin names: strings and numerals. Strings are sequences of symbols enclosed by double-quotes and numerals are sequences of digits. These builtin literals, together with the corresponding sorts and some basic operations, are listed in Figure 3.8 and their semantics are given in Figure 3.8. Strings and numerals are interpreted in any algebra as themselves (that is why they are called “literals”). Because of possible leading zeros, numerals are different from natural numbers: for example, *001* is not the same as *1*. Leading zeros are significant if a numeral is used in the fractional part of a floating point number. Numerals can be used to denote natural numbers by defining an implicit function from numerals to the non-builtin sort `NatS` of natural numbers (see Chapter 5 for implicit functions). The minus-notation can be realized by defining `-` as a unary function on numbers and the dot-notation for reals can be realized by defining `.` (dot) as a binary operation on numerals (see Section 5.5).

The builtin literals and their operations are added for convenience. It depends on the applications of a language what is convenient. For example, if efficient execution

```
BuiltinM:StringS
BuiltinM:s      : -> BuiltinM:StringS
BuiltinM:Concat : BuiltinM:StringS, BuiltinM:StringS
                  -> BuiltinM:StringS
BuiltinM:NumeralS
BuiltinM:Null   : -> BuiltinM:NumeralS
BuiltinM:n      : -> BuiltinM:NumeralS
BuiltinM:Concat : BuiltinM:NumeralS, BuiltinM:NumeralS
                  -> BuiltinM:NumeralS
```

$$
\begin{aligned}
\mathcal{A}(\texttt{StringS}) &= \text{set of all strings} \\
\mathcal{A}(s)() &= s \\
\mathcal{A}(\texttt{Concat})(s_1, s_2) &= \text{the concatenation of } s_1 \text{ and } s_2 \\
\mathcal{A}(\texttt{NumeralS}) &= \text{set of finite sequences of digits} \\
\mathcal{A}(\texttt{Null})() &= \text{the empty sequence} \\
\mathcal{A}(n)() &= n \\
\mathcal{A}(\texttt{Concat})(n_1, n_2) &= \text{the concatenation of } n_1 \text{ and } n_2
\end{aligned}
$$

Fig. 3.8: Builtin literals plus operations and their semantics. (Given signature Σ and Σ-algebra $\mathcal{A}$. For strings s, s_1, and s_2 and digit sequences n, n_1, and n_2.)

```
<term>          ::=  <variable>
                   | <args> <function> <args>
                   | "(" <variable> [ ":" <sort> ] "|" <term> ")"
                   | <informal-term>
<args>          ::=  "(" { <term> "," } * ")"
                   | <term>
                   |
<informal-term> ::=  "{" <informal> * "}"
<informal>      ::=  <natural-language>
                   | "'" <name> "'"
                   | "'" <term> "'"
```

Fig. 3.9: Syntax of terms.

of specifications is an issue different kinds of numbers should be a builtin together with a large collection of efficient operations. Also, it could be useful to have lists as builtin data types, including a special list-notation. Since AFSL is a prototype language, its collection of builtin names is very limited, only to avoid the roughest notational edges.

3.8 Terms

A *term* is either a variable v, a *function application* $a_1 f a_2$, a *lambda abstraction* $(v:s \mid t)$, or an *informal term* (see Figure 3.9). A single term t as function argument is shorthand for (t) and no argument is equivalent to the empty argument $()$. The sort s in a lambda abstraction is optional, it may be omitted provided v is declared as a variable of type s.

All function applications in AFSL are actual infix applications. But, because empty arguments are allowed, both prefix and postfix notation is possible. The syntax also allows less usual forms of function application, such as $(x,y)f$ and $x\ f(y,z)$. The second form enables a kind of object-oriented notation where the receiver (first argument) is written in front of a message (function plus the rest of the arguments). Section 3.9 discuses how ambiguous function applications are parsed. Because $(t_1, \ldots)f(\ldots, t_n)$ is equivalent to $f(t_1, \ldots, t_n)$, the discussion of function applications will use only the second, more familiar and simpler, notation.

A term that contains abbreviated names is called an *abbreviated term*, otherwise it is a *complete term*. A term that contains overloaded functions is called an *overloaded term*. Each completion of an overloaded function can have a different value. Therefore, it is in general not possible to associate a value with an overloaded term,

$$\frac{}{v \twoheadrightarrow v}$$

$$\frac{f \twoheadrightarrow f' \qquad t_1 \twoheadrightarrow t_1' \quad \cdots \quad t_n \twoheadrightarrow t_n'}{f(t_1, \ldots, t_n) \twoheadrightarrow f'(t_1', \ldots, t_n')}$$

$$\frac{s \twoheadrightarrow s' \qquad t \twoheadrightarrow t'}{(v : s \mid t) \twoheadrightarrow (v : s' \mid t')}$$

$$\frac{e_1 \rightarrowtail e_1' \quad \cdots \quad e_n \rightarrowtail e_n'}{\{e_1 \ldots e_n\} \twoheadrightarrow \{e_1' \ldots e_n'\}}$$

$$\frac{}{nl \rightarrowtail nl} \qquad \frac{nm \twoheadrightarrow nm'}{\text{'}nm\text{'} \rightarrowtail \text{'}nm'\text{'}} \qquad \frac{t \twoheadrightarrow t'}{\text{'}t\text{'} \rightarrowtail \text{'}t'\text{'}}$$

Fig. 3.10: Rules for abbreviation expansion. (Given a signature. For variable v; sorts s and s'; functions f and f'; terms t, t', t_1, t_1', $\ldots$ t_n, t_n'; informals $e_1, \ldots e_n$; natural language nl; and names nm and nm'.)

```
<priorities>   ::=  "PRIORITIES" <group> * "END" "PRIORIES"
     <group>   ::=  "GROUP" { <operator> "," }*
  <operator>   ::=  <function-id> [ <associativity> ]
<associativity> ::=  "(LEFT)" | "(RIGHT)"
```

Fig. 3.11: Syntax of a priority section.

unless all overloaded functions are replaced by completions first. Given a signature, term t' is a *expansion* of t (denoted by $t \twoheadrightarrow t'$) if and only if t' can be obtained from t by replacing all its abbreviations by complete names, which is formally defined in Figure 3.10. Here $e \rightarrowtail e'$ denotes that the informal e (natural language or quoted name/term) can be expanded to e'. The notion "expansion" will be extended in Section 5.8.

3.9 Priorities

The syntax of terms is ambiguous because arguments are optional and parentheses may be omitted if an argument is a single term. For example, how should $f\ g\ h$ be parsed for functions f, g, and h? To enable unambiguous parsing of terms without using many parentheses, a relative *priority* and an *associativity* ("left", "right", or "none") is associated with each function. By default, all functions have the same priority and are right-associative, this causes $f\ g\ h$ to be parsed as $f(g(h))$, or actually $()f(()g(()h()))$.

Non-default priorities and associativities can be declared in a *priority section*, the syntax of which is defined in Figure 3.11. Each specification has exactly one priority section. The one used throughout this thesis is given in Figure 3.12. A priority section declares the parsing information of function identifiers, not of functions. Therefore, all functions with the same identifier have the same priority and associativity. This prevents ambiguous parsing information for function name abbreviations. Non-default priorities are always weaker than the default, identifiers within one group have the same priority, and groups are listed in order of decreasing priority. Note that a function can only really be used infix or postfix if it has a non-default priority. If not, for example, `1+2` would be parsed as `1(+(2))` instead of `(1)+(2)` (`1` and `2` are functions too!). Terms that cannot be parsed uniquely based on the priority section are considered ill-formed and hence disallowed. In AFSL function types play no role at all in the parsing of terms (contrary to, for example, ASF+SDF).

The version of AFSL used in the case studies did not use a priority section. Instead, function names could be declared with priority attributes `PRIORITY` n, `LEFT`, and `RIGHT`, where n is a natural number indicating the absolute priority of the func-

```
PRIORITIES

GROUP ., E
GROUP @ (LEFT), # (LEFT)
GROUP & (RIGHT), ++ (LEFT)
GROUP * (LEFT), / (LEFT), Div (LEFT), Mod (LEFT)
GROUP + (LEFT), - (LEFT)
GROUP :=
GROUP >>= (LEFT), =<< (RIGHT), +<< (RIGHT), ++<< (RIGHT),
      >> (RIGHT), << (LEFT)
GROUP =, /=, <, >, <=, >=, ==, /==
GROUP Not
GROUP And (RIGHT), Or (RIGHT)
GROUP <=>, ==> (RIGHT), <== (LEFT)

END PRIORITIES
```

Fig. 3.12: Priorities used throughout this thesis.

tion. This mechanism had some serious drawbacks. If priorities are declared in attributes scattered all over the specification, there is no clear view on the mutual relations in the priority hierarchy. Declaring absolute priorities makes it difficult to insert an operation in the priority hierarchy without renumbering the priorities of functions already declared. The priority of a name abbreviation can become ambiguous if priorities are linked to complete names. Finally, the use of priority attributes makes it harder to parse a module, since all imported modules have to be parsed first in order to collect all parsing information.

3.10 Type and Semantics of Terms

Given a signature Σ and a type context Γ, a complete term t can have a sort s as its *type*, which is denoted by $\Sigma, \Gamma \vdash t : s$ (or just $t : s$ if it is clear what Σ and Γ are). The rules for determining the type of a term are given in Figure 3.13. Here the notation $\Gamma[v \mapsto s]$ is used for the assignment of sort s to variable v in type context Γ (leaving the type of all other variables unchanged). See Section 3.11 for an explanation of the typing rule for informal terms.

Terms that have a type according to these rules are called *well-typed*, the others are called *ill-typed*. A *formula* is a non-overloaded term of type `BoolS`. Formulas are treated as ordinary terms as a consequence of the choice to model relations by functions (see Section 2.1.2).

Given an algebra $\mathcal{A}$ and a value context α, a complete term $t : s$ has a *value* in $\mathcal{A}(s)$, which is denoted by $[\![t]\!]_{\mathcal{A},\alpha}$ (or just $[\![t]\!]$ if it is clear what $\mathcal{A}$ and α are). The

$$\frac{\Sigma \vdash s \qquad \Gamma \vdash v : s}{\Sigma, \Gamma \vdash v : s}$$

$$\frac{\Sigma \vdash f : s_1, \ldots, s_n \to s \qquad \Sigma, \Gamma \vdash t_1 : s_1 \quad \cdots \quad \Sigma, \Gamma \vdash t_n : s_n}{\Sigma, \Gamma \vdash f\,(t_1\,, \ldots\,, t_n) : s}$$

$$\frac{\Sigma \vdash s_1 \qquad \Sigma, \Gamma[v \mapsto s_1] \vdash t : s_2}{\Sigma, \Gamma \vdash (v : s_1 \mid t) : \texttt{FuncS}[s_1\,, s_2]}$$

$$\frac{}{\Sigma, \Gamma \vdash \{e_1 \ldots e_n\} : s}$$

Fig. 3.13: Typing rules for non-abbreviated terms. (Given signature Σ and type context Γ. For any variable v; complete sorts s, s_1, ... , s_n; complete function f; complete terms t, t_1, ... , t_n; and informals e_1, ... e_n such that quoted terms have a unique type.)

$$\begin{aligned} [\![v]\!]_{\mathcal{A},\alpha} &= \alpha(v) \\ [\![f\,(t_1\,, \ldots\,, t_n)]\!]_{\mathcal{A},\alpha} &= \mathcal{A}(f)([\![t_1]\!]_{\mathcal{A},\alpha}, \ldots, [\![t_n]\!]_{\mathcal{A},\alpha}) \\ [\![(v : s \mid t)]\!]_{\mathcal{A},\alpha}(x) &= [\![t]\!]_{\mathcal{A},\alpha[v \mapsto x]} \end{aligned}$$

Fig. 3.14: Semantics of terms. For any algebra $\mathcal{A}$, value context α, variable v, function f, terms t, t_1, ... , t_n, sort s, and value $x \in \mathcal{A}(s)$.

definition of $[\![\ ]\!]$ is given in Figure 3.14. The value of a variable is taken directly from the value context. The value of a function application is obtained by applying the mapping denoted by the function name to the values of its arguments. The value of a lambda abstraction is the mapping that maps any possible value of its bound variable to the value of its body (which may contain occurrences of the bound variable and, therefore, is dependent on its value). Note that $[\![(v\texttt{:}s \mid t)]\!]_{\mathcal{A},\alpha}$ is a mapping and, therefore, can be applied to an object x. The notation $\alpha[v \mapsto x]$ is used to denote the assignment of value x to variable v in α (leaving the value of all other variables unchanged, but possibly overwriting the previous value of α for v). The value of an informal term is undefined, which is due to the fact that natural language does not have a formal semantics. In Section 3.11 the informal value of informal terms is discussed.

3.10.1 Formula

A formula is a complete term of type `BoolS`. Each assertion $q\ t$ (with qualifier q and term t) in a specification must be such that t has exactly one expansion ϕ that is a formula (that is, ϕ is the only formula such that $t \twoheadrightarrow \phi$). This guarantees that, for a given algebra and value context, all assertions in a specification have an unambiguous boolean value.

A formula ϕ is called *satisfied* by an algebra $\mathcal{A}$ if and only if the (informal) value of ϕ for $\mathcal{A}$ is $true$ for any arbitrary value context. If $\mathcal{M}$ has no imports, then an algebra $\mathcal{A}$ is called a *model* of $\mathcal{M}$ if and only if all expanded assertions of $\mathcal{M}$ are satisfied by $\mathcal{A}$. If $\mathcal{M}$ does have imports, then $\mathcal{A}$ is a model of $\mathcal{M}$ if and only if $\mathcal{A}$ is a model of the unfolding of $\mathcal{M}$ (see Section 3.14). A formula ϕ is called a *consequence* of $\mathcal{M}$ if and only if all the models of $\mathcal{M}$ satisfy ϕ.

3.11 Informal Terms

An informal term is a term that contains natural language and as such has no formal semantics. This explains the name *Almost* Formal Specification Language. Informal terms can be used if the exact formulation of an assertion is not that important, or as an informal explanation besides a formal assertion. Examples of the use of informal terms can be found in both `Bit1M` and `BitList1M`:

```
LEMMA  {'BitS' is the sort of bits with two distinct elements
       'Zero' and 'One'.}
AXIOM  l1 ++ l2 = {the concatenation of 'l1' and 'l2'}
```

3.11.1 Syntax

An *informal term* is a sequence (enclosed by brackets) of *natural language* (which can be almost any sequence of symbols, see Section 3.2), *quoted names* (a name enclosed by `'`), and *quoted terms* (a term enclosed by `'`). Examples of informal terms are:

```
{'BitS' is the sort of bits}
{the concatenation of 'l1' and 'l2'}
{'l1++l2' is the concatenation of 'l1' and 'l2'}
```

Quoted names and terms allow tools to recognize the formal parts of an informal term and check whether they are well-formed. A quoted function name f is considered a quoted name rather than a quoted term (f is a name but also an alternative notation for the term `()`f`()`).

3.11.2 Type and Semantics

Due to its informal nature it is impossible to associate a formal type with an informal term. Therefore, an informal term can have any type, provided its quoted names are non-overloaded and its quoted terms have exactly one well-typed expansion. This explains the last typing rule in Figure 3.13. Quoted terms must have unique well-typed expansions because their context does not provide any information to resolve overloading.

An informal term t also does not have a formal value $[\![t]\!]_{\mathcal{A},\alpha}$. However, a complete informal term $t : s$ does have an *informal value*, which is the *natural interpretation* of t in s. This means that t should be interpreted according to the "common" meaning of the natural language in t and the formal meaning of the quoted names and terms within t. If t does not have a natural interpretation in s, then the informal value of t is arbitrary. Some examples of natural interpretation are given in Figure 3.15.

3.11.3 Informal Terms vs. Comment

Informal terms are different from comments. First there is a difference in use. Comments are used for documentation only, they do not contribute to the definition of the names declared in a specification. Informal terms, however, are used to put restrictions on the definitions of names. They cannot be left out without changing the informal meaning of a specification. Secondly, there is a syntactic difference: an informal term can be used *instead* of a formal term, whereas comment can only be *added* to a specification. Finally, the typing rules extend to the quoted terms within informal terms, whereas nothing prevents comment from containing ill-typed terms.

$$[\![\{\texttt{'1+2'}\}]\!]_{\mathcal{A},\alpha} = 3$$

$$[\![\{\texttt{'BitS' is the sort of bits}\}]\!]_{\mathcal{A},\alpha} = \begin{cases} \mathit{true} & \text{if } \mathcal{A}(\texttt{BitS}) \text{ is the sort of bits} \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$[\![\{\texttt{the concatenation of 'l1' and 'l2'}\}]\!]_{\mathcal{A},\alpha} = \text{the concatenation of } \alpha(\texttt{l1}) \text{ and } \alpha(\texttt{l2})$$

$$[\![\{\texttt{'l1++l2' is the concatenation of 'l1' and 'l2'}\}]\!]_{\mathcal{A},\alpha} = \begin{cases} \mathit{true} & \text{if } \mathcal{A}(\texttt{++})(\alpha(\texttt{l1}), \alpha(\texttt{l2})) \text{ is the concatenation} \\ & \text{of } [\![\texttt{l1}]\!]_{\mathcal{A},\alpha} \text{ and } [\![\texttt{l2}]\!]_{\mathcal{A},\alpha} \\ \mathit{false} & \text{otherwise} \end{cases}$$

Fig. 3.15: The informal value of some example informal terms. (For given algebra $\mathcal{A}$ and value context α.)

3.12 Assertions

There are three kinds of assertions: *axioms* (indicated by the qualifier `AXIOM`), *requirements* (indicated by `REQ`), and *lemmas* (indicated by `LEMMA`). All three forms can be found in the example module `BitList1M`. There is no difference between the three kinds of assertions in syntax, typing, or semantics. The different kinds of assertion are used for different purposes and are therefore qualified as such. It is the same distinction that is common in mathematical theories that usually contain definitions (axioms), assumptions (requirements), and conclusions (lemmas).

The three kinds of assertions give rise to three different classes of models: the *axiom models* that satisfy all axioms, the *lemma models* that satisfy all lemmas, and the *requirement models* that satisfy all requirements.

Axioms are used to do the actual specification (that is, definition of sorts and functions). If one is only interested in what is specified by a module, lemmas and requirements can be ignored.

Requirements are used whenever actual specification of a name x declared in module $\mathcal{M}$ is not yet possible or desirable in $\mathcal{M}$, but some restrictions need to be put on future specification of x in any module that imports $\mathcal{M}$. A module is called *satisfied* if and only if all its requirements are a consequence of its axioms (that is, the class of requirement models includes the class of axiom models). For example, in module `BitList1M` of Figure 3.2 the relation `<=` is declared without a definition, but with the requirement that `<=` must be a total order; a module that imports `BitList1M` inherits this requirement. This example is somewhat artificial. However, suppose an ordering on lists of bits is needed for efficient searching in a database of lists, then the actual ordering is irrelevant, as long as it is a total order. Requirements are also used to restrict the possible values of formal parameters (see Chapter 4).

Lemmas are primarily intended for validation purposes (a specification adequately models some system). A module is said to be *valid* if and only if all its lemmas are a consequence of its axioms and requirements (that is, the class of lemma models includes the intersection of the classes of axiom and requirement models). Lemmas can also be seen as a way of documenting a specification, since lemmas give a different view on what is specified by the axioms and requirements.

Why can lemmas and requirements not all be changed into axioms? In fact they can, but that conceals the fact that definitions should be consistent with the lemmas and requirements, which then are not recognizable as such anymore. Of course, all definitions should be consistent with other axioms; but proving validity of lemmas and satisfaction of requirements is less work than proving consistency of all axioms, and probably more relevant. Another advantage of separating axioms from lemmas and requirements is that the latter are not part of the actual specification. If a specification is used as the basis of the realization of a system, the lemmas and requirements are ignored.

Module validation (proving it is valid) and verification (proving it is satisfied) can in principle be done by formal proof, possibly supported by an automatic proof assistant. However, formal proof rules for AFSL are outside the scope of this thesis, which is primary focused on the expressiveness of formal specification languages rather than their mathematical properties.

A minor disadvantage of the use of requirements, instead of axioms, for the specification a name x is that the eventual definition of x often needs to replicate the requirements as axioms. If the requirements were replaced by axioms, the specification only needs to be extended to complete the definition. It is unlikely that this is a real disadvantage, since it is clearer to have a self contained definition anyhow.

3.13 Inductive Sorts

A sort can be declared to be inductively defined, which means that all its elements can be constructed by using a finite number of so-called constructors. An *inductive sort* is a sort with the attribute `INDUCTIVE`. A *constructor* of an inductive sort s is a function with co-domain s and attribute `CONSTRUCTOR`. Constructors of s have to be declared in the same module as s in order to guarantee the monotonicity of module import. That is, if module $\mathcal{M}_1$ imports $\mathcal{M}_2$ then all derived properties of $\mathcal{M}_2$ also hold for $\mathcal{M}_1$. It is allowed for the domain of a constructor to contain non-inductive sorts.

In the example modules, `BitS` and `BitListS` are inductive sorts and `Zero`, `One`, `Empty`, and `&` are constructors. No non-inductive sorts are declared in the example, but in general modules may contain non-inductive sorts. Free sorts occur in particular as formal parameters (see Section 3.3). For example, in Figure 4.4 a general specification of lists is given where the type `XS` of the elements in the lists is a formal parameter and as such a non-inductive sort. But also non-parameters can be non-inductive sorts, for example, if a sort cannot be fully specified (yet).

The role of inductive sorts is similar to that of algebraic data types in functional programming. For example, the specification of the inductive sorts `BitS` and `BitListS` is similar to the following definitions of algebraic data types in the programming language Haskell (Bird 1998) (the identifiers `Zero`, `One`, and `&` are not allowed as functions in Haskell and, therefore, are replaced by `Zero`, `One`, and `Add`):

```
data BitS     = Zero | One
data BitListS = Empty | Add BitS BitListS
```

Contrary to Haskell, in AFSL inductiveness of a sort s does not imply that elements of s that are constructed with different constructors are not equal. For example, without the inequality axiom in `Bit1M`, it is not prohibited that `Zero` is equal to `One`.

3.13.1 Formal Definition of Inductiveness

Some auxiliary definitions are needed in order to define what it exactly means for a sort to be inductive. Given a signature and type context, a *semi ground term* is a term that consists of constructors and variables of non-inductive sorts. Given a signature Σ, an algebra $\mathcal{A}$ is *minimal* if and only if for each sort s in Σ and object $x \in \mathcal{A}(s)$ there exist a type context Γ, value context α, and semi ground term $t : s$ such that $x = [\![t]\!]_{\mathcal{A},\alpha}$ (that is, t represents x). Note that each non-inductive sort satisfies this condition by definition. In a minimal algebra all elements of inductive sorts can be constructed by a finite number of constructors (and maybe some variables as placeholder for objects of non-inductive sorts). A semi ground term is similar to what is known as a ground term in the theory of initial semantics for algebraic languages, except that a ground term may contain any function (all functions are considered constructors), and no variables of non-inductive sorts (all sorts are inductive with initial semantics).

All models of an AFSL specification must be minimal, which limits the possible interpretation of inductive sorts. For example, the only elements of `BitS` are (the values of) `Zero` and `One`, since these are the only semi ground terms for `BitS`. A semi ground term for `BitListS` is of the form `Empty` or t_1`&`$\cdots$`&`t_n`&Empty` for terms $t_1, \ldots, t_n \in \{$"`Zero`", "`One`"$\}$). So, `BitListS` contains the empty list and lists with a finite number of elements of `BitS`. In these examples all relevant sorts are inductive, no variables are possible in the semi ground terms of either `BitS` or `BitListS`. In the parameterized module `List1M` (Figure 4.4) the formal parameter `XS`, the sort of the elements of the list, is non inductive. There the semi ground terms of `ListS[XS]` are of the form `Empty` or v_1`&`$\cdots$`&`v_n`&Empty` for some type context that declares the variables $v_1, \ldots, v_n$ to be of type `XS`. Now, `ListS[XS]` still contains the empty list and lists with a finite number of elements of `XS`, but the variables in the semi ground term function as placeholders for arbitrary elements of `XS`.

3.14 Module Import

The import of a module has the effect of including the body of the imported module. The process of recursively replacing each import in a module $\mathcal{M}$ by the body of the imported module is called *unfolding*. For example, the unfolding of the specification of lists of bits `BitList1M` is given in Figure 3.16. With respect to the expressiveness of the language the import mechanism is redundant, since a module can always be replaced by its unfolded version. The import mechanism is, however, an important tool for structuring and reuse of specifications.

By definition, the signature of a structured module is the signature of its unfolding, a structured module is well-formed if its unfolding is well-formed. The semantics of a structured module equals the meaning of the unfolded version.

```
MODULE BitList1M

VAR    x : BitS

SORT   Bit1M:BitS (INDUCTIVE)
FUNC   Bit1M:Zero : -> BitS (CONSTRUCTOR)
FUNC   Bit1M:One  : -> BitS (CONSTRUCTOR)
LEMMA  {'BitS' is the sort of bits with two distinct elements
       'Zero' and 'One'.}
AXIOM  Zero /= One

FUNC   Bit1M:* : BitS, BitS -> BitS
AXIOM  Zero * x = Zero
AXIOM  One  * x = x

%... unfolding rest of Bit1M, unfolding of LogicM and NatM

VAR    b : BitS
VAR    l : BitListS

SORT   BitListS (INDUCTIVE)
FUNC   Empty :                -> BitListS (CONSTRUCTOR)
FUNC   &     : BitS, BitListS -> BitListS (CONSTRUCTOR)
AXIOM  b&l   /= Empty
AXIOM  b1&l1 /= b2&l2  <==  b1 /= b2  Or  l1 /= l2

%... rest of the body of BitList1M

END MODULE
```

Fig. 3.16: Unfolding module `BitList1M` of Figure 3.2.

Unfolding involves a bit more than just copying the bodies of imported modules. Some changes have to be made in the bodies of the imported modules to ensure that the unfolded module has the right signature and assertions:

- All occurences of names in the imported module must be completed. This is necessary because the unfolded module has a larger signature than the imported module. This may lead to an overloading in the included body which cannot be resolved in the context of the importing module or will be resolved differently. Completion also avoids that names declared in the imported module get the name of the importing module as their origin (that is, `BitList1M:Zero` instead of `Bit1M:Zero`). Completion of names in assertions involves overloading expansion, implicit functions expansion (see Chapter 5), and implicit binding expansion (See Chapter 7).

- If an imported module $\mathcal{M}$ is parameterized (see Chapter 4) all occurrences of formal parameters in the body of $\mathcal{M}$ are replaced by the corresponding actual parameter before the body of $\mathcal{M}$ is copied to the importing module.

- If, after unfolding, the same name declaration occurs multiple times, then all but the first are removed. Multiple declaration occurs if the same module is imported multiple times or because formal parameters are replaced by actual parameters.

- If necessary, variables of the imported module are renamed to avoid multiple declarations of the same variable. This is the reason why the variable `b` of `Bit1M` is renamed to `x` in Figure 3.16.

The result of unfolding of a module $\mathcal{M}$ depends on the specification which $\mathcal{M}$ is part of. The bodies of imported modules are obtained from that specification. If an imported module cannot be found in the specification $\mathcal{M}$ is ill-formed.

Unfolding would result in an infinite body if a module imports itself directly or through intermediate module. Therefore, circular imports are not allowed.

Completion of names and expansion of assertions may lead to a syntactic explosion of imported modules. For example, the complete specification of `BitS` in Figure 3.16 should really be:

```
SORT   Bit1M:BitS (INDUCTIVE)
FUNC   Bit1M:Zero : -> Bit1M:BitS (CONSTRUCTOR)
FUNC   Bit1M:One  : -> Bit1M:BitS (CONSTRUCTOR)
LEMMA  {'Bit1M:BitS' is the sort of bits with two distinct
       elements 'Bit1M:Zero:->Bit1M:BitS' and
       'Bit1M:One:->Bit1M:BitS'.}
AXIOM  Bit1M:Zero:->Bit1M:BitS
       BuiltinM:/=:Bit1M:BitS,Bit1M:BitS->BuiltinM:BoolS
       Bit1M:One:->Bit1M:BitS
```

To reduce the syntactic overhead unfolded modules may be simplified (by using abbreviations, implicit functions, and implicit bindings) provided the simplified module has the same expansion as the non-simplified version. The examples of unfolded modules in this thesis will always be simplified. Note that it is a non-trivial exercise to automate such a simplification task.

4. GENERIC MODULES

Abstract

A *generic module* is a parameterized module, that is, a module with a non-empty list of formal parameters. Module parameters are names that have to be declared somewhere within the module. The imports of a generic module must be supplied with names as actual parameters. When unfolding the import of a generic module all occurrences of its formal parameters are replaced by the corresponding actual parameters before it is copied to the importing module. This includes occurrences of formal parameters in names (as indexes or as part of function types). This causes these names to behave like a kind of polymorphic sorts/functions.

The parameter mechanism of AFSL is significantly different from that of most other specification languages. The pro's and cons of the different kinds of generic modules are discussed in Section 2.1.4. This chapter gives a number of examples of different uses of generic modules and discusses restrictions that apply to parameters.

4.1 Example: Generic Assertions

A simple use of a generic module is to collect assertions for a parameterized name x in a separate building block, which can be used many times with different substitutions for x. An example of such a collection of generic assertions is given in Figure 4.1, where it is required that the parameterized relation `R` is a total order (`XS` is an auxiliary formal parameter needed to pass the domain type of `R`). Module `TotalOrderM` is used in Figure 4.2 for the specification of lists of bits where it replaces the four requirements for `<=` of `BitList1M` in Figure 3.2. The unfolding of this new version of `BitList2M` (given in Figure 4.3) is equivalent to `BitList1M`.

The effect of replacing formal parameters by actual parameters during unfolding is similar to renaming (see Section 2.1.4). AFSL does not have a renaming construct, but a hypothetical extension of the language could allow an import with renaming like:

```
MODULE TotalOrderM [XS, R]

VAR    x : XS

IMPORT LogicM % Fig. 3.3 page 43

SORT   XS

FUNC   R : XS, XS -> BoolS
REQ    x  R x
REQ    x1 R x2  And  x2 R x1  ==>  x1 = x2
REQ    x1 R x2  And  x2 R x3  ==>  x1 R x3
REQ    x1 R x2  Or   x2 R x1

END MODULE
```

Fig. 4.1: Specification of a total ordering.

```
MODULE BitList2M

VAR    b : BitS
VAR    l : BitListS

IMPORT Bit1M % Fig. 3.1 page 40

SORT   BitListS
%... definition of BitListS

FUNC   <= : BitListS, BitListS -> BoolS
IMPORT TotalOrderM [BitListS, BitList2M:<=] % Fig. 4.1 page 68

%... definition of operations on lists

END MODULE
```

Fig. 4.2: Reformulation of the specification of lists of bits of Figure 3.2, now using a generic module for the requirements on `<=`. Note that the import argument `<=` needs its origin because it is overloaded, there is also one for `BitS`.

```
MODULE BitListM

VAR    b : BitS
VAR    l : BitListS

%... unfolding of BitM
%... unfolding of SystemM

SORT   BitListS
%... definition of BitListS

FUNC   <= : BitListS, BitListS -> BoolS
VAR    x : BitS
REQ    x  <= x
REQ    x1 <= x2  And  x2 <= x3  ==>  x1 <= x3
REQ    x1 <= x2  And  x2 <= x1  ==>  x1 = x2
REQ    x1 <= x2  Or   x2 <= x1

%... definition of operations on lists

END MODULE
```

Fig. 4.3: Unfolding of module `BitList2M` of Figure 4.2.

```
IMPORT TotalOrderM RENAMING XS TO BitListS
                   ,        R  TO BitList2M:<=
                   END
```

as an alternative for the import statement of `TotalOrderM` (the parameter list in `TotalOrderM` can then be omitted). A renaming construct is more flexible than the parameter mechanism because it is not fixed which names have to be renamed. On the other hand, imports with actual parameters have less syntactic overhead, since the names that have to be replaced do not have to be mentioned.

Parameter replacement differs from renaming in a subtle way: formal parameters are *replaced* by existing names. The actual parameters of an import must be names that are declared before the import. That is why declarations of formal parameters are removed from the imported module during unfolding: the replacements are already declared (see Section 3.14). Renaming does only *change the syntax* of names declared in the imported module. The declarations of the names that are renamed are not affected in any way. For example, if the import of `TotalOrderM` in `BitList2M` were replaced by the above mentioned import with renaming, then the unfolding of `BitList2M` would contain declarations of two different versions of `BitListS` and `<=`. One couple will have origin `BitList2M`, the other `TotalOrderM`. Both versions will be syntactically similar, but semantically unrelated. This is not the intention of the import of `TotalOrderM`, which is meant to add requirements for the `<=` of `BitList2M`.

4.2 Restrictions on Parameters

Within a module there is no distinction between a formal parameter and other names: it must be declared, assertions can use it, it can be used as an index, it can be used as actual parameter, etc. Formal parameters do not need to be introduced directly in the beginning of a module. This can be useful when the type of a parameter contains a sort which has to be imported first from another module. A formal parameter is not allowed to be an indexed name, inductive sort, constructor, or implicit function (implicit functions are discussed Chapter 5).

The actual parameters of an import must be declared before the import. In order to ensure that an unfolded module is well-typed, each actual parameter x must be of the same kind (sort or function) and type (in case of a function) as the corresponding formal parameter y. If y is a function, the type of y may contain other formal parameters. For example, the type of `R` in `TotalOrderM` contains `XS`. In that case, x must be of the same type as y after substitution of the formal parameters in the type of y by the corresponding actual parameters. For example, for the import of `TotalOrderM` in `BitList1M` the actual parameter `<=` must be of type `BitListS, BitListS -> BoolS`, which it indeed is.

```
MODULE List1M [XS]

VAR    x : XS
VAR    l : ListS[XS]

SORT   XS

IMPORT LogicM % Fig. 3.3 page 43
IMPORT NatM   % Fig. 5.8 page 94

SORT   ListS[XS] (INDUCTIVE)
FUNC   Empty :               -> ListS[XS] (CONSTRUCTOR)
FUNC   &     : XS, ListS[XS] -> ListS[XS] (CONSTRUCTOR)

AXIOM  x&l   /= Empty
AXIOM  x1&l1 /= x2&l2  <==  x1 /= x2  Or l1 /= l2

FUNC   <= : ListS[XS], ListS[XS] -> BoolS
IMPORT TotalOrderM [ListS[XS], <=] % Fig. 4.1 page 68

FUNC   Length : ListS[XS] -> NatS
AXIOM  Length Empty  = 0
AXIOM  Length (x&l)  = Length l + 1

%... definition of other operations on lists

END MODULE
```

Fig. 4.4: Specification of typed lists.

4.3 Example: Parametric Overloading

A *generic name* is a name which contains some formal parameters as part of its complete name (as indexes or in its type info). Recall that an indexed name is a name which has a sequence of names (its indexes) directly following its identifier (see Section 3.4). In the previous section `TotalOrderM` has no generic names. The generic specification of lists in Figure 4.4 does declare a generic sort name `ListS[XS]` (with index `XS`) and generic functions `Empty`, `&`, `<=`, and `Length` (`XS` occurs in the types of these functions).

When unfolding a module import the occurrences of formal parameters in a generic name are replaced by the corresponding actual parameters. This changes the generic name into a new name called an *instantiation*. Thus, an import of a generic module results in a renaming of all its generic names. This is true renaming: for-

```
MODULE ListExampleM

IMPORT Bit1M            % Fig. 3.1 page 40
IMPORT List1M [BitS]    % Fig. 4.4 page 71
IMPORT List1M [StringS] % Fig. 4.4 page 71

END MODULE
```

Fig. 4.5: Example of imports of the parameterized list module.

mal parameters are replaced, generic names are renamed. Multiple imports of the same module may result in different instantiations of the same generic name. This causes generic names to behave like a kind of polymorphic sorts/functions. This is demonstrated in `ListExampleM` given in Figure 4.5, the unfolding of which is given in Figure 4.6. In `ListExampleM` two different list sorts are declared (`ListS[BitS]` and `ListS[StringS]`), each with their own set of operations. Note that in the unfolding of the definitions of the two `Length` functions the occurrences of `Empty` are, partly, completed in order to avoid overloaded abbreviations.

4.4 Polymorphism

The use of indexes is essential for specifying polymorphic sorts, which is the reason why indexes are available in AFSL. Without the index `XS` in `ListS[XS]` two different imports of `List1M` would declare only one list sort `ListS` (the sort of *all* lists) with one constant `Empty` (*the* empty list), and two versions of `&` (since `XS` is then still part of the type of `&`). Because the type of a function is part of its complete name, there is no need to use indexes for functions here. For example, in `ListExampleM` two distinct constants for the empty list are declared: `Empty:->ListS[BitS]` and `Empty:->ListS[StringS]`. Functions can have indexes, but those are used for other purposes than polymorphism (see the example in Section 4.8).

The form of polymorphism demonstrated here is not quite the same as it is known in many functional programming languages, where type variables can be replaced by any arbitrary type (sorts are called types in programming languages) without explicit instantiation. For example, the module `List1M` can be formulated in the functional language Haskell (Fasel et al. 1992, Bird 1998) as follows (here `&` is renamed to `Add` and `Length` to `len`):

```
MODULE ListExampleM

%... unfolding of Bit1M

VAR    b  : BitS
VAR    bl : ListS[BitS]

%... unfolding of LogicM and NatM

SORT   List1M:ListS[BitS] (INDUCTIVE)
FUNC   List1M:Empty :                    -> ListS[BitS] %...
FUNC   List1M:&     : BitS, ListS[BitS] -> ListS[BitS] %...
%... equality axioms for ListS[BitS]

FUNC   <= : ListS[BitS], ListS[BitS] -> BoolS
%... unfolding of TotalOrderM [ListS[BitS], <=]

FUNC   List1M:Length : ListS[BitS] -> NatS
AXIOM  Length Empty:->ListS[BitS] = 0
AXIOM  Length (b&bl)              = Length bl + 1
%... unfolding rest of List1M [BitS]

VAR    s  : StringS
VAR    sl : ListS[StringS]

%... unfolding of LogicM and NatM

SORT   List1M:ListS[StringS] (INDUCTIVE)
FUNC   List1M:Empty :                       -> ListS[StringS] %...
FUNC   List1M:&     : StringS, ListS[StringS] -> ListS[StringS] %...
%... equality axioms for ListS[StringS]

FUNC   <= : ListS[StringS], ListS[StringS] -> BoolS
%... unfolding of TotalOrderM [ListS[StringS], <=]

FUNC   List1M:Length : ListS[StringS] -> NatS
AXIOM  Length Empty:->ListS[StringS] = 0
AXIOM  Length (s&sl)                 = Length sl + 1
%... unfolding rest of List1M [StringS]

END MODULE
```

Fig. 4.6: The unfolding of `ListExampleM`.

```
module ListM where

data ListS xs  = Empty | Add xs (ListS xs)

len :: ListS xs -> Int
len Empty      = 0
len (Add x xl) = (len xl) + 1
```

One import of `ListM` is sufficient to declare `ListS xs` for any possible substitution of `xs`. In AFSL each instantiation of `ListS[XS]` has to be created explicitly by an import of `List1M`.

In Haskell `len` is a function with a polymorphic function type. AFSL does not have polymorphic function types. Instead, the parameter mechanism is used to create overloaded versions of `Length`, one for each type of list. Actually, this kind of overloading resembles the concept of type class in Haskell rather than that of polymorphic function types.

Overloading is sometimes called *ad hoc polymorphism* and the kind of polymorphism used in the Haskell example is then called *parametric polymorphism*. Overloading is considered ad hoc because in general different instances of overloaded names are semantically unrelated, whereas a parametric polymorphic function behaves uniformly for each instance. The way different instances `Length` are overloaded is not that ad hoc since they all are defined in the same way. Therefore, the terminology *parametric overloading* is used here for the kind of overloading caused by different instantiations of a generic name.

To keep the language simple, adding parametric polymorphism to AFSL has never been considered although potentially it has some important benefits (see Section 8.5). Parametric overloading has some advantages too. First, it is a relatively simple language feature. Provided a language already has overloading and generic modules, it only needs indexed names; which is a simple syntactic mechanism that does not affect the semantics of the language. Second, parametric overloading is more flexible than parametric polymorphism, as is shown in the examples given in Sections 4.5 and 4.8.

4.5 Example: Restricted Overloading

For some generic names not all instantiations are desirable. Explicit instantiation of generic names provides control over which instantiations are allowed. With parametric polymorphism this is not possible: then it is all or nothing. Restrictions on instantiations of a generic name are given by requirements. For example, module `TotalOrderedM` in Figure 4.7 does specify generic names `<=` and `<=`. The imported requirements from `TotalOrderM` restrict instantiations of `>=` to total orders. That is,

```
MODULE TotalOrderedM [XS]

VAR    x : XS

SORT   XS

FUNC   <= : XS, XS -> BoolS
IMPORT TotalOrderM [XS, <=] % Fig. 4.1 page 68

FUNC   >= : XS, XS -> BoolS
AXIOM  x1 >= x2  <=>  x2 <= x1

%... other operations that can be defined with <=, such as < and Min

END MODULE
```

Fig. 4.7: Example specification of generic functions.

in a satisfied module an instantiation of `<=` is only allowed if it is defined as a total order.

Without restrictions the definition of an instantiation of `<=` can be completely arbitrary. This may easily lead to confusing specifications, since the symbol `<=` suggests a total order (or at least, some kind of well-behaving order). Therefore, requirements are put on the future definitions of `<=` to guarantee some common properties for different instantiations of `<=`.

The relation `<=` cannot be defined within `TotalOrderedM` because nothing is known about `XS` there. The definition of `<=` is, therefore, postponed to the modules that import `TotalOrderedM`. Note that `>=` is completely defined within the module `TotalOrderedM`, although its actual meaning depends on the future definition of `<=`.

Figure 4.8 shows how bits can be specified without locally declaring `<=`, using the generic `<=` imported by `TotalOrderedM` instead. The restrictions on the definition of `<=` in `TotalOrderedM` are satisfied by the defining axioms for `<=` in `Bit2M`. It does not need to be the case that the restrictions are satisfied right away in the importing module. For example, in Figure 4.4 the declaration of `<=` together with the import of `TotalOrderM` can be replaced by an import of `TotalOrderedM`, the definition of `<=` is then forwarded to the module that imports `List1M`.

4.6 Coherent Overloading

Apart from the usual advantages of modularization, the declaration of overloaded names as a single generic name with a forwarded definition (such as the function `<=`

```
MODULE Bit2M

VAR    b : BitS

SORT   BitS (INDUCTIVE)
FUNC   Zero : -> BitS (CONSTRUCTOR)
FUNC   One  : -> BitS (CONSTRUCTOR)
LEMMA  {'BitS' is the sort of bits with two distinct elements
       'Zero' and 'One'}
AXIOM  Zero /= One

FUNC   * : BitS, BitS -> BitS
AXIOM  Zero * b = Zero
AXIOM  One  * b = b

IMPORT TotalOrderedM [BitS] % Fig. 4.7 page 75

AXIOM  Zero <= b
AXIOM  b    <= One

END MODULE
```

Fig. 4.8: Reformulation of the specification of bits of Figure 3.1, now using a generic module to introduce <=.

in `TotalOrderedM`) has an additional benefit. The requirements on the definition of a generic name n guarantee some coherence among the definitions of the different instances of n. For example, we know for sure that in a satisfied specification (see Section 3.12) every instance of the `<=` of `TotalOrderedM` is a total ordering. Although the definition of `<=` is not contained in the generic module that declared it, parametric overloading is less ad hoc than unrelated overloading.

The advantage of this *coherent overloading* is that it makes the properties of overloaded names more predictable. It is a good modeling principle to have a clear relation between names and their meaning; it makes the meaning of names easier to understand and remember. Knowing the meaning of the individual words is essential for understanding a text. Maintaining a relationship between syntax and semantics is difficult for an overloaded name since it has more than one meaning. The context of the use of an overloaded name must provide additional clues about its meaning in that case (see Section 3.8). Coherent overloading guarantees that the different meanings of an overloaded name have at least some properties in common, regardless of the context of its use. Within the FSA case studies it was a guideline to avoid unrelated overloading whenever possible, either by using parametric overloading or by using inheritance (see Chapter 5). It turned out that elimination of unrelated overloading can be a driving force in the process of structuring a specification and creating reusable modules.

Coherent overloading does not come automatically, even if the only form of overloading used is parametric. The level of agreement in the semantics of different instances of a generic name depends on the actual choice of requirements. This is a modeling decision which has to be made with care. Too many restrictions make a generic module hard to reuse, to few weakens coherence. A balance between reusability and coherence is not easy to find if it is not known in advance how a generic name is going to be used. That is why well-established mathematical properties were used as requirements for the generic names declared in the library of the FAN case study (Groenboom 1997) (for example, `<=` must always be a total order). Even if mathematical properties are used, there can be discussion about the level of coherence. For example, should `<=` be a total order or just a partial order?

If coherent overloading is that desirable, why not forbid any other form of overloading than parametric? For example, the only way overloading can be realized in the programming language Haskell is through the use of type classes (which are a kind of generic modules with generic names). However, the possibility to allow plain overloading can be handy in some situations. First, if modules are written independently of each other unintended overloading may occur. Secondly, in the early stage of writing a specification elimination of ad hoc overloading may not yet be a priority. Finally, there can be forms of non-parametric overloading that are still coherent, such as restriction overloading (see Section 4.5).

```
MODULE PlusM [XS]

VAR    x : XS

SORT   XS

FUNC   Zero : -> XS

FUNC   + : XS, XS -> XS
REQ    x  + Zero      = x
REQ    x1 + x2        = x2 + x1
REQ    (x1 + x2) + x3 = x1 + (x2 + x3)

END MODULE
```

Fig. 4.9: Example specification of generic functions.

4.7 Example: Arithmetic Operations

Coherent overloading is demonstrated in Figure 4.9 where `+` is an addition operation that forms an Abelian monoid with identity element `Zero`. Figure 4.10 extends this specification with `--` as a unary minus that forms an Abelian group with `+` and `Zero`. Note that without its requirements `PlusMinusM` is not a valid module, since the lemmas do depend on the requirements. `PlusMinusM` is used in the specification of integers given in Figure 4.11.

The specification of arithmetic operations could be extended to a hierarchy of modules like in Figure 4.12. The AFSL library discussed in Groenboom (1997) contains a small fragment of this hierarchy (with different names than used here) plus the specifications of other generic functions like boolean operations and ordering relations. The properties “Abelian monoid” and “Abelian group” could be defined in separate modules, just like “total order” is defined in `TotalOrderM`. Such modules are omitted here for brevity. The library of Groenboom (1997) does specify general algebraic structures like field, group, and ordering. These specifications follow the definitions in the Larch library (Guttag & Horning 1993).

Some of the axioms for `+` in `Int1M` (for example, the first axiom) could be omitted if the requirements of the imported modules were changed into axioms. In general, one can reduce the total number of assertions in a specification if all requirements are changed into axioms. If the assertion for a generic function are axioms, instead of requirements, they serve as a partial definition which have to be completed in a later stage. It is a matter of style whether generic functions are restricted by requirements (as in the given examples) or partly defined by axioms. However, the use of requirements is preferred for a number of reasons. First, it is clearer if all axioms for

```
MODULE PlusMinusM [XS]

VAR    x : XS

SORT   XS

IMPORT PlusM [XS] % Fig. 4.9 page 78

FUNC   -- : XS -> XS
REQ    x + (-- x)   = Zero
LEMMA  -- Zero      = Zero
LEMMA  -- -- x      = x
LEMMA  -- (x1 + x2) = --x1 + --x2

FUNC   - : XS, XS -> XS
AXIOM  x1 - x2 = x1 + --x2
LEMMA  x1 - x2 = -- (x2 - x1)

END MODULE
```

Fig. 4.10: Example specification of generic functions. (Here `--` is used as unary operation because `-` is used as infix operator.)

a particular instantiation of a generic function are grouped together; if a definition is scattered, contradicting axioms may be overlooked. Second, requirements stand out as properties that have to be met (maybe even proven) by the eventual definitions.

4.8 Example: Higher-Order Names

Up till now indexes only occurred in polymorphic sorts like `FuncS[`s_1`,`s_2`]` and `ListS[`s`]`. Here both the indexed names as well as the indexes are sorts. However, indexes are not restricted to sorts, functions also can have indexes and indexes can be functions. It is unlikely there is much use for a sort as index of a function because its type will most likely already contain that sort. But, functions with functions as indexes can be used to model higher-order functions (which is a function that takes another function as an argument or returns a function as result).

Higher-order constructs in programming languages have proven to be effective tools for abstraction, for example, to pass an ordering as argument to a sorting operation. Higher-order functions can be modeled in AFSL using function representations (see Section 3.6.2). For example, a higher-order function of type $(s_1 \to s_2) \to s_3$ can be modeled by a first order function f of type `FuncS[`s_1`,`s_2`]` $\to s_3$. However, a function $g : s_1 \to s_2$ cannot be passed directly as an argument to f, *function* g must

```
MODULE Int1M

VAR    i : IntS

SORT   IntS (INDUCTIVE)
FUNC   ZeroInt :      -> IntS (CONSTRUCTOR)
FUNC   Succ    : IntS -> IntS (CONSTRUCTOR)
FUNC   Pred    : IntS -> IntS (CONSTRUCTOR)

%... equality axioms for IntS

IMPORT PlusMinusM [IntS] % Fig. 4.10 page 79

AXIOM  Zero = ZeroInt

AXIOM  i  + Zero    = i
AXIOM  i1 + Succ i2 = Succ (i1 + i2)
AXIOM  i1 + Pred i2 = Pred (i1 + i2)

AXIOM  -- Zero   = Zero
AXIOM  -- Succ i = Pred -- i
AXIOM  -- Pred i = Succ -- i

%... other operations for IntS

END MODULE
```

Fig. 4.11: Specification of integers using generic functions `Zero`, `+` and `--`. (The constructor `ZeroInt` cannot be renamed to `Zero` here because that would result in unresolvable overloading of `Zero`.)

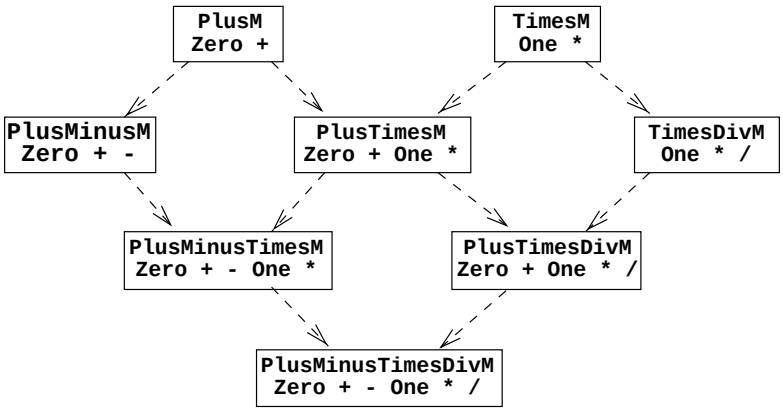

Fig. 4.12: A possible hierarchy of modules that specify generic arithmetical operations.

first be transformed to a *function representation* of type FuncS[s_1,s_2] (for example, by using the lambda-abstraction `(x:`s_1`|`g`(x))`).

A function g can be passed directly as a kind of argument to a function f if g is made an index of f. This is demonstrated in Figure 4.13 where `Sort` takes a total order `R` as "argument" to sort a list. Within `SortListM` formal parameter `R` can be viewed as the formal parameter of `Ordered`, `Insert`, and `Sort`. If `SortListM` is imported, for example, by:

```
IMPORT SortListM [BitS, <=]
```

then `Sort[<=]` will be defined on lists of bits. Here the index `<=` functions as an "argument" of `Sort`. Note that here `XS` does not need to be an index of `Ordered`, `Insert`, and `Sort` because it is already part of the types of these functions. Since these types are part of the complete names, different instantiations of `XS` will result in different declarations of these functions.

In module `SortListM` function `Sort` is not a higher-order function in the usual sense of the word. `Sort` is not a function at all, it is an integral part of the indexed name `Sort[R]`. For each function that occurs as an "argument" of `Sort` a separate import of `SortListM` is needed. And, finally, there is no form of lambda abstraction that can be used to create anonymous "arguments" for `Sort`, only named functions can be used as actual parameters. These shortcomings are the reason to consider `Sort` a *poor-mans higher-order function*.

The definition of poor-mans higher-order function `Sort` in `SortListM` has one major advantage over a true higher-order definition: the formal parameter `R` is explicitly restricted to total orders. This restriction is essential for the correct definition

```
MODULE SortListM [XS, R]

VAR    x  : XS
VAR    l  : ListS[XS]

SORT   XS

IMPORT List1M [XS] % Fig. 4.4 page 71

FUNC   R : XS, XS -> BoolS
IMPORT TotalOrderM [XS, R] % Fig. 4.1 page 68

FUNC   Ordered[R] : ListS[XS] -> BoolS
AXIOM  Ordered[R] Empty
AXIOM  Ordered[R] (x&Empty)
AXIOM  Ordered[R] (x1&x2&l)  <=>  x1 R x2  And  Ordered[R] (x2&l)

FUNC   Insert[R] : XS, ListS[XS] -> ListS[XS]
AXIOM  Insert[R] (x,  Empty) = x & Empty
AXIOM  Insert[R] (x1, x2&l)  = x1 & x2 & l           <==  x1 R x2
AXIOM  Insert[R] (x1, x2&l)  = x2 & Insert[R](x1,l)  <==  Not x1 R x2

FUNC   Sort[R] : ListS[XS] -> ListS[XS]
AXIOM  Sort[R] Empty = Empty
AXIOM  Sort[R] (x&l) = Insert[R] (x, Sort[R] l)
LEMMA  Ordered[R] Sort[R] l

END MODULE
```

Fig. 4.13: Specification of poor-man's higher-order functions `Ordered`, `Insert`, and `Sort`.

```
MODULE OrderedListM [XS, R]

VAR    x  : XS
VAR    ol : OrderedListS[R]

SORT   XS
FUNC   R : XS, XS -> BoolS

IMPORT TotalOrderM [XS, R] % Fig. 4.1 page 68

IMPORT SortListM [XS, R] % Fig. 4.13 page 82

SORT   OrderedListS[R] (INDUCTIVE)
FUNC   Empty  :                      -> OrderedListS[R] (CONSTRUCTOR)
FUNC   Insert : XS, OrderedListS[R] -> OrderedListS[R] (CONSTRUCTOR)

AXIOM  ol1 = ol2  <==  Elements ol1 = Elements ol2

FUNC   Elements : OrderedListS[R] -> ListS[XS]
AXIOM  Elements Empty          = Empty
AXIOM  Elements Insert (x, ol) = Insert[R] (x, Elements ol)
LEMMA  Ordered[R] Elements ol

END MODULE
```

Fig. 4.14: Specification of ordered lists.

of `Sort`. As a matter of fact, `SortListM` would not be formally valid without the import of `TotalOrderM`; it is needed to make the lemma true.

Functions can also occur as indexes of sorts, where they too serve as a kind of higher-order parameters. For example, in Figure 4.14 `OrderedListS[R]` is the sort of lists that are ordered according to the total order `R`. Higher order sorts are not available in common higher-order languages.

5. IMPLICIT FUNCTIONS

Abstract

In object-oriented methods for system modeling, or programming, the organization of datatypes in subclass hierarchies plays a central role in the structuring of models. The use of inheritance simplifies the construction, maintenance, and reuse of conceptual models. Since this is also a point of concern for formal specification, which is a form of system modeling, it seems worthwhile to incorporate an inheritance mechanism in a specification language.

In AFSL inheritance is realized by allowing implicit functions. Some example specifications are given that use this form of inheritance in a number of different ways. The examples are followed by a discussion of the ambiguity problems that can be caused by inheritance. A mechanism is introduced that can resolve many of these ambiguities.

5.1 Inheritance

The key aspect of the subclass relationship is that a "class" c_1 "inherits" all "properties" of any of its superclasses c_2, meaning that any operation defined for objects of class c_2 can also be applied to objects of class c_1. In AFSL inheritance is realized by allowing *implicit functions*, which are unary functions that are declared with the attribute `IMPLICIT`. An implicit function $f : s_1 \rightarrow s_2$ can be applied to the argument of a function $g : s_2 \rightarrow s_3$ without actually showing f. That is, for term $t : s_1$ the application $g(t)$ can be used as shorthand for $g(f(t))$. Here the effect of f being implicit is that s_1 inherits property g from s_2.

Implicit functions are also known as *coercions* (Cardelli & Wegner 1985). Implicit functions are similar to *injections* in ASF+SDF (Klint 1993), except that these injections are syntax-less, that is, they are denoted by an invisible name. The terminology "implicit function" is used in this thesis instead of "injection" because implicit functions do not need to be injective in the mathematical sense (that is, mapping unequal arguments to unequal results).

It is not the intention of AFSL to be an object-oriented language (according to whatever definition). Apart from implicit functions, there are no other language con-

structs that specifically facilitate object-oriented modeling. On the other hand, implicit functions can be used in situations where traditional object-oriented inheritance is not applicable, such as inheritance for has-a relationships (see the examples given in this chapter). In object-oriented methods objects usually have an internal state. For a proper modeling of states in AFSL so-called implicit lifting is used, which is not introduced until Chapter 6. Therefore, the given examples do not use state; in that respect the examples are atypical for object-oriented models.

There are no special restrictions on the possible values of an implicit function; implicit functions are in every respect ordinary functions, except that they can be used implicitly. We write $s_1 < s_2$ if and only if there is at least one implicit function $f : s_1 \to s_2$. Sort s_1 is said to be an *inheritor* of s_2 if and only if $s_1 <^* s_2$, where $<^*$ is the reflexive transitive closure of $<$. The terminology "s_1 is an inheritor of s_2" is used here instead of "s_1 is subsort of s_2" because the word "subsort" suggests that s_1 represents a subset of s_2, which does not need to be the case.

Making implicit functions in a term explicit is part of the expansion mechanism that was introduced in Section 3.8. That is, $t \twoheadrightarrow t'$ denotes that term t can be expanded to term t' by completing names and adding implicit functions, for example:

$$g(t) \twoheadrightarrow g(\underline{f}(t))$$

For clarity, added implicit functions are sometimes underlined. For simplicity, in examples the names in an expansion are often abbreviated. An implicit function that is inserted in a expansion is called a *conversion* (f in the example). The formal (re)definition of $\twoheadrightarrow$ is given in Section 5.8.

5.2 Example: List Notation

AFSL does not have a builtin sort for lists, therefore a generic specification of lists was given in 4.3. This self made definition of list does not allow the usual notation for lists (such as `[1,2,3]`) many languages have builtin. Therefore, a minor inconvenience is that each list has to be terminated by the constant `Empty` (such as `1&2&3&Empty`). This can be remedied by adding to the definition of lists (Figure 4.4) an implicit function that maps elements `x` of `XS` to the singleton list containing only `x`:

```
FUNC   Singleton : XS -> ListS[XS] (IMPLICIT)
AXIOM  Singleton x = x&Empty
```

Now `1&2&3` can be used as shorthand for `1&2&3&Empty`.

Here the implicit function `Singleton` is injective. Therefore, `XS` is a real subsort of `ListS[XS]` in the sense that `XS` is "included" in `ListS[XS]`. A generic implicit function for subsorts is defined in `SubsortM` (Figure 5.1). Using this module it is clearer to define the implicit function from elements to singleton lists by:

```
MODULE SubsortM [XS, YS]

VAR    x : XS
VAR    y : YS

SORT   XS
SORT   YS

IMPORT LogicM % Fig. 3.3 page 43

FUNC   Inject : XS -> YS (IMPLICIT)
REQ    x1 /= x2  ==>  Inject x1 /= Inject x2

END MODULE
```

Fig. 5.1: Specification of subsort relationship.

```
IMPORT SubsortM [XS, ListS[XS]]
AXIOM  Inject x = x&Empty
```

Note that `Inject` is not a parameter of module `SubsortM` because that would require the definition of an "inject" function for each import of `SubsortM`.

5.3 Example: Aliases

Sort names with nested indexes can become lengthy and barely descriptive. For example, if information about persons is stored as a list of pairs of type:

```
ListS[PairS[PersonS,PersonInfoS]]
```

it can be useful to rename this sort to `PersonTableS`, but AFSL does not have a renaming mechanism. An earlier version of AFSL had a sort definition construct that allowed `PersonTableS` to be declared as an alias of the nested name. Aliases where used frequently in the FAN case study.

If implicit functions are available no special language construct for sort definitions is needed. Instead, two sorts s_1 and s_2 can be semi-identified by declaring that they are each others subsorts. This is done in the generic module `AliasM` in Figure 5.2. Now `PersonTableS` can be defined to be an alias by:

```
SORT   PersonTableS
IMPORT AliasM [PersonTableS, ListS[PairS[PersonS,PersonInfoS]]]
```

Note that the axioms of `AliasM` are ambiguous since implicit functions can be added in numerous ways, for example:

```
MODULE AliasM [XS, YS]

VAR    x : XS
VAR    y : YS

SORT   XS
SORT   YS

IMPORT SubsortM [XS, YS] % Fig. 5.1 page 87
IMPORT SubsortM [YS, XS] % Fig. 5.1 page 87

AXIOM  Inject Inject y = y
AXIOM  Inject Inject x = x

END MODULE
```

Fig. 5.2: Specification of alias relationship.

```
AXIOM  Inject Inject Inject y = Inject y
AXIOM  Inject Inject y        = Inject Inject y
```

Resolving inheritance ambiguities like this is discussed in Section 5.7.

5.4 Example: Shapes

A typical object-oriented example model is the description of different types of geometrical shapes, which here are two-dimensional objects placed in an imaginary three-dimensional space (for example, a window in a graphical user-interface). A simplified model is given which serves to demonstrate the use of inheritance. Shapes (Figure 5.4) have a position and an area, no more features are assumed for shapes in general. The position of a shape is a three-dimensional coordinate (Figure 5.3). De z-coordinate of the position is the depth at which the shape is placed in space. The components of a coordinate and the area are assumed to be floating point numbers for simplicity, but should include a unit of measurement (such as “meters” and “square meters”) to be precise. Three special forms of shapes are specified: circles (Figure 5.5), with an additional radius feature; rectangles (Figure 5.6), with height and width features; and squares, which are rectangles that have a height equal to their width (Figure 5.7).

The sorts in this example are not defined inductively; instead they are specified by listing a number of key features which fully determine the observable properties of its elements. Here a *feature* of sort s is a function whose first domain type is s. The *key features* of s are the features of s that together determine the equality of s.

```
MODULE CoordinateM

IMPORT FloatM % Fig. 5.10 page 96

SORT   CoordS
FUNC   XCoord : CoordS -> FloatS
FUNC   YCoord : CoordS -> FloatS
FUNC   ZCoord : CoordS -> FloatS

%... further definition of functions for CoordS

END MODULE
```

Fig. 5.3: Specification of coordinates in three dimensional space.

```
MODULE ShapeM

VAR    sh : ShapeS

IMPORT LogicM      % Fig. 3.3 page 43
IMPORT CoordinateM % Fig. 5.3 page 89

SORT   ShapeS
FUNC   Position : ShapeS -> CoordS (IMPLICIT)
FUNC   Area     : ShapeS -> FloatS
AXIOM  Position sh1 = Position sh2  And
       Area sh1 = Area sh2
       ==>
       sh1 = sh2

FUNC   InFrontOf : ShapeS, ShapeS -> BoolS
AXIOM  sh1 InFrontOf sh2  <=>  ZCoord sh1 <= ZCoord sh2

FUNC   BiggerThan : ShapeS, ShapeS -> BoolS
AXIOM  sh1 BiggerThan sh2  <=>  Area sh1 >= Area sh2

%... Other operations for shapes

END MODULE
```

Fig. 5.4: Specification of shapes.

```
MODULE CircleM

VAR    circ : CircleS

IMPORT ShapeM % Fig. 5.4 page 89

SORT   CircleS
FUNC   Shape  : CircleS -> ShapeS (IMPLICIT)
FUNC   Radius : CircleS -> FloatS

AXIOM  Shape circ1 = Shape circ2  And
       Radius circ1 = Radius circ2
       ==>
       circ1 = circ2

AXIOM  Area circ = Pi * Radius circ * Radius circ

%... definition of other functions for circles

END MODULE
```

Fig. 5.5: Specification of circles.

For example, the key features of `ShapeS` are `Position` and `Area`. That is, shapes with the same area and position cannot be distinguished as elements of `ShapeS`. In fact, shapes can be viewed as records with fields `Position` and `Area`.

`ShapeS` is an abstract sort in the sense that it has derived features (`Area`) that cannot be defined for shapes in general, but that can be defined for some of its inheritors (`CircleS` and `RectangleS`). In this respect `Area` behaves like what is sometimes called a virtual method in object-oriented programming.

It can be argued that in a true object-oriented specification the equality axioms should be omitted since, for example, a rectangle is a shape and two rectangles are not equal even when they have the same position and area. It might even be true that any definition of equality for shapes is not needed and therefore over-specific. However, defining equality for any sort s has the advantage that it makes clear right away which properties of the elements of s are relevant for the specification, even if the equality itself is not used. For example, no matter how they are defined, `InFrontOf` and `Bigger` at most depend on the position and area of their arguments.

`Position` is an implicit function so that the individual coordinates of (the position of) a shape can be referred to directly using one of the coordinate selection functions, such as `ZCoord`. The resulting inheritance by `ShapeS` from `CoordS` is used in the axiom that defines the relation `InFrontOf` in terms of the z-coordinate of

```
MODULE RectangleM

VAR    rect : RectangleS

IMPORT ShapeM % Fig. 5.4 page 89

SORT   RectangleS
FUNC   Shape  : RectangleS -> ShapeS (IMPLICIT)
FUNC   Height : RectangleS -> FloatS
FUNC   Width  : RectangleS -> FloatS

AXIOM  Shape rect1  = Shape rect2   And
       Height rect1 = Height rect2  And
       Width rect1  = Width rect2
       ==>
       rect1 = rect2

AXIOM  Area rect = Height rect * Width rect

%... Operations for rectangles

END MODULE
```

Fig. 5.6: Specification of rectangles.

```
MODULE SquareM

VAR    sq   : SquareS

IMPORT RectangleM % Fig. 5.6 page 91

SORT   SquareS
FUNC   Rectangle : SquareS -> RectangleS (IMPLICIT)

AXIOM  Rectangle sq1 = Rectangle sq2
       ==>
       sq1 = sq2

AXIOM  Height sq = Width sq
LEMMA  Area sq = (Height sq) * (Height sq)

%... Operations for squares

END MODULE
```

Fig. 5.7: Specification of squares.

shapes. This axiom is shorthand for the expansion:

```
AXIOM sh1 InFrontOf sh2
      <=> ZCoord Position sh1 <= ZCoord Position sh2
```

`CircleS` is made an inheritor of `ShapeS` by the implicit function `Shape`. One key feature `Radius` is added to circles as an extension of shapes. Note that `Position` and `Area` are indirectly key features of `Circle` because `Shape` is a key feature. The dependency between the area and the radius of a circle is laid down in an axiom that uses inheritance by `CircleS` from `ShapeS`, it is shorthand for the expansion:

```
AXIOM Area Shape circ = Pi * Radius circ * Radius circ
```

In a similar way `RectangleS` is an inheritor of `ShapeS` and `SquareS` of `RectangleS`. For squares no additional key features are added to its rectangle features, only the possible values of `Height` and `Width` are restricted. Inheritance is used in the definition of `Area` for rectangles, in the axiom that restricts squares to rectangles with a height equal to their width, and in the lemma for squares.

The inheritance from coordinates to shapes through `Position` is conceptually different from the inheritance from shapes to circles through `Shape`. The first case corresponds to a so-called *has-a relationship* (a shape has a position) and the second to an *is-a relationship* (a circle is a shape). Technically, in AFSL there is no difference between these two kinds of inheritance. This is where the implicit function

approach diverges from inheritance in object-oriented methods which treat is-a and has-a differently (it is common that there is no inheritance for has-a relationships).

5.5 Example: Numbers

The next example demonstrates forms of inheritance that cannot be achieved in common object-oriented languages, but are quite common in programming languages. Three types of numbers and the implicit conversions among them are specified: naturals (that is, natural numbers) in Figure 5.8, integers in Figure 5.9, and floats (that is, floating point numbers) in Figure 5.10. The specification of integers given in Figure 5.9 is an alternative for the one in Figure 4.11 (which does not define integers in terms of naturals).

These examples instantiate the standard set of arithmetic operations that is discussed in Section 4.7. The binary function `.` (the dot) in `FloatM` can be used to construct the usual denotations for floats (for example, `2.20371`). Note that `.` is declared on numerals (see Section 3.7) rather than naturals because the leading zeros of the fractional part are significant (otherwise `1.05` would be equal to `1.5`).

Subsort relationships are defined to convert naturals to integers and integers to floats. The implicit function `Nat` convert numerals to naturals. This is not a subsorts relationship because `Nat` is not injective. The relationship can both be viewed as an is-a relationship (a numeral is a representation of a natural) or a has-a relationship (a numeral has a natural as value). Fortunately, the implicit function mechanism does not force one to choose.

This example is different from a typical object-oriented model like the shape example of the previous section. The three number sorts are defined inductively instead of listing key features. The inheritance relationships between the three sorts are, therefore, not declared by implicit key features. Object-oriented languages in general do not have inductive types. But, apart from that, in a object-oriented language c_1 can only be declared to be a subclass of c_2 as part of the declaration of c_1. In the example given here the situation is opposite: the "subclasses" are defined independently of their "superclasses". For example, `NatS` is made a subsort of `IntS` as part of the definition of `IntS`. This flexibility in defining inheritance relationships is possible because implicit functions are declared independently of sorts.

5.6 Example: Retract Functions

From an object-oriented point of view numeric operations such as `+` should be declared only once at the most general level (`FloatS`), and passed to more specific levels by inheritance. But, the addition of two numbers will then always be of type `FloatS`, even if the arguments are of type `IntS`. Terms like `5 Div (1+2)` will then

```
MODULE NatM

VAR    n   : NatS
VAR    num : NumeralS

IMPORT LogicM % Fig. 3.3 page 43

SORT   NatS (INDUCTIVE)
FUNC   ZeroNat :      -> NatS (CONSTRUCTOR)
FUNC   Succ    : NatS -> NatS (CONSTRUCTOR)

AXIOM  ZeroNat /= Succ n
AXIOM  Succ n1 /= Succ n2  <==  n1 /= n2

FUNC   Nat : NumeralS -> NatS (IMPLICIT)
AXIOM  Nat num = {the natural number represented by 'num'}

IMPORT PlusTimesM [NatS]

AXIOM  Zero = ZeroNat

AXIOM  Zero    + n  = n
AXIOM  Succ n1 + n2 = Succ (n1 + n2)

AXIOM  One = Succ Zero

AXIOM  Zero    * n  = Zero
AXIOM  Succ n1 * n2 = n1 * n2 + n2

%... more operations for naturals

END MODULE
```

Fig. 5.8: Specification of natural numbers.

```
MODULE Int2M

VAR    n : NatS
VAR    i : IntS

IMPORT NatM % Fig. 5.8 page 94

SORT   IntS (INDUCTIVE)
FUNC   Int : NatS -> IntS (CONSTRUCTOR)
FUNC   Min : NatS -> IntS (CONSTRUCTOR)

AXIOM  Int n1 /= Int n2  <==  n1 /= n2
AXIOM  Min n1 /= Min n2  <==  n1 /= n2
AXIOM  Int 0   = Min 0
AXIOM  Int n1 /= Min n2  <==  n1 /= 0  Or  n2 /= 0

IMPORT SubsortM [NatS, IntS] % Fig. 5.1 page 87

AXIOM  Inject n = Int n

IMPORT PlusMinusTimesM [IntS]

AXIOM  Zero:->IntS = Int Zero

AXIOM  Int n1      + Int n2      = Int (n1 + n2)
AXIOM  Min n1      + Min n2      = Min (n1 + n2)
AXIOM  Int Succ n1 + Min Succ n2 = Int n1 + Min n2
AXIOM  Min Succ n1 + Int Succ n2 = Min n1 + Int n2

AXIOM  -- Int n = Min n
AXIOM  -- Min n = Int n

AXIOM  Int n1 * Int n2 = Int (n1 * n2)
AXIOM  Min n1 * Min n2 = Int (n1 * n2)
AXIOM  Int n1 * Min n2 = Min (n1 * n2)
AXIOM  Min n1 * Int n2 = Min (n1 * n2)

%... more functions for integers such as Div and Mod

END MODULE
```

Fig. 5.9: Specification of integers. This specification is an alternative for the one given in Figure 4.11.

```
MODULE FloatM

VAR    i  : IntS
VAR    f  : FloatS
VAR    ds : NumeralS

IMPORT Int2M % Fig. 5.9 page 95

SORT   FloatS (INDUCTIVE)
FUNC   E : IntS, IntS -> FloatS (CONSTRUCTOR)

AXIOM  (i1 * 10) E i2 = i1 E (i2 + 1)
AXIOM  i1 /= i5 * 10  And  i3 /= i6 * 10  And
       i1 /= i3  And  i2 /= i4
       ==>
       i1 E i2 /= i3 E i4

IMPORT SubsortM [IntS, FloatS] % Fig. 5.1 page 87
AXIOM  Inject i = i E 0

FUNC   . : NumeralS, NumeralS -> FloatS
AXIOM  ds1.ds2
       = {the float with whole part 'ds1' and fractional part 'ds2'}

IMPORT PlusMinusTimesM [FloatS]

AXIOM  Zero:->FloatS = Zero E Zero

AXIOM  (i1 E i2) + (i3 E i2) = (i1 + i3) E i2

AXIOM  -- (i1 E i2) = (-- i1) E i2

AXIOM  (i1 E i2) * (i3 E i4) = (i1 * i3) E (i2 + i4)

FUNC   Pi : -> FloatS
AXIOM  Pi = 3.14

IMPORT TotalOrderedM [FloatS] % Fig. 4.7 page 75

%... definition of <= and other operations for FloatS

END MODULE
```

Fig. 5.10: Specification of floating point numbers.

```
MODULE Retract1M [XS, YS]

VAR    x : XS

SORT   XS
SORT   YS

IMPORT SubsortM [XS, YS] % Fig. 5.1 page 87

FUNC   Retract : YS -> XS (IMPLICIT)
AXIOM  Retract Inject x = x

END MODULE
```

Fig. 5.11: Specification of generic retract function.

be ill-typed (assuming the co-domain of `Div` is `IntS`). Typed object-oriented programming languages face the same problem; a solution some languages offer is the use of type-casting to make `1+2` officially a term of type `IntS` (for example, by writing `5 Div (1+2:IntS)`). It is possible to define an implicit type-cast operation, a so-called retract function (named after the similar concept of retracts in OBJ (Goguen et al. 1988)).

The *retract function* from a sort `XS` to a subsort `YS` is the inverse of the injection from `YS` to `XS`. The specification of retract functions is given in Figure 5.11. Strictly, `Retract` is a partial function because not all elements of `XS` have to be in the co-domain of `Inject`. Partiality is not discussed until Chapter 6. In Section 6.5 it is shown how the retract function can be specified as a partial function.

Using the retract function from `FloatS` to `IntS` (made available by replacing the import of `SubsortM` in module `FloatM` by `Retract1M`) is an alternative for having overloaded arithmetic operations for both floats and integers. For example, the retract function causes `5 Div (1+2)` to be well-typed because it then is an abbreviation of `5 Div (Retract (1+2))`. See Figure 5.11.

A disadvantage of defining the retract function from floats to integers is that all terms of type `FloatS` can then pass as integers, even if they do not represent an integer. The retract function causes `FloatS` to behave like an alias of `IntS`. Retract functions have to be used with care, otherwise the separation between sorts gets blurred and too many terms are well-typed.

5.7 Inheritance Ambiguity

The given examples contain a number of ambiguous assertions; that is, conversions can be inserted in more than one way. Declaration of implicit functions often results in ambiguous assertions, making it virtually impossible to write unambiguous specifications. However, these ambiguities are often harmless because either the assertion is semantically unambiguous (that is, all possible expansions have the same semantics) or there are good reasons to prefer one particular expansion over all others. In the current section a number of examples of inheritance ambiguities are given. In the next section a preference mechanism is presented that reduces inheritance ambiguity to an acceptable level.

The effect of inheritance on ambiguity is similar to that of overloading. For example, consider an implicit function f and a non-implicit function g such that:

$$f : s_1 \to s_2$$
$$g : s_2, s_3 \to s_4$$

then effectively there is a second "overloaded" version of g of type $s_1, s_3 \to s_4$. If there is also an overloaded version of g of type $s_1, s_3 \to s_4$, then effectively there are three overloaded versions of g of which two have the same type. There are two types of ambiguities that can be caused by inheritance: overloading conflicts and repeated inheritance.

5.7.1 Overloading Conflicts

Overloading conflicts occur when in a function application $f(\ldots)$ the overloaded abbreviation f can be completed in more than one way. For example, using the definitions of the number example, the term `0.5+(1+2)=2.5` is ambiguous since the numerals `1` and `2` can be summed using the overloaded `+` either for naturals, integers, or floats:

```
0.5 + (Float Int ((Nat 1) + (Nat 2))) = 3.5
0.5 + (Float ((Int Nat 1) + (Int Nat 2))) = 3.5
0.5 + ((Float Int Nat 1) + (Float Int Nat 2))) = 3.5
```

All of these expansions have the same semantics and, therefore, it is unsatisfactory to consider `0.5+(1+2)=3.5` ill-formed because of its syntactic ambiguity.

In general it is not true that different expansions have the same semantics. For example, using the definitions of the shape example, the equation `rect1=rect2` (where `rect1` and `rect2` are rectangle variables) has well-typed expansions (the first one without any conversions):

```
rect1 = rect2
(Shape rect1) = (Shape rect2)
(Position Shape rect1) = (Position Shape rect2)
```

These three expansions do not need to have the same semantics. Assume, for example, that `rect1` and `rect2` denote different rectangles that have the same area and position. Then the first expansion is false, but the other two are true. Two unequal rectangles can be equal as shapes because position and area are the key features of shapes, not of rectangles. Since a rectangle is a shape, this could be considered bad modeling practice and a reason to remove the equality axioms for `ShapeS`; adding an axiom for the injectivity of the implicit function `Shape` instead. The last expansion then still would have a different value than the first two because `Position` is not injective. This could be a reason to reject the use of inheritance for has-a relationships. Even if the ambiguities are not eliminated, the first expansion is the most obvious reading of `rect1=rect2` because it does not use any conversions. So, even if a term is semantically ambiguous there can be good reasons to prefer one particular expansion over the others.

An ambiguous term can be both semantically unambiguous and have a most obvious reading at the same time. For example, possible expansions of `1+2=3` are:

```
(Nat 1) + (Nat 2) = (Nat 3)
(Int Nat 1) + (Int Nat 2) = (Int Nat 3)
(Int ((Nat 1) + (Nat 2))) = (Int Nat 3)
(Float Int Nat 1) + (Float Int Nat 2) = (Float Nat 3)
(Float ((Int Nat 1) + (Int Nat 2))) = (Float Nat 3)
(Float Int ((Nat 1) + (Nat 2))) = (Float Nat 3)
```

All these expansions have the same semantics, but it is also pointless to insert conversions from naturals to integers or floats since the equation can already be evaluated at the level of naturals. Also, for `0.5+(1+2)`=3.5 it can be argued that the first expansion is the most obvious reading since there is no point in converting `1` and `2` to integers or floats before adding them.

5.7.2 Repeated Inheritance

The second kind of inheritance related ambiguity is caused by *repeated inheritance* by a sort s_1 from s_2. That is, if there are different sequences of implicit functions from s_1 to s_2.

The simplest case is *direct repeated inheritance* where there are implicit functions f_1 and f_2 both of type $s_1 \to s_2$. Then, for function $g : s_2 \to s_3$ and term $t : s_1$, the term $g(t)$ can be expanded to $g(f_1(t))$ and $g(f_2(t))$. In this situation there is no reason to favor one expansion over the other. But, there seems to be no reason to have multiple implicit functions from s_1 to s_2 in the first place, one is enough (that is why

no relevant example of this form of repeated inheritance is given here). So, in case of ambiguity caused by direct repeated inheritance it is acceptable to reject an assertion as being ill-formed.

A more complicated situation arises with *indirect repeated inheritance*, where $s_1 <^* x <^* s_2$ and $s_1 <^* y <^* s_2$ for distinct intermediate sorts x and y. Such a form of repeated inheritance, where the properties of x and y are combined in s_1, is similar to multiple inheritance in object-oriented languages. It can sometimes be useful; Meyer (1988) discusses repeated inheritance and gives the example of transcontinental drivers, which are drivers that drive cars on two different continents. An adaptation of this example is given in Figure 5.12 (here "France" and "US" refer to the places where cars are driven, not citizenship). This example is a bit farfetched, but it does illustrate the problem. Meyer notes that repeated inheritance does occur in practice, but not frequently. In `DriverM` the axiom

```
AXIOM  NrOfViolations fud
       = FranceViolations fud + USViolations fud
```

would be ambiguous without the definition of the additional operation `Driver`, because then `NrOfViolations fud` could be expanded in two ways:

```
NrOfViolations Driver France fud
NrOfViolations Driver US fud
```

The symmetry of the example prohibits that one expansion can be favored over the other. This is remedied by the additional conversion function `Driver` from `FranceUSDriverS` to `DriverS`. With this implicit function the expansion

```
NrOfViolations Driver fud
```

may be favored over the other two because it uses less implicit functions to convert an element of `FranceUSDriverS` to `DriverS`. Moreover, the axioms for `Driver fud` also guarantee semantic unambiguity. It is not an elegant solution, but it does do the job.

A special form of repeated inheritance is *repeated self inheritance*, that is, repeated inheritance between a sort s and s itself. By definition s inherits from itself through the empty chain of implicit functions. Thus, if there is at least one non-empty chain of implicit functions from s to s, then there is repeated self inheritance. This occurs, for example, in the definition of aliases (see Section 5.3) and retract functions (see Section 5.6).

Repeated self inheritance always causes ambiguous expansion if a term $t : s$ is used in an assertion since conversions from s to s can be applied to t an arbitrary number of times. However, such cyclic conversions are always redundant, therefore, expansions without them are the most obvious.

```
MODULE DriverM

VAR    fud : FranceUSDriverS

IMPORT Int2M % Fig. 5.9 page 95

SORT   DriverS
FUNC   Age            : DriverS -> IntS
FUNC   Address        : DriverS -> StringS
FUNC   NrOfViolations : DriverS -> IntS

SORT   FranceDriverS
FUNC   Driver           : FranceDriverS -> DriverS (IMPLICIT)
FUNC   FranceViolations : FranceDriverS -> IntS

SORT   USDriverS
FUNC   Driver       : USDriverS -> DriverS (IMPLICIT)
FUNC   USViolations : USDriverS -> IntS

SORT   FranceUSDriverS
FUNC   FranceDriver : FranceUSDriverS -> FranceDriverS (IMPLICIT)
FUNC   USDriver     : FranceUSDriverS -> USDriverS (IMPLICIT)
AXIOM  NrOfViolations fud = FranceViolations fud + USViolations fud

% Additional operation to prevent ambiguity
FUNC   Driver : FranceUSDriverS -> DriverS (IMPLICIT)
AXIOM  Driver fud = Driver FranceDriver fud
AXIOM  Driver fud = Driver USDriver fud

END MODULE
```

Fig. 5.12: Specification of transcontinental drivers.

5.8 Preferred Expansions

If an assertion is ambiguous solely because of overloading it is reasonable to reject it as being ill-formed. There is not much choice here: the different instances of overloaded functions are probably semantically unrelated. But even if they were related, it is hard for an automatic type-checker to verify that an ambiguous term is semantically unambiguous. On the syntactic side, there is no such thing as "most obvious reading" of a term with ambiguous overloading, because there is no measure to compare different expansions. Inheritance ambiguity is different in this respect. The previous section shows that sometimes the ambiguities are innocent, because either the possible expansions are semantically equivalent or ambiguity is caused by unnecessary conversions.

Therefore, a preference mechanism is introduced that, in case of innocent inheritance ambiguity, picks one of the expansions as *the* expansion. In case of a semantically unambiguous assertion an arbitrary expansion can be chosen. However, semantic unambiguity cannot be verified automatically by a type-checker (unless an automatic proof tool is used). In case of unnecessary conversions, it is feasible to choose the expansion that has the least number of conversions, because there is no point in adding redundant conversions. It will turn out that "least number of conversions" can be defined in such a way that semantically unambiguous assertions like the ones in the given examples, also have a preferred expansion. From an object-oriented point of view minimizing the number of conversions reflects the idea that in case of an overloaded method the most specific instance possible is used (that is, the instance belonging to the class closest to the class of the receiving object).

We can prefer expansions with a minimal number of conversions for each individual function argument. This works well in some cases: for example, the preferred expansions for `1+2=3` and `rect1=rect2` then are:

```
(Nat 1) + (Nat 2) = (Nat 3)
rect1 = rect2
```

However, none of the expansions of `0.5+(1+2)=3.5` has a minimum number of conversions for all individual arguments. The problem here is that conversion can be inserted at different levels, either for the arguments of `1+2` or for its result. Also minimizing the total number of conversions does not work. For example, expansions of `--2+2.5=0.5` with the same minimal overall number of conversions are:

```
(Float -- Int Nat 2) + 2.5 = 0.5
(-- Float Int Nat 2) + 2.5 = 0.5
```

Since treating all arguments within a term equal does not solve all harmless ambiguities, they will be handled in a particular order: expansion of arguments in a term is minimized in postorder (that is, relative to the parse tree of the term). Now the preferred expansion of `0.5+(1+2)=3.5` is:

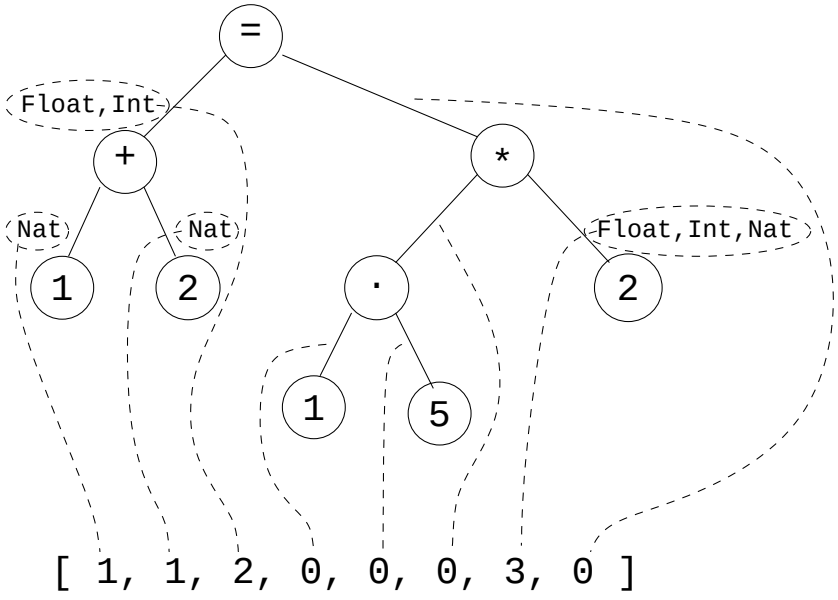

Fig. 5.13: The computation of the costs of the expansion of `1+2=1.5*2` to `Float Int (Nat 1 + Nat 2) = 1.5 * (Float Int Nat 2)`.

```
0.5 + (Float Int ((Nat 1) + (Nat 2)) = 3.5
```

With this form of preference the previous examples `1+2=3` and `rect1=rect2` still have the same preferred expansion.

This preference mechanism is defined by associating a cost with each expansion, $t \twoheadrightarrow t' : c$ denotes that t' is an expansion of t with costs c. Here c is a sequence of natural numbers, where each number represents the number of conversions added to one particular argument. That is, the cost of an expansion of $f(t_1, t_2, \ldots)$ first contains the cost of expanding t_1, then the number of conversions applied to t_1, then the cost for expanding t_2, the number of conversions applied to t_2, etcetera. An example of the computation of expansion costs is given in Figure 5.13. If a term has multiple expansions, the one with the minimum cost is preferred, where the costs are ordered lexically as lists. For example, the three expansions of `1+2=3` are (using abbreviated names):

$$\texttt{1+2=3} \twoheadrightarrow \texttt{(Nat 1) + (Nat 2) = (Nat 3)} : [1, 1, 0, 1]$$
$$\texttt{1+2=3} \twoheadrightarrow \texttt{(Int Nat 1) + (Int Nat 2) = (Int Nat 3)} : [2, 2, 0, 2]$$
$$\texttt{1+2=3} \twoheadrightarrow \texttt{(Int ((Nat 1) + (Nat 2))) = (Int Nat 3)} : [1, 1, 1, 2]$$

Here the preferred expansion is the first one with cost $[1, 1, 0, 1]$.

$$\frac{}{v \twoheadrightarrow v : [\,]}$$

$$\frac{\begin{array}{c} f \hookrightarrow f' \\ t_1 \twoheadrightarrow t_1' : c_1 \quad \cdots \quad t_n \twoheadrightarrow t_n' : c_n \\ t_1' \rightsquigarrow t_1'' : a_1 \quad \cdots \quad t_n' \rightsquigarrow t_n'' : a_n \end{array}}{f(t_1, \ldots, t_n) \twoheadrightarrow f'(t_1'', \ldots, t_n'') : c_1 +\!\!+ [a_1] +\!\!+ \ldots +\!\!+ c_n +\!\!+ [a_n]}$$

$$\frac{\begin{array}{c} s \hookrightarrow s' \\ t \twoheadrightarrow t' : c \end{array}}{(v : s \mid t) \twoheadrightarrow (v : s' \mid t') : c}$$

$$\frac{e_1 \rightarrowtail e_1' : c_1 \quad \cdots \quad e_n \rightarrowtail e_n' : c_n}{\{e_1 \ldots e_n\} \twoheadrightarrow \{e_1' \ldots e_n'\} : c_1 +\!\!+ \ldots +\!\!+ c_n}$$

$$\frac{}{nl \rightarrowtail nl : [\,]} \qquad \frac{nm \hookrightarrow nm'}{\text{'}nm\text{'} \rightarrowtail \text{'}nm'\text{'} : [\,]} \qquad \frac{t \twoheadrightarrow t' : c}{\text{'}t\text{'} \rightarrowtail \text{'}t'\text{'} : c}$$

$$\frac{}{t \rightsquigarrow t : 0} \qquad \frac{t \rightsquigarrow t' : a}{t \rightsquigarrow \pi(t') : a + 1}$$

Fig. 5.14: Definition of expansion rules, including implicit functions. For any implicit function π. (For any variable v; sorts s and s'; functions f and f'; terms $t_1, \ldots, t_n$, $t_1', \ldots, t_n'$, $t_1'', \ldots, t_n''$, t, and t'; integer lists $c_1, \ldots c_n$ and c; informals $e_1, \ldots e_n$ and $e_1', \ldots e_n'$; natural language nl; names nm and nm'; and integers $a_1, \ldots, a_n$, and a.)

The redefinition of the expansion relation $\twoheadrightarrow$ is given in Figure 5.14 (it replaces the definition of $\twoheadrightarrow$ in Figure 3.10). The auxiliary relation $\rightsquigarrow$ is used to add the actual conversions to function arguments: $t \rightsquigarrow t' : a$ denotes that t' is optained from t by applying a implicit functions to it. The operation $+\!\!+$ concatenates lists. Note that with respect to the calculation of the cost of an expansion, only function applications are relevant, variables, lambda-abstractions, and informal terms are ignored. The definition of $\twoheadrightarrow$ is extended further in Section 7.5 with so-called application redirection.

6. NON-FUNCTIONAL FEATURES

Abstract

Specification of software often involves operations with non-functional features, which are operations that are not functional. There are formal languages that do allow certain kinds of non-functional features, such as operations with side-effects and partial functions. The semantics of these languages is inherently more complex than for simple algebraic languages and they have a fixed set of non-functional features they can handle. This chapter shows, by example, how operations with non-functional features can be modeled by total functions. Three examples are given: partial functions, state-dependent operations, and state-changing operations. Unfortunately, the technique described has some disadvantages which will be discussed, together with some solutions, in Chapter 7.

6.1 Actions

An operation with non-functional features is an operation that cannot fully be described by a total function from its arguments to its results (see Section 2.1.1). Examples of non-functional features are: partiality, exceptions, nondeterminism, user interaction, state dependency, state changes, real-time behavior, efficiency, and concurrency. Of course, a non-functional feature must have a formal status before it can be modeled in any formal language. Therefore, we consider only those non-functional features that can be formalized in some form. This may exclude non-functional features that are not directly implementable, such as “maintainability”, “robustness”, “costs”, and “reliability”.

Many operations with non-functional features can be modeled by a total function by passing extra arguments and returning enhanced results which contain, besides the actual result, additional information about the non-functional behavior of the operation. For example, a partial function can be modeled by a total function that returns a “partial value” (which is either an exception or a normal value) and a state-dependent function can be modeled by a total function that takes the state as an additional argument.

In fact, the additional arguments can also be incorporated in an enhanced result by making that result a function, similar to Currying in functional programming. For example, a state-dependent function can be modeled by a total function that returns a state-dependent value (that is, a mapping from states to values).

So, operations with non-functional features can be modeled by total functions by considering these features a property of the result rather than a property of the operation itself. In this thesis results that model values with non-functional features are called actions, which suggests that something has to be done (the action has to be *performed*) before an actual result is returned. For example, performing a partial value either fails or returns a value and performing a state-dependent value involves querying the state. Note that here "action" and "performing" are itself no formal notions. Each type of non-functional feature will have its own formal definition of "action" accompanied with a formal definition of "performing an action", which is actually incorporated in the definition of "lifting", which will be discussed in Chapter 7.

6.2 Specification of Actions

Some generic properties of actions are specified in Figure 6.1 where `ActionXS` is a sort of `XS`-actions with *value type* `XS`. Not much is specified here, only that every value is also an action (that is, `XS` is a subsort of `ActionXS`). The intention of the subsort relationship is that the injection of an element `x` of `XS` into `ActionXS` is a *trivial action* that does nothing except returning the value `x`.

Some auxiliary functions are defined in `ActionM` that will be used to construct actions. `If` can be used to make conditional actions, that is, conditional in the sense that performing `If(...)` performs either the second or third argument, not both. The functions `<<` and `>>` will be used to combine the non-functional features of two actions, returning the result of the first and second argument, respectively. `UnitS` is specified in Figure 6.2; it contains exactly one element `Unit` which is used as the result type of actions that are only relevant for their non-functional features. The role of `UnitS` is similar to that of `void` in C. The definitions of `<<` and `>>` are somewhat peculiar since they do not operate on actions at all. It even seems pointless to use them: all they do is ignore one of their arguments (why write `x << u` if it is `x` you need?). However, as will be shown at the end of Section 6.7, the so-called lifted versions of `<<` and `>>` do combine the non-functional features of both arguments.

6.3 Example: Partial Functions

A *partial function* is an operation that does not have a result for all of its possible arguments. Examples of partial functions are the division function for numbers (division by zero is not defined) and the operation that returns the head (that is, first

```
MODULE ActionM [XS, ActionXS]

VAR    x  : XS
VAR    ax : ActionXS
VAR    u  : UnitS

SORT   XS
SORT   ActionXS

IMPORT LogicM                  % Fig. 3.3 page 43
IMPORT UnitM                   % Fig. 6.2 page 109
IMPORT SubsortM [XS, ActionXS] % Fig. 5.1 page 87

FUNC   If : BoolS, ActionXS, ActionXS -> ActionXS
AXIOM  If (True,  ax1, ax2) = ax1
AXIOM  If (False, ax1, ax2) = ax2

FUNC   << : XS, UnitS -> XS
AXIOM  x << u = x

FUNC   >> : UnitS, XS -> XS
AXIOM  u >> x = x

%... other functions for actions

END MODULE
```

Fig. 6.1: Generic specification of actions.

```
MODULE UnitM

SORT   UnitS (INDUCTIVE)
FUNC   Unit : -> UnitS (CONSTRUCTOR)

END MODULE
```

Fig. 6.2: Specification of the unit sort.

```
MODULE PartialM [XS]

VAR    x : XS

SORT   XS

SORT   PartialS[XS]                (INDUCTIVE)
FUNC   Def   : XS -> PartialS[XS]  (CONSTRUCTOR)
FUNC   Undef :    -> PartialS[XS]  (CONSTRUCTOR)

AXIOM  Def x  /= Undef
AXIOM  Def x1 /= Def x2  <==  x1 /= x2

IMPORT ActionM [XS, PartialS[XS]] % Fig. 6.1 page 109

AXIOM  Inject x = Def x

END MODULE
```

Fig. 6.3: Specification of partial values.

element) of a list (an empty list does not have a head).

Partial functions can often be avoided by returning arbitrary results for "illegal" arguments. For example, the result of division by zero can be zero, or even unspecified. For a generic head operation applied to the empty list it is impossible to choose an arbitrary value because the type of the list elements is unknown in a generic list specification. The only option then is that the head of an empty list is an arbitrary object. But, this assumes that the element type is non-empty (otherwise there is no arbitrary object in it), which it not always is. If at all possible, choosing arbitrary results covers up the fact that some arguments should be considered erroneous, which may obscure a specification. However, the usefulness of partial functions is not the issue here, they merely serve to illustrate non-functional features.

A partial function can be made total by adding exception values to its co-domain, which are used as return values for those arguments for which the partial function is undefined. In Figure 6.3 the sort `PartialS[XS]` of *partial* `XS`*-values* is specified, which is a copy of `XS` (where the elements of `XS` are injected in `PartialS[XS]` via `Def`) plus one exception value `Undef`. A partial function with co-domain `XS` can now be modeled as a total function with co-domain `PartialS[XS]`.

There is no particular reason why `PartialS[XS]` contains only one exception value, except that this simplifies the example: the number of exceptions can be extended in a straightforward manner, for example, by adding to `Undef` a string argument for passing an error message.

```
MODULE List2M [XS]

VAR    x : XS
VAR    l : ListS[XS]

SORT   XS

%... definition of ListS[XS], Empty, &, and Length
%... (same as in Fig. 4.4 page 71)

IMPORT PartialM [XS]
IMPORT PartialM [ListS[XS]]

FUNC   Head : ListS[XS] -> PartialS[XS]
AXIOM  Head Empty  = Undef
AXIOM  Head (x&l) = x

FUNC   Tail : ListS[XS] -> PartialS[ListS[XS]]
AXIOM  Tail Empty = Undef
AXIOM  Tail (x&l) = l

LEMMA  Tail l1 = Def l2  ==>  Length l1 = Length l2 + 1
LEMMA  Head l1 = Def x  And  Tail l1 = Def l2  ==>  l1 = x&l2

END MODULE
```

Fig. 6.4: Extension of the list specification of Figure 4.4 with the partial head and tail functions.

In Figure 6.4 the partial `Head` and `Tail` functions for lists are defined in this way. Unfortunately the result of `Head` or `Tail` cannot be used straightforwardly. Actions have to be "unpacked" before they can be used as an argument for functions that do not know how to handle the non-functional information. For example, in the lemmas the values of `Tail l1` and `Head l1` are unpacked before they can be passed to `&` or `Length`. `Def` is used here instead of the implicit `Inject` to show clearly that `Tail l1` is not equal to `l2` but to some action "containing" `l2`.

6.4 Function Lifting

One might prefer a formulation of the lemmas in Figure 6.4 in which the partial values are not unpacked, like in:

```
LEMMA  l /= Empty  ==>  Length l = Length Tail l + 1
LEMMA  l /= Empty  ==>  l = (Head l)&(Tail l)
```

But, this lemma is ill-formed since `Length` and `&` do not accept partial values. The condition that `l` is non-empty guarantees that the result of `Tail l` is defined, but they are of the form `Def l2` and as such cannot be handled by `Length` and `&`. This is unfortunate, since the alternative formulation of the lemmas is simpler and more to the point than the one used in in Figure 6.4. As a matter of fact, the alternative formulation is common in everyday mathematical notation.

The functions `&` and `Length` can be used directly on partial values only if they are defined as lifted versions of the original `&` and `Length`. A *lifted* version of a function f is a function that accepts actions as arguments by first performing these actions and then passing the results to f. Just as "performing" is not a formal notion, "lifting" is not formal either (for the moment; lifting is treated formally in the next chapter). Lifted (and overloaded) versions of `&` and `Length` are defined in Figure 6.5. Now the simpler, previously ill-typed, formulation of the lemmas is well-formed. Note that it is necessary to also define a lifted version of `+` that accepts the partial result of the lifted version of `Length`. It is necessary in `List3M` to explicitly use `Def` (instead of the implicit `Inject`) in some of the axioms that define the lifted functions, since otherwise the expansion of these axioms would contain the non-lifted functions on the left side of the equations rather than the lifted ones.

The same approach of modeling partial functions as total functions with partial results can be applied in any other situation. Then partial results have to be unpacked too before they can be passed to ordinary functions. For example, a division function for reals would return a partial real, which makes it impossible to write:

```
LEMMA i2 /= 0  ==>  (i1/i2) * i2 = i1
```

unless lifted versions of `/` and `*` are defined. The function `*` needs to be lifted because the lifted version of `/` returns a partial real. Alternatively this property can be formulated as:

```
MODULE List3M [XS]

VAR    x  : XS
VAR    l  : ListS[XS]
VAR    n  : NatS
VAR    px : PartialS[XS]
VAR    pl : PartialS[ListS[XS]]

SORT   XS

%... definition of ListS[XS] and its operations
%... (same as in Fig. 6.4 page 111)

IMPORT PartialM [NatS]

FUNC   Length : PartialS[ListS[XS]] -> PartialS[NatS]
AXIOM  Length Undef = Undef
AXIOM  Length Def l = Length l

FUNC   + : PartialS[NatS], NatS -> PartialS[NatS]
AXIOM  (Def n1) + n2 = Def (n1 + n2)
AXIOM  Undef    + n  = Undef

FUNC   & : PartialS[XS], PartialS[ListS[XS]] -> PartialS[ListS[XS]]
AXIOM  (Def x) & (Def l) = x&l
AXIOM  Undef   & pl      = Undef
AXIOM  px      & Undef   = Undef

LEMMA  l /= Empty  ==>  Length l = Length Tail l + 1
LEMMA  l /= Empty  ==>  l = (Head l)&(Tail l)

END MODULE
```

Fig. 6.5: Extension of the list specification of Figure 6.4 with lifted `Length`, `+`, `&` for partial values.

```
MODULE Retract2M [XS, YS]

VAR    x : XS
VAR    y : YS

SORT   XS
SORT   YS

IMPORT SubsortM [XS, YS] % Fig. 5.1 page 87
IMPORT PartialM [XS]     % Fig. 6.3 page 110

FUNC   Retract : YS -> PartialS[XS] (IMPLICIT)
AXIOM  Retract y = Def x  <=>  y = Inject x

END MODULE
```

Fig. 6.6: Specification of partial retract function.

```
LEMMA i1/i2 = Def i3  ==>  i3 * i2 = i1
```

6.5 Example: Partial Retract Functions

Another example of the use of partial functions is the proper definition of the generic retract function (see Section 5.6) in Figure 6.6. Here the partial `Retract` is implicit, which may lead to implicit conversion of a defined value to `Undef`. For example, `1.5` may be converted to `Undef` if an integer is needed. Unfortunately, partial type cast is not of much use: for example, it does not make `5 Div (0.5+2.5)` a well-formed term since `Div` cannot handle partial values, unless a lifted version of `Div` is defined.

6.6 Example: State Dependent Operations

A *state-dependent operation* is an operation whose result depends on some hidden parameter called the *state*. A state can, for example, be the current content of a computer memory, the current time, or the current phase of some research project. Corresponding state-dependent operations are the dereference of a program variable, the age of a person, or the next thing to do to complete this thesis. State dependent functions can be useful when axioms use many functions that share some common argument. Hiding such a common argument as part of the state can improve the readability of axioms. Often it is even natural to do so: one would rather say "Bob's age is greater than Clair's age" (time is implicit here) than "Bob's age at time t is greater that Clair's age at time t" (time t is explicit here). On the other hand this will

```
<expression>  ::=  <integer>
                 | <simpl-variable>
                 | "read"
                 | "(" <expression> "+" <expression> ")"
                 | "(" <expression> "-" <expression> ")"
<statement>   ::=  "skip"
                 | <simpl-variable> ":=" <expression>
                 | "write" <expression>
                 | "condition" <expression> "then" <statement>
                 | "while" <expression> "do" <statement>
                 | "(" <statement> ";" <statement> ")"
```

Fig. 6.7: Syntax of the simple imperative programming language SImPL.

complicate the understanding of a specification, functional purists will claim.

In imperative programming languages the state of the computer memory is an implicit parameter of all memory dependent operations. We show how the semantics of the expressions of a simple imperative programming language SImPL can be defined using state-dependent operations. First, the state-dependent operations are modeled by functions that take the state as an additional argument; then the action types of state-dependent values are introduced and used. Often state-dependent operations are used in combination with state-changing operations (that is, operations with side-effects). These are discussed in Section 6.7 where the semantics of the statements of the SImPL are defined; the current section only defines the evaluation of SImPL expressions.

6.6.1 SImPL

The syntax of SImPL is defined in Figure 6.7 (given some set of SImPL program variables). A SImPL program is a statement, which is evaluated in the context of an environment, which maps variables to integers, and input/output streams of integers. The expressions and statements have their usual semantics (details are given later), where `read` returns the first element of the input stream, `write` e adds the value of e to the output stream, and the guard of an `condition`- or `while`-statement succeeds if and only if its value is zero.

SImPL is similar to the programming language PICO, which is algebraically specified in Bergstra et al. (1989) (without using non-functional features). The main differences between the two languages are: SImPL is untyped (that is, the only type of data is integer), SImPL variables are not declared, and SImPL has operations for

```
MODULE SImPLSyntaxM

IMPORT Int2M % Fig. 5.9 page 95

SORT   ExprS (INDUCTIVE)
FUNC   Const : IntS         -> ExprS (CONSTRUCTOR)
FUNC   Var   : StringS      -> ExprS (CONSTRUCTOR)
FUNC   Read  :              -> ExprS (CONSTRUCTOR)
FUNC   +     : ExprS, ExprS -> ExprS (CONSTRUCTOR)
FUNC   -     : ExprS, ExprS -> ExprS (CONSTRUCTOR)

SORT   StatementS (INDUCTIVE)
FUNC   Skip  :                          -> StatementS (CONSTRUCTOR)
FUNC   :=    : StringS, ExprS           -> StatementS (CONSTRUCTOR)
FUNC   Write : ExprS                    -> StatementS (CONSTRUCTOR)
FUNC   Cond  : ExprS, StatementS        -> StatementS (CONSTRUCTOR)
FUNC   While : ExprS, StatementS        -> StatementS (CONSTRUCTOR)
FUNC   ;     : StatementS, StatementS -> StatementS (CONSTRUCTOR)

END MODULE
```

Fig. 6.8: The abstract syntax of SImPL.

reading/writing values from/to an input/output stream.

In Figure 6.8 the sorts `ExprS` and `StatementS` of abstract syntax trees of expressions and statements are defined. Each constructor represents one of the production rules of the concrete syntax; where `Const` i is the integer constant i, `Var` s is a variable with identifier s, and `Cond(`e`,`s`)` is the condition-statement with condition e.

6.6.2 Semantics of SImPL Expressions

The value of an expression depends on the *environment* that contains the value of the program variables. Environments are represented by so-called *tables*, a generic specification of which is given in Figure 6.9. The value of the lookup in an empty table is left unspecified here in order to make `Lookup` a total function. `Lookup` can be changed into a partial function by using `PartialS[YS]` as co-domain (see Section 6.3). However, that would involve combining partiality and state-dependency, which obscures the current example. `TableS[XS,YS]` could be modeled as an alias of `FuncS[XS,YS]`. But, we try to use function sorts as few as possible since they are not common in algebraic specification.

In Figure 6.10 the evaluation function `Eval` is defined that maps an expressions

```
MODULE TableM [XS, YS]

VAR    x   : XS
VAR    y   : YS
VAR    tbl : TableS[XS,YS]

SORT   XS
SORT   YS

IMPORT LogicM % Fig. 3.3 page 43

SORT   TableS[XS,YS] (INDUCTIVE)
FUNC   Empty  :                        -> TableS[XS,YS] (CONSTRUCTOR)
FUNC   Assign : TableS[XS,YS], XS, YS -> TableS[XS,YS] (CONSTRUCTOR)

FUNC   Lookup : TableS[XS,YS], XS -> YS
AXIOM  Lookup (Assign(tbl,x,y) , x)  = y
AXIOM  Lookup (Assign(tbl,x2,y), x1) = Lookup (tbl, x1)  <==  x1 /= x2

END MODULE
```

Fig. 6.9: Specification of tables.

`e` and an environment `env` to the value of `e` for `env`. The sort `EnvS` of environments is defined as alias for `TableS[StringS,IntS]` in order to simplify notation. The evaluation of `Read` is left unspecified here because we ignore side-effects for the moment (the read operation changes the input stream as a side-effect).

The given definition is the common way to specify semantics of imperative languages in an algebraic specification language, such as the definition of PICO in Bergstra et al. (1989). What is typical for these kind of definitions is that the evaluation of a compound expression (such as `e1+e2`) for a given environment is defined in terms of the evaluation of its components (`e1` and `e2`) for that same environment. This makes the environment a good candidate for being hidden as part of a state.

6.6.3 State-Dependent Values

State-dependent functions can be modeled using the type of actions specified in Figure 6.11. `PeekS[XS]` is the sort of state-dependent `XS`-values; that is, an entity that has a value in `XS` for any value of `StateS`. State-dependent operations from `XS` to `YS` can now be modeled by an ordinary function from `XS` to `PeekS[YS]`. The terminology “peek” refers to the operation of the programming language BASIC that is used to read (peek) the content of a memory location. Associated with each state-

```
MODULE ExpressionSemantics1M

VAR    i   : IntS
VAR    v   : StringS
VAR    e   : ExprS
VAR    env : EnvS

IMPORT SImPLSyntaxM % Fig. 6.8 page 116

SORT   EnvS
IMPORT TableM [StringS, IntS]              % Fig. 6.9 page 117
IMPORT AliasM [EnvS, TableS[StringS,IntS]] % Fig. 5.2 page 88

FUNC   Eval : ExprS, EnvS -> IntS
AXIOM  Eval (Const i, env) = i
AXIOM  Eval (Var v  , env) = Lookup (env, v)
AXIOM  Eval (e1+e2  , env) = Eval (e1, env) + Eval (e2, env)
AXIOM  Eval (e1-e2  , env) = Eval (e1, env) - Eval (e2, env)

END MODULE
```

Fig. 6.10: Semantics of SImPL expressions without side-effects. The evaluation of `Read` is left unspecified.

```
MODULE PeekM [XS]

VAR    x  : XS
VAR    st : StateS

SORT   XS

IMPORT StateM % Fig. 6.12 page 119

SORT   PeekS[XS]
FUNC   Comp : PeekS[XS], StateS -> XS

IMPORT ActionM [XS, PeekS[XS]] % Fig. 6.1 page 109

AXIOM  Comp (Inject x, st) = x

END MODULE
```

Fig. 6.11: Specification of state-dependent values.

```
MODULE StateM

SORT   StateS

END MODULE
```

Fig. 6.12: Definition of states.

dependent value is a mapping from states to the value for each state. This is modeled by the function `Comp` (for "compute") that maps a state-dependent value `p` and a state `st` to the value of `p` for `st`. The trivial state-dependent value for `x` always has value `x`, regardless of the state it is executed in. Other instances of `PeekS[XS]` have to be defined by specifying the value of `Comp` for any state.

The sort `StateS` of states is declared in Figure 6.12. For the moment `StateS` is left unspecified in module `StateM` so that it can be tailored to the examples in the current and the next section. In Section 6.8 `StateS` will get its actual definition as an environment of typed state variables, similar to the state of a computer program. Alternatively, `StateS` could be made a parameter of the module `PeekM` and an index of `PeekS` (that is, `PeekS[StateS,XS]` instead of `PeekS[XS]`). For each use of state-dependent values an appropriate instance of `StateS` can then be used. However, using different notions of "state" obscures specifications.

6.6.4 States for SImPL

A suitable definition of states for the evaluation of SImPL expressions is given in Figure 6.13, where `StateS` is simply defined to be an alias for environments (that is, states are the same as environments). This definition of `StateS` is rather trivial, but it suits our purposes here. Module `SImPLState1M` contains a first example of a state-dependent operation: the function `Value` returns the value of a SImPL variable. `Value` can be viewed as a kind of, infinite, array with strings as indexes.

6.6.5 State-Dependent Semantics of Expressions

In Figure 6.14 the evaluation function `Eval` is redefined as a state-dependent function. Although the table is no longer a parameter of `Eval`, this does not bring much relief here because the use of `Comp` on the result of `Eval` and `st` makes the environment (disguised as state) a parameter of `Eval` via a detour (and even introduces extra overhead). The idea was to hide the state from the definition of `Eval`, but it is still there.

```
MODULE SImPLState1M

VAR    v   : StringS
VAR    i   : IntS
VAR    env : EnvS

IMPORT Int2M  % Fig. 5.9 page 95
IMPORT StateM % Fig. 6.12 page 119

SORT   EnvS
IMPORT TableM [StringS, IntS]              % Fig. 6.9 page 117
IMPORT AliasM [EnvS, TableS[StringS,IntS]] % Fig. 5.2 page 88

IMPORT AliasM [StateS, EnvS] % Fig. 5.2 page 88

IMPORT PeekM [IntS] % Fig. 6.11 page 118

FUNC   Value : StringS -> PeekS[IntS]
AXIOM  Comp (Value v, env) = Lookup (env,v)

END MODULE
```

Fig. 6.13: Definition of the state used for the semantics of SImPL expressions.

```
MODULE ExpressionSemantics2M

VAR    i  : IntS
VAR    v  : StringS
VAR    e  : ExprS
VAR    st : StateS

IMPORT SImPLSyntaxM % Fig. 6.8 page 116
IMPORT SImPLState1M % Fig. 6.13 page 120
IMPORT PeekM[IntS]  % Fig. 6.11 page 118

FUNC   Eval : ExprS -> PeekS[IntS]
AXIOM  Comp (Eval Const i, st)
       = i
AXIOM  Comp (Eval Var v  , st)
       = Comp (Value v, st)
AXIOM  Comp (Eval (e1+e2), st)
       = Comp (Eval e1, st) + Comp (Eval e2, st)
AXIOM  Comp (Eval (e1-e2), st)
       = Comp (Eval e1, st) - Comp (Eval e2, st)

END MODULE
```

Fig. 6.14: Definition of the semantics of SImPL expressions using state-dependent values in order to “hide” the environment.

```
MODULE ExpressionSemantics3M

VAR    i    : IntS
VAR    v    : StringS
VAR    e    : ExprS
VAR    st   : StateS
VAR    pi   : PeekS[IntS]

IMPORT SImPLSyntaxM % Fig. 6.8 page 116
IMPORT SImPLState1M % Fig. 6.13 page 120
IMPORT PeekM[IntS]  % Fig. 6.11 page 118

FUNC   + : PeekS[IntS], PeekS[IntS] -> PeekS[IntS]
AXIOM  Comp (pi1+pi2, st) = Comp(pi1,st) + Comp(pi2,st)

FUNC   - : PeekS[IntS], PeekS[IntS] -> PeekS[IntS]
AXIOM  Comp (pi1-pi2, st) = Comp(pi1,st) - Comp(pi2,st)

FUNC   Eval : ExprS -> PeekS[IntS]
AXIOM  Eval Const i = i
AXIOM  Eval Var v   = Value v
AXIOM  Eval (e1+e2) = (Eval e1) + (Eval e2)
AXIOM  Eval (e1-e2) = (Eval e1) - (Eval e2)

END MODULE
```

Fig. 6.15: Reformulation of the definition of the semantics of SImPL expressions of Figure 6.14, now using lifted versions of `+` and `-`. This allows the environment to be removed from the definition of `Eval`.

6.6.6 Using Lifted Functions

If lifted functions are used, it is possible to hide the state completely from the definition of `Eval` as is shown in Figure 6.15. In the first axiom the state is hidden using the implicit function `Inject`. In the second axiom the state can be omitted without further ado. In the axioms for `+` and `-` the state is only passed, without accessing it, to the evaluation of the sub-expressions. This can be left implicit by defining lifted versions of `+` and `-`. The resulting definition of `Eval` is much more to the point and, as a matter of fact, similar to the way a SImPL evaluator would be programmed in an imperative programming language.

In effect the definition of `Eval` in Figure 6.15 determines exactly the same equations as in Figure 6.14, for example:

```
Comp (Eval (e1+e2), st)
=     % definition of Eval for +
Comp ((Eval e1) + (Eval e2), st)
=     % definition of lifted +
Comp (Eval e1, st) + Comp (Eval e2, st)
```

6.7 Example: State-Changing Operations

The definition of the semantics of SImPL in the previous section does not yet handle the side-effects caused by reading input (removing the first element from the input stream), writing output (adding an element to the output stream), and variable assignment (changing the environment). This section shows how the definition of the evaluation function `Eval` can be adapted to incorporate these side-effects. A traditional algebraic specification can be given were the evaluation function takes, besides the environment, additional arguments for the input and output streams and that returns the changed environment and streams. Although these definitions are straightforward, they show how the overhead of explicitly passing arguments can lead to unreadable specifications. Of course, it is a matter of taste and training what is considered unreadable and what not; but, at some point specifications like this become too hard to read and, therefore, difficult to understand, maintain, and reuse.

The use of non-functional features can bring relief here, as is shown in an alternative definition of a state-changing version of the evaluation function. A *state-changing operation* is an operation whose result possibly depends on a hidden state (like a state-dependent operation) and which also defines a state transition. Examples of state-changing operations are the read and write operations of SImPL. As will be shown, state-changing operations too can be modeled by total functions that return an element of a suitable action type, in this case a state-changing value.

6.7.1 Semantics of SImPL Statements

First a traditional algebraic definition of the semantics of SImPL statements is given, similar to that of PICO in Bergstra et al. (1989).

Input and output of SImPL programs are modeled by streams of integers. *Streams* are infinite lists. As such they always have a head and a tail. The use of streams to model input avoids the complication of partial semantics caused by the possible read from an empty input stream (output could be modeled by finite lists without this complication). The data type of streams is specified in Figure 6.16. In order to have functions that return combined data structures, 2-tuples are specified in Figure 6.17. It is assumed a module `Tuple3M` of 3-tuples is defined similarly.

Figure 6.18 defines the semantics of expressions and statements in the traditional way. The evaluation of expression `e` takes as arguments the environment and the

```
MODULE StreamM [XS]

VAR    x      : XS
VAR    stream : StreamS[XS]

SORT   XS

SORT   StreamS[XS]
FUNC   Head : StreamS[XS] -> XS
FUNC   Tail : StreamS[XS] -> StreamS[XS]

FUNC   & : XS, StreamS[XS] -> StreamS[XS]
AXIOM  Head (x&stream) = x
AXIOM  Tail (x&stream) = stream

%... other functions

END MODULE
```

Fig. 6.16: Specification of infinite streams.

```
MODULE Tuple2M [XS, YS]

VAR    x : XS
VAR    y : YS

SORT   XS
SORT   YS

IMPORT LogicM

SORT   TupleS[XS,YS] (INDUCTIVE)
FUNC   ‘ : XS, YS -> TupleS[XS,YS] (CONSTRUCTOR)

AXIOM  ‘(x1, y1) /= ‘(x2, y2)  <==  x1 /= x2  Or  y1 /= y2

%... other functions for tuples

END MODULE
```

Fig. 6.17: Definition of pairs.

```
MODULE StatementSemantics1M

VAR i : IntS VAR v : StringS VAR e : ExprS VAR s : StatementS
VAR env : EnvS VAR in : StreamS[IntS] VAR out : StreamS[IntS]

IMPORT LogicM       % Fig. 3.3 page 43
IMPORT SImPLSyntaxM % Fig. 6.8 page 116

SORT   EnvS
IMPORT TableM [StringS, IntS]              % Fig. 6.9 page 117
IMPORT AliasM [EnvS, TableS[StringS,IntS]] % Fig. 5.2 page 88

IMPORT StreamM [IntS]                % Fig. 6.16 page 124
IMPORT Tuple2M [IntS, StreamS[IntS]] % Fig. 6.17 page 124
IMPORT Tuple3M [EnvS, StreamS[IntS], StreamS[IntS]]

FUNC   Eval : ExprS, EnvS, StreamS[IntS] -> TupleS[IntS,StreamS[IntS]]
AXIOM  Eval (Const i, env, in) = `(i, in)
AXIOM  Eval (Var v, env, in) = `(Lookup (env, v), in)
AXIOM  Eval (Read, env, in) = `(Head in, Tail in)
AXIOM  Eval (e1, env, in1) = `(i1, in2) And
       Eval (e2, env, in2) = `(i2, in3) ==>
       Eval (e1+e2, env, in1) = `(i1+i2, in3) And
       Eval (e1-e2, env, in1) = `(i1-i2, in3)

FUNC   Eval : StatementS, EnvS, StreamS[IntS], StreamS[IntS]
              -> TupleS[EnvS,StreamS[IntS],StreamS[IntS]]
AXIOM  Eval (Skip, env, in ,out) = `(env, in, out)
AXIOM  Eval (e, env , in1) = `(i, in2) ==>
       Eval (v:=e, env, in1, out) = `(Assign(env,v,i), in2, out) And
       Eval (Write e, env , in1, out ) = `(env, in2, i&out)
AXIOM  Eval (e, env, in1) = `(0, in2) ==>
       Eval (Cond(e,s), env, in1, out) = Eval (s, env, in2, out)
AXIOM  Eval (e, env, in1) = `(i, in2) And i /= 0 ==>
       Eval (Cond(e,s), env, in1, out) = `(env, in2, out)
AXIOM  Eval (While(e,s), env, in, out)
       = Eval (Cond(e,s;While(e,s)), env, in, out)
AXIOM  Eval (s1, env1, in1, out1) = `(env2, in2, out2) ==>
       Eval (s1;s2, env1, in1, out1)  = Eval (s2, env2, in2, out2)

END MODULE
```

Fig. 6.18: Definition of the semantics of SImPL expressions and statements with side-effects.

```
MODULE PokeM [XS]

VAR    st   : StateS
VAR    x    : XS
VAR    peek : PeekS[XS]

SORT   XS

IMPORT StateM               % Fig. 6.12 page 119
IMPORT Tuple2M [XS, StateS] % Fig. 6.17 page 124

SORT   PokeS[XS]
FUNC   Exec : PokeS[XS], StateS -> TupleS[XS,StateS]

IMPORT ActionM [XS, PokeS[XS]] % Fig. 6.1 page 109

AXIOM  Exec (Inject x, st) = `(x, st)

IMPORT PeekM [XS]                       % Fig. 6.11 page 118
IMPORT SubsortM [PeekS[XS], PokeS[XS]] % Fig. 5.1 page 87

AXIOM  Exec (Inject peek, st) =  `(Comp (peek, st), st)

END MODULE
```

Fig. 6.19: Definition of state changing values

input stream. It returns the value of `e` together with the input stream that remains after executing `e`. The evaluation of statement `s` takes as arguments the environment, the input stream (`s` may contain read operations), and the output stream (the output generated up till now; last output in front). It returns the value of `s` together with the remaining input stream and the extended output stream.

Different choices for the arguments and results of both `Eval`'s are possible. For example, the evaluation of statements does not need the output as argument and only has to return the list of elements added to the output. However, the choices made here better fit the state based definition of evaluation which is discussed next.

6.7.2 State-Changing Values

The sort `PokeS[XS]` of state-changing `XS`-values is defined in Figure 6.19. The definition of `PokeS` is similar to that of `PeekS` in Figure 6.11; the difference between `Comp` and `Exec` (for "execute") is that the latter also returns the changed state besides the actual value. The state returned by `Exec(p,st)` models the updated state after

performing the side-effect of `p` on `st`. For example, assume that a state models a file system by keeping the current contents of the files and the position of the file pointers, and `ReadStr` is the state-changing operation that reads a string from a text file; then `Exec(ReadStr f, st1)` should return:

- the string in file `f` at the position of the file pointer (in state `st1`), and
- the state `st2` which is derived from `st1` by moving the file pointer of `f` to the next string.

The trivial state-changing value `Inject x` returns `x` without changing the state.

A state-dependent value can be seen as a state-changing value that does not actually changes the state. Therefore in `PokeM PeekS` is defined to be a subsort of `PokeS`; state-dependent values can therefore be used as if they were state-changing values. The corresponding injection `Inject` transforms a state-dependent value in a state-changing value without side-effects. This is defined in the last axiom where the execution of (the injection of) state-dependent value `peek` in state `st` returns the value of `peek` (in state `st`) and the unchanged state `st`.

6.7.3 States for SImPL Statements

The definition of `StateS` that suits SImPL side-effects is given in Figure 6.20. Here states are triples containing an environment, input stream, and output stream. Same state-dependent operations are defined to access the different components of a state (`Value`, `Input`, and `Output`) and state-changing operations to modify the PartialS of a state (`SetValue`, `SetInput`, and `SetOutput`).

6.7.4 State-Changing Semantics of Statements

The definitions of the state-changing evaluation functions for SImPL are given in Figure 6.21. Note that the `Eval` for expressions has more implicit arguments than needed, it could do without the output part of the state; however, two different versions of `PokeS` would then be needed, each with its own definition of state.

As with the definition of the state-dependent `Eval` in Figure 6.14, the use of actions only hides the states from the types of the `Eval`'s, not from their definitions. Still, the new definition is a small improvement over the one in Figure 6.18 since the structure of states is hidden (that is, environment and streams do not occur explicitly).

6.7.5 Using Lifted Functions

As with partial and state-dependent values, specifications with state-changing values suffer from the overhead of unpacking the actions. This can be avoided by defining the appropriate lifted functions, as is shown in Figure 6.22. The resulting definitions

```
MODULE SImPLState2M

VAR v : StringS VAR i : IntS VAR env : EnvS
VAR in : StreamS[IntS] VAR out : StreamS[IntS]

IMPORT Int2M  % Fig. 5.9 page 95
IMPORT StateM % Fig. 6.12 page 119

SORT   EnvS
IMPORT TableM [StringS, IntS]              % Fig. 6.9 page 117
IMPORT AliasM [EnvS, TableS[StringS,IntS]] % Fig. 5.2 page 88

IMPORT StreamM [IntS]          % Fig. 6.16 page 124
IMPORT Tuple3M [EnvS, StreamS[IntS], StreamS[IntS]]
IMPORT AliasM [StateS, TupleS[EnvS,StreamS[IntS],StreamS[IntS]]]
IMPORT UnitM                   % Fig. 6.2 page 109
IMPORT PeekM [IntS]            % Fig. 6.11 page 118
IMPORT PeekM [StreamS[IntS]]   % Fig. 6.11 page 118
IMPORT PokeM [UnitS]           % Fig. 6.19 page 126

FUNC   Value : StringS -> PeekS[IntS]
AXIOM  Comp (Value v, `(env,in,out)) = Lookup (env, v)

FUNC   Input : -> PeekS[StreamS[IntS]]
AXIOM  Comp (Input, `(env,in,out)) = in

FUNC   Output : -> PeekS[StreamS[IntS]]
AXIOM  Comp (Output, `(env,in,out)) = out

FUNC   SetValue : StringS, IntS -> PokeS[UnitS]
AXIOM  Exec (SetValue(v,i), `(env,in,out))
       = `(Unit, `(Assign(env,v,i), in, out))

FUNC   SetInput : StreamS[IntS] -> PokeS[UnitS]
AXIOM  Exec (SetInput in1, `(env,in2,out)) = `(Unit, `(env,in1,out))

FUNC   SetOutput : StreamS[IntS] -> PokeS[UnitS]
AXIOM  Exec (SetOutput out1, `(env,in,out2)) = `(Unit, `(env,in,out1))

END MODULE
```

Fig. 6.20: Definition of the states used for the semantics of SImPL with side-effects. This is an adaption of the definition of StateS in Figure 6.13.

```
MODULE StatementSemantics2M

VAR i : IntS VAR v : StringS VAR e : ExprS VAR s : StatementS
VAR st : StateS VAR in : StreamS[IntS] VAR out : StreamS[IntS]

IMPORT SImPLSyntaxM  % Fig. 6.8 page 116
IMPORT SImPLState2M  % Fig. 6.20 page 128
IMPORT PokeM [IntS]  % Fig. 6.19 page 126
IMPORT PokeM [UnitS] % Fig. 6.19 page 126

FUNC   Eval : ExprS -> PokeS[IntS]
AXIOM  Exec (Eval Const i, st) = `(i, st)
AXIOM  Comp (Value v, st) = i  ==>
       Exec (Eval Var v, st) = `(i, st)
AXIOM  Comp (Input, st1) = in  And
       Exec (SetInput Tail in, st1) = `(Unit, st2)  ==>
       Exec (Eval Read, st1) = `(Head in, st2)
AXIOM  Exec (Eval e1, st1) = `(i1, st2)  And
       Exec (Eval e2, st2) = `(i2, st3)  ==>
       Exec (Eval (e1+e2), st1) = `(i1+i2, st3)  And
       Exec (Eval (e1-e2), st1) = `(i1-i2, st3)

FUNC   Eval : StatementS -> PokeS[UnitS]
AXIOM  Exec (Eval Skip, st) = `(Unit, st)
AXIOM  Exec (Eval e, st1) = `(i, st2)  ==>
       Exec (Eval (v:=e), st1) = Exec (SetValue (v,i), st2)
AXIOM  Exec (Eval e, st1) = `(i, st2)  And
       Comp (Output, st2) = out  ==>
       Exec (Eval Write e, st1) = Exec (SetOutput (i&out), st2)
AXIOM  Exec (Eval e, st1) = `(0, st2)  ==>
       Exec (Eval Cond (e,s), st1) = Exec (Eval s, st2)
AXIOM  Exec (Eval e, st1) = `(i, st2)  And  i /= 0  ==>
       Exec (Eval Cond (e,s), st1) = `(Unit, st2)
AXIOM  Exec (Eval While (e,s), st)
       = Exec (Eval Cond (e, s;While(e,s)), st)
AXIOM  Exec (Eval s1, st1) = `(Unit, st2)  ==>
       Exec (Eval (s1;s2), st1) = Exec (Eval s2, st2)

END MODULE
```

Fig. 6.21: Definition of the semantics of expressions and statements using state-changing values.

```
MODULE StatementSemantics3M

VAR i : IntS VAR b : BoolS VAR v : StringS VAR e : ExprS
VAR s : StatementS VAR st : StateS VAR pi : PokeS[IntS]
VAR pb : PokeS[BoolS] VAR pu : PokeS[UnitS]

%... imports (see Fig. 6.21 page 129)
IMPORT PokeM [BoolS]         % Fig. 6.19 page 126
IMPORT PokeM [StreamS[IntS]] % Fig. 6.19 page 126

FUNC   + : PokeS[IntS], PokeS[IntS] -> PokeS[IntS]
AXIOM  Exec (pi1, st1) = `(i1, st2)  And
       Exec (pi2, st2) = `(i2, st3)  ==>
       Exec (pi1+pi2, st1) = `(i1+i2, st3)
FUNC   = : PokeS[IntS], IntS -> PokeS[BoolS]
AXIOM  Exec (pi, st1) = `(i1, st2)  ==>
       Exec (pi=i2, st1) = `(i1=i2, st2)
FUNC   If : PokeS[BoolS], PokeS[UnitS], PokeS[UnitS] -> PokeS[UnitS]
AXIOM  Exec (pb, st1) = `(b, st2)  ==>
       Exec (If(pb,pu1,pu2), st1) = Exec (If(b,pu1,pu2), st2)
FUNC   << : PokeS[IntS], PokeS[UnitS] -> PokeS[IntS]
AXIOM  Exec (pi, st1) = `(i, st2)  And
       Exec (pu, st2) = `(u, st3)  ==>
       Exec (pi<<pu, st1) = `(i<<u, st3)
%... lifting - & Head Tail SetValue SetInput SetOutput >>

FUNC   Eval : ExprS -> PokeS[IntS]
AXIOM  Eval Const i  = i
AXIOM  Eval Var v    = Value v
AXIOM  Eval Read     = (Head Input) << (SetInput Tail Input)
AXIOM  Eval (e1+e2)  = (Eval e1) + (Eval e2)
AXIOM  Eval (e1-e2)  = (Eval e1) - (Eval e2)

FUNC   Eval : StatementS -> PokeS[UnitS]
AXIOM  Eval Skip        = Unit
AXIOM  Eval (v:=e)      = SetValue (v, Eval e)
AXIOM  Eval Write e     = SetOutput ((Eval e)&Output)
AXIOM  Eval Cond (e,s)  = If (Eval e = 0, Eval s, Unit)
AXIOM  Eval While (e,s) = Eval Cond (e, s;While(e,s))
AXIOM  Eval (s1;s2)     = (Eval s1) >> (Eval s2)

END MODULE
```

Fig. 6.22: Definition of the semantics of SImPL expressions and statements using lifted functions.

of the `Eval`'s are much more concise than the ones in Figure 6.18 and Figure 6.21. However, numerous lifted functions have to be defined to make this possible. As a matter of fact, the overhead of unpacking actions is only shifted to the definition of the lifted function. Chapter 7 shows how this can be avoided by making lifting an implicit operation. This will allow a definition of `Eval` which looks exactly the same as in Figure 6.22, but without the definition of the lifted functions (although, unfortunately, some new overhead will be introduced in the form of numerous imports of special "lifting" modules).

`StatementSemantics3M` shows the use of the lifted version of `>>` and `<<` which were defined as rather useless functions in module `ActionM` (Figure 6.1). The lifted `<<` first executes both its arguments (from left to right) and then returns the result of the first argument (and similarly `>>`). Thus, for state-changing values the lifted `<<` and `>>` operate like sequential composition in imperative programming (the difference between the lifted `<<` and `>>` is the result they return). As mentioned before, the non-lifted versions of these composition functions are rather useless. Their lifting nicely shows that lifting of functions to state-changing arguments involves sequential composition of the side-effects of the arguments.

There are functions that are only lifted in some of their arguments. For example, `=` is not lifted in its second argument, although it would not hurt either. This is different for the lifted `If`, which only can be lifted in its first argument (since the second and third argument are already state-changing values). It is essential for the laziness of `If` that it is not lifted in its second and third argument; otherwise execution of the lifted `If` would involve executing both these arguments.

6.8 Example: State Variables

The way states were defined in Figure 6.13 and later on redefined in Figure 6.20 is rather unpractical. For each state component separate access and modification operations have to be defined. Furthermore, it is impossible to extend these definitions in a later stage. That is why `StateS` had to be completely redefined in Figure 6.20 to incorporate in- and output streams. This makes it impossible to use `StateS` for different purposes within a single specification. In the current section a more general definition of `StateS` is given that has generic access and modification operations and that does allow extension.

The idea is to define a state as a mapping from a new data type of state variables to values, together with operations to read and set the value of variables. Such a state models the memory of an imperative programming language, where the read operation corresponds to dereference (which is often implicit in imperative programming) and the set operation to assignment.

6.8.1 Typing State Variables

There is a problem in formalizing such a generic definition of states: it should be possible to store elements of arbitrary type in a state (for example, integers and streams), combined with some form of static type checking on the read and set operations. That is, a distinction should be made between variables of different types, such that, for example, the value of an integer variable can only be used as an integer and only integers can be assigned to integer variables.

It may seem that this requires a type system with parametric polymorphism (see Section 4.3). However, such a type system is probably not capable of defining a state with typed variables. For example, in Haskell it is not possible to define:

```
type Var a = ...?
type State = Var a -> a
```

The problem here is that type parameter `a` cannot be used in the definition of `State` because it is not a parameter of `State`. Moreover, what should the definition of `Var a` be? AFSL does not have parametric polymorphism anyway, we have to resort to parametric overloading (see also Section 4.3).

Variables of different types have to be related to each other in order to specify that an assignment to an s_1-variable does not alter the value of any s_2-variable (for different sorts s_1 and s_2). This cannot be done if s_1 and s_2-variables belong to different sorts because only elements of the same sort can be compared in AFSL. To overcome this problem both untyped and typed variables are specified. The sorts of typed variables are treated as subsorts of the sort of untyped variables.

6.8.2 Definition of States and Variables

In Figure 6.23 `StateS` is defined as a table from `VarS` (untyped variables) to `ValueS` (untyped values). Sorts `VarS` and `ValueS` are left unspecified. The sort `VarS[XS]` of typed `XS`-variables is declared as a subsort of `VarS`, which allows the treatment of typed variables as untyped. The sort `VarS[XS]` is left unspecified here. It is up to the modules that use typed variables to introduce elements of `VarS[XS]` (that is, declare state variables). In order to allow the treatment of elements of `XS` as untyped values the wrapping function `Untype` is declared. In order to allow `Exec(^xvar,st)` always to be defined its value is left unspecified whenever `Lookup(st,xvar)` does not have a value of the form `Untype x` (this can be avoided if `^` is made a state-dependent partial function). The read (`^`) and write (`:=`) operations are defined by disguising typed variables and typed values as untyped ones (using `Inject` and `Untype` respectively).

The treatment of typed variables given here may seem a form of illusion: how can an untyped state become typed, something must be wrong here! And indeed,

```
MODULE VarM [XS]

VAR    st   : StateS
VAR    x    : XS
VAR    var  : VarS
VAR    varx : VarS[XS]

SORT   XS

IMPORT StateM        % Fig. 6.12 page 119
IMPORT UnitM         % Fig. 6.23 page 133
IMPORT PeekM [XS]    % Fig. 6.11 page 118
IMPORT PokeM [UnitS] % Fig. 6.19 page 126

SORT   VarS

SORT   ValueS

IMPORT TableM [VarS, ValueS] % Fig. 6.9 page 117

IMPORT AliasM [StateS, TableS[VarS,ValueS]] % Fig. 5.2 page 88

SORT   VarS[XS]

IMPORT SubsortM [VarS[XS], VarS] % Fig. 5.1 page 87

FUNC   Untype : XS -> ValueS

FUNC   ^ : VarS[XS] -> PeekS[XS]
AXIOM  Lookup (st, varx) = Untype x ==> Comp (^varx, st) = x

FUNC   := : VarS[XS], XS -> PokeS[UnitS]
AXIOM  Exec (varx:=x, st) = `(Unit, Assign (st, var, Untype x))

END MODULE
```

Fig. 6.23: Definition of states as untyped variable environments and specification of typed state variables.

it is still possible to assign a value of an incorrect type to a variable `v` by directly accessing the state using `Assign`. After such an ill-typed assignment the value of `^v` is unspecified, but it can still be used. There is yet another reason why `^v` might be unspecified: if no value ever is assigned to `v`. In both situations the value of `^v` should really be undefined rather than unspecified. That is, `^` should be a partial state-dependent operation. That would involve a slightly different version of `PeekS` that maps states to partial values. This is very well possible, but is omitted here to keep the example as simple as possible.

6.8.3 Using State Variables

Using state variables, in Figure 6.24 the semantics of SImPL can now be defined without having to define a tailored version of `StateS`. The definitions of the two `Eval`'s here is not much different from Figure 6.22, except that the read and set operations now have an additional argument (the variable to be read or set). What is different is that `Value`, `Input`, and `Output` are now declared as variable-valued functions rather than defined as state-dependent operations accompanied by corresponding state-changing operations.

The inequality axioms for the state variables are essential to this specification. They guarantee that all the variables used are distinct and therefore an assignment to one of the variables does not alter the value of the others. Variables of different types (such as `Input` and `Value v`) can be compared here because of the implicit injection that converts typed variables to untyped variables. The injectivity of `Value` implies that distinct SImPL variables are mapped to distinct untyped state variables. Therefore, an assignment to one SImPL variable does not alter the value of all other SImPL variables.

The need for inequality axioms results in an explosion of axioms if many state variables are used: for any pair of state variables there must be an inequality axiom. This may be unacceptable for large specifications, in particular if state variables are declared in independent modules. A possible solution for this is the extension of AFSL with a "uniqueness" attribute for functions. Two different applications of a unique function always denote different values. This means that unique functions are injective, and that two distinct unique functions have non-overlapping ranges (even if they have the same co-domain type).

For the state variables example the functions `Inject` (from typed to untyped variables), `Value`, `Input`, and `Output`, should be unique in order to avoid the inequality axioms. For example, the inequality of `Value v1` and `Value v2` follows from the injectivity of `Value`. The inequality of the untyped versions of `Output` and `Input` follows from the fact that two different instances of the overloaded function `Inject` (with different ranges) are used to convert `Output` and `Input` to type `VarS`. Uniqueness can also be used for specifying inequality of inductive data types by declaring

```
MODULE StatementSemantics4M

VAR    i : IntS
VAR    v : StringS
VAR    e : ExprS
VAR    s : StatementS

IMPORT SImPLSyntaxM          % Fig. 6.8 page 116
IMPORT VarM [IntS]           % Fig. 6.23 page 133
IMPORT StreamM [IntS]        % Fig. 6.16 page 124
IMPORT VarM [StreamS[IntS]]  % Fig. 6.23 page 133
IMPORT PokeM [IntS]          % Fig. 6.19 page 126
IMPORT PokeM [UnitS]         % Fig. 6.19 page 126

FUNC   Value : StringS -> VarS[IntS]
AXIOM  Value v1 /= Value v2  <==  v1 /= v2

FUNC   Input : -> VarS[StreamS[IntS]]
AXIOM  Input /= Value v

FUNC   Output : -> VarS[StreamS[IntS]]
AXIOM  Output /= Value v
AXIOM  Output /= Input

%... definition of lifted versions of Head, <<, etc.

FUNC   Eval : ExprS -> PokeS[IntS]
AXIOM  Eval Const i  = i
AXIOM  Eval Var v    = ^(Value v)
AXIOM  Eval Read     = (Head ^Input) << (Input := Tail ^Input)
AXIOM  Eval (e1+e2)  = (Eval e1) + (Eval e2)
AXIOM  Eval (e1-e2)  = (Eval e1) - (Eval e2)

FUNC   Eval : StatementS -> PokeS[UnitS]
AXIOM  Eval Skip        = Unit
AXIOM  Eval (v:=e)      = (Value v) := (Eval e)
AXIOM  Eval Write e     = Output := (Eval e)&(^Output)
AXIOM  Eval Cond (e,s)  = If (Eval e = 0, Eval s, Unit)
AXIOM  Eval While (e,s) = Eval Cond (e, s;While(e,s))
AXIOM  Eval (s1;s2)     = (Eval s1) >> (Eval s2)

END MODULE
```

Fig. 6.24: Alternative definition of the semantics of SImPL using typed state variables.

the constructors to be unique.

6.9 Note on Lifted Equality

One has to be careful with the use of lifted versions of the equality functions = and /=, like in the definition of Eval for Cond in the previous section. The problem is that there are builtin instances of = for each sort, in particular there is one with type:

```
= : PokeS[IntS], PokeS[IntS] -> BoolS
```

Therefore, from a typing point of view it is not necessary to use a lifted = in the axiom for Cond, the normal non-lifted = can be used too. If a lifted = for PokeS[IntS] is defined it is preferred in the expansion of the axiom for Cond, because that involves fewer conversions.

The point to note here is that it is very much relevant which instance of = is used because the non-lifted and lifted versions have different meanings. In the non-lifted case, Eval e = 0 is a *state-independent* boolean that is true if and only if Eval e and 0 are the same actions (that is, the equation is true if and only if Eval e has no side-effects and is zero in every state). Whereas, if the lifted = is used, Eval e = 0 is a *state-changing* boolean (with the same side-effects as Eval e) that is true in those states where the value of Eval e is equal to 0. In the definition of Eval for Cond the second interpretation of = is the one needed. First of all, the side-effects of evaluating e must be part of the evaluation of the conditional statement. Secondly, the choice of evaluating s has to be made for each state independently (based on the evaluation of e in that state).

A similar distinction between non-lifted and lifted equality holds for other types of actions. For example, assume there is a lifted = for PartialS[NatS], that is:

```
FUNC   = : PartialS[NatS], PartialS[NatS] -> PartialS[BoolS]
AXIOM  (Def n1 = Def n2) = Def (n1 = n2)
AXIOM  (Undef  = Def n2) = Undef
AXIOM  (Def n1 = Undef)  = Undef
```

If t_1 and t_2 are terms of type PartialS[NatS], then t_1=t_2 could be expanded in two ways to a term of type PartialS[BoolS]. One uses the non-lifted equality and is always defined (true if both t_1 and t_2 are undefined and false if just one is undefined), the other uses the lifted = and might be undefined (if t_1 and/or t_2 are/is undefined).

Non-lifted equality corresponds to what is sometimes referred to as *strong equality*, where the two values are compared including their non-functional information. Lifted equality corresponds to the so-called *strict equality* (also-called "weak equality"), where the actions are first performed before they are compared, and the non-functional information of the arguments is passed to the result of the equation. The

distinction between “strong equality” and “strict equality” usually refers to the behavior of equality with respect to partial values, but it applies to non-functional features in general. The terminology “strong” can be confusing since from a logical point of view strong equality is weaker than strict equality. That is, strict equality implies strong equality.

What is confusing about using lifted (in)equality is the overloading of the symbols `=` and `/=`. One has to be aware of the type that the result of an equation must have (either boolean or boolean valued action) in order to know which version of the (in)equality function applies, whereas for most overloaded functions it is enough to know the type of its arguments to resolve overloading. Most languages with partial functions use different syntax for strong and strict equality (for example COLD (Feijs et al. 1994)). Indeed, it might help to avoid confusion to use a different notation for lifted (in)equality in AFSL (for example, `==` and `/==`). However, that would require a special status for `=` and `/=`. In the next chapter so-called application redirection is introduced which makes the definition of lifted functions obsolete by lifting functions implicitly if necessary. The (in)equality functions should be excluded from this mechanism in order to avoid the problem above.

7. BINDING AND APPLICATION REDIRECTION

Abstract

The use of actions allows operations with non-functional features to be modeled by total functions, as described in the previous chapter. The advantage of using non-functional features is that some aspects can be left implicit in assertions. However, the method described also has some serious drawbacks. The current chapter discusses these problems and offers a solution. This involves the formal definition of the notion of *binding*. A mechanism called *application redirection* is introduced which allows binding to be applied implicitly. This leads to a notation very close to that of languages which have special provision for non-functional features (such as imperative programming languages), but still in a purely algebraic setting.

7.1 Problems with Non-Functional Features

Knowledge of the structure of actions is needed whenever they are passed to functions that expect "normal" values (that is, without non-functional features) as arguments. For example, recall the lemma from Section 6.3:

```
LEMMA  Tail l1 = Def l2  ==>  Length l1 = Length l2 + 1
```

Here one has to be aware of the fact that the partial value of `Tail l1` is of the form `Def l2` or `Undef` and as such cannot be passed as argument to `Length`. This obscures the actual property that is formalized. Using lifted versions of functions (here `Length`) shifts the problem to the definition of these functions.

There is no fixed way in which action are passed to functions that do not accept actions as arguments; this can lead to arbitrary definitions. For example, it is clear what the length of defined lists should be, but the length of `Undef` is chosen rather arbitrarily. Although it is plausible to return undefined results for undefined arguments, one can diverge from that. For example, a lifted "`-`" on partial numbers could be defined such that `Undef-Undef` is `0` instead of `Undef`. Such arbitrariness in handling actions makes specifications harder to understand and, consequently, harder to maintain and (re)use.

Languages that have builtin provisions for non-functional features do not have these problems. For example, in languages with partial functions a partial value x can be passed directly to a total function f without overhead (that is, the need to unpack x or define a special lifted version of f) and will always be handled in the same way.

In AFSL the use of parameterized modules may reduce the problems somewhat: modules can be defined that take a function f as parameter and define the lifted version of f for a particular type of action. The problems mentioned above will then be limited to these modules. However, this is not a satisfying solution: separate modules have to be available for every function arity and for every kind of non-functional feature; for each lifted function an import has to be done; and the lifted functions that are imported will have a different name than the non-lifted function (for example, `Lift[+]` as the lifted version of `+`) possibly with a different priority and associativity.

In the current chapter a different approach is taken. It is shown how "passing an action as argument" can formally be defined within a specification. Each kind of non-functional feature has its own so-called binding operation which can be used to apply a function to an action. Using this operation direct knowledge about the structure of actions is not needed anymore (since that information is incorporated within the definition of the binding operation) and actions will be passed consequently (since the same binding operation is used each time).

7.2 Binding Operations

The current section gives a generic definition of the binding operation `>>=`. Subsequent sections will instantiate `>>=` for the three example non-functional features given in the previous chapter (partiality, state-dependency, and state-change).

Binding is an operation on functions: it passes an action to a function. As such it cannot be formalized in a first-order language. Therefore, AFSL includes function representations as first-order data types. Recall from Section 3.6.2 that `FuncS[`s_1`,`s_2`]` is the builtin sort of representations of total mappings from sort s_1 to sort s_2, the builtin function `@` represents function application, and lambda abstractions can be used to construct function representations.

Using these higher-order concepts, Figure 7.1 specifies the generic operation `>>=`. For any kind of action `>>=` is supposed to be such that `ax >>= ff` first performs `ax`, returning a value `x` (that is, "binding" the value of `ax`), and then performs `ff @ x`. As noted before, "performing an action" is not a formal notion. It is the definition of `>>=` that actually defines what "performing an action" means for a particular kind of action. The requirement guarantees that performing the trivial action does nothing but returning a value.

```
MODULE BindM [XS,YS,ActionXS,ActionYS]

VAR    x  : XS
VAR    ax : ActionXS
VAR    f  : FuncS[XS,YS]
VAR    ff : FuncS[XS,ActionYS]

SORT   XS
SORT   YS
SORT   ActionXS
SORT   ActionYS

IMPORT ActionM [XS, ActionXS] % Fig. 6.1 page 109
IMPORT ActionM [YS, ActionYS] % Fig. 6.1 page 109

FUNC   >>= : ActionXS, FuncS[XS,ActionYS] -> ActionYS
REQ    (Inject x) >>= ff = ff @ x

FUNC   =<< : FuncS[XS,ActionYS], ActionXS -> ActionYS (APPLICATION)
AXIOM  ff =<< ax = ax >>= ff

FUNC   +<< : FuncS[XS,YS], ActionXS -> ActionYS (APPLICATION)
AXIOM  f +<< ax = ax >>= (x| Inject (f @ x))

END MODULE
```

Fig. 7.1: Generic definition of binding operation >>=.

The functions =<< and +<< are variations on >>= that will be discussed in Section 7.5. The attribute APPLICATION indicates that these are application operations, that is, they can be used to apply functions to arguments. These special operations are used for application redirection in Section 7.5.

The sorts ActionXS and ActionYS are assumed to be the same kind of action sorts (for example, both sorts of partial values). Unfortunately, this cannot be formalized here. Since AFSL does not have sort constructors, the action kind cannot be passed to module BindM as a parameter. See Section 8.5 for a discussion of a possible extension of AFSL with polymorphism that does allow a proper definition of >>=).

The operation >>= is inspired by the mathematical notion of monad (Moggi 1991, Wadler 1995). The >>= specified here plays the same role as in Haskell (Bird 1998). Besides the embedding from values to actions (here the subsort injection from the value type to the action type) and the binding operation >>=, monads also have three laws which restrict these two operations in order to guarantee a "sensible" notion of action. The first monad law is the requirement in BindM. Because of the lack of polymorphism the other two laws cannot be formalized in AFSL (see Section 8.5). The absence of monad laws is innocent in the sense that it cannot directly lead to "wrong" specifications. But, they do leave room for unintended definitions of >>=, which may cause confusion and, therefore, lead indirectly to mistakes.

7.3 Example: Partial Functions

In Figure 7.2 binding is defined for partial functions (see Section 6.3); in Figure 7.3 >>= is used to define the lifted version of list operations Length, +, and &.

In order to define the lifted version of a function using >>=, it has to be transformed into a function representation using lambda-abstraction. Here Inject is used to make Length l a partial value, which is needed since >>= only works on functions that return an action. The implicit function Inject must be used explicitly here since functions can only be applied implicitly to function arguments, not to the body of a lambda-abstraction. It is not always necessary to use Inject to define a lifted function. To illustrate this, the definition of a lifted version of Head has been included in Figure 7.3. Head is already a partial function, therefore, no application of Inject is needed in the definition of the lifted Head.

That the definitions of the lifted Length and & given in Figure 7.3 are indeed equivalent to the ones of Figure 6.5 is shown by the derivations in Figures 7.4 and 7.5 (for & only one case is proven).

>>= is only defined for unary functions, but functions with n arguments can be lifted by using >>= n times on a nested lambda-abstraction (such as in the definition of &). The general format of defining the lifted version of a function f is (for suitable declarations of the variables):

```
MODULE BindPartialM [XS,YS]

VAR    x  : XS
VAR    ff : FuncS[XS,PartialS[YS]]

SORT   XS
SORT   YS

IMPORT PartialM [XS] % Fig. 6.3 page 110
IMPORT PartialM [YS] % Fig. 6.3 page 110

IMPORT BindM [XS,YS,PartialS[XS],PartialS[YS]] % Fig. 7.1 page 141

AXIOM  Def x >>= ff = ff @ x
AXIOM  Undef >>= ff = Undef

END MODULE
```

Fig. 7.2: Definition of binding for partial functions.

```
AXIOM f(xa,xb,...,xx)
      = xa >>= (a| xb >>= (b|...xx >>= (x| f(a,b,...,x))...))
```

The order in which the arguments `xa` ... `xx` are bound may seem arbitrary: for example, the definition of `&` in Figure 7.3 is equivalent to:

```
AXIOM  px&pl = pl >>= (l| px >>= (x| x&l))
```

However, this is purely accidental since the order in which partial values are "performed" is irrelevant. But, if `pl` and `px` were, for example, state-changing values, then the order does matter. Since the usual order of evaluation in programming languages is from left to right, we always use left-to-right binding, even if order does not matter.

7.4 Example: State-Dependent Functions

Binding for state-dependent functions is defined in Figure 7.6 and used in Figure 7.7 in the definition of the semantics of SImPL expressions (ignoring for the moment the side-effects). No lifted functions are defined here, `>>=` is used directly to define `Eval`. Alternatively, lifted versions of `+` and `-` could be defined and used in the definition of `Eval`. In Figure 7.8 it is shown that the definition of `Eval` for `+` is indeed correct.

```
MODULE List4M [XS]

VAR    x  : XS
VAR    l  : ListS[XS]
VAR    n  : NatS
VAR    pl : PartialS[ListS[XS]]
VAR    pn : PartialS[NatS]
VAR    px : PartialS[XS]

SORT   XS

%... definition of ListS[XS] and operations (see Fig. 6.4 page 111)

IMPORT PartialM [NatS]                      % Fig. 6.3 page 110
IMPORT BindPartialM [ListS[XS], NatS]       % Fig. 7.2 page 143
IMPORT BindPartialM [NatS, NatS]            % Fig. 7.2 page 143
IMPORT BindPartialM [XS, ListS[XS]]         % Fig. 7.2 page 143
IMPORT BindPartialM [ListS[XS], ListS[XS]] % Fig. 7.2 page 143
IMPORT BindPartialM [ListS[XS], XS]         % Fig. 7.2 page 143

FUNC   Length : PartialS[ListS[XS]] -> PartialS[NatS]
AXIOM  Length pl = pl >>= (l| Inject Length l)

FUNC   + : PartialS[NatS], NatS -> PartialS[NatS]
AXIOM  pn1 + n2 = pn >>= (n1| Inject (n1 + n2))

FUNC   & : PartialS[XS], PartialS[ListS[XS]] -> PartialS[ListS[XS]]
AXIOM  px&pl = px >>= (x| pl >>= (l| Inject x&l))

FUNC   Head : PartialS[ListS[XS]] -> PartialS[XS]
AXIOM  Head pl = pl >>= (l| Head l)

LEMMA  l /= Empty  ==>  Length l = Length Tail l + 1
LEMMA  l /= Empty  ==>  l = (Head l)&(Tail l)

END MODULE
```

Fig. 7.3: Specification of lifted list operations using binding.

```
Length Def l1
=     % definition of lifted Length in Figure 7.3
Def l1 >>= (l2| Inject Length l2)
=     % definition of >>= in Figure 7.2
(l2| Inject Length l2) @ l1
=     % beta-reduction, definition Inject in  Figure 6.3
Def Length l1


Length Undef
=     % definition of lifted Length in Figure 7.3
Undef >>= (l2| Inject Length l2)
=     % definition of Inject in Figure 7.2
Undef
```

Fig. 7.4: Correctness of the definition of the lifted `Length` in Figure 7.3.

```
(Def x1) & Undef
=     % definition of lifted & in Figure 7.3
Def x1 >>= (x2| Undef >>= (l| Inject (x2&l)))
=     % definition of >>= in Figure 7.2
Def x1 >>= (x2| Undef)
=     % definition of >>= in Figure 7.2
(x2| Undef) @ x1
=     % beta-reduction
Undef
```

Fig. 7.5: Correctness of the definition of the lifted `&` in Figure 7.3 (only one case).

```
MODULE BindPeekM [XS,YS]

VAR    x  : XS
VAR    ex : PeekS[XS]
VAR    ff : FuncS[XS,PeekS[YS]]
VAR    st : StateS

SORT   XS
SORT   YS

IMPORT PeekM [XS] % Fig. 6.11 page 118
IMPORT PeekM [YS] % Fig. 6.11 page 118

IMPORT BindM [XS,YS,PeekS[XS],PeekS[YS]] % Fig. 7.1 page 141

AXIOM  Comp (ex >>= ff, st) = Comp (ff @ Comp(ex,st), st)

END MODULE
```

Fig. 7.6: Definition of binding for state-dependent functions.

```
MODULE ExpressionSemantics4M

VAR    i   : IntS
VAR    v   : StringS
VAR    e   : ExprS

IMPORT SImPLSyntaxM           % Fig. 6.8 page 116
IMPORT VarM [IntS]            % Fig. 6.23 page 133
IMPORT PeekM[IntS]            % Fig. 6.11 page 118
IMPORT BindPeekM [IntS, IntS] % Fig. 7.6 page 146

FUNC   Value : StringS -> VarS[IntS]
AXIOM  Value v1 /= Value v2  <==  v1 /= v2

FUNC   Eval : ExprS -> PeekS[IntS]
AXIOM  Eval Const i = i
AXIOM  Eval Var v   = ^(Value v)
AXIOM  Eval (e1+e2) = Eval e1 >>= (i1| Eval e2 >>= (i2| Inject i1+i2))
AXIOM  Eval (e1-e2) = Eval e1 >>= (i1| Eval e2 >>= (i2| Inject i1-i2))

END MODULE
```

Fig. 7.7: Definition of the semantics of SImPL expressions using state variable and let-terms.

```
Comp (Eval (e1+e2), st)
=     % definition of Eval in Figure 7.7
Comp (Eval e1 >>= (i1| Eval e2 >>= (i2| Inject i1+i2)), st)
=     % definition of >>= in Figure 7.6
Comp ((i1| Eval e2 >>= (i2| Inject i1+i2)) @ Comp(Eval e1,st), st)
=     % beta-reduction
Comp (Eval e2 >>= (i2| Inject (Comp(Eval e1,st) + i2)), st)
=     % definition of >>= in Figure 7.6
Comp ((i2| Inject (Comp(Eval e1,st) + i2)) @ Comp(Eval e2,st), st)
=     % beta-reduction
Comp (Inject (Comp(Eval e1,st) + Comp(Eval e2,st)), st)
=     % definition of Inject in Figure 6.11
Comp(Eval e1,st) + Comp(Eval e2,st)
```

Fig. 7.8: Correctness of the definitions of `Eval` for `+` in Figure 7.7.

7.5 Application Redirection

When using `>>=` to pass actions as arguments to functions there is still some overhead. Ideally, it should be possible to use actions directly as arguments without the need to use `>>=` and lambda abstraction. For example, the definition of `Eval` in Figure 7.7 would be clearer if we could write (without defining lifted `+` and `-`):

```
AXIOM  Eval (e1+e2) = (Eval e1) + (Eval e2)
AXIOM  Eval (e1-e2) = (Eval e1) - (Eval e2)
```

In order to allow special kinds of function application, like passing actions as normal arguments, a mechanism called *application redirection* is introduced. This allows function application to be trapped and redirected through a self-defined *application operation*, which is a function with the attribute `APPLICATION`. Examples of application operations are `=<<` and `+<<` in Figure 7.1. These operations incorporate the binding operation, thus redirection via these operations effectively means implicit binding.

Using application redirection the above "direct" axioms for `Eval` are abbreviations for:

```
AXIOM  Eval (e1+e2)
       = (i1| (i2| i1+i2) +<< (Eval e2)) =<< (Eval e1)
AXIOM  Eval (e1-e2)
       = (i1| (i2| i1-i2) +<< (Eval e2)) =<< (Eval e1)
```

which are indeed equivalent to the corresponding axioms for `Eval` in Figure 7.7

As with overloading and implicit functions, application redirection is incorporated in the rules for term expansion. The redefinition of expansion relation $\twoheadrightarrow$ is

$$\frac{}{v \twoheadrightarrow v : [\,]}$$

$$\frac{\begin{array}{c} f \leftarrowtail f' \\ t_1 \twoheadrightarrow t_1' : c_1 \quad \cdots \quad t_n \twoheadrightarrow t_n' : c_n \\ t_1' \rightsquigarrow t_1'' : a_1 \quad \cdots \quad t_n' \rightsquigarrow t_n'' : a_n \\ f'(t_1'', \ldots, t_n'') \multimap_1 t'' \end{array}}{f(t_1, \ldots, t_n) \twoheadrightarrow t'' : c_1 +\!\!+ [a_1] +\!\!+ \ldots +\!\!+ c_n +\!\!+ [a_n]}$$

$$\frac{\begin{array}{c} s \leftarrowtail s' \\ t \twoheadrightarrow t' : c \end{array}}{(v : s \mid t) \twoheadrightarrow (v : s' \mid t') : c}$$

$$\frac{e_1 \rightarrowtail e_1' : c_1 \quad \cdots \quad e_n \rightarrowtail e_n' : c_n}{\{e_1 \ldots e_n\} \twoheadrightarrow \{e_1' \ldots e_n'\} : c_1 +\!\!+ \ldots +\!\!+ c_n}$$

$$\frac{}{nl \rightarrowtail nl : [\,]} \qquad \frac{nm \leftarrowtail nm'}{\text{'}nm\text{'} \rightarrowtail \text{'}nm'\text{'} : [\,]} \qquad \frac{t \twoheadrightarrow t' : c}{\text{'}t\text{'} \rightarrowtail \text{'}t'\text{'} : c}$$

$$\frac{}{t \rightsquigarrow t' : 0} \qquad \frac{\begin{array}{c} t \rightsquigarrow t' : a \\ \pi(t') \multimap_1 t'' \end{array}}{t \rightsquigarrow t'' : a + 1}$$

$$\frac{}{f(t_1, \ldots, t_n) \multimap_{n+1} f(t_1, \ldots, t_n)}$$

$$\frac{f(t_1, \ldots, t_n) \multimap_{i+1} t'}{f(t_1, \ldots, t_n) \multimap_i t'}$$

$$\frac{f(t_1, \ldots, t_{i-1}, \mu, t_{i+1}, \ldots, t_n) \multimap_{i+1} t'}{f(t_1, \ldots, t_n) \multimap_i \phi((\mu : s \mid t'), t_i')}$$

Fig. 7.9: Definition of expansion rules, including implicit functions. For any implicit function π, variable μ not occurring free in $t_1, \ldots, t_n$, and application operation ϕ. (For any variable v; sorts s and s'; functions f and f'; terms $t_1, \ldots, t_n$, $t_1', \ldots, t_n'$, $t_1'', \ldots, t_n''$, t, t', and t''; integer lists $c_1, \ldots c_n$ and c; informals $e_1, \ldots e_n$ and $e_1', \ldots e_n'$; natural language nl; names nm and nm'; integers $a_1, \ldots, a_n$, and a; and natural numbers $n \geq 0$ and $1 \leq i \leq n$.)

given in Figure 7.9 (it replaces the definition of $\twoheadrightarrow$ in Figure 5.14 on page 104). Here $f(\ldots) \multimap_i t'$ denotes that the function application $f(\ldots)$ can be expanded to t' by possibly redirecting the application of all but the first $i-1$ arguments. The first rule for $\multimap$ is the base case in which there are no applications to redirect; the second rule corresponds to the direct application (that is, without redirection) of f to argument t_i; and the third rule corresponds to the redirected application of f to argument t_i.

Redirection is applied to implicit functions because they also may need to be lifted. For example, if `Div` is a partial function on `IntS`, then `(5 Div 2) * 0.3` is expanded to the partial number:

```
(r:FloatS| r*0.3) +<<
((i:IntS| Inject i) +<< ((Inject Nat 5) Div (Inject Nat 2)))
```

where the first `Inject` (from integers to floats) is lifted such that it accepts partial integers.

Application redirection can be used to reformulate the lemmas of `ListM` of Figure 6.4:

```
IMPORT BindPartialM [ListS[XS], NatS]
IMPORT BindPartialM [NatS, NatS]

LEMMA  l /= Empty ==> Length l = Length Tail l + 1
```

This new formulation of the lemma is exactly as in Figure 6.4, but now without the need to define lifted versions of list operations `&` and `Length`. The (simplified) expansion of the above lemma is:

```
LEMMA  l /= Empty
       ==>
       Length l = (n| n+1) +<< ((l1| Length l1) +<< (Tail l))
```

Note that `=` and `==>` are not implicitly lifted here because assertions cannot be partial (they must always be of type `BoolS`). This is not a problem here because there is a `=` for partial numbers. However, in general one has to be careful with undefined terms in assertions. For example, the following example causes a problem (assuming `Div` is a partial function):

```
LEMMA  i > 0 ==> 1 Div i > 0
```

Since `1/n` is a partial value, both `>` and `==>` have to be lifted:

```
LEMMA  (b| i > 0 ==> b) +<< ((i2| i2 > 0) +<< (1 Div i))
```

But, this is not allowed because it makes the lemma partial. A possible solution is using an (implicit) function that maps `Undef` to `False` or a conditional implication operation, as in:

```
MODULE BindPokeM [XS,YS]

VAR    x  : XS
VAR    ax : PokeS[XS]
VAR    ff : FuncS[XS,PokeS[YS]]
VAR    pf : FuncS[XS,PeekS[YS]]
VAR    st : StateS

SORT   XS
SORT   YS

IMPORT PokeM [XS] % Fig. 6.19 page 126
IMPORT PokeM [YS] % Fig. 6.19 page 126

IMPORT BindM [XS, YS, PokeS[XS], PokeS[YS]] % Fig. 7.1 page 141

AXIOM  Exec (ax, st1) = `(x, st2)  ==>
       Exec (ax >>= ff, st1) = Exec (ff @ x, st2)

FUNC   ++< : FuncS[XS,PeekS[YS]], PokeS[XS] -> PokeS[YS] (APPLICATION)
AXIOM  pf ++< ax = ax >>= (x| Inject (pf @ x))

END MODULE
```

Fig. 7.10: Lifting of state-changing operations.

```
FUNC   |=> : BoolS, PartialS[BoolS] -> BoolS
AXIOM  (b1 |=> Def b2) = (b1 ==> b2)
AXIOM  (b  |=> Undef)  = (Not b)

LEMMA  i > 0 |=> 1 Div i > 0
```

7.6 Example: State Changing Functions

Binding for state-changing values is defined in Figure 7.10. An additional application operation `++<<` is defined here which can be used to bind state-changing values to state-dependent functions. This form of binding is possible because `PeekS[XS]` is a subsort of `PokeS[XS]`.

An alternative definition of the semantics of SImPL statements using application redirection is given in Figure 7.11. The correctness of the definition of `Eval` for `+` is shown in Figure 7.12. Note that for state-changing functions the order in which arguments are executed does matter.

```
MODULE StatementSemantics5M

VAR    i : IntS
VAR    v : StringS
VAR    e : ExprS
VAR    s : StatementS

%... imports and declaration of state variables, see Fig. 6.24 page
135

IMPORT BindPokeM [IntS, IntS]                    % Fig. 7.10 page 150
IMPORT BindPokeM [UnitS, IntS]                   % Fig. 7.10 page 150
IMPORT BindPokeM [IntS, UnitS]                   % Fig. 7.10 page 150
IMPORT BindPeekM [StreamS[IntS], IntS]           % Fig. 7.10 page 150
IMPORT BindPokeM [StreamS[IntS], UnitS]          % Fig. 7.10 page 150
IMPORT BindPokeM [IntS, StreamS[IntS]]           % Fig. 7.10 page 150
IMPORT BindPokeM [StreamS[IntS], StreamS[IntS]] % Fig. 7.10 page 150
IMPORT BindPeekM [IntS, StreamS[IntS]]           % Fig. 7.6 page 146
IMPORT BindPeekM [StreamS[IntS], StreamS[IntS]] % Fig. 7.6 page 146
IMPORT BindPokeM [UnitS, UnitS]                  % Fig. 7.6 page 146
IMPORT BindPokeM [IntS, BoolS]                   % Fig. 7.6 page 146
IMPORT BindPokeM [BoolS, UnitS]                  % Fig. 7.6 page 146

FUNC   Eval : ExprS -> PokeS[IntS]
AXIOM  Eval Const i = i
AXIOM  Eval Var v   = ^(Value v)
AXIOM  Eval Read    = Head ^Input << Input := Tail ^Input
AXIOM  Eval (e1+e2) = Eval e1 + Eval e2
AXIOM  Eval (e1-e2) = Eval e1 - Eval e2

FUNC   Eval : StatementS -> PokeS[UnitS]
AXIOM  Eval Skip       = Unit
AXIOM  Eval (v:=e)     = (Value v) := Eval e
AXIOM  Eval Write e    = Output := (Eval e)&(^Output)
AXIOM  Eval Cond(e,s)  = If (Eval e = 0, Eval s, Unit)
AXIOM  Eval While(e,s) = Eval Cond (e, s;While(e,s))
AXIOM  Eval (s1;s2)    = Eval s1 >> Eval s2

END MODULE
```

Fig. 7.11: Definition of the semantics of SImPL expressions and statements using state variables and application redirection. This module is an adaptation of Figure 6.24.

```
assume: Exec (Eval e1, st1) = ‘(i1, st2)
        Exec (Eval e2, st2) = ‘(i2, st3)

Exec (Eval (e1+e2), st1)
=     % definition Eval for + in Figure 7.11
Exec (Eval e1 + Eval e2, st1)
=     % expansion
Exec ((j1| (j2| j1+j2) +<< (Eval e2)) =<< (Eval e1), st1)
=     % definition of =<< in Figure 7.1
Exec ((Eval e1) >>= (j1| (j2| j1+j2) +<< (Eval e2)), st1)
=     % definition >>= in Figure 7.10
Exec ((j1| (j2| j1+j2) +<< (Eval e2)) @ i1, st2)
=     % beta-reduction
Exec ((j2| i1+j2) +<< (Eval e2), st2)
=     % definition +<< in  Figure 7.1,
Exec ((Eval e2) >>= (i| Inject ((j2| i1+j2) @ i), st2)
=     % definition >>= in  Figure 7.10
Exec ((i| Inject ((j2| i1+j2) @ i) @ i2, st3)
=     % beta-reduction
Exec (Inject (i1+i2), st3)
=     % definition Inject from XS to PokeS[XS] in Figure 6.19
‘(i1+i2, st3)
```

Fig. 7.12: The semantics of the addition operation + as defined in Figure 7.11.

```
Assume: Exec (Eval e, st1) = ‘(i1, st2)
        Lookup (st2, Inject Value (v, i1)) = Untype i2

Exec (Eval (v#e), st1)
=     % definition Eval for #
Exec (^(Value (v, Eval e)), st1)
=     % expansion
Exec ((var| ^var) ++<< ((j| Value (v, j)) +<< (Eval e)), st1)
=     % definition of ++<< in  Figure 7.10
Exec (((j| Value (v, j)) +<< (Eval e)) >>= (var| Inject ^var), st1)
=     % Exec ((j| Value (v, j)) +<< (Eval e), st1)
      % =  % definition +<< in  Figure 7.1,
      % Exec ((Eval e) >>= (j| Inject Value (v, j)), st1)
      % =  % definition >>= in  Figure 7.10
      % Exec (Inject Value (v, i1)), st2)
      % =  % definition Inject from XS to PokeS[XS] in Figure 6.19
      % ‘(Value (v, i1), st2)
Exec ((var| Inject ^var) @ (Value (v, i1), st2)
=     % beta-reduction
Exec (Inject ^Value (v, i1), st2)
=     % definition Inject from PeekS[XS] to PokeS[XS] in Figure 6.19
‘(Comp (^Value (v, i1), st2), st2)
=     % definition ^ in Figure 6.23
‘(i2, st2)
```

Fig. 7.13: The semantics of the array indexing operation `#`.

7.7 Example: Variable Arrays

The definition of the semantics of SImPL in Figure 7.11 does not make use of the application operation `++<<` of Figure 7.10 that binds state-changing values to state-dependent functions. In order to show the use of this form of binding, the SImPL program variables are changed to variable arrays.

Assume that, in the abstract syntax of SImPL in Figure 6.8, the constructors `Var` and `:=` are replaced by:

```
FUNC   #   : StringS, ExprS  -> ExprS (CONSTRUCTOR)
FUNC   :=  : StringS, ExprS, ExprS  -> StatementS (CONSTRUCTOR)
```

Here `a#e` denotes the value of array `a` at position `e` (usually written as `a[e]`, but that is not allowed by the syntax of AFSL) and `(a,e1):=e2` denotes the update of `a` at position `e1` to value `e2` (usually written as `a[e1]:=e2`).

The declaration of `Value` (see Figure 6.24) is changed such that it can store the values of arrays:

```
FUNC   Value : StringS, IntS -> VarS[IntS]
AXIOM  v1 /= v2  Or  i1 /= i2
       ==>
       Value (v1, i1) /= Value (v2, i2)
```

The definition of `Eval` for the array operations is:

```
AXIOM  Eval (v#e)        = ^(Value (v, Eval e))
AXIOM  Eval ((v,e1):=e2) = (Value (v, Eval e1) := Eval e2)
```

In Figure 7.13 the correctness of the definition of `Eval` for `#` is shown. In the before-last step the injection from `PeekS[XS]` to `PokeS[XS]` is used.

7.8 Explicit Binding

It can sometimes be useful to use explicit binding instead of implicit binding through application redirection, in particular to bind the value of an action that is going to be used more than once. For example, consider the next two definitions of the evaluation of the `Read` operation of SImPL :

```
AXIOM  Eval Read = Head ^Input << Input := Tail ^Input
AXIOM  Eval Read = ^Input >>= (in| Input := Tail in >> Head in)
```

The first axiom is taken from Figure 7.11, the second is an alternative, though equivalent, definition. In the second axiom it is emphasized that the sub-term `^Input` is used twice. Moreover, the explicit binding of `in` to the result of `^Input` allows us to do the assignment to `Input` before returning `Head in` as a result, which is the common order of doing things. As a matter of fact, imperative programming languages do not have an equivalent for `<<`, whereas `>>` corresponds to sequential composition.

A special situation arises when performing an action; that is, whenever performing the same action twice has a different effect than just performing it once. A typical case of non-idempotent actions are state-changing actions. Consider, for example, the following three state-changing terms (with the definition of `Eval Read` from the previous example):

```
(Eval Read) + (Eval Read)
(Eval Read) >>= (i1| (Eval Read) >>= (i2| i1 + i2))
(Eval Read) >>= (i| i + i)
```

The first two terms are equivalent, both read two integers `i1` and `i2` from the input stream and return the sum of `i1` and `i2`. However, the last term reads only one integer `i` from input and returns twice the value of `i`. Not only the value of the last term is possibly different from the former two, but also the returned state. In the first two cases two elements are removed from the input stream, in the final case only one. So, whenever the value of a non-idempotent action is used more than once, one should use let-terms for explicit binding of the value.

8. DISCUSSION

Abstract

This chapter contains a critical discussion of some of the design decisions of AFSL (see Chapter 2). Although AFSL does satisfy its original requirements, some improvements can be made to improve its usability and definition. Also it is discussed how the FSA specification method could be developed further.

8.1 First-Order Functions

AFSL was designed as a first-order language in order to keep the language definition as simple as possible. It was thought that first-order functions, in combination with generic modules, would be powerful enough for specification purposes (many existing specification languages are first order). Although it is true that much can be modeled using first-order functions, it does not always result in clean (expanded) specifications. The addition of function representations to the builtin objects (see Section 3.6) is an ad-hoc extension which was needed for the application redirection (which in turn is needed to model non-function features). Functions must be transformed into function representations (using lambda abstraction) before application redirection can be applied, which is rather elaborate. A higher-order language would allow a cleaner language definition (see Section 8.8).

When it was decided to add application redirection to AFSL, the type system was fixed to a first-order system (in fact, the language was already in use). It was (and still is) hard to assess whether true higher-order functions would conflict with other language features (in particular, the syntax of function application, implicit functions, and application redirection).

8.2 Syntax

If a flexible syntax is required (for example, to allow `if␣then␣else␣` notation), a specification language really needs mixfix operation calls (see Section 2.1.2). It seems obvious to use a mechanism similar to SDF (Bergstra, Heering & Klint 1987, Heering, Hendriks, Klint & Rekers 1992). SDF is based on the elegant idea that

declarations of mixfix operations can be viewed as a grammar definition where the sorts are the non-terminals of the grammar (see the example ASF+SDF specification in Figure 2.1). Unfortunately, there are some other wishes (polymorphism, higher-order functions, implicit functions, and application redirection) that may stand in the way of a straightforward employment of a SDF-like syntax within AFSL because it complicates parsing.

8.3 Informal Terms

Informal terms (Section 3.11) do facilitate stepwise refinement (Section 2.2.1) in a straightforward way. In the case studies informal terms turned out to be very useful for a first definition of names. However, informal terms were only used on a limited scale: informal terms were not used as sub-terms of formal terms and quoted terms were hardly used. The remaining advantage of such informal terms over ordinary comments is that the type-checker tests whether the quoted names are indeed declared.

The limited use of informal terms may justify a simpler mechanism. An alternative is the use of *description items* which are made up of natural language with quoted names and terms, for example:

```
FUNC   + : NatS, NatS -> NatS
DESCR  'n1+n2' is the sum of 'n1' and 'n2'.
AXIOM  Zero + n     = n
AXIOM  Succ n1 + n2 = Succ (n1+n2)
```

8.4 Non-executable Semantics

Definitions in AFSL are non-executable (see Section 2.1.3) in order not to limit its expressiveness. However, within the FSA case studies (see Section 1.6) it was recognized that executable prototypes are an important tool to validate specifications (that is, to find out whether a specification specifies what it is supposed to). It was assumed that turning specifications into prototypes would be a straightforward exercise, but that turned out to be wrong.

In the MP case study, AFSL axioms were translated manually to (more or less) logically equivalent Prolog clauses. In principle this works well, but the translation turned out to be very laborious. Not all axioms were Horn clauses and, therefore, needed to be converted first. Because this was done manually, the prototypes and specifications diverged more and more over time, which made any valuable feedback from prototype to specification impossible.

For FAN a small interpreter was made that could execute *some* of the axioms that were specific for that case. This also never resulted in a workable situation because

the implementation of the interpreter was part of the case study itself, whereas it should have been available as a tool from day one.

In retrospect, I think that definitions (that is, axioms, not lemmas and requirements) should always be executable in order to allow instant prototyping. In addition, many people find it simpler to write executable definitions instead of abstract property-based definitions. An operational understanding of a definition often makes it easier to envision its implications (there is only one mental model to consider). Also, the restrictions imposed by executable definitions lead to more uniform definitions with less room for mistakes.

When I have to design a new specification language in the future, I will consider an executable language with special provisions for the *forward declaration* of names. That is, names can be declared without a definition, which is forwarded to any module down the import chain. Most definitions in this thesis are in the form of rewrite rules, so it should not be difficult to change these into executable definitions. Forward declarations may contain requirements, which are boolean terms. A requirement does not supply any operational information about the declared name, but is itself a computable term (given values for the free variables). When requirements are accompanied by a list of test values for their free variables, then the definitions can automatically be tested by computing all the requirements for the given test values and check if their value is `True`. Similarly, declaration may contain lemmas with test values. Validity of a specification can then be tested by computing these lemmas.

8.5 Modules

The generic module mechanism defined in Chapter 4 is very useful as an abstraction, structuring, and reuse mechanism. Moreover, it has a straightforward definition based on unfolding of structured modules, which makes it easy to understand structured specifications. There is, however, one serious drawback in using parameterized modules as an abstraction mechanism: for each required instance of a polymorphic operation one import is needed. Particularly, in the case of modules that define binding operations (Chapter 7) this leads to many imports.

Also, it is not clear how useful the mechanism is for an executable language. An interpreter/compiler that handles structured specifications by simply unfolding all imports may face an explosion of code. Maybe, there are techniques that can avoid this (for example, by eliminating duplicate definitions).

Extension of AFSL with *parametric polymorphism* (see Section 4.4) can help to reduce the number of module imports. For example, a module that specifies a polymorphic sort of lists such as in Figure 8.1 has to be imported only once to declare lists of any element type (compare this with to the specification of lists given in Figure 4.4). Here an ad-hoc extension of AFSL is used in which sort names ending

```
MODULE ListM

VAR    x  : X$
VAR    xl : ListS[X$]

IMPORT LogicM % Fig. 3.3 page 43

SORT   ListS[X$] (INDUCTIVE)
FUNC   Empty :               -> ListS[X$] (CONSTRUCTOR)
FUNC   &     : X$, ListS[X$] -> ListS[X$] (CONSTRUCTOR)

AXIOM  x&xl   /= Empty
AXIOM  x1&xl1 /= x2&xl2  <==  x1  /= x2
AXIOM  x1&xl1 /= x2&xl2  <==  xl1 /= xl2

%... definition of other operations on lists

END MODULE
```

Fig. 8.1: Specification of typed lists using type polymorphism.

with `$` are *sort variables* that can be replaced by any other sort name. The declaration of `ListS[X$]` now declares a whole collection of sorts names (`ListS[BoolS]`, `ListS[IntS]`, `ListS[ListS[IntS]]`, etc.). So, only one import of this polymorphic `ListM` is sufficient for the definition of all this different list types.

Parametric polymorphism reduces the amount of module imports (and possible code explosion). Note, however, that polymorphism does not make module parameters obsolete. There are situations in which we do not want to instantiate a generic specification for arbitrary actual parameters. See, for example, the module `TotalOrderedM` in Figure 4.7: we do not want a `>=` for every sort. Here the parameterized modules of AFSL play a role similar to classes in Haskell (Fasel et al. 1992, Bird 1998), which also supports polymorphism.

Parametric polymorphism is particularly useful when more than one sort variable is involved. See, for example, the two specifications of the composition operation `O` for function representations given in Figure 8.2. The first definition of `O` has to be imported separately for each combination of values for `AS`, `BS`, and `CS`, the second has to be imported only once. Here we can see an additional advantage of parametric polymorphism. The lemma in the second version of `ComposeM` cannot even be formulated within the current AFSL because multiple instances of `O` are involved here (in the first version of `ComposeM` there is only one instance of `O`). Different instances of `O` can only be declared by imports of `ComposeM`. Even if it were allowed to import modules into themselves, that would not be sufficient because the lemma must hold

```
MODULE ComposeM [AS,BS,CS]

VAR    f : FuncS[AS,BS]
VAR    g : FuncS[BS,CS]
VAR    a : AS

SORT   AS
SORT   BS
SORT   CS

FUNC   O : FuncS[BS,CS], FuncS[AS,BS] -> FuncS[AS,CS]
AXIOM  (g O f) @ a = g @ (f @ a)

END MODULE


MODULE ComposeM

VAR    f : FuncS[A$,B$]
VAR    g : FuncS[B$,C$]
VAR    h : FuncS[C$,D$]
VAR    a : A$

FUNC   O : FuncS[B$,C$], FuncS[A$,B$] -> FuncS[A$,C$]
AXIOM  (g O f) @ a = g @ (f @ a)
LEMMA  (h O g) O f = h O (g O f)

END MODULE
```

Fig. 8.2: Two definitions of the composition operation on function representations; the first using module parameters, the second using sort variables.

```
MODULE ActionM [ActionS[X$]]

VAR    x  : X$
VAR    ax : ActionS[X$]
VAR    f  : FuncS[X$,ActionS[Y$]]
VAR    g  : FuncS[Y$,ActionS[Z$]]

SORT   ActionS[X$]

IMPORT SubsortM [X$, Action[X$]] % Fig. 5.1 page 87

IMPORT ComposeM % Fig. 8.2 page 159

FUNC   >>= : Action[X$], FuncS[X$,Action[Y$]] -> Action[Y$]
REQ    (Inject x) >>= f        = f @ x
REQ    ax >>= (x| Inject x)    = ax
REQ    ax >>= (x| (f@x) >>= g) = (ax >>= f) >>= g

%... definition of =<<, +<<, If, etc.

END MODULE
```

Fig. 8.3: Alternative definition of actions and binding, including the monad laws.

for *arbitrary* instances of the sort variables, even for sort names not yet declared.

Parametric polymorphism would be very useful for modeling non-functional features (Chapter 7). Currently, for each possible binding operation (Section 7.2) a separate import is needed in order to define the right binding operation. Because binding is often applied implicitly by application redirection (Section 7.5), one can easily forget some necessary imports. Moreover, the current specification of `>>=` in Figure 7.1 fails to satisfy two of the so-called monad laws (see Section 7.2, Moggi (1991) and Wadler (1995)) because these involve multiple instances of `>>=`. The monad laws are requirements that have to be satisfied by any "tidy" definition of binding (the reasons why is outside the scope of this thesis). Figure 8.3 shows how binding can be specified using sort variables, including the monad laws (the requirements at the end of the module).

Here a new form of formal parameter (`ActionS[X$]`) is needed. This is a *polymorphic parameter* that may only be instantiated by names with an equal number of sort variables (such as `ListS[X$]` from Figure 8.1). Note that the new `ActionM` combines two previous modules (`ActionM` of Figure 6.1 and `BindM` of Figure 7.1). The original separation was needed because `BindM` uses multiple instances of sort `ActionS[]`.

8.6 Implicit Functions

The implicit function mechanism is well suited to model inheritance. Because of its generality it is more flexible than true object-oriented inheritance. It can, for example, be used for has-a relationships and defining aliases (using mutual inheritance).

The main disadvantage is that a preference mechanism is necessary to resolve ambiguities caused by multiple inheritance and overloading (Section 5.8). This preference mechanism is chosen rather ad-hoc, although conceptually it is quite simple and it is relatively easy to implement (if efficiency is not an issue, see Section 8.9). But, that may also be true for other definitions of preference.

The left-to-right bias in minimizing the number of conversions introduces an asymmetry in the expansion mechanism. The order of function arguments influences the way assertions are expanded, and therefore their interpretation for a given algebra. From a language design point of view this is unwanted since the semantics of a language should be independent of its syntax.

It is not clear to what extent the chosen preference mechanism leads to practical problems more severe than any other solution for the inheritance ambiguity problem. Before any preference mechanism can be evaluated, criteria should be formulated. A possible criterion is that the preferred expansion of an assertion is equivalent (that is, has the same value) to all other expansions. Another option is that the preferred expansion should be the logically weakest one possible (making sure the assertion has no unintended implications).

8.7 Non-Functional Features

As demonstrated in Chapter 7, application redirection allows a treatment of non-functional features without notational overhead (that is, with a notation similar to languages with builtin non-functional features). The mechanism was added to the language based on the FSA case studies, but still has to be tested in realistic case studies. For practical use the application redirection mechanism may turn out to be too complicated.

Application redirection (Section 7.5) completely eliminates the overhead that was introduced by modeling non-functional features by actions. Unfortunately, overhead that still remains are the imports of the appropriate binding modules (`BindPartialM`, etc.) that must ensure that all necessary application operations (such as `=<<`) are indeed defined. Many imports may be needed (with different actual parameters), which is a potential source of errors since it is hard to oversee which imports are exactly needed. The application operations that need to be imported are not used explicitly and therefore easily over-seen.

The problem is not so much application redirection itself, but the lack of polymorphism in AFSL. If polymorphism was available, the binding modules would not

need formal parameters, instead they would declare polymorphic versions of `>>=`. At most one import of each binding module would then be needed. Adding polymorphism to AFSL is, however, outside the scope of this thesis; see Section 8.5 for a discussion of a possible extension of AFSL with polymorphism.

Recall the discussion in Section 6.9 about the potential confusion that can arise when using lifted versions of the equality operations `=` and `/=`. Now that binding can be implicit one should be even more aware of this problem. Informal functions (Section 3.11) have a similar problem. Informal functions are extremely overloaded, they can have any domain (similar to the equality operations), but also any co-domain type (unlike the equality operations). Whenever an argument in the application of an informal function f (that is, a quoted term in an informal term) is an action, for example, of type `PartialS[IntS]`, and the corresponding instance of `=<<` is defined, then overloading ambiguity occurs since there is a choice between a version of f that works on `IntS` or one that works on `PartialS[IntS]`. These complications could be a reason for excluding `=`, `/=`, and informal terms from application redirection.

8.8 Application Redirection

Application redirection (Section 7.5) could potentially lead to incomprehensible specifications since one is free to define as many application operation as wanted, with an arbitrary definition. If this is done thoughtlessly, some function application may not behave as expected. Maybe some restrictions on the use of application redirection are in order, but at this point it is not clear how these should be formulated

Although in itself a straightforward concept, the definition of application redirection is rather circumstantial. To a large extent this is due to the lack of higher-order functions (see Section 8.1). If higher-order function were allowed, it might be possible to handle application redirection by fittings. A *fitting* is a function that can be applied implicitly to functions, rather than to arguments. For example, the definition of actions in Figure 8.3 could be extended with a fitting `Lift` (using parametric polymorphism as discussed in Section 8.5):

```
FUNC   Lift : (X$ -> ActionS[Y$])
              -> (ActionS[X$] -> ActionS[Y$]) (FITTING)
AXIOM  (Lift f) @ ax  =  ax >>= f
```

That is, `Lift f` is the lifted version of `f`. Using lifting as a fitting makes it unnecessary to convert a function to a function representation before application redirection can be used. Assuming the following declarations:

```
FUNC  F : RealS -> PartialS[RealS]
FUNC  R :       -> PartialS[RealS]
```

the application `F R` can be expanded to `(Lift F) R`

In the current AFSL, the interaction between application redirection and implicit functions (which may also need to be redirected) is quite complicated (see definition of $\twoheadrightarrow$ in Figure 7.9). Maybe, these rules can be simplified by replacing implicit functions by fittings. This leads to a kind of inheritance which is more general than the object-oriented form. For example, the implicit conversion from naturals to integers and from integers to reals can be realized by the following fittings:

```
FUNC  FitNat : (IntS  -> X$) -> (NatS -> X$)  (FITTING)
FUNC  FitInt : (RealS -> X$) -> (IntS -> X$)  (FITTING)
```

Assuming the following declarations:

```
FUNC  F : RealS -> PartialS[RealS]
FUNC  X :       -> NatS
FUNC  Y :       -> PartialS[NatS]
```

the application `F X` can be expanded to:

```
(FitNat (FitInt F)) X
```

That is, first `F` is fitted to accept integers instead of reals and then to accept naturals instead of integers. The expansion of the application `F Y` involves a lifting in order to adapt `F` to partial arguments:

```
(Lift (FitNat (FitInt F))) Y
```

This mechanism looks very promising, but needs further investigation.

8.9 Type Checking

For the version of AFSL that was used in the FSA case studies (Groenboom 1997, Veenstra 1998) a type checker was implemented in the functional programming language SML (Harper et al. 1986, Myers et al. 1993). The version of AFSL presented in this thesis has evolved from that early version. Apart from some minor syntactical and conceptual details, there are two main differences between the two versions.

The preference mechanism for adding implicit functions to terms (see Section 5.8) for the old version was hardwired into the implementation of the type checker. There was no clear definition of this mechanism such as in Figure 5.14. In fact it was quite difficult to fully understand the preference mechanism (although this did not complicate the actual use of the language).

In the case studies no use was made of actions for modeling non-functional features (see Chapter 6), therefore no application redirection (see Section 7.5) was needed. Instead the old AFSL had builtin partial functions.

Moreover, the type checker was optimized for performance and included version management and error handling. All this made it impractical to adapt the original

type checker to the final version of AFSL in a straightforward way. Therefore, a new type checker has been implemented in order to check the specification given in this thesis. This checker is a "quick and dirty" prototype which is not meant to be used for any other purposes. No attention was paid to performance, robustness, error messages, and version management.

The new type checker is implemented in the functional programming language Haskell (Fasel et al. 1992, Bird 1998). The main reason to use this language is that it uses lazy evaluation, which SML does not. Laziness is essential here because expansion of a term t is implemented by generating the (potentially infinite) list of all expansions of t in order of their costs (the cheapest first). The preferred expansion of t is the first in the list (provided it is the only expansion with lowest costs). Generating the list of all expansions makes it easy to write a program that stays close to the rules given in Figure 7.9. As a matter of fact, there has been feedback in both directions between the formulation of the rules and their implementation. The laziness of Haskell guarantees that only those expansions are computed that are really needed.

The type-checker uses a parser that is generated by a parser generator for the syntax definition language SDF-2 (Visser 1997). Joost Visser provided the tools that allowed the representations of the parse trees, constructed by the SGLR parser (Rekers 1992, Visser 1997) for SDF-2, to be read from within a Haskell program. SDF tools were used for pretty printing (de Jonge 1999, van den Brand & de Jonge 1999) the syntax definition of Appendix A.

If a language can declare its own priorities (like AFSL) it is impossible to use parser generators that assume fixed priority for operations (such as YACC (Levine, Mason & Brown 1992) or the ASF+SDF meta-environment (*The ASF+SDF Meta-environment User's Guide* 1995)). In principle it is possible to use such generators if the scanner can be programmed such that it first parses the priority section. When scanning a function identifier in a module it can then return special tokens for each priority group/associativity combination. But, this leads to an explosion of the number of syntax rules for function application.

Therefore, the type checker parses specifications in two stages. The first stage uses a generated parser that assumes all functions have the same priority and are right associative. This parser is generated from the syntax given in Appendix A. The second stage uses a hand coded program that rearranges the parse trees based on the information read from the priorities file.

8.10 Specification Method

The FSA method as presented in Section 2.2 is far from a complete method that can guide and support people in making formal specifications. What has to be done to make it a complete specification method?

I distinguish two kinds of techniques that make up a method: hard and soft techniques. Hard techniques have a clear status and have to be either completely accepted or rejected (for example, the use of formal languages, proof rules, libraries, or computer tools). Soft techniques can be interpreted in many different ways and function as suggestions for good practice (such as "which classes do we need?" or "which things belong in a module?").

When the FSA project started I expected that it would be possible to formulate useful soft techniques that can help people to make the right decisions. However, their are few existing soft techniques that could be used (such as clear guidelines for setting up an object-oriented model). Moreover, soft techniques tend to be under discussion all the time (even more than the definition of a language) and everyone changes the rules at will. Now I think that it are the hard techniques that are the driving force of a method because they *force* people to adapt a particular way of working. From this point of view, the success of object-orientation depends on its languages and tools rather than its methodology.

Development of a method of specification should concentrate on language, tools, and library development. A nice idea is the development of a graphical modeling tool where object-oriented models are drawn on the computer screen and large parts of the specification are generated from those models. Details that cannot be captured in the graphical notation then have to be added by hand. Such tools already exist for generating computer programs. For example, there are Together (see www.togethersoft.com) and Rational Rose (see www.rational.com) which use the object oriented modeling language UML (Booch, Rumbaugh & Jacobson 1999) and generate skeleton code in some programming language (that is, programs that have to completed by hand). Maybe any of these tools can be configured such that it generates skeleton specifications instead of programs.

8.11 Conclusions

AFSL has served very well as a specification language for the FSA case studies. It is simple, though expressive enough. However, as discussed in the current chapter there are still a few improvements that can be made:

- Type polymorphism for more expressiveness and fewer module imports.
- Mixfix function application for a more flexible syntax.
- True higher-order functions for a cleaner language definition.
- Integration of the preference mechanism for implicit functions and application redirection for a cleaner language definition.
- Executable definitions for a quick validation of specifications.

Future research may contribute to these points.

From the point of view of the original goals of the FSA project (Section 1.5) the decision to develop a new language turned out to be not the best choice. Although the arguments to do so (see Chapter 2) are still valid, the amount of work involved in making a new language was underestimated. It is so much work because designing a formal language (or any other completely new artifact) is inherently difficult. Moreover, there are very few techniques available that help one to make a new language. In particular, the lack of a generally accepted formal meta-language (plus tools) for the definition of language semantics was a handicap. As a result, AFSL is developed at the expense of the development of the method of specification, which was the main motivation for starting the FSA project in the first place.

This does not imply that the development of AFSL was useless. If we forget about the original goals of the FSA project, it is fair to say that a number of important aspects of formal specification are addressed in the design of AFSL. The requirements listed in Section 2.3 can be relevant for any specification method. Although AFSL does not completely satisfy all these requirements, it has a unique combination of features that do form a sound basis for further development. In particular, the way application redirection facilitates non-functional features allows a flexibility in the semantics of operations no other language offers, but which is essential for a truly wide-spectrum and general purpose language.

APPENDIX

A. SYNTAX DEFINITION

This appendix contains the syntax definition of AFSL in the syntax definition language SDF-2 (Visser 1997). Note that in the syntax given here all functions have the same priority and are right associative (see Section 8.9).

Lexical Syntax

module *Lexical*
exports
sorts ModuleId SortId FunctionId Numeral Str Variable NaturalLanguage NaturalNumber Special

lexical syntax

[\␣\t\n**]**	→ **LAYOUT**
[\%**]** **~[**\n**]* [**\n**]**	→ **LAYOUT**
[****] **~[**\n**]* [**\n**]**	→ **LAYOUT**
[A-Z**]** **[**A-Za-z0-9**]* [**M**]**	→ ModuleId
[A-Z**]** **[**A-Za-z0-9**]* [**S**]**	→ SortId
[A-Z**]** **[**A-Za-z0-9**]* [**A-LN-RT-Za-z0-9**]**	→ FunctionId
[A-LN-RT-Z**]**	→ FunctionId
Numeral	→ FunctionId
Str	→ FunctionId
Special	→ FunctionId
[0-9**]+**	→ Numeral
“\”” **~[**\”**]** “\””	→ Str
[a-z**]** **[**a-z**]* [**0-9**]***	→ Variable
~[\{\}\’\␣\t\n\%**]+**	→ NaturalLanguage
[\!\#\$\%\&*\+\-\.\:\;\<\=\>\?\@\^_\‘|\~**]+**	→ Special

Priorities Declarations

module *Priorities*
exports
imports *Lexical*

exports
sorts Priorities Group Operator Associativity

context-free syntax

"PRIORITIES" Group***** "END" "PRIORITIES"	→ Priorities
"GROUP" {Operator ","}*****	→ Group
FunctionId Associativity	→ Operator
	→ Associativity
"(" "LEFT" ")"	→ Associativity
"(" "RIGHT" ")"	→ Associativity

Specifications

module *Module*
exports
imports *Term*

exports
sorts Module NameList Item Declaration Attribs Attribute Qualifier

context-free syntax

"MODULE" ModuleId NameList Item***** "END" "MODULE"	→ Module
	→ NameList
"[" {Name ","}***** "]"	→ NameList
"IMPORT" ModuleId NameList	→ Item
Declaration	→ Item
Qualifier Term	→ Item
"SORT" Sort Attribs	→ Declaration
"FUNC" Function Attribs	→ Declaration
"VAR" Variable ":" Sort	→ Declaration
	→ Attribs
"(" {Attribute ","}***** ")"	→ Attribs
"IMPLICIT"	→ Attribute
"INDUCTIVE"	→ Attribute
"CONSTRUCTOR"	→ Attribute
"APPLICATION"	→ Attribute
"AXIOM"	→ Qualifier
"LEMMA"	→ Qualifier
"REQ"	→ Qualifier

module *Name*
exports
imports *Lexical*

exports
sorts Sort Function Name Origin Indexes TypeInfo FunctionType

context-free syntax

Origin SortId Indexes	→ Sort
Origin FunctionId Indexes TypeInfo	→ Function
Sort	→ Name
Function	→ Name
	→ Origin
ModuleId ":"	→ Origin
	→ Indexes
"[" {Name ","}* "]"	→ Indexes
	→ TypeInfo
":" FunctionType	→ TypeInfo
{Sort ","}* "→" Sort	→ FunctionType

module *Term*
exports
imports *Name*

exports
sorts Term NonAppl Appl VarType Informal Args

context-free syntax

Variable	→ NonAppl
Args Function Args	→ Appl
Args Function Appl	→ Appl
"(" Variable VarType "|" Term ")"	→ NonAppl
"{" Informal* "}"	→ NonAppl
NonAppl	→ Term
Appl	→ Term
	→ Args
NonAppl	→ Args
"(" {Term ","}* ")"	→ Args
	→ VarType
":" Sort	→ VarType
NaturalLanguage	→ Informal
"'" Name "'"	→ Informal
"'" Term "'"	→ Informal

context-free priorities
"'" Name "'" → Informal >
"'" Term "'" → Informal

module *Main*
imports *Module Priorities*

exports
context-free restrictions
(**LAYOUT**)**?** -/- **[**\␣\t\n\%**]**
Informal -/- **~[**\{\}\'\␣\t\n\%**]**
Variable -/- **[**0-9**]**

lexical restrictions
Special -/- **[**\!\#\$\%\&*\+\-\.\:\∨\;\<\=\>\?\@\^_\‘\|\~**]**

context-free syntax
“REQ” → FunctionId {**reject**}
“AXIOM” → FunctionId {**reject**}
“LEMMA” → FunctionId {**reject**}

BIBLIOGRAPHY

Abrial, J.-R., Börger, E. & Langmaack, H. (eds) (1996). *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, LNCS 1165, Springer, Berlin.

Astesiano, E., Kreowski, H. J. & Krieg-Brückner, B. (eds) (1999). *Algebraic Foundations of Systems Specifications*, Springer, Berlin.

Barringer, H., Cheng, J. H. & Jones, C. B. (1984). A logic covering undefinedness in program proofs, *Acta Informatica* **21**: 251–269.

Barwise, J. (1989). Mathematical proofs of computer system correctness, *Notices of the American Mathematical Society* **36**: 844–851.

Bergstra, J. A., Bethke, I. & Rodenburg, P. H. (1994). A propositional logic with 4 values: true, false, divergent and meaningless, *Technical Report P9406*, Programming Research Group, University of Amsterdam, Amsterdam.

Bergstra, J. A., Heering, J. & Klint, P. (1987). ASF: an algebraic specification formalism, *Technical Report CS-R8705*, Department of Software Technology, Centre for Mathematics and Computer Science, Amsterdam.

Bergstra, J. A., Heering, J. & Klint, P. (eds) (1989). *Algebraic Specification*, ACM Press/Addison-Wesley, New York.

Bird, R. (1998). *Introduction to Functional Programming using Haskell*, second edn, Prentice Hall Europe.

Blum, B. I. (1993). Representing open requirements with a fragment-based specification, *IEEE Transactions on Systems, Man, and Cybernetics* **23**(3): 724–736.

Blum, B. I. (1994). A taxonomy of software development methods, *Communications of the ACM* **37**(11): 82–94.

Booch, G., Rumbaugh, J. & Jacobson, I. (1999). *The Unified Modeling Language User Guide*, Addison-Wesley, Reading.

Bowen, J. P. & Hinchey, M. G. (1994). Seven more myths of formal methods, *Technical Report PRG-TR-7-94*, Oxford University Computing Laboratory, Oxford.

Brookhuis, K. A. & Oude Egbrink, H. J. H. (1992). Proceedings of the first workshop on detection, tutoring & enforcement systems, *Technical Report DETER D1 (500A)*, Commission of the European Community.

Brooks, F. P. (1987). No silver bullet: essence and accidents of software engineering, *IEEE Computer* **20**(4): 10–19.

Cardelli, L. & Wegner, P. (1985). On understanding types, data abstraction, and polymorphism, *ACM Computing Surveys* **17**(4): 471–522.

Cheng, J. H. (1986). *A Logic for Partial Functions*, Ph.D. thesis, University of Manchester, Computer Science Department.

Cheng, J. H. (1990). On the usability of logics which handle partial functions, *Technical Report UMCS-90-3-1*, Department of Computer Science, University of Manchester, Manchester.

Chomsky, N. (1992). A minimalist program for linguistic theory, *Technical report*, MIT. MIT Occasional Papers in Linguistics.

CoFI Task Group on Language Design (1997). CASL - the CoFI algebraic common language - rationale, version 0.97, *Technical report*, CoFI: Common Framework Initiative.

CoFI Task Group on Language Design (1999). CASL, the common algebraic specification language: Summary, *Technical report*, CoFI: Common Framework Initiative. Version 1.0.

de Bunje, A. (1992). User's manual of COLD-1, *Technical Report RWR-508-RE-92106*, Philips Research Laboratories, IST, Eindhoven.

de Geus, A. F. & Rotterdam, E. R. (1992). *Decision Support in Anaesthesia*, PhD thesis, Department of Anesthesiology, University of Groningen.

de Geus, F., Rotterdam, E., van Denneheuvel, S. & van Emde Boas, P. (1991). Physiological modeling using RL, *in* M. Stefanelli, A. Hasman, M. Fieschi & J. Talmon (eds), *Proceedings of AIME '91*, Springer Verlag, Berlin, pp. 198–210.

de Jonge, M. (1999). boxenv.sty: A latex style file for formatting box expressions, *Technical Report SEN-R9911*, CWI, Amsterdam.

de Waard, D., Brookhuis, K. A., van der Hulst, M. & van der Laan, J. D. (1994). Behaviour comparator prototype test in a driving simulator, *Technical Report V2009/DETER/Deliverable 10 (321B) to the Commission of the European Union*, Traffic Research Centre, University of Groningen, Haren.

Diller, A. (1994). *Z: An Introduction to Formal Methods*, second edn, John Wiley & Sons, Chichester.

Fasel, J. H., Hudak, P., Peyton-Jones, S. & Wadler, P. (1992). SIGPLAN notices special issue on the functional programming language Haskell, *ACM SIGPLAN Notices* **27**(5).

Feijs, L. M. G. & Jonkers, H. B. M. (1992). *Formal Specification and Design*, Vol. 35 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press, Cambridge.

Feijs, L. M. G., Jonkers, H. B. M., Koymans, C. P. J. & Renardel de Lavalette, G. (1987). Formal definition of the design language COLD, *Technical Report Technical Note 234/87*, Philips Research Laboratories. Also appeared as ESPRIT Document METEOR/t7/PRLE/7.

Feijs, L. M. G., Jonkers, H. B. M. & Middelburg, C. A. (1994). *Notations for Software Design*, Springer-Verlag, London.

Fensel, D. (1995). *The Knowledge Acquisition and Representation Language KARL*, Kluwer Academic Publisher, Boston.

Floyd, C. (1995). Theory and practice of software development: stages in a debate, *in* P. D. Mosses, N. Nielsen & M. I. Schwartzbach (eds), *TAPSOFT '95: Theory and Practice of Software Development*, Vol. 915 of *Lecture Notes in Computer Science*, Springer, Berlin.

Floyd, C., Züllighoven, H., Budde, R. & Keil-Slawik, R. (eds) (1992). *Software Development and Reality Construction*, Springer-Verlag, Berlin.

Fraser, M. D., Kumar, K. & Vaishnavi, V. K. (1994). Strategies for incorporating formal methods in software development, *Communications of the ACM* **37**(10): 74–85.

Fröhlick, M. & Werner, M. (1995). daVinci V1.4 user manual, *Technical report*, Department of Computer Science, University of Bremen.

Ghezzi, C., Mandrioli, D. & Morzenti, A. (1990). TRIO, a logic language for executable specifications of real-time systems, *The Journal of Systems and Software* **12**(2).

Glass, R. L. (1999). The realities of software technology payoffs, *Communications of the ACM* **42**(2): 74–79.

Goguen, J. A. & Burstall, R. (1992). Institutions: Abstract model theory for specification and programming, *Journal of the ACM* **39**(1): 95–146.

Goguen, J. A. & Winkler, T. (1988). Introducing OBJ3, *Technical Report SRI-CSL-88-9*, Computer Science Laboratory, SRI International.

Goguen, J., Kirchner, C., Kirchner, H., Mégrelis, A. & Meseguer, J. (1988). An introduction to OBJ3, *in* J.-P. Jouannaud & S. Kaplan (eds), *Proceedings of the Workshop on Conditional Term Rewriting Systems (Orsay, France)*, Vol. 308 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 258–263.

Groenboom, R. (1997). *Formalizing Knowledge Domains: Static and Dynamic Aspects*, PhD thesis, University of Groningen.

Groenboom, R. & Renardel de Lavalette, G. R. (1996). FAN: Formalizing ANesthesia, *Technical Report CSN9601*, Department of Computer Science, University of Groningen.

Guttag, J. V. & Horning, J. J. (1993). *Larch: Languages and Tools for Formal Specification*, Texts and Monographs in Computer Science, Springer-Verlag, New York.

Guttag, J. V., Horning, J. J. & Modet, A. (1990). Report on the Larch shared language: Version 2.3, *Technical Report SRC Research Report 58*, Digital Systems Research Center.

Harper, R., McQueen, D. & Milner, R. (1986). Standard ML, *LFCS report series ECS-LFCS-86-2*, Laboratory for Foundations of Computer Science, Dept. of Computer Science, University of Edinburgh.

Hayes, I. J. & Jones, C. B. (1989). Specifications are not (necessarily) executable, *Technical Report UMCS-89-12-1*, Department of Computer Science, University of Manchester, Manchester.

Heering, J., Hendriks, P. R. H., Klint, P. & Rekers, J. (1992). The syntax definition formalism SDF (reference manual).

Heitmeyer, C. & Mandrioli, D. (eds) (1996). *Formal Methods for Real-Time Computing*, John Wiley & Sons.

Ishihara, Y., Seki, H. & Kasami, T. (1992). A translation method from natural language specifications into formal specifications using contextual dependencies, *IEEE International Symposium on Requirements Engineering*, pp. 232–239.

Jirotka, M. & Goguen, J. (eds) (1994). *Requirements Engineering: Social and Technical Issues*, Academic Press, London.

Jones, C. B. (1986). *Systematic Software Development Using VDM*, Prentice-Hall International Series in Computer Science, Prentice-Hall International, Englewood Cliffs.

Klint, P. (1993). A meta-environment for generating programming environments, *ACM Transactions on Software Engineering and Methodology* **2**(2): 176–201.

Lamport, L. & Paulson, L. C. (1999). Should your specification language be typed?

Leith, P. (1990). *Formalism in AI and Computer Science*, Ellis Horwood.

Levine, J. R., Mason, T. & Brown, D. (1992). *Lex & Yacc*, O'Reilly & Associates.

McDermid, J. A. (ed.) (1991). *Software Engineer's Reference Book*, Butterworth Heinemann, Oxford.

Meyer, B. (1988). *Object-Oriented Software Contruction*, Prentice Hall International Series in Computer Science, Prentice Hall, New York.

Moggi, E. (1991). Notions of computation and monads, *Information and Computation* **93**(1): 55–92.

Monahan, B. & Shaw, R. (1991). Model-based specifications, *in* J. A. McDermid (ed.), *Software Engineer's Reference Book*, Butterworth Heinemann, Oxford.

Myers, C., Clack, C. & Poon, E. (1993). *Programming with Standard ML*, Prentice Hall.

Orejas, F. (1999). Structuring and modularity, *in* E. Astesiano, H. J. Kreowski & B. Krieg-Brückner (eds), *Algebraic Foundations of Systems Specifications*, Springer, Berlin.

Rekers, J. (1992). *Parser Generation for Interactive Environments*, PhD thesis, University of Amsterdam.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W. (1991). *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs.

Rushby, J., von Henke, F. & Owre, S. (1991). An introduction to formal specification and verification using EHDM, *Technical report*, Computer Science Laboratory SRI International, Menlo Park.

Saaman, E., Politiek, P. P. & Brookhuis, K. A. (1994). Specification and design of InDETER-1, a prototype driving behaviour assessment system, *Technical Report V2009/DETER/Deliverable 9 (321A) to the Commission of the European Union*, Traffic Research Centre, University of Groningen, Haren.

Sannella, D. & Wirsing, M. (1999). Specification languages, *in* E. Astesiano, H. J. Kreowski & B. Krieg-Brückner (eds), *Algebraic Foundations of Systems Specifications*, Springer, Berlin.

Si Alhir, S. (1998). *UML in a Nutshell: A Desktop Quick Reference*, O'Reilly.

Stabler, E. P. (1992). *The Logical Approach to Syntax*, MIT Press, Cambridge.

Sterling, L. & Shapiro, E. (1994). *The Art of Prolog*, MIT Press.

Tarlecki, A. (1999). Institutions: An abstract framework for formal specifications, *in* E. Astesiano, H. J. Kreowski & B. Krieg-Brückner (eds), *Algebraic Foundations of Systems Specifications*, Springer, Berlin.

The ASF+SDF Meta-environment User's Guide (1995).

van den Brand, M. & de Jonge, M. (1999). Pretty printing within the asf+sdf meta-environment: A generic approach, *Technical Report SEN-R9904*, CWI, Amsterdam.

van Deursen, A. (1994). *Executable Language Definitions: Case Studies and Origin Tracking Techniques*, PhD thesis, Universiteit van Amsterdam, Amsterdam. ILLC Dissertation Series 1994-5.

van Eijk, P. H. J., Vissers, C. A. & Diaz, M. (eds) (1989). *The Formal Description Technique LOTOS*, Elsevier Science Publishers.

van Harmelen, F. & Balder, J. R. (1992). $(ML)^2$: A formal language for KADS models of expertise, *Knowledge Acquisition* **4**(1).

van Leeuwen, J. (ed.) (1990). *Handbook of Theoretical Computer Science Volume B: Formal Models and Semantics*, Elsevier Science Publishers.

Veenstra, M. (1998). *Formalizing the Minimalist Program*, PhD thesis, Rijksuniversiteit Groningen.

Visser, E. (1997). *Syntax Definition for Language Prototyping*, PhD thesis, University of Amsterdam.

Wadler, P. (1992). The essence of functional programming, *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico*, ACM Press, New York, pp. 1–14.

Wadler, P. (1993). Monads and functional programming, *in* M. Broy (ed.), *Proceedings of the 1992 Marktoberdorf international summer school on program design calculi*, Springer Verlag.

Wadler, P. (1995). Monads for functional programming, *Lecture Notes in Computer Science* **925**.

Wirsing, M. (1990). Algebraic specification, *in* J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Vol. B, Elsevier Science Publishers.

INDEX

SAMENVATTING

Dit proefschrift is het verslag van de ontwikkeling van de nieuwe formele specificatietaal AFSL die speciaal bedoeld is voor de beschrijving van software. Behandeld worden de rol van formele specificatie in software-ontwikkeling, de algemene principes van formele specificatietalen, de ontstaansgeschiedenis van AFSL, de redenen om een nieuwe taal te maken, de definitie van de taal, ontwerpoverwegingen, voorbeelden van het gebruik en ideeën voor mogelijke verbeteringen aan de taal. De afkorting AFSL staat voor "Almost Formal Specification Language" waarbij "Almost" refereert naar de mogelijkheid om (niet formele) natuurlijke taal te gebruiken binnen het verder formele AFSL. In de titel "Another Formal Specification Language" refereert "Another" naar het feit dat er al veel formele specificatietalen bestaan (misschien al genoeg), maar het geeft ook aan dat AFSL anders is.

Een specificatie is een beschrijving van de eigenschappen van een ding. Een specificatie is formeel als hij aan bepaalde vormafspraken voldoet zodat de doelgroep de beschrijving begrijpt. Voorbeelden van specificaties zijn de partituur van een muziekstuk, een patroon van een bruidsjurk en het elektronisch schema van een radio. Het is vaak van belang om een specificatie te hebben voordat het ding gemaakt wordt omdat het proces anders in chaos ontaardt. Je kunt een groep bouwvakkers immers geen huis laten bouwen door ze alleen te vertellen wat het gewenste vloeroppervlak en aantal kamers zijn.

De hierboven gegeven voorbeelden hebben allemaal een verschillende mate van formaliteit. De noten in de partituur geven bijvoorbeeld toonhoogte aan maar bepalen niet de klankkleur van de noten. Dat is de reden waarom dirigent en musici zo'n belangrijke rol spelen bij de interpretatie van van een muziekstuk. Het patroon maakt gebruik van speciale symbolen die een geschoolde kleermaker voldoende aanwijzingen geeft om de jurk precies op maat te maken, ook al zegt het nog niks over de kleur stof. Het elektronisch schema tenslotte maakt gebruik van zulke gestandaardiseerde symbolen dat elke elektronicus er een radio mee kan maken.

De notatie die gebruikt wordt voor een specificatie noemen we een specificatietaal. Notenschrift is een specificatietaal, maar ook de symbolen die gebruikt worden in een patroon. Een specificatietaal is formeel als deze zodanig is gestandaardiseerd dat er geen ruimte is voor interpretatieverschillen. Notenschrift is dus niet echt een formele specificatietaal maar de notatie voor elektronische schema's wel.

Hebben we specificaties nodig voor het maken van software? Er bestaat veel ergenis over de kwaliteit van software en de slechte beheersbaarheid van het bijbehorende ontwikkelproces. Er is een aantal redenen waarom het moeilijk is software te maken. Software is meestal te omvangrijk om door één persoon begrepen te worden. Het is moeilijk om vast te stellen aan welke eisen precies voldaan moet voldoen. De onderlinge samenhang tussen verschillende onderdelen maakt het moeilijk om veranderingen aan te brengen. Het maken van software is een denkproces waarbij de redenen om een bepaalde beslissing te nemen niet altijd ex-

pliciet worden gemaakt. Ten slotte is de omgeving waarbinnen de software moet opereren is vaak moeilijk te doorgronden. Bij al deze aspecten is communicatie een belangrijke element, specificaties kunnen daarbij een belangrijke rol spelen. Bijvoorbeeld bij het vastleggen van afspraken, documenteren van ontwerpbeslissingen, beschrijven van de verschillende onderdelen van de software, beschrijven van de omgeving, beschrijving van standaarden, etc. Een bijkomend voordeel van het maken van specificaties is dat het de betrokken partijen dwingt goed na te denken over het op te lossen probleem.

Er zijn verschillende soorten specificatietalen. Het meest voorkomend zijn symbolische talen (waarbij met letters en woorden gewerkt wordt) en grafische talen (waarbij met plaatjes gewerkt wordt). Over het algemeen vatten grafische beschrijvingen slechts deelaspecten van een compleet systeem, zoals bijvoorbeeld de relaties tussen gegevens die verwerkt worden. Als een grafische taal gebruikt wordt voor een volledige specificatie worden de plaatje al gauw te ingewikkeld. Een uitgangpunt van dit proefschrift is het kunnen maken van volledige specificaties, daarom is gekozen voor een symbolische taal. Veel symbolische specificaties van software worden geschreven in natuurlijke taal (Nederlands, Engels, etc.). Het nadeel van natuurlijke taal is dat hij vaak niet goed gebruikt wordt, op verschillende manieren begrepen kan worden en moeilijk automatisch te verwerken is. Een andere symbolische taal die in kleine kring gebruikt wordt, vooral in de wetenschap, is die van de wiskunde. Deze taal is minder dubbelzinnig dan natuurlijke taal en leent zich bovendien bij uitstek voor systematische analyse van software, zoals het correct bewijzen van programma's. Toch is ook de taal van de wiskunde niet gestandaardiseerd en leent zich ook slecht voor automatische verwerking. Daar komt nog eens bij dat hij moeilijk in het gebruik is.

Kortom, voor de specificatie van software is een echte formele taal het meest gewenst. Er bestaat een veelheid aan formele talen. De bekendste van deze talen is de eerste-orde predikatenlogica en gerelateerd daaraan de logische programmeertaal Prolog. In principe kunnen de meeste formele talen gebruikt worden voor de specificatie van software. In de praktijk leidt dat echter tot omslachtige en daardoor onbegrijpelijke specificaties. Bovendien sluiten deze algemene talen slecht aan bij de rest van het software-ontwikkelproces. Daarom bestaan er talen die speciaal gemaakt zijn voor de specificatie van software. Veel van deze talen zijn ontwikkeld om gebruikt te kunnen worden voor het correct bewijzen van programma's, maar daar is AFSL niet voor bedoeld. Als gevolg daarvan ontbreken bijvoorbeeld bewijsregels.

Nu we weten dat specificeren zinvol kan zijn en dat daar bij voorkeur een formele taal voor gebruikt moet worden kunnen we dus aan de slag. Helaas ligt dat niet zo simpel. De huidige formele specificatietalen zijn zo ingewikkeld dat ze een opleiding vergen die een stuk verder gaat dan een programmeeropleiding. Los daarvan is het boven tafel krijgen van specificaties geen eenvoudige taak, daarvoor is veel inzicht en ervaring nodig. Het zou daarom helpen als er een methode was voor het maken van formele specificaties. Deze methode zou moeten bestaan uit een specificatietaal plus aanwijzingen en hulpmiddelen voor het gebruik van deze taal. Zo'n methode bestaat niet, met name ontbreekt het aan concrete aanwijzingen voor het opstellen van specificaties. Dit is de aanleiding geweest voor het starten van een onderzoek met als doel het ontwikkelen van een specificatiemethode. Uitgangspunt vormde daarbij de ideeën uit objectgeorienteerde methoden, aangezien die zich nadrukkelijk richten op het beschrijven van systemen. Onderdeel van het onderzoek was een tweetal praktijkstudies.

Gebaseerd op de wensen voor de specificatiemethode is een aantal eisen opgesteld waar-

aan de te gebruiken specificatietaal moest voldoen. De taal moet eenvoudig zijn zodat hij gemakkelijk te leren en automatisch te verwerken is. Natuurlijke taal moet toegestaan zijn zodat een informele specificatie stapsgewijs omgezet kan worden naar een formele. De taal moet bruikbaar zijn in alle fasen van software-ontwikkeling om vertaalproblemen tussen opeenvolgende fasen te voorkomen. De taal moet algemeen toepasbaar voor willekeurige probleemgebieden zodat methode en deelspecificaties hergebruikt kunnen worden. Objectgeorienteerde concepten zoals klassificatie, inheritance, encapsulation en polymorfisme moeten ondersteund worden. Een modulemechanisme met parameters moet hergebruik ondersteunen. Er bleek geen taal beschikbaar te zijn die in voldoende mate voldeed aan al deze eisen. Daarom is besloten een nieuwe taal te maken, dat is AFSL geworden.

AFSL is gebaseerd op de eerste-orde predikatenlogica, daarnaast hebben de specificatietalen COLD, EHDM en LSL grote invloed gehad. Belangrijke eigenschappen van AFSL zijn: eenvoudige basistaal waarbij alle geavanceerde mogelijkheden terugvertaald kunnen worden naar deze basis; natuurlijke taal is toegestaan in informele definities; functies kunnen impliciet gebruikt worden voor het converteren van functie-argumenten naar de benodigde vorm, dit maakt een vorm van inheritance mogelijk; modulen zijn sjablonen, bij import worden moduleparameters tekstueel vervangen door de opgegeven argumenten; polymorfie is gebaseerd op overloading.

De belangrijkste innovatie van AFSL is echter de manier waarop zogenaamde niet-functionele eigenschappen worden behandeld. De niet-functionele eigenschappen van een operatie zijn verborgen aspecten die vallen buiten het pure functionele input/output-gedrag. Voorbeelden hiervan zijn: afhandeling van foutmeldingen, lezen en schrijven van variabelen en communicatie met andere processen. Niet-functionele eigenschappen worden in AFSL op een vergelijkbare manier behandeld als bij de programmeertaal Haskell (het zogenaamde monads-mechanisme). Uitgangspunt hierbij is dat niet-functionele informatie wordt verpakt in het resultaat van de operatie. Speciale uitpakoperaties zijn nodig om de niet-functionele informatie verder te verwerken. Het nadeel hiervan is dat de verwerking van niet-functionele eigenschappen zichtbaar is binnen de specificatie, waardoor deze soms moeilijk te begrijpen is. AFSL heeft daarom de unieke mogelijkheid om via zogenaamde application redirection de uitpakoperaties impliciet toe te passen, waardoor ze niet meer zichtbaar zijn.

Alhoewel de huidige versie van AFSL heel dicht in de buurt komt van de originele eisen, zijn er toch een aantal verbeteringen mogelijk: type-polymorfisme zal specificaties eenvoudiger maken, met name door de beperking van het aantal module-imports; flexibelere syntax voor functie-applicaties zal formuleringen toestaan die dichter staan bij wat gebruikelijk is binnen een bepaald domein; hogere-orde functies zullen een eenvoudiger behandeling van niet-functionele eigenschappen mogelijk maken; operationele semantiek maakt validatie en implementaie van specificaties eenvoudiger. Al deze verbeteringen vergen echter fundamentele aanpassingen aan AFSL, daarom zijn ze in de huidige versie niet opgenomen.

Het is uiteindelijk niet gelukt om een methode van specificeren te ontwikkelen. Hiervoor zijn twee oorzaken aan te wijzen. Ten eerste kostte het ontwerpen en bestuderen van AFSL veel meer werk dan verwacht. De verschillende aspecten van een taal hebben vaak invloed op elkaar. Zo staan syntax, overloading, impliciete functies en application redirection in verband met elkaar omdat ze allemaal tot ambiguiteiten kunnen leiden. Verandering aan één aspect heeft daardoor vaak gevolgen voor de rest. Daarnaast zijn er weinig richtlijnen en hulpmiddelen voor het ontwikkelen van formele talen. Met name het ontbreken van een algemeen geaccepteerd mechanisme voor het definiëren van de semantiek was een grote handicap. De

tweede moeilijkheid bij het ontwikkelen van de methode was het ontbreken van bruikbare kennis waar op voortgebouwd kon worden. Bestaande objectgeorienteerde methoden bleken weinig concrete aanwijzingen te bevatten voor het het maken van modellen, meestal beperken ze zich tot de uitleg van de gebruikte notatie en voorbeelden van het gebruik. Achteraf is het de vraag of het wel mogelijk is richtlijnen te formuleren voor het maken van specificaties. Daarvoor is het proces nog te onbegrepen, bovendien is het heel moeilijk richtlijnen eenduidig te formuleren. In eerste instantie zullen we ons daarom moeten richten op het ontwikkelen van goede talen en automatische hulpmiddelen.

Ondanks het niet vinden van de heilige graal is het onwikkelen van AFSL geen zinloze onderneming geweest. Een aantal belangrijke aspecten zijn aan de orde gekomen die van belang zijn voor de brede toepasbaarheid van formele specificatie. Als zodanig vormt dit proefschrift een proeftuin voor taalontwerp. De combinatie van taaleigenschappen is hierbij van groot belang, te vaak worden deze in isolatie bestudeerd zonder daarbij aandacht te besteden aan de practische toepasbaarheid binnen een groter geheel.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1996-1

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms.* Faculty of Mathematics and Computer Science, KUN. 1996-2

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation.* Faculty of Mathematics and Computer Science, KUN. 1996-3

M.G.A. Verhoeven. *Parallel Local Search.* Faculty of Mathematics and Computing Science, TUE. 1996-4

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory.* Faculty of Mathematics and Computer Science, KUN. 1996-5

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems.* Faculty of Mathematics and Computing Science, TUE. 1996-6

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance.* Faculty of Mathematics and Computer Science, UvA. 1996-7

H. Doornbos. *Reductivity Arguments and Program Construction.* Faculty of Mathematics and Computing Science, TUE. 1996-8

D. Turi. *Functorial Operational Semantics and its Denotational Dual.* Faculty of Mathematics and Computer Science, VUA. 1996-9

A.M.G. Peeters. *Single-Rail Handshake Circuits.* Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism.* Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference.* Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking.* Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics.* Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth.* Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context.* Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types.* Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics.* Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution.* Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing.* Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing.* Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering.* Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation.* Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller.* Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model.* Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing.* Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes.* Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search.* Faculty of Mathematics and Natural Sciences, Univ. Leiden. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection.* Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Schedulere Optimization in Real-Time Distributed Databases.* Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics.* Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems.* Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods.* Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems.* Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems.* Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules.* Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System.* Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars.* Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Progam Construction.* Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic.* Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms.* Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes.* Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols.* Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor.* Faculty of Mathematics and Computing Science, TUE. 2000-05

D. Bosnacki. *Enhancing State Space Reduction Techniques for Model Checking.* Faculty of Mathematics and Computing Science, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs.* Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection.* Faculty of Mathematics and Computing Science, TUE. 2000-09

E. Saaman. *Another Formal Specification Language.* Faculty of Mathematics and Natural Sciences, RUG. 2000-10