Design, Implementation and Evaluation of a Declarative Object-Oriented Programming Language

Adolfo J. Socorro Ramos
Wolfson College

A thesis submitted in partial fulfilment of the
requirements for the degree of Doctor of Philosophy
at the University of Oxford, Trinity Term, 1993

Abstract

This thesis is a detailed study of FOOPS, a "wide spectrum" object-oriented programming language. FOOPS supports all of the classical features of the object paradigm, including classes, overloading, polymorphism, and multiple class inheritance with overriding and dynamic binding. However, it goes beyond other object-oriented languages in its facilities for the specification, composition and reuse of modules. FOOPS is patterned after OBJ, a functional programming language, and from which it derives several of these facilities.

The type system of FOOPS distinguishes between **sorts**, which are collections of **values** (immutable entities), and **classes**, which are collections of **objects** (mutable entities). Moreover, both of these are different from modules, which may declare together several related sorts and classes. Inheritance exists for all of these. Sort and class inheritance concern the hierarchical classification of values and objects; module inheritance supports code reuse by importation.

Theories are special kinds of modules that serve to classify other theories and modules by the syntactic and semantic properties that they satisfy; they are mostly used to constrain the actual arguments to parameterised modules. So-called **views** are bindings that express how a theory is satisfied by another theory or module, allowing many-many relationships between them.

FOOPS is declarative in that it uses **axioms** to define the properties of functions, attributes and methods. Also, there is a formal semantics given by a deduction system, which can be used to prove properties of FOOPS programs.

FOOPS supports design in the same framework as specification and coding. Designs are given as **module expressions**, and when they consist of **executable modules**, can be composed to produce rapid prototypes. Module expressions can describe both **vertical structure**, which relates to implementation layers, and **horizontal structure**, which concerns module aggregation and specialisation. Furthermore, **built-in modules** can be used to interface other languages, and can also appear in module expressions. Finally, views can be employed to capture relationships of refinement and evolution of system designs.

This thesis considers the design of FOOPS, explaining all its features and examining their application to the design and development of object-oriented systems. Also, it describes a prototype implementation of FOOPS that was built using facilities given by the implementation of OBJ3, and which supports most features. Moreover, this thesis performs an in-depth evaluation of FOOPS that focuses on large-grain issues such as the distinction between classes and modules, module instantiation with views, vertical and horizontal structuring, and integrated support for specification and prototyping; comparisons with many other languages are given. Additionally, this thesis presents a detailed summary of current work towards a mathematical semantics for FOOPS, including order-sorted algebra, hidden order-sorted algebra, and the theory of institutions. Examples motivate our discussions throughout, and an appendix expands some of those that appear in the body of the thesis.

Acknowledgements

If I have seen further it is because I stand on the shoulders of giants.

— Isaac Newton

I first wisb to thank my supervisor, Professor Joseph Goguen, whose work is the basis for this thesis, and who provided careful advice and direction to the rcsearch that is reported here. I am also very gratefnl for his continuous efforts to help me concentrate on fundamental aspects, and for stimulating my interest in being a better writer.

My fellow students and research officers in the Declarative Languages Group gave me a forum to express not only ideas. but also doubts. I am particularly indebted to Antonio Alencar, Paulo Borba, Jason Brown, Răzvan Diaconescu, Hendrik Hilberdink, Tom Kemp, Grant Malcolm and Lucia Rapanotti. The Category Theory Study Group helped me understand mathematical foundations: José Barros, Jason Brown, Lutz Hamel, and especially Antonio Alencar.

I was fortunate for the kindness of those who read drafts of this thesis and provided valnable suggestions on how to improve it. My examiners, Professor Hans-Dieter Ehrich and Bernard Sufrin, noted several aspects that needed clarification and uncovered typographical errors. Augnsto Sampaio contributed extensive comments on several chapters, and Ignacio Trejos-Zelaya carefully read some chapters. Paulo Borba and Răzvan Diaconescu also deserve my gratitude.

I have had the pleasnre to correspond with people external to Oxford who have helped me understand wbat I was trying to do. I wish to thank Professor Don Batory, of the University of Texas, and Professor Ehrich and his group at the Technische Universität Braunschweig, Germany. Also, my supervisor at the University of Massachusetts, Professor David Stemple, is responsible for much of my earlier interest in Computing Science and encouraged me to come to Oxford.

I am very thankful to Tracy Fuzzard of Wolfson College who so willingly took care of much administrative drudgery.

In Oxford I met tbree people whom I now consider family, and they are responsible for much of my residual sanity. They always had time to listen, to help, to laugh, and to worry, not to mention cook (the little one also screamed and cried). They are among the kindest people I have ever met. *Claudia, Augusto e Gabi, eu vou ter muitas saudades de vocês.*

Other friends in Oxford have made it all more bearable. My sincere thanks go to Paulo Borba and Ignacio Trejos-Zelaya, and also to Janelle and Luis Montaner.

My friend Frank shared the diffieulties of being a PhD student, and he always had an ear to lend and good advice to give from this vantage point. *Thank you, Frank.*

A large amount of my everyday inspiration to work was due to the encouragement and love of my family and friends at home, and to the thought that one of the rewards of

finishing :his thesis would be our reunion. Even though I was not always able to explain exactly what I was doing, why I had come so far to do it, or why it was taking so long, they firmly stood by me.

My mother has lived up to that title like no other person I know, and her unfailing support in both the peaks and the slumps has been vital. This achievement is as much hers as it is mine. *Gracias Mami.*

My grandmother supplied weekly cheer to work hard and concentrate, always in the characteristic style with which she loves. *Gracias Mima.*

My aunt Saída has also seen me through some tough times and I have been blessed with receiving much of the affection that she so happily gives to all. *Gracias titi Saída.*

Some other aunts and uncles have always believed in me. I am grateful to tío Chiqui, tío Junior, tío Víctor, tío Arturo and titi Milagros, and tío Meco and titi Zulma. Two other relatives whom I loved dearly are no longer with us, but their memory has been a constant source of inspiration: Abita and titi Bayé remain in my heart.

A close friend of our family has unceasingly been a supporter of mine, and I owe her much gratitude. *Gracias Miriam.*

My friends at home always receive me with open arms, and fortunately our closeness has not withered across the miles. Gossie and Posky always had time to put pen to paper; Chino has been a great ally; and Héctor, Javi, Juan Carlos and Luis have been remarkable. *Gracias hermanos.*

Two more-recent friends have also been heartening: Don William and Doña Miriam.

Last on this list of relations, but by no means least, my girlfriend Tita has given me love, support, encouragement, aud much needed hugs and kisses. *Linda, we beat all the odds. Nuestro momento llega.*

Finally, I wish to gratefully acknowledge the followiug for their financial support throughout my studies: Administración de Fomento Económico de Puerto Rico; Committee of Vice-Chancellors aud Principals, United Kingdom; National Science Foundation, U.S.A.; Science and Engineering Research Council, United Kingdom; Wolfson College, Oxford; and Mami, titi Saída and Miriam.

This thesis is dedicated to my mother.

Deep roots are not reached by the frost.
— Bilbo Baggins

Contents

Chapter 1

Introduction

Complexity [...] seems to be an essential property of all large software systems. By essential we mean that we may master this complexity, but we can never make it go away.

— Grady Booch

The object paradigm advocates the design and development of systems as structured collections of objects that communicate with each other, have local storage, and persist and evolve with time. The paradigm represents an evolution and maturation of ideas in programming languages, data abstraction, modularity, communication and hierarchical system organisation. Because system structure is based on the entities being modelled, it is claimed that the use of object-oriented techniques leads to systems that are easier to maintain and, furthermore, that the resulting software is more reusable. The first claim has to do with the observation that what changes most in a system are its functions, not the entities it manipulates; the reusability claim is rooted in the belief that object descriptions often transcend particular applications. These two promises have aroused much excitement in the computer industry, which has had difficulty coping with software systems of increasing size and complexity. In addition, the continuously diminishing cost of hardware, together with spectacular advances in machine capacity and performance, has placed software issues on centre stage.

Objects are also natural units of concurrency and distribution, because their local storage and communication aspects directly reflect the dispersal of entities in a system and of memory in computers. Moreover, objects are a unifying concept in Computing Science; they arise in programming languages, databases, knowledge-representation systems, graphical user interfaces, machine architectures and in many other places. Finally, the basic intuition of software artifacts that correspond to the entities of the application domain appeals to both software engineers and individuals from other disciplines, and this should improve their ability to communicate.

The purpose of this thesis is to discuss the design, prototype implementation and prag-

matics of FOOPS, a wide-spectrum object-oriented language[1]. FOOPS provides abstract data types, objects, classes, overloading, polymorphism, inheritance with overriding, and many additional facilities that go beyond what other current-generation object-oriented languages offer, including parameterised modules with semantic interface requirements, a module interconnection language that can be used to compose modules both vertically and horizontally, and "mixfix" syntax for functions, attributes and methods (Section 1.4 explains all these features). FOOPS is patterned after OBJ [53], a functional specification language, from which it derives several of these facilities; in fact, FOOPS retains OBJ as a sublanguage.

FOOPS was first described by Goguen and Meseguer in [48]. On the one hand, it was recognised that functional programming offered declarativeness and simplicity of language design, but that its lack of a notion of state made very unnatural the specification and implementation of networks and database systems, for example. On the other hand, object-oriented programming provided state but lacked declarativeness and formal foundations. Thus FOOPS used abstract data types as its foundation, with a formal semantics based on algebra and category theory. Much influence came from earlier work by the same authors on order-sorted algebra [49], a logic of inheritance, and on abstract machines [47]; a more recent development by Goguen is hidden-sorted algebra, which formally captures basic intuitions about encapsulation and information hiding [39, 43]. Furthermore, FOOPS can be seen as bringing state-of-the-art module technology to object orientation.

In the sections that follow we discuss the object-oriented paradigm in more detail, and also trace its early influences. We then discuss the reuse of software, followed by an overview of the characteristic aspects of FOOPS. To conclude, we examine the contributions of this thesis and provide an outline of the chapters to come.

1.1 Object-oriented Systems

Each object in an object-oriented system has a unique **identifier** and belongs to a **class**, where each class is associated with a set of attributes and a set of methods. **Attributes** access parts of the local storage of the objects of the class, while **methods** are the operations that can change the state of objects. A form of information hiding is built into this paradigm, in that an object can only be updated through the methods associated with its class. This style of interaction between objects is called **message passing**, but it is usually no more than procedure invocation. There is also **class inheritance**, which refers to the hierarchical organisation of classes to reflect that objects can simultaneously belong to several classes (e.g., at the same time, a person may be a tennis player, a teacher and a mother). When a class `B` inherits from a class `A`, we say that `B` is a **subclass** of `A` and that `A` is a **superclass** of `B`; moreover, inheritance is a transitive relationship, so that if `C` is a subclass of `B`, then it is also a subclass of `A`. **Multiple inheritance** allows a class to have more than one immediate superclass. A class whose attributes and methods refer to objects of other classes is said

[1] The acronym stands for Functional and Object-Oriented Programming System, although we generally use it to refer to the language that it supports.

to be their **client**. For example, a class `Queue` would be a client of the class of the objects that queues store.

Class inheritance supports a form of reuse, because the attributes and methods associated with a class include those of its superclasses. This gives rise to so-called **subclass polymorphism**, which allows an object of some class to be placed wherever an object of any of its superclasses is expected. For example, if `SquareWindow` is a subclass of `RectangularWindow`, any software that manipulates rectangular windows can also manipulate square windows, *without any mediating conversions or "case" statements*. Furthermore, a subclass may **redefine** some of the methods it inherits; this may be desirable because the nature of the class allows them to be more efficiently implemented, or because new attributes need to be updated. For example, `SquareWindow` may reimplement the method `perimeter` using the fact that squares have four sides of equal length. Then, an application of method `perimeter` to an object bound to some variable `X` of class `RectangularWindow` will be resolved *at run-time*, based on the exact class of the object that `X` refers to: if the object is of class `RectangularWindow`, then the original perimeter method will be chosen; if the object is of class `SquareWindow`, then the new perimeter method will be selected. This mechanism is called **dynamic binding**.

Subclass polymorphism and dynamic binding allow for variation in data structures and algorithms in a way such that software is automatically accommodated to deal with objects of classes derived from existing ones. Said differently, these two mechanisms permit a reconciliation between reusability and extendibility [77], and their combination is one of the salient features of this paradigm.

Lastly, we note that there exist several variations on the theme, such as the replacement of class inheritance by various forms of inheritance at the level of objects (for example, see [69, 115]). Other variations occur in languages which support concurrency.

1.1.1 Origins and Current Research

Object-orientation grew out of Simula-67 [87], a discrete-event simulation language developed in Norway by Nygaard and Dahl as an extension of Algol-60. Its emphasis on simulation gave rise to the general idea that software artifacts should directly reflect the entities being modelled. Simula-67 (or Simula, for short) included classes, objects with local state, dynamic object creation, single inheritance, and redefinition and dynamic binding. The language also introduced garbage collection and heap allocation. Clearly, Simula was ahead of its time.

In the late 1960s and early 1970s, the work of Alan Kay and his colleagues at Xerox's Learning Research Group in California led to the Smalltalk system, which is credited with much of the early popularity of object-orientation. Smalltalk has never been just a language, but a revolutionary single-user environment. Besides the Simula influence, Smalltalk is a synthesis of ideas from algebra (sets with associated operations), biology (encapsulated cells that communicate with each other), operating systems (capabilities), Lisp (simplicity and declarativeness) and interactive, graphical user interfaces (including the use of the mouse, pens, and pop-up menus) [26, 63]. Kay's principal objective was personal computing, and

several prototype machines that ran Smalltalk were then built at Xerox, and they included the user-interface facilities just mentioned.

The late 1970's saw the advent of "C with classes," which evolved into C++ [112], and of the Smalltalk-76 and Smalltalk-80 systems [56] (there was also Smalltalk-72). C++ and Smalltalk-80 are currently the most widely used object-oriented languages.

The 1980s and early 1990s have experienced an explosion in the amount of research dedicated to object-oriented systems. There is much interest in linguistic, implementation and environment issues, and several design and analysis notations have been proposed (for example, see [8, 20, 116]). Also, there is a growing community of theoreticians trying to explain the formal semantics of object-orientation.

1.2 Object-oriented Design

Object-oriented design focuses on the data components—the objects—and not on algorithmic abstractions or on the functions a system is supposed to perform. There exist various reasons for this. First, a successful system is soon asked to support further functionality [11, 24]; therefore, if its design was based on functions performed it will probably require major structural changes. Also, changes in functionality are more common than changes in the kind of objects a system manipulates. In the words of Coad and Yourdon [20, page 29]:

> The most stable aspects of a system are the classes and objects which strictly depict the problem domain and the system's responsibility within that domain. Whether one specifies a very low budget or a very sophisticated system, one will still have the same basic classes and objects with which to organise the analysis and ultimately the specification. A more expensive system might have: more attributes for certain objects; more sophisticated services (methods); and, additional classes and objects. Yet the basic classes and objects in the problem domain will remain the same.

Meyer [77] gives the following examples to illustrate the situation. Initially, a compiler is developed to translate one language into another. But later on, it might be modified to provide pretty printing, to support a schematic editor, to gather statistics on common syntactic errors, to generate different code, and so on. Even though the tasks performed by the compiler might have changed quite dramatically, the kinds of data it manipulates will remain more or less intact: tokens, grammar, syntax trees, etc. Another familiar example is payroll systems, which are soon called upon to gather employee statistics, compute taxes and employee benefits, and to interact with various media, while the data items, such as persons, cheques, and time cards, remain undisturbed. Thus, object-oriented design attempts to provide a system with "stable intermediate forms" [8] that will make it more resilient to changes in requirements. Furthermore, because the "user's view of reality" [60] is embodied in the system's structure, object-oriented design is closer to a true design method, which must be repeatable, teachable, and reliable [6].

This style of software development is in a sense the opposite of the functional decomposition method, a top-down, sequeutial approach in which one is forced to make (and freeze) the most important design decisions at the very beginning, exactly when the problem is least understood [77]. This premature hinding of affairs tends to create inflexible architectures, and ignores issues of data abstraction and information hiding [8]. Furthermore, functional decomposition is unpredictable and variable, as it is often unclear whether the decomposition should be with respect to time order, data flow, access to common resources, control flow, etc. [6]; such accidental dependencies also tend to hamper reuse [90]. Jackson [60, page 370] summarises some of these deficiencies as follows:

> Top-down is a reasonahle way of describiug things which are already fully understood [...] When the developer of a system already has a clear idea of the completed result in his mind, he can use top-down to describe on paper what is iu his mind. [But] the method of description is confused with the method of development. A method of development must allow the designer to solve problems to which he does uot already know the solution. Top-down development compels the developer to make the largest and most far-reaching decisions at the beginning.

Of course, at some point one must establish some sequential constraints; the differeuce is that with object-orientation they are not a primary concern. On the positive side, we note that functional decomposition is a well-understood and generally applicable method that has been widely used.

Booch [8] summarises object-oriented development as an iterative and incremental process involving the following steps:

(a) *Identify the classes and objects at a given level of abstraction.* According to Stroustrup [111], classes exist at three levels:

  (1) **application**, which includes classes for user-level concepts such as cars, and for generalisations of these, such as vehicles.

  (2) **machine**, which iucludes classes that model hardware resources (e.g., memory), and system resources (e.g., input-output facilities).

  (3) **implementation**, which includes classes for data structures such as lists.

  (This separation is not strict, but serves to illustrate the overall situation.)

(b) *Identify the semantics of these classes and objects.* This involves the association of attributes and methods with classes.

(c) *Identify the relationships between these classes and objects.* This involves the declaration of inheritance relationships, and the design of interfaces between the objects of the various classes; this step is the hardest oue.

(d) *Implement these classes and objects.* This entails decisions of large-grain structure and organisation, such as gathering related classes into modules.

We wish to note that while object-oriented design and programming may not constitute Brooks' "silver bullet" [11], it is widely acknowledged that the object paradigm advances the art of software engineering. There is no denying that achieving good designs requires sound management, experience, good taste, intelligence, and perhaps even luck [84]. Object-orientation is an attempt to provide better tools for the sorcerers and sorceresses.

1.3 Software Reuse

"The most radical possible solution for software is not to construct it at all." With these words, Brooks [11, page 16] summarised his view about the complexity of building software systems, adding his voice to the increasing number of computer professionals lobbying for more reuse in software construction. Object-orientation has brought renewed interest, because objects and classes are natural units of reuse[2].

The vision of an off-the-shelf components industry that would support the building of new software systems based on standardised components is due to McIlroy and dates back to 1968 [68]. He also argued for libraries with components that could be customised to fit particular needs. Twenty-five years later, McIlroy's vision still remains largely unrealised [25, 68, 77].

Besides complexity, there are other natural and economic forces that motivate reuse. First, according to Standish [109] (who cites a study by Boehm [7]), the cost of software increases exponentially with its size. Therefore, the presence of a components industry in which buying a component is cheaper than building it from scratch would reduce software costs and allow more sophisticated (i.e., larger) systems to be built[3]. Reuse could cut down development time, including testing, and could have a positive impact on maintenance. Additionally, there are commercial benefits associated with reuse. The most immediate one is that ideas would reach their realisation in software more rapidly. Now that systems are also being built for innovative and strategic reasons [31], fast delivery is crucial. (This scenario assumes that a purchased component is reliable, at least in the sense of having satisfied some form of rigorous testing.)

Second, companies could use pre-fabricated components to build rapid (and perhaps even multiple) prototypes to not only advertise ideas, but also to help capture requirements more effectively [11, 31].

Lastly, the astonishing advances in hardware have made it clear that software is the major bottleneck.

Today, most successful reuse occurs at the level of subroutines [25, 30]. Examples include routines from mathematical libraries (most written in FORTRAN), EMACS functions, and UNIX shell utilities (which can be easily glued together with "pipes"). Another popular technique, albeit much less structured, is called "code scavenging" [68]; it refers to program-

[2]However, this thesis will argue that *modules*, which may declare several related classes, are more appropriate units of reuse

[3]Standish also notes that Boehm's study predicts that in the coming years there will not be enough programmers to satisfy the demand for software—unless a more efficient approach to building systems is adopted.

mers picking out arbitrary pieces of old software and patching them onto their code. This is a tedious and difficult activity, especially with regards to searching through old software, understanding the context of the extracted bits, modifying them, and then debugging the resulting whole.

The kind of reuse being advocated at present involves not only *logical* fragments of code, hut also logical fragments of designs and documentation [30, 38, 50, 68]. Additionally, there is a need for flexible reconfiguration tools that will allow components to be adapted before being reused; rapid and multiple prototypes would particularly gain from such facilities.

Apart from these linguistic issues, much difficulty lies in that developing reusable components is as much an art as software engineering is; components should be neither too specific nor too general. The practice requires a commitment for building reusable parts [111], and also some way of classifying the components built so that they can later on be found by others [94]. Recent success stories seem to indicate that reuse occurs most profitably within narrow domains, such as database and network software [4, 19]. A further difficulty is due to the lack of standard interfaces for software components; in hardware design and in other disciplines, reuse is commonplace because of standardisation [25, 103].

1.4 Aspects of FOOPS

The type system of FOOPS introduced two important distinctions. First, data elements are not objects. **Data elements** are stateless and thus cannot change; examples include the natural numbers and the colours. **Objects** have an internal state and persist and evolve with time; examples include gardens and video screens. When these two concepts are merged, as they are in many object-oriented languages, much confusion arises because it is then possible, for example, to "send a message" to the colour blue so that it "adds a shade of yellow" to itself. Clearly, there is no such thing: blue will always be blue. However, it is possible to have **functions** that accept data elements as arguments; for instance, one that gives as result the colour arising from the combination of two other colours. Consequently, in FOOPS data elements are collected into **sorts** and objects are collected into **classes**. Following the ADJ tradition [51] and the work of Goguen and Meseguer, an **abstract data type** in FOOPS is a sort together with its set of associated functions. Also, an **abstract object type** is a class together with its associated attributes and methods.

Second, classes are not modules. The modules of FOOPS may declare several related classes together, and constitute its main programming unit. By contrast, most other object-oriented languages take as their main programming unit a syntactic construction for the definition of a single class with its associated attributes and methods. While this simplifies language design, it is nevertheless a step backwards from the advances introduced by languages such as Ada [59], Modula [118], OBJ and many others in supporting the more general construction.

Given the disjunction between sorts and classes in FOOPS, we say that it has a **functional level** and an **object level**. Abstract data types exist at the functional level; abstract object types exist at the object level. At each level there are two kinds of module, one which

encapsulates executable code and the other which declares properties. The former are simply called modules, or **functional modules** and **object modules** when their level needs to be made explicit; the latter are called **theories**, or **functional theories** and **object theories** when there is a need to make their level clear.

Theories serve to classify other theories and modules by the syntactic and semantic properties that they satisfy, and are mostly used in FOOPS to constrain the actual arguments to parameterised modules (see below). They constitute the purely declarative aspect of FOOPS. So-called **views** are bindings that express how a theory is satisfied by another theory or module. This allows the capture of the rather common case in which a theory is satisfied by more than one module, or in which a particular module satisfies a theory in more than one way. For example, the natural numbers form a partially-ordered set under either the less-than or the greater-than relations; also, the strings over the Roman alphabet form a partially-ordered set under the usual lexicographical ordering.

FOOPS also offers (and distinguishes) inheritance for sorts, classes and modules. Inheritance of sorts and classes has to do with the hierarchical classification of data elements and objects. For example, the sort `Nat` of natural numbers inherits, or is a **subsort** of, the sort `Int` of integers, because all natural numbers are also integers; similarly, the class `LandVehicle` *is a* **subclass** of the class `Vehicle`. These two kinds of inheritance have a set-inclusion semantics given by order-sorted and hidden order-sorted algebra, respectively [39, 43, 49]. Module inheritance supports code reuse by importation. This other kind of inheritance allows old sorts and classes to be imported and enriched with operations derived from the ones originally associated with them. For example, a module that defines trigonometric functions may be defined by extending a pre-existing module for floating-point numbers with declarations for sine, cosine, etc. Or a generic module that declares iteration methods over special kinds of data structures may simply define the new methods as combinations of already existing methods on these data structures. Note that these examples do not use either class inheritance or clientship, and thus could not be done using the features that are typically available in languages that identify classes and modules. The semantics of module inheritance is based on category theory [41].

Both modules and theories use **equations**, or **axioms**, to define the properties of functions, attributes and methods. FOOPS is a logical language in the sense that its formal semantics defines a deduction system which can be used to derive new axioms from old ones; said differently, its deduction system can be used to prove properties about FOOPS programs. (Much more details about this are given in Chapter 4.)

When axioms have a particular form, they can be regarded as executable code; thus, one syntactic difference between modules and theories is that axioms declared in modules are required to have this form. For functional modules, these axioms are interpreted as left-to-right rewrite rules in the classical term-rewriting sense [67], except that rewriting takes into account sort orderings or hierarchies, and is called **order-sorted term rewriting** [44]. For object modules, axioms are in general considered from left to right, but as descriptions of updates to an implicit object database [48]. In FOOPS, however, the notion of "main program" is absent. Rather, computations are started by supplying **terms**, or expressions,

to the top level of the system, which evaluates them with respect to the axioms in a given module.

1.4.1 Parameterised Programming

While much work in object-orientation has concentrated on issues such as low-level type systems, not enough attention has been devoted to the study of system-level phenomena, such as overall structure, large-grain properties, sub-component compatibility, variants and configurations [28, 85]. FOOPS addresses these concerns.

FOOPS is equipped with facilities for composing modules, including renaming, sum, parameterisation, instantiation and importation. These constitute **parameterised programming** [36], which can be seen as functional programming with modules as values, theories as types, and module expressions as (functional) programs. **Renaming** allows the sorts, classes, attributes and methods of modules to get new names, while **sum** is a kind of parallel composition of modules that takes account of sharing. The interfaces of parameterised modules are defined by **theories**. **Instantiation** is specified by a view from an interface theory to an actual module, describing a binding of parts in the theory to parts in the actual module; **default views** can be used to give "obvious" bindings. **Importation** allows multiple inheritance at the module level. Parameterised programming was first implemented in OBJ [53], has a rigorous semantics based on category theory [29, 39, 41], and is a development of ideas in the Clear specification language [14]. Much of the power of parameterised programming comes from treating theories and views as first class citizens. For example, it can provide a higher order capability in a first order setting [37, 50].

A major advantage of parameterised programming is its support for *design* in the same framework as specification and coding [50]. Designs are expressed as **module expressions**, and they can be executed symbolically if specifications having a suitable form are available. This gives a convenient form of prototyping. An interesting feature of the approach we advocate is its distinction between horizontal and vertical structuring, genericity and compositionality. **Vertical structure** relates to layers of abstraction, where lower layers implement or support higher layers. **Horizontal structure** is concerned with module aggregation, enrichment and specialisation. Both kinds of structure can appear in module expressions, and both are evaluated when a module expression is evaluated. There is also support for rather efficient prototyping through **built-in** modules, which can be composed just like other modules, and give a way to combine symbolic execution with access to an underlying implementation language.

Parameterised programming is considerably more general than the module systems of languages like Ada, CLU [71] and Modula-3 [83], which provide only limited support for module composition. For example, interfaces can only express purely syntactic restrictions on actual arguments, cannot be horizontally structured, and cannot be reused. But in parameterised programming, theories are modules which can be generic and can be combined using instantiation, sum, renaming, and importation. Recent work of Batory [3, 104] shares many of our concerns, and in particular distinguishes between components and "realm interfaces," which correspond to theories in parameterised programming, although without

any semantic constraints. Batory's approach is primarily based on vertical parameterisation, although a limited form a horizontal parameterisation allows constants and types, without any horizontal composition. Another difference is that FOOPS allows non-trivial views, whereas Batory's approach only has (implicit) default views. Related work has also been done by Tracz [114], whose LILEANNA system implements the horizontal and vertical composition ideas of LIL [35] for the Ada language, using ANNA [73] as its specification language.

1.5 Contributions of this Thesis

As its title suggests, the contributions of this thesis lie mainly in three areas: design, implementation and evaluation of FOOPS. Because we build upon previous research by others, we here give details of what distingnishes our work.

Our starting point was [48], the first publication on FOOPS, together with the work that led to it, particularly [36] and [49]. We began by providing more precise definitions for the language features proposed in [48], including object destruction, the interpretation of groups of axioms that define methods, redefinition and dynamic binding, and class and sort inheritance conflicts and their resolution.

Some other features of the language were extended. For example, object creation is now much more flexible (e.g., identifiers need not always be explicitly given), attributes can be "derived" (i.e., defined in terms of others) and can have multiple arguments, and a method can return any result, not jnst the same object it modifies.

Furthermore, some aspects of the langnage are new or have been reworked from earlier proposals. For example, we defined the encapsnlation rules of the language and designed its information hiding mechanism, except for vertical parameterisation, which was added to FOOPS in joint work with Joseph Goguen [50], nsing ideas from LIL [35]. Module blocks had been proposed earlier [55], but we give a more detailed design. Also, abstract classes are a new feature that we helped to develop. (These last two aspects are also reported in [50].)

Our prototype implementation of FOOPS builds upon facilities given by the OBJ3 implementation [53]. Early design help was provided by Goguen and by Timothy Winkler of SRI International, California. This prototype implemeutation has served as the basis for simulating a concurrent version of FOOPS [9], and to support the implementation of OOZE [1].

This thesis also evalnates FOOPS and compares it with other languages. In addition, it provides extended discnssions on the applicability and benefits of parameterised programming (as defined here) for the design and implementation of object-oriented systems. We have focused on large-grain issues such as module reuse and composition, and have found added leverage in several of the aspects mentioned in the previons section, snch as: the distinction between classes and modules, including the different kinds of inheritance; module instantiation with views; vertical module parameterisation, which allows fine-tuning the implementation of modules by supplying different vertical parameters; and integrated sup-

port for specification and prototyping, including the use of views to express refinement and evolution relationships between systems defined by module expressions. Some of this work was inspired by a paper by Goguen and Wolfram [54]; we offer a more comprehensive and in-depth analysis, and include in it the aspects of FOOPS that are new with this thesis. Also, the comparisons with other languages is new (a small part of it was added to [50]).

Moreover, we feel that we will contribute to the understanding of the semantics of FOOPS by threading together several separate publications on the subject, from the perspective of someone who is not a theoretician.

Overall, one of the main points of this thesis is that parameterised programming clarifies and enriches several aspects of the object paradigm. Another main point is the use of semantic foundations to explicate, propose and analyse features and applications of FOOPS.

1.6 Overview of Subsequent Chapters

The rest of this thesis is organised as follows:

Chapter 2 discusses the form and informal meaning of the declarations that modules encapsulate, such as those for sorts, classes, inheritance relationships, axioms, and so on; it also describes object creation and destruction. This chapter deals exclusively with executable modules, although most details carry over to theories.

Chapter 3 explains facilities for composing modules and designing systems, including theories, views, parameterisation and module expressions. It also examines abstract classes, module importation, module blocks and information hiding capabilities.

Chapter 4 summarises current work towards a mathematical semantics for FOOPS, including order-sorted algebra, hidden order-sorted algebra, and the theory of institutions.

Chapter 5 describes a prototype implementation of FOOPS. It supports a majority of the features discussed in this thesis, and gives ideas on how to implement some which are not currently available.

Chapter 6 is an evaluation of FOOPS carried out by comparing several of its facilities to those present in other object-oriented languages, with particular emphasis on constructs for programming-in-the-large. Among other aspects, it treats the distinctions between modules and classes, and examines renaming, parameterised modules and information hiding capabilities. Around fifteen languages are considered, including the major ones in use today.

Chapter 7 concludes this thesis with a summary of what was achieved and an outline of areas that remain unexplored.

Appendix A gives the full syntax of FOOPS.

Appendix B provides additional examples.

Chapter 2

Modules

Object orientation comes to full fruition only when combined with modularity and strict typing of data.

— Niklaus Wirth

The main programming unit of FOOPS is the module, which encapsulates executable code. This chapter describes the form and informal meaning of module declarations, including sorts, classes, functions, attributes, methods and axioms. Also, we discuss the models of computation at the functional and object levels of FOOPS. This presentation is based on Sections 2 and 3 of [95], the reference manual for the language and our prototype implementation. The discussion of abstract classes, encapsulation rules, and information hiding facilities in FOOPS is delayed until the next chapter, as they can only be fully understood once module inheritance is explained. Lastly, the functional sublanguage of FOOPS is a syntactic variant of OBJ3, and therefore we do not attempt to cover it in as much detail as other documents (e.g., [53]); Appendix A gives the exact syntactic correspondence between the two. Our main concern is with the object level of FOOPS.

Syntactic descriptions are presented incrementally in **syntax boxes**, which have the form

Syntax *NN* (*Name*)
text
□

where *NN* is the box number, *Name* is the name of the syntactic unit being described and *text* gives the formal syntax of the unit (and often some English commentary). Syntax is described in the following extended BNF notation: the symbols { and } are used as meta-parentheses; the symbol | is used to separate alternatives; [and] pairs enclose optional syntax; ⟨*NonT*⟩ is a non-terminal symbol; ... indicates 0 or more repetitions of the preceding unit; and **x** denotes x literally. As an application of this notation,

A {, **A**} ...

is an idiom used for non-empty lists of A's separated by commas.

2.1 Functional-level Modules

A functional module defines one or more abstract data types, which consist of a set of data elements and operations on them. Data elements (also called "values") are stateless entities, such as the numbers and the colours. A set of data elements is called a **sort**, while operations on data elements are called **functions**. Sorts may be placed in a partial order, interpreted as subset inclusion. This order defines an inheritance hierarchy among sorts, in which a sort `A` inherits from a sort `B` if `A` < `B`; in this case we say that `A` is a **subsort** of `B`. A **signature** is a group of sort, subsort and function declarations, and an **algebra** is a partially-ordered collection of sorts together with interpretations (or definitions) for each function symbol. A **term** at the functional level is an expression that is built up from function symbols and from variables denoting data elements. The model of computation at this level is **order-sorted term rewriting**, which regards the axioms that define the properties of functions as left-to-right rewrite rules. (Chapter 4 gives formal definitions for all these concepts; informal definitions will suffice for the purposes of this chapter.) When not parameterised, functional modules have this form:

Syntax 2.1 (Unparameterised Functional Modules)

```
fmod ⟨ModId⟩ is
  ⟨fModElt⟩ ...
endf
```

where ⟨*ModId*⟩ stands for the name of the module, by convention given in upper case letters. ⟨*fModElt*⟩ stands for the things that may be declared by a functional module, namely sorts, functions, variables, and axioms. Functional modules may also import other functional modules. □

2.1.1 Sorts and Subsorts

Sort declarations have the following syntax:

Syntax 2.2 (Sorts)

```
sorts ⟨SortIdList⟩ .
```

where ⟨*SortIdList*⟩ is a non-empty list of sort names separated by blanks; by convention, sort names are capitalised. Since it may sometimes be more natural to use the singular rather than the plural, the keyword `sort` is allowed as a synonym to `sorts`. □

For example, a single sort `Nat` may be declared like this:

```
sort Nat .
```

while to declare together the sorts `Nat`, `Int` and `Rat` we write:

```
sorts Nat Int Rat .
```

Subsort declarations have the following syntax:

Syntax 2.3 (Subsorts)

`subsorts` ⟨*SortList*⟩ `<` ⟨*SortList*⟩ {`<` ⟨*SortList*⟩}... .

where ⟨*SortList*⟩ is a non-empty list of sort names, separated by blanks and possibly qualified (this is explained in Section 2.1.3). This syntax specifies that the sorts mentioned in the first list are all subsorts of those in the second list, and so on. The keyword `subsort` is allowed as a synonym to `subsorts`. Also, note that the intended meaning is that of less-than-or-equal, although for typographical convenience the less-than symbol has been used. □

For instance, we may declare a sort `Nat`, denoting the natural numbers, to be a subsort of the sort `Int`, denoting the integers, because all natural numbers are also integers:

```
subsort Nat < Int .
```

Since the integers are in turn a subset of the rationals (of sort `Rat`, say), we could also write:

```
subsort Nat < Int < Rat .
```

or, similarly,

```
subsort Int < Rat .
subsort Nat < Int .
```

or even,

```
subsorts Nat Int < Rat .
subsort Nat < Int .
```

2.1.2 Functions and Terms

When the language of a particular application domain can be readily used in a specification, programs are easier to read and to write. Towards that end, FOOPS allows each function to be given a "syntactic form" that describes whether the function is to be referred to in infix, prefix, postfix, outfix or, in general, **mixfix** syntax. Also, function names may be overloaded, in that there can be two or more functions with the same name. These options require some sophisticated parsing, and a term in FOOPS is considered to be **well-formed** if and only if it has a unique *least* parse, where the ordering is derived from the subsort relation. The next subsection gives more details on this.

Syntactic specifications are of two forms. The first is the **standard form**, which gives ordinary prefix-with-parenthesis syntax to a function, with arguments separated by commas. For example, the function `cons` for prepending an element onto a list is normally written with this syntax. For a natural number `X` and a list `L` of natural numbers, a simple term involving `cons` is `cons(X,L)`. In FOOPS, this syntax is declared like this:

```
fn cons : Nat List -> List .
```

Here fn is a keyword and cons is the name, or form, of the function. The list of sorts between the ":" and the "->" gives the sort of each argument; this list is called the function's **arity**. The sort between the "->" and the "." is called the **coarity**, or value sort, of the function. The **rank** of a function is its arity and coarity taken together; for example, the rank of cons may be written (Nat List,List). More formally, standard form syntax is:

Syntax 2.4 (Standard Form Syntax)

fn ⟨*StdOpForm*⟩ : ⟨*Sort*⟩... -> ⟨*Sort*⟩ .

where ⟨*StdOpForm*⟩ is a string of symbols that cannot include underbars (i.e., "_"), and ⟨*Sort*⟩ is the name of a sort, possibly qualified. □

Constant, or nullary, functions are those whose arity is empty. For instance, the natural number 0 may be specified as the following constant:

```
fn 0 : -> Nat .
```

and the empty list of sort List as

```
fn nil : -> List .
```

With these declarations, the following are well-formed terms:

```
0
cons(0,cons(0,nil))
```

The first term has sort Nat and the second has sort List.

The other syntactic form is the **mixfix form**. It has this syntax:

Syntax 2.5 (Mixfix Form Syntax)

fn ⟨*MixfixOpForm*⟩ : ⟨*Sort*⟩... -> ⟨*Sort*⟩ .

where ⟨*MixfixOpForm*⟩ is like ⟨*StdOpForm*⟩ except that underscores are permitted. Underscores serve as placeholders for the arguments to the function, and there must be exactly as many as there are sorts in its arity, to which they correspond in order. Single underbars or just blanks are not valid syntactic forms. □

For example, the symbol "+" for infix addition of naturals may be declared like this:

```
fn _+_ : Nat Nat -> Nat .
```

For N and M of sort Nat, N + M is a well-formed term using this syntax. Also, the successor function on naturals is usually prefix:

```
fn succ_ : Nat -> Nat .
```

Factorial, on the other hand, is usually postfix:

```
fn _! : Nat -> Nat .
```

These are well-formed terms of sort Nat:

```
succ 0
(succ succ 0)!
```

To illustrate outfix syntax, consider a sort Set for sets of natural numbers. Singleton sets may be syntactically described in this way:

```
fn {_} : Nat -> Set .
```

Finally, syntax may be mixfix, as in:

```
if_then_else_fi : Bool Nat Nat -> Nat .
```

where Bool would denote the sort of boolean values. (By the way, Bool is declared in a module called BOOL that is automatically imported into every other module; see Section 2.1.5.5 for more details about BOOL.) In the context of these declarations, the following terms may be formed:

```
cons(0,nil)
{ succ 0 }
```

When two or more functions have the same rank, they may be declared together, as in:

```
fns (_+_) (_*_) : Nat Nat -> Nat .
```

Note here the use of the keyword fns and that the form of each function has to be enclosed in parentheses. However, the parentheses may he omitted for constants or when standard-form syntax is desired:

```
fns 0 1 2 : -> Nat .
fns plus times : Nat Nat -> Nat .
```

Formally, the syntax is:

Syntax 2.6 (Multiple Function Declarations)

fns ⟨*OpForm*⟩ ⟨*OpForm*⟩... : ⟨*Sort*⟩... -> ⟨*Sort*⟩ .

where ⟨*OpForm*⟩ is either ⟨*StdOpForm*⟩ or ⟨*MixfixOpForm*⟩. □

Example 2.7 The following is a simple syntactic specification of the hexadecimal numbers:

```
fmod HEX is
  sorts HexDigit HexNum .
  subsort HexDigit < HexNum .
  fns 0 1 2 3 4 5 6 7 8 9 : -> HexDigit .
  fns A B C D E F : -> HexDigit .
  fn _ . _ : HexDigit HexNum -> HexNum .
endf
```

Example terms from HEX are:

```
0
1 . F
1 . 5 . A . 2
```

The first has (lowest) sort HexDigit, while the others have sort HexNum. □

2.1.3 Parsing and Qualification

The flexible syntax of FOOPS offers many opportunities for ambiguity, and sometimes extra information must be attached to terms, or perhaps extra parentheses added, in order to ensure unique parses.

When functions are overloaded, terms may need to be **qualified** with sort information to resolve ambiguities. A sort-qualified term has this syntax:

Syntax 2.8 (Qualification with Sort Names)

⟨*Term*⟩).⟨*SortId*⟩

where ⟨*SortId*⟩ is the name of a sort. □

For example, if the declarations

```
sort Nat .
fn 0 : -> Nat .
```

are added to module HEX, the term 0 is ambiguous (it could be either of sort Nat or of sort HexDigit). To resolve this ambiguity, we need to specify which zero we are referring to, by using either (0).Nat or (0).HexDigit.

For a slightly more complicated example, consider these further declarations:

```
fn _+_ : HexNum HexNum -> HexNum .
fn _+_ : Nat Nat -> Nat .
```

Then the parse of 0 + 0 is also ambiguous, but this time in more than one way. To resolve it, we can either qualify the term on the outside or qualify one of the zeroes. For example,

```
(0 + 0).HexNum
```

and

```
(0).Nat + 0
```

are both unambiguous. As with the second zero in the last term, context information resolves ambiguities in many cases, and makes explicit qualifications unnecessary.

Sometimes sort qualification is not enough, because two different modules might introduce sorts with the same name; then, qualification with module names is required. For example, each of the modules NAT-LIST and COLOUR-LIST may define both a sort List and

a constant `nil` of that sort. To disambignate `nil`, we need to write either `(nil).NAT-LIST` or `(nil).COLOUR-LIST`. Sorts may also ueed to be qualified; for example, in syntactic specifications. Formally, qualification with module names has this syntax:

Syntax 2.9 (Qualification with Module Names)

$(\langle Term \rangle).\langle ModId \rangle$
$\langle SortId \rangle.\langle ModId \rangle$

Note that the second form does not require parentheses. □

However, in cases where a sort and a module have the same name the ambiguity may persist. On the other hand, if our naming conventions are followed (module names in upper case, sort names capitalised) this will not be a problem. Qualification with *module expressions* is discussed in Section 3.7.

2.1.3.1 Least Parses

It is possible to have `B` < `A` and the functions

```
fn f : A -> A .
fn f : B -> B .
```

This is of course overloading, but of a more subtle kind, as it involves not only names but also sorts that are related. From the previous discussion we know that a term `f(x)` for `x` of sort `B` will have sort `B`, because that is the "least" of the two possible parses. More specifically, rank ⟨`S1 S2 ... SN,R1`⟩ is less than rank ⟨`T1 T2 ... TN,R2`⟩ if and only if for `i = 1..N`, `Si` ≤ `Ti` and `R1` ≤ `R2`; below we also speak of orderings among arities, with a similar meaning. For unique least parses to exist, a condition on signatures, called "regularity," must be obeyed [49]. Regularity depends on a signature being **monotonic**, which means that for each pair

```
fn f : S1 S2 ... SN -> R1 .
fn f : T1 T2 ... TN -> R2 .
```

if the first arity is less than the second then `R1` ≤ `R2`. For example, the declarations

```
fn f : A A -> B .
fn f : B B -> A .
```

violate monotonicity, because the coarities are related in the opposite direction. A signature is **regular** if and only if

- it is monotonic, and
- given a function symbol `f` and a lower bound w_0 for its arity, there is a least arity for `f` among those `f`'s with arity greater than or equal to w_0.

For example, given `B < A`, the declarations

```
fn f : A B -> A .
fn f : B A -> B .
```

violate regularity, because neither rank is less than the other. (A lower bound arity here is `(B B)`.)

2.1.4 Inheritance Diagrams

Let us now consider some general examples of multiple inheritance and relate them to the preceding discussion on parsing. As a graphical aid, we will use diagrams in which an arrow from `X` to `Y` indicates that `X` is a subsort of `Y`. The diagram in Figure 2.1 shows three sorts `B`, `C` and `D`, where `D` is a subsort of both `B` and `C`; it also shows that each of `B` and `C` have a function `f` associated with them. This situation is erroneous because it violates regularity. (Hint: take $w_0 = D$.)

The diagram in Figure 2.2 shows `D` itself with an `f`. This is fine because regularity is obeyed. We call this situation **merging**.

Finally, the diagram in Figure 2.3 shows a rather common situation: inheritance of the same sort via distinct paths. A key question here is whether `D` is associated with two functions `f`, by virtue of there being two paths from `D` to `A`. As the diagram gives away, in FOOPS there is only one sort `A` from which `D` inherits, and therefore `D` is associated with only one `f`. Another way of viewing this is to consider the transitive closure of the subsort relation, which gives {(B,A), (C,A), (D,B), (D,C), (D,A)}, clearly showing that multiple ways of seeing `D` as a subsort of `A` have no effect on the relationship between the two.

2.1.5 Axioms and Evaluation

In the functional level of FOOPS, an **axiom** declares that two terms are equal. The terms in an axiom are more general than those given above as examples because they may also involve **variables**, which can be thought of as placeholders for arbitrary terms of their declared sort. Terms in axioms are therefore called **patterns**, or templates of possible terms; a term without variables is called a **ground term**. For example, given a variable `N` of sort `Nat` and a function `pred_` (predecessor) on natural numbers, the following is an axiom:

```
ax pred succ N = N .
```

Axioms may also be **conditional**, meaning that the equality holds only if a certain condition, given by a `Bool`-valued term, is true; furthermore, conditions may involve variables.

By regarding axioms as left-to-right rewrite rules, FOOPS takes **order-sorted term rewriting** as the model of computation for functional modules. Computations are started by supplying terms to the top level of the system. They proceed by repeatedly matching subterms against the left-hand sides of the rewrite rules and then rewriting, or replacing, the matched subterms with the right-hand side of the corresponding matching rules, until no more matches are found.

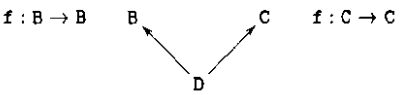


Figure 2.1: Multiple inheritance of a similar function.

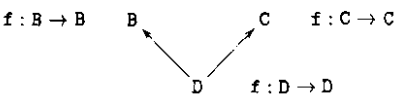


Figure 2.2: An example of merging.

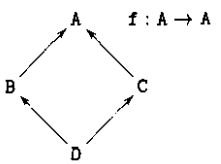


Figure 2.3: Inheritance via distinct paths.

2.1.5.1 Variables

Variable declarations associate an identifier with a sort, and have this syntax:

Syntax 2.10 (Variables)

`var` ⟨*VarId*⟩ `:` ⟨*Sort*⟩ `.`
`vars` ⟨*VarIdList*⟩ `:` ⟨*Sort*⟩ `.`

where ⟨*VarId*⟩ is the name of the variable, by convention written in upper case, and ⟨*VarIdList*⟩ is a list of ⟨*VarId*⟩. □

Example applications of this syntax are

```
var N : Nat .
vars FIRST MID LAST : Nat .
```

2.1.5.2 Axioms

Unconditional axioms have the following syntax:

Syntax 2.11 (Unconditional Axioms)

`ax` ⟨*Term*⟩ `=` ⟨*Term*⟩ `.`

The equal sign separates the left from the right-hand side of the axiom. The sort of the term on the right-hand side must be less than or equal to the sort of the term on the left-hand side; otherwise, under certain conditions, *retracts* might be added to the right-hand side, as explained in Section 2.1.6. □

Given the variable `N` above, another example axiom is

```
ax N + 0 = N .
```

Conditional axioms have this syntax:

Syntax 2.12 (Conditional Axioms)

`cax` ⟨*Term*⟩ `=` ⟨*Term*⟩ `if` ⟨*BoolTerm*⟩ `.`

where ⟨*BoolTerm*⟩ is a `Bool`-valued term. □

For example, given variables `N` and `M` of sort `Nat`, and a declaration of the greater-than function on naturals,

```
fn _>_ : Nat Nat -> Bool .
```

the following is a couditional axiom:

```
cax N + M > N = true  if M > 0 .
```

So that an axiom may be interpreted as a rewrite rule, the variables that appear in its right-hand side and in its condition (if conditional) must be a subset of those that appear in its left-hand side. Also, the left-hand side of an axiom may not be a single variable.

2.1.5.3 Evaluating Terms

The following informal definitions will help understand what a computation is in the order-sorted term rewriting model; formal mathematical definitions are given in Chapter 4. An **assignment** is a function $a : X \to T$ from a set X of variable symbols to a set T of ground terms over some signature; for $V \in X$, $a(V)$ is called the **value** of V (under a). An assignment $a : X \to T$ can be extended to a **term-assignment** $a^* : T(X) \to T$ which takes terms with variables from X to ground terms, by using a to assign values to each variable; for a term $t \in T(X)$, $a^*(t)$ is called the **instantiation** of t (under a^*). A ground term t **matches** a term p if there exists an assignment A such that p instantiated with A yields a term that is equal to t.

Given a set of rewrite rules, a computation starts with a term t and proceeds by repeatedly searching for a rewrite rule whose left-hand side is matched by some subterm s of t, instantiating the right-hand side of the rule with the assignment that generated the match, and rewriting, or replacing, s in t with the instantiated right-hand side. A conditional rule has its left-hand side matched similarly, but the rewrite takes place only if the instantiated condition evaluates to the constant `true`. If it does not, the matching process resumes by considering other rules. A computation terminates (if at all) when there are no more matches. This entire process is called the **evaluation** or **reduction** of t, and if it terminates, the resulting ground term is called the **normal form** of t. In general, FOOPS performs matching modulo associativity and commutativity, using facilities given by OBJ3; Section 2.1.7 gives details of this.

A set of rewrite rules is said to be **terminating** when the evaluation of terms terminates. Also, a set of rewrite rules is called **confluent**, or **Church-Rosser**, if whenever a term t can be rewritten to two different terms t_1 and t_2, these two terms can themselves be rewritten to some term t_3. FOOPS does not require that sets of rewrite rules satisfy either of these properties (an undecidable task anyway).

A further monotonicity condition exists for algebras, and states the following for each pair

```
fn f : S1 S2 ... SN -> R1 .
fn f : T1 T2 ... TN -> R2 .
```

in which the first rank is less than the second: for arguments of sorts `S1 S2 ... SN`, respectively, the two must give the same result. This phenomenon in which different instances of a function symbol are related by inheritance such that the result does not depend on the instance used is known as **subsort polymorphism** [49].

Below we give a module that defines lists of colours, in the context of which we will exemplify evaluations in FOOPS:

```
fmod LIST-OF-COLOUR is
  sorts Colour List .
  subsort Colour < List .
  protecting NAT .
```

```
  fns red blue yellow : -> Colour .
  fn __ : Colour List -> List .
  fn length_ : List -> Nat .
  var C : Colour .  var L : List .
  ax length C = 1 .
  ax length(C L) = 1 + length L .
endf
```

Colour is declared to be a subsort of List purely for the convenience of having single colours themselves as lists. The line "protecting NAT ." indicates that the module NAT should be imported into it (this is explained in more detail in the next chapter). NAT is part of the FOOPS default environment, and declares a sort Nat that represents the natural numbers plus the nsual operations on them. Nat has a subsort called Zero whose only element is the constant 0. and a subsort called NzNat whose elements are the natural numbers except zero. Note the juxtaposition notation used for constructing lists, with syntax __ (two underbars). The first axiom describes the length of single-element lists; the second axiom defines the length of lists with 2 or more elements in them.

Some evaluations are now in order. The top-level command for evaluating terms in FOOPS has this syntax:

Syntax 2.13 (Eval)

```
eval ⟨Term⟩ .
```

□

For instance, executing

```
eval blue .
```

results in the following output being displayed:

```
evaluate in LIST-OF-COLOUR : blue
rewrites: 0
result Colour: blue
```

Since blue does not match any left-hand side, no rewrites were possible. But for length blue there is one match, and the result of executing

```
eval length blue .
```

is

```
evaluate in LIST-OF-COLOUR : length blue
rewrites: 1
result NzNat: 1
```

Also,

```
eval length (red yellow) .
```

gives

```
evaluate in LIST-OF-COLOUR : length (red yellow)
rewrites: 3
result NzNat: 2
```

where (initially) `red` matches `C` and `yellow` matches `L` in the second axiom.

The axioms of module `LIST-OF-COLOUR` are confluent because a given term can only be matched by one axiom, never by both, as their left-hand sides define mutually exclusive patterns. The axioms are also terminating.

2.1.5.4 Traces

The **trace** of an evaluation of a term t is a sequence of terms beginning with t such that each subsequent term is the result of applying a rewrite rule to the previous term. For example, assuming a rewrite rule that expresses the result of adding two numbers, a trace of the evaluation of `2 + 3 + 5` is:

```
2 + 3 + 5
5 + 5
10
```

As a debugging and teaching aid, our prototype implementation of FOOPS offers a facility that allows following its evaluations step by step (this facility is inherited from OBJ3). A trace shows, for each matched subterm, the rule being applied, the assignment that generated the match, and the result of the rewrite.

2.1.5.5 Term Equality and the Module `BOOL`

FOOPS supports a polymorphic operator for testing the equality of two terms. For a sort `S`, it has syntax

```
fn _==_ : S S -> Bool .
```

It works by first evaluating both terms and then checking if the results are identical (modulo associativity and commutativity; see Section 2.1.7). Therefore, it is only appropriate to use it if the rewrite rules are terminating and confluent. There is also inequality, with syntax `_=/=_`, and if-then-else, with syntax `if_then_else_fi`. These three functions, along with the sort `Bool`, its constants `true` and `false`, and the logical functions `_and_`, `_or_` and `_implies_`, are part of a module called `BOOL` that is by default imported into every other module.

2.1.5.6 A Larger Example

This section presents an extended version of LIST-OF-COLOUR to exemplify more features of FOOPS. The extension includes functions for testing membership in a list and for reversing a list. The membership test illustrates the use of the equality predicate and of if-then-else. The reversal function illustrates more sophisticated pattern matching, and employs this algorithm: split the list in two, reverse the two parts, and then append the (reversed) second part onto the (reversed) first part; where to split the list is left to the pattern matcher. Note also that the append function is generalised, so that its first argument is a list and not just a single colour; it is also declared to be *associative* (see Section 2.1.7). Lastly, we use "--->", a command that prints whatever follows it, to document the expected results. When used without the >, nothing is printed, thus serving as a passive comment. Both comment forms can also be used inside modules.

```
fmod LIST-OF-COLOUR is
  sorts Colour List .
  protecting NAT .
  subsort Colour < List .
  fns red blue yellow : -> Colour .
  fn __ : List List -> List [assoc] .
  fn length_ : List -> Nat .
  fn _in_ : Colour List -> Bool .
  fn rev  : List -> List .
  vars C C1 C2 : Colour .  vars L L1 L2 : List .
  ax length C = 1 .
  ax length(C L) = 1 + length L .
  ax C1 in C2 = C1 == C2 .
  ax C1 in (C2 L) = if C1 == C2 then true else C1 in L fi .
  ax rev(C) = C .
  ax rev(L1 L2) = rev(L2) rev(L1) .
endf

eval red in (red blue) .            ---> should be true
eval red in (blue yellow) .         ---> should be false
eval rev(red blue yellow) .         ---> should be (yellow blue red)
eval rev(rev(red blue yellow)) .    ---> should be (red blue yellow)
```

2.1.6 Flexible Typing and Error Handling

Sometimes static type-checking is too restrictive because it rejects expressions that, while not completely well-formed, could achieve a correct type at run time. On the other hand, dynamic typing is too liberal, allowing truly nonsensical expressions to go undetected until run-time, with possibly disastrous consequences. To provide a middle ground, the FOOPS type checker does as much static typing as possible, but gives "the benefit of the doubt" to

certain expressions that show the potential of becoming type correct as evaluation proceeds.

The determination of when to do this is based upon the semantics of the subsort relation. The approach is this: if a function expects an argument of sort S but is given an argument of sort T, and S < T, then the type-checker inserts a function that will try to lower the sort of the actual argument to S. If the argument evaluates to a term of sort S (or perhaps lower), then the function that was inserted will disappear at run-time, and all is fine. If it does not, the function will not disappear but will instead remain in the expression as an informative error message. This kind of function is called a **retract**. In general, there is a retract function from a sort T to a sort S if S < T or if S and T have a common supersort. This technique naturally supports error detection and recovery, while at the same time avoiding the complexities associated with partial functions and the arbitrariness of the exception-handling mechanisms of some languages.

First a simple example illustrating retracts. Consider the following (incomplete) module:

```
fmod NUMBERS is
  sorts Rat Nat .
  subsort Nat < Rat .
  fn _/_ : Nat Nat -> Rat .
  fn _! : Nat -> Nat .
  ...
endf
```

where Nat is intended to represent the natural numbers and Rat the rationals. For N and M of sort Nat, an expression such as

```
(N / M)!
```

does not strictly type-check, because the division function has value sort Rat but the factorial function expects a Nat. However, certain divisions, such as 4 divided by 2, actually result in a natural number, and thus

```
(4 / 2)!
```

makes sense. Since Nat < Rat, the FOOPS type checker will insert a retract function in this expression to try to lower the sort of the result of the division. In this case, the retract function is named r:Rat>Nat and would be defined by these declarations (altbough it is actually built-in[1]):

```
fn r:Rat>Nat : Rat -> Nat .
var N : Nat .
ax r:Rat>Nat(N) = N .
```

Note that the axiom says exactly what we want: if the argument is a natural number, then the retract function disappears. The previous expression would therefore be converted by the parser to

[1] In OBJ3, as well as in FOOPS, a **built-in** feature is one which is automatically provided, defined either in terms of other features or in terms of the facilities of an underlying implementation language.

```
(r:Rat>Nat(4 / 2))!
```

and its evaluation yields 2.

We now consider how to do error detection and recovery in FOOPS. A familiar benchmark is stacks, because they force us to deal with situations such as `top(empty)` and `pop(empty)`. In the example that follows, the approach is to restrict the domain of these functions to the non-empty stacks by defining a subsort `NeStack` of `Stack`, and declaring `push` to have coarity `NeStack`. This code realises this idea:

```
fmod STACK-OF-NAT is
  sorts Stack NeStack .
  subsort NeStack < Stack .
  protecting NAT .
  fn empty : -> Stack .
  fn push : Nat Stack -> NeStack .
  fn top_ : NeStack -> Nat .
  fn pop_ : NeStack -> Stack .
  var X : Nat .   var S : Stack .
  ax top push(X,S) = X .
  ax pop push(X,S) = S .
endf
```

Then, an expression such as `top push(5,empty)` is well-formed and evaluates to

```
result NzNat: 5
```

but `top pop push(1,push(2,empty))` needs a retract, and is converted to

```
top r:Stack>NeStack(pop push(1,push(2,empty)))
```

During its evaluation the retract disappears, and yields

```
result NzNat: 2
```

On the other hand, `top pop push(5,empty)` parses as

```
top r:Stack>NeStack(pop push(5,empty))
```

but its evaluation cannot make the retract disappear, and gives

```
result Nat: top r:Stack>NeStack(empty)
```

The retract then serves as an indication of how and where something went wrong.

There are at least three ways to add error recovery code to this specification. First, `top` and `pop` may be overloaded (to accept arguments of sort `Stack`), and then axioms of the following form may be declared:

```
ax top empty = ...
ax pop empty = ...
```

Second, retracts can be explicitly used on the left-hand side of axioms. For example:

```
ax top r:Stack>NeStack(empty) = ...
ax pop r:Stack>NeStack(empty) = ...
```

(A large example of the use of retracts in this way is given in [42].)

Lastly, [53] presents an approach that involves declaring error supersorts to contain messages for exceptional conditions. In FOOPS, however, error sorts are *automatically* declared, and named by appending the string "?" to the name of the original sort. For example, declaring a sort S causes the interpreter to declare a sort S? with S < S?.

2.1.7 Function Properties

It is of great advantage and convenience to be able to *declare* whether a function has certain properties, and for a system to directly recognise them, rather than having to encode these properties as axioms. This section describes three properties that can be given to functions in FOOPS, namely associativity, commutativity and identities, and their effect on pattern-matching and rewriting. Some other properties, such as precedence and memoisation, are discussed in [53][2].

Properties are given as part of the syntactic form declaration of a function, between the coarity and the final period, and are enclosed in square brackets. In full, the syntax for functions with properties is:

Syntax 2.14 (Functions with Properties)

`fn` ⟨*OpForm*⟩ `:` ⟨*Sort*⟩`...` `->` ⟨*Sort*⟩ `[`⟨*Props*⟩`]` `.`

where ⟨*Props*⟩ is a list of properties, and ⟨*OpForm*⟩ is either ⟨*StdOpForm*⟩ or ⟨*MixfixOpForm*⟩ (as before). The associative property has syntax `assoc`, the commutative property has syntax `comm`, and the identity property has syntax

`id:` `(`⟨*Term*⟩`)`

The order in which properties are given is irrelevant. □

For instance, boolean conjunction would be declared to be associative:

```
fn _and_ : Bool Bool -> Bool [assoc] .
```

This allows us to write, say, the term

```
true and false and true
```

[2] That document calls properties "attributes." We avoid this terminology because "attributes" also has a technical meaning at the object level of FOOPS.

without the parentheses that would otherwise be required to disambiguate its two possible parses. More important, perhaps, is that the pattern matcher will take this information into account and ignore parentheses when matching terms that involve associative functions, even if we explicitly write the parentheses for readability. The associative property also affects the equality and inequality predicates `_==_` and `_=/=_`, and the system will be able to determine that, for example,

```
true and (false and true)  ==  (true and false) and true
```

is true. The `assoc` property is only meaningful for a binary function with rank `(A B,C)` when `C < A` and `C < B`; however, retracts may be inserted if either `A < C` or `B < C`.

Since Boolean conjunction is also commutative, it would be declared as such with the `comm` property:

```
fn _and_ : Bool Bool -> Bool [assoc comm] .
```

Here the situation is more subtle because the axiom that expresses commutativity, i.e.,

```
ax P and Q = Q and P .
```

causes termination problems when considered as a left-to-right rewrite rule. If instead we let the pattern matcher of FOOPS take care of this property, the termination problem is avoided, because it tries all possible orderings of arguments to commutative functions when attempting matches. For example, there is no need to write two versions of the axiom

```
ax P and false = false .
```

because the terms `true and false` and `false and true` are both matched by the left-hand side of the axiom. In addition, the equality and inequality predicates will also take commutativity into account, and the system will be able to determine that

```
true and (false and true)  ==  (true and true) and false
```

is true (because `_and_` is also associative). The `comm` property is only meaningful for a binary function whose two arity sorts have a common supersort.

The kind of matching that results when a function is given associativity and commutativity properties is called **A/C matching**. Although it is an NP-complete problem, A/C matching is highly optimised in the implementation of FOOPS, using facilities given by OBJ3.

Lastly, functions may be given identities. For example, Boolean disjunction has `false` as a left and a right identity. Thus, it would be declared like this:

```
fn _or_ : Bool Bool -> Bool [id: (false)] .
```

This gives the effects of the axioms

```
ax false or P = P .
ax P or false = P .
```

(Of course, `_or_` would also be declared to be associative and commutative.) More specifically, a left-identity equation is added if the sort of the identity is less than the sort of the first argument, and a right-identity equation is added if the sort of the identity is less than the sort of the second argument. See [53] for a presentation of the propositional calculus that uses all of the above properties.

2.1.8 Order of Evaluation

Rather than adopting a fixed strategy for evaluating the arguments of a function, FOOPS allows the user to specify the order in which arguments are to be evaluated. Thus, strategies such as lazy, eager, or any mixture of those, can be specified on a per-function basis. This flexibility has wide-ranging applications, including language specification and operating system scheduling [46]. To make matters simple for the programmer, FOOPS computes a default strategy for functions that are not explicitly given one; for most cases of interest, it is left-to-right and eager.

The evaluation strategy, or **E-strategy**, of a function is also given as part of its declaration, in the section for properties. It consists of a list of argument indexes that in the simplest case gives the order in which arguments should be evaluated; laziness on an argument is specified by omitting its index from the list. An example is if-then-else, declared like this for some sort `S`:

```
fn if_then_else_fi : Bool S S -> S [strat (1 0)] .
```

The strategy is the parenthesised list given after the keyword `strat`. It indicates that the first argument should be evaluated first, followed by rewrites at "the top,", i.e., involving `if_then_else_fi` (indicated by "`0`"). In other words, the **then** and **else** branches are evaluated lazily.

Section C.5 of [53] presents another example of the use of lazy evaluation, where the Sieve of Erathostenes is used to find all prime numbers.

2.2 Object-level Modules

An object module defines one or more classes, which are collections of (potential) objects. The attributes and methods associated with a class give the description of the internal state of its objects and the operations that can change that state. Objects are created and deleted dynamically and are accessed with unique **object identifiers** that are assigned to them at the time of creation. In FOOPS objects also *persist*, meaning that once created they become part of the environment and remain there until explicitly deleted[3]. In addition, **metaclasses** are provided; these are lists of the current objects of a certain class, and may be used to effect changes on groups of objects of the same class. Furthermore, object modules may declare abstract data types.

Many of the concepts and mechanisms presented in the previous section also apply to the object level. First, classes may be organised in inheritance hierarchies, and when a class

[3] This persistence is *per session*

A inherits from a class B we say that A is a **subclass** of B. Second, the informal definitions of signature and algebra carry over as expected to include classes, subclasses, attributes and methods (but see Chapter 4 for precise definitions). Also, mixfix syntax is available. Terms at this level may then be constructed using attributes, methods, variables and even function symbols, and the same qualification notation for disambiguating parses at the functional level may be used. In addition, the concept of a least parse carries over.

A further feature at this level is **redefinition**, or **overriding**, by which subclasses may replace by new ones the definitions of attributes and methods associated with their superclasses. Therefore, subclass relationships cannot be strictly interpreted as inclusions, although operationally an object of some class B can be placed wherever an object of any of its superclasses is expected, giving rise to **subclass polymorphism**. At run time, a mechanism called **dynamic binding** selects the most specific version of a method, based on the class of the object to which the method is applied; similarly for attributes.

Lastly, the model of computation at this level is a generalised form of term rewriting in which implicit reference is made to a database of objects. We will use examples to explain the interpretation of axioms under this model.

When not parameterised, object modules have this form:

Syntax 2.15 (Unparameterised Object Modules)

```
omod ⟨ModId⟩ is
  ⟨oModElt⟩ ...
endo
```

where ⟨*ModId*⟩ stands for the name of the module, by convention given in upper case letters. ⟨*oModElt*⟩ stands for the things that may be declared by an object module, which include classes, attributes, methods and axioms, but also anything that can be declared by a functional module. Object modules may also import other functional or object modules. □

2.2.1 Classes and Subclasses

Class declarations have the following syntax:

Syntax 2.16 (Classes)

```
classes ⟨ClassIdList⟩ .
```

where ⟨*ClassIdList*⟩ is a non-empty list of class names separated by blanks; by convention, class names are capitalised. The keyword `class` is allowed as a synonym to `classes`. □

Example applications of this syntax are:

```
class LinkedList .
classes Teacher Student .
```

Subclass declarations have the following syntax:

Syntax 2.17 (Subclasses)

`subclasses` ⟨*ClassList*⟩ `<` ⟨*ClassList*⟩ {`<` ⟨*ClassList*⟩}... .

where ⟨*ClassList*⟩ is a non-empty list of class names, separated by blanks and possibly qualified (this was explained in Section 2.1.3). This syntax specifies that the classes mentioned in the first list are all subclasses of those in the second list, and so on. The keyword `subclass` is allowed as a synonym to `subclasses`. □

For example, we may declare the following subclass relationships:

```
subclass Teacher < Person .
subclasses TeachingAssistant < Teacher Student < Person .
```

2.2.2 Attributes

The syntax for declaring attributes is similar to that for declaring functions. It is:

Syntax 2.18 (Attributes)

`at` ⟨*OpForm*⟩ `:` ⟨*KindList*⟩ `->` ⟨*Kind*⟩ .

where ⟨*OpForm*⟩ is the syntactic form of the attribute (just as for functions), ⟨*Kind*⟩ is the name of a sort or a class (possibly qualified), and ⟨*KindList*⟩ is a non-empty list of ⟨*Kind*⟩. The arity must include at least one class. □

For example, a class `Person` might have an attribute called `age_`, declared like this:

```
at age_ : Person -> Nat .
```

Likewise, an attribute called `nth-person` for a class `PersonList` may be declared like this:

```
at nth-person : PersonList Nat -> Person .
```

Note from these declarations that an object of class `Person` has only one age, but that an object of class `PersonList` can have many values for its `nth-person` attribute.

Like `fns`, the keyword `ats` is used to declare together attributes with the same rank, as in

```
ats (age_) (number-of-children_) : Person -> Nat .
```

An attribute is always associated with the first class mentioned in its arity. For instance, if `Circle` and `Square` are classes, then

```
at fits-inside : Circle Square -> Bool .
```

is regarded as an attribute of objects of class `Circle`. Attributes whose coarity is a sort are commonly referred to as **sort-valued** attributes, and those whose coarity is a class as **object-valued** or **complex** attributes. An object with complex attributes is called a **complex object**.

Attributes are also classified by whether their values are **stored** or **derived**. The value of a stored attribute is explicitly kept in an object and can be directly updated by methods. The value of a derived attribute is given by an expression that may involve various function and attribute symbols, and thus cannot be directly updated; said differently, the value of a derived attribute is functionally determined from the stored state of the object. (Derived attributes are analogous to what the database community calls *computed* or *virtual fields*.) It is often clear whether an attribute should be stored or derived, but occasionally an attribute can be either, and the choice is one of convenience. An example of this would be an attribute `height` for a class of balanced binary trees. If stored, the operations that change the structure of trees (e.g., `insert`) must ensure that the value of `height` is always up to date; if derived, this value may be computed on demand by traversing trees from their root to any leaf, giving as result the number of nodes visited minus one. Another example of an attribute requiring a similar decision is `ntb-person`, which could be computed on demand or stored in the object for every currently valid index.

In FOOPS there is no special syntax for distinguishing stored from derived attributes. An attribute is assumed stored unless an axiom that defines its value is given. For instance, the value of an attribute `age_` for persons might be defined with an axiom such as

```
ax age P = year-difference(current-date,birth-date(P)) .
```

where `current-date` would be a built-in nullary function returning the current date and `birth-date` a stored attribute.

For each stored attribute there is a built-in axiom that indicates how to fetch its value from the database of objects in the FOOPS environment. Axioms for both derived and stored attributes are interpreted operationally as rewrite rules in this extended sense.

Finally, attributes can be declared associative but not commutative, and can have evaluation strategies. See Section 2.2.4 for more details about evaluation order.

2.2.3 Creating Objects

Objects in FOOPS may be created either dynamically or at module-entry time. For dynamic creation, FOOPS provides a method that is built-in for every class. For a class `C`, it is a standard-form method called `new.C` that accepts as arguments a unique object identifier and initial values for each of the stored attributes of the object; it gives as result the identifier. Object identifiers are simple symbols such as `Johnny`, `TheRedArmy` and `Chapter1`. Initial values for attributes are specified by giving the attribute's syntactic form, then an equal sign, and finally the term for the value itself. To illustrate object creation, consider this simple module:

```
omod NAT-LINKABLE is
```

```
  class Linkable .
  protecting NAT .
  at value_ : Linkable -> Nat .
  at next_  : Linkable -> Linkable .
endo
```

NAT-LINKABLE defines a class of nodes similar to those that would be used in a linked implementation of lists. To create an object of class Linkable, we can execute

```
eval new.Linkable(SomeLinkable, value_ = 10, next_ = AnotherLinkable) .
```

where SomeLinkable is the (unique) identifier of the new object, and AnotherLinkable is the identifier of another object of class Linkable that must have been created previously[4]. The semantics of creation calls for attributes to be initialised in parallel, so that the expressions on the right-hand side of the equal sign cannot refer to the new object's state.

It is an error to attempt to create an object whose identifier is not unique within its class. However, objects of unrelated classes may have the same identifier, and qualification may be required to resolve ambiguities.

Object creation is very flexible and powerful in FOOPS, as we now illustrate. First, there is no fixed order in which initial values for attributes must be given, so that

```
new.Linkable(SomeLinkable, next_ = AnotherLinkable, value_ = 10)
```

is the same as

```
new.Linkable(SomeLinkable, value_ = 10, next_ = AnotherLinkable)
```

Moreover, object identifiers are optional, and in their absence generated automatically by the system. For example, it is possible to write

```
new.Linkable(value_ = 10, next_ = AnotherLinkable)
```

and let the system choose a unique identifier for the new Linkable object; this is useful for data structures in which we do not care about the names of internal components.

Finally, initialising an attribute is optional. An attribute that is not initialised is assigned a *default value*, which is either given explicitly or else is determined from the environment (see below for the details). For example, the execution of

```
eval new.Linkable(value_ = 25) .
```

will involve assigning a default value to next_. Combining all of these conventions, we see that

```
new.Linkable()
```

[4] There is a circularity problem here. how can a Linkable object be created without there being other linkables already in existence? Further below we show how this is solved.

is a valid expression in FOOPS. It creates an object of class `Linkable` which is given a unique identifier and default values for its attributes.

Formally, the syntax for object creation is:

Syntax 2.19 (Object Creation)

`new.` ⟨*ClassId*⟩ `(`[⟨*NewObjArgs*⟩]`)`

where ⟨*NewObjArgs*⟩ is defined to be:

⟨*ObjectId*⟩ { `,` ⟨*OpForm*⟩ `=` ⟨*Term*⟩}...

□

2.2.3.1 Entry-time Creation

Objects created at module-entry time may serve to denote special situations, in much the same way that constants are sometimes used at the functional level; for example, the constant `nil` is used to denote empty lists. The attributes of these objects may be given initial values by declaring the appropriate axioms: those attributes not initialised receive default values, as explained below. The syntax for specifying the identifiers of these objects is to declare a method without arguments:

Syntax 2.20 (Entry-time Objects)

`me` ⟨*ObjId*⟩ `:` `->` ⟨*Class*⟩ `.`

where ⟨*ObjId*⟩ is an object identifier, as described earlier. □

(Section 2.2.4 describes how methods with arguments are used for updating objects.)

For example, we may declare a class of stacks and a constant `empty` to denote empty stacks, as follows:

```
omod STACK-OF-NAT is
  class Stack .
  extending NAT .
  fn no-top : -> Nat? .
  at top      : Stack -> Nat? [default: (no-top)] .
  at rest     : Stack -> Stack .
  at is-empty : Stack -> Bool .
  me empty    : -> Stack .
  me pop      : Stack -> Stack .
  me push     : Nat Stack -> Stack .
  var S : Stack .  var N : Nat .
  --- the next two axioms declare initial values for empty:
  ax  top(empty) = no-top .
  ax rest(empty) = empty .
```

```
  ax is-empty(S) = top(S) == no-top and rest(S) == empty .
  ax top(pop(S)) = top(rest(S)) .
  ax rest(pop(S)) = rest(rest(S)) .
  ax top(push(N,S)) = N .
  ax rest(push(N,S)) = new.Stack(top = top(S), rest = rest(S)) .
endo
```

(The line "`extending NAT .`" declares that `NAT` is imported in `extending` mode. The details of this mode may be ignored here; the next chapter gives more information.)

Two remarks are in order. First, the axiom

```
ax rest(push(N,S)) = S .
```

would have been incorrect: it specifies that after a `push`, the `rest` of a stack is itself; this is so because `S` denotes an object identifier and not the state of some stack. The axiom in the module says that the current state of the stack is copied onto its `rest`, so that this circularity problem is avoided; this seems to be a good example of the difference between reference and copy semantics for assignment in programming languages.

Second, terms such as `push(5,empty)` are valid. However, if `empty` is updated its intended meaning is destroyed. This misuse can be prevented by declaring that `empty` is *private* to `STACK-OF-NAT` (and indeed `rest` would need to be private too, so that `empty` could not be accessed indirectly either); Section 3.9 describes how to do this.

2.2.3.2 Default Values for Attributes

When an attribute of a new object is not initialised it is assigned a default value. There are two kinds of default value: explicit and implicit. *Explicit* default values are those specified as part of an attribute's declaration, in the section for properties, as in:

```
at age_ : Person -> Nat [default: (1)] .
```

which indicates that if `age_` is not given a value in a call to `new.Person`, for example, it should be set to `1` automatically. The syntax for specifying an explicit default value is:

Syntax 2.21 (Explicit Default Value)

`default: (`⟨*Term*⟩`)`

□

If an attribute is not given an initial value in a call to `new` or with an axiom (for entry-time objects) and it does not have an explicit default value, an *implicit* default value for it is determined from the environment. This helps with the creation of objects with complex structure, and works as follows. For a sort-valued attribute of sort `S`, the implicit default value is the **principal constant** of `S`, which is the first-declared constant of that sort[5].

[5] A more satisfactory definition of principal constants takes into account module hierarchies; it is given in Section 3.1.1.

If S does not have a principal constant, the implicit default value for the attribute is an automatically provided constant called `void-S` of sort `S?`, the error supersort of `S`. For example, in

```
new.Linkable(SomeLinkable, next_ = AnotherLinkable)
```

attribute `value_` is assigned `0`, because that is the principal constant of sort `Nat` in module `NAT` (see [95] for a complete list of the principal constants of the sorts of modules in the default environment).

For an object-valued attribute with class `C` as coarity, the implicit default value is the first-declared entry-time object of that class. For example, in the context of module `STACK-OF-NAT`, `new.Stack()` creates an empty stack (as determined by `is-empty`); i.e., `is-empty(new.Stack())` evaluates to `true`. If such an entry-time object does not exist, then the implicit default value is a new object of class `C` whose attributes are all initialised with defaults. The termination of this recursive strategy can be guaranteed by remembering the classes that have been instantiated and stopping at the point where the default value for an attribute requires the instantiation of a class for the second time. Then, that attribute is assigned an object of class `C?`, the error superclass of `C`, whose identifier is `void-C`. This object is automatically provided by FOOPS and is by convention used to denote a null reference. For example, in

```
new.Linkable(SomeLinkable, value_ = 15)
```

attribute `next_` is assigned `void-Linkable`. More examples of default value computations are given in Section 2.2.3.4.

2.2.3.3 Metaclasses

Sometimes there is a need to define operations that act upon all of the objects of a certain class. To facilitate this, every class in FOOPS has an associated "metaclass," which is a list of its objects; in fact, object creation and deletion can be seen as methods associated with metaclasses (deletion is explained in Section 2.2.5). The metaclass of a class `C` is accessed with the nullary operation `all-C`. The next section and Appendix B provide more details and examples.

2.2.3.4 An Example

This section further demonstrates the various features presented thus far with two modules and several evaluations[6]; also, it shows that objects persist in the FOOPS environment. The first module declares a class `Person` and two subclasses of it, `Male` and `Female`. Objects of class `Person` have two stored attributes, `name_` and `age_`, and one derived attribute, `gender_`. Because `Male` and `Female` are subclasses of `Person`, their objects also have these attributes; however, no others are declared for them in particular. Attribute `name_` is given

[6] These modules appeared originally as examples in [48], but we have changed some of the details in order to use new language features.

an explicit default value but age_ is not. Its default value is implicitly the natural number 0, because that is the principal constant of sort Nat in module NAT.

PERSON imports QID, a functional module provided in the FOOPS default environment. This module declares a sort named Id whose elements are quoted symbols, such as 'Car and 'Book. (QID has nothing to do with object identifiers.)

The evaluations are straightforward.

```
omod PERSON is
  classes Person Male Female .
  subclasses Male Female < Person .
  sort Gender .
  fns male female : -> Gender .
  protecting QID .
  protecting NAT .
  --- stored attributes:
  at name_ : Person -> Id [default: ('NoName)] .
  at age_  : Person -> Nat .
  --- a derived attribute:
  at gender_ : Person -> Gender .
  var F : Female .  var M : Male .
  ax gender F = female .
  ax gender M = male .
endo

eval new.Female(Wilma, name_ = 'WilmaPebble, age_ = 30) .
                                        ---> should be Wilma
eval new.Male(Fred, name_ = 'FredFlinstone, age_ = 35) .
                                        ---> should be Fred
eval new.Person(Somebody, age_ = 23) .  ---> should be Somebody
eval name Somebody .                    ---> should be 'NoName

eval gender Wilma .       ---> should be female
eval gender Fred .        ---> should be male
eval gender Somebody .    ---> should not evaluate any further

---> examine metaclasses
eval all-Person .         ---> should be Wilma, Fred, and Somebody
eval all-Male .           ---> should be Fred
eval all-Female .         ---> should be Wilma
```

The second module declares a class of families and imports module PERSON. Objects of class Family have two stored attributes, wife_ and husband_, and by their coarities it is clear that only females can be wives and only males can be husbands. There is also a

derived attribute, **name_**, defined to be the name of the husband.

The evaluations make use of the fact that objects persist in the environment: the objects created in the context of module PERSON are available when carrying out evaluations in the context of module FAMILY, because PERSON is imported into FAMILY.

```
omod FAMILY is
  class Family .
  protecting PERSON .
  --- stored attributes:
  at wife_ :     Family -> Female .
  at husband_ : Family -> Male .
  --- a derived attribute:
  at name_ : Family -> Id .
  var F : Family .
  ax name F = name husband F .
endo

---> the next evaluation uses objects created previously
eval new.Family(TheFlinstones, wife_ = Wilma, husband_ = Fred) .
eval name TheFlinstones .        ---> should be 'FredFlinstone
eval age wife TheFlinstones .    ---> should be 30

---> initial values can also be calls to new:
eval new.Family(TheMunsters,
          husband_ = new.Male(Herman, age_ = 42, name_ = 'HermanMunster),
          wife_ = new.Female(Lily, age_ = 37, name_ = 'LilyMunster)) .
eval age husband TheMunsters .     ---> should be 42
eval name wife TheMunsters .       ---> should be 'LilyMunster
eval name TheMunsters .            ---> should be 'HermanMunster

---> implicit default values for complex attributes
---> need to be computed next:
eval new.Family(TheNeighbours) .
eval wife TheNeighbours .          ---> should be some Female identifier
eval age husband TheNeighbours .   ---> should be 0
eval name TheNeighbours .          ---> should be 'NoName

---> ids are also optional
eval new.Person(age_ = 20, name_ = 'GrandpaMunster) .
eval new.Family() .
eval new.Person() .
```

2.2.4 Methods

Methods (with arguments) are the operations that change the state of objects by assigning new values to their attributes. In FOOPS, the effect of a method is described declaratively by axioms. Also, methods may be combined to form **method expressions** that perform complex updates on various objects.

The syntax for declaring methods is similar to that for declaring attributes and functions:

Syntax 2.22 (Methods)

```
me ⟨OpForm⟩ : ⟨KindList⟩ -> ⟨Kind⟩ .
```

where ⟨*Kind*⟩ is the name of a sort or a class (possibly qualified), and ⟨*KindList*⟩ is a non-empty list of ⟨*Kind*⟩. The arity must include at least one class. □

Like attributes, methods are associated with the first class mentioned in their arity, so that for classes `NatTree` and `NatList`, the following are methods on objects of class `NatTree`:

```
me insert_in_  : Nat NatTree -> NatTree .
me insert-each : NatTree NatList -> NatTree .
```

The keyword `mes` may be used to declare together methods with the same rank, as in

```
mes (insert_in_) (delete_from_) : Nat NatTree -> NatTree .
```

Method axioms can be of two forms. The first is the **direct** form, which specifies the attribute to be updated and the value of the attribute after the execution of the method. For a standard-syntax attribute `a` and a standard-syntax method `m`, the general form of a direct method axiom (DMA) is:

```
ax a(m(O,args)) = ⟨Term⟩ .
```

where `O` is a variable that denotes the object that `m` updates, `args` stands for the other arguments to `m` (if any)[7] and ⟨*Term*⟩ cannot contain any method symbols. The axiom specifies that the value of `a` after the execution of `m` is equal to the term on the right-hand side. The term on the right-hand side may mention `a`, so that its new value can be defined in terms of its old value. Moreover, when a *group* of DMAs define a method, their right-hand sides are evaluated before any attributes are changed.

It is not necessary to give a DMA for each of the object's attributes. If a DMA is not given for a particular one, then the attribute retains its old value. This is termed a **frame assumption**, and reduces the number of axioms that must be written. Finally, note that methods described with DMAs evaluate to the identifier of the object they update. Therefore, their coarity must be the class of this object (otherwise, the left-hand side of the DMAs would not parse properly).

Let us consider some examples in the context of these declarations:

[7] To simplify the exposition we have assumed that the object to update is given as the first argument.

```
class Pair .
ats fst snd : Pair -> Nat .
```

which define a class of objects whose state consists of a pair of natural numbers. A method that increments attribute `fst` by a certain amount may be declared like this:

```
me incr-fst : Pair Nat -> Pair .
```

Its effect is captured by the following axiom:

```
ax fst(incr-fst(P,N)) = fst(P) + N .
```

where `P` is a variable of class `Pair` and `N` is a variable of sort `Nat`. These evaluations illustrate how the method works:

```
eval new.Pair(p, fst = 0, snd = 0) .
sval incr-fst(p,5) .  ---> should be p
eval fst(p) .         ---> should be 5
eval snd(p) .         ---> should be 0 (no change)
```

The evaluations also show that each DMA is not interpreted directly as a rewrite rule. The DMAs that describe a method may be thought of as *one* rewrite rule whose left-hand side has the method as its top symbol and whose right-hand side gives the updated object. (Again, this is not term rewriting in the classical sense, but an extension of it that takes into account an implicit object database.) However, we regard the declarative reading of individual DMAs as primary.

Now consider a method `swap` that sets the value of `fst` to that of `snd` and vice versa. It is defined by these declarations:

```
me swap : Pair -> Pair .
ax fst(swap(P)) = snd(P) .
ax snd(swap(P)) = fst(P) .
```

Note that the declarativeness of DMAs affords an exceptionally compact description of this method. Again, evaluations exemplify the situation:

```
eval new.Pair(p2, fst = 0, snd = 1) .
eval swap(p2) .  ---> should be p2
eval fst(p2) .   ---> should be 1
eval snd(p2) .   ---> should be 0
---> now try a method expression!
eval swap(incr-fst(p2,1)) .  ---> should be p2
eval fst(p2) .   ---> should be 0
eval snd(p2) .   ---> should be 2
```

A method `copy` that takes as arguments two pairs and sets the values of the attributes of the first to be equal to those of the second is defined as follows:

```
me copy : Pair Pair -> Pair .
vars P1 P2 : Pair .
ax fst(copy(P1,P2)) = fst(P2) .
ax snd(copy(P1,P2)) = snd(P2) .
```

The axioms specify changes to the attributes of the object denoted by P1, following the convention stated earlier. Now some evaluations:

```
eval new.Pair(p3, fst = 5, snd = 10) .
eval new.Pair(p4, fst = 7, snd = 14) .
eval copy(p3,p4) .  ---> should be p3
eval fst(p3) .  ---> should be 7
eval snd(p3) .  ---> should be 14
eval fst(p4) .  ---> should be 7  (no change)
eval snd(p4) .  ---> should be 14 (no change)
```

DMAs may also be conditional. Their general form is:

```
cax a(m(O,args)) = ⟨Term⟩ if ⟨BoolTerm⟩ .
```

where ⟨*BoolTerm*⟩ is a Bool-valued term that may not contain any method symbols; everything else is as before. This axiom says that m makes attribute a of O equal to ⟨*Term*⟩ if the condition is satisfied; otherwise, the attribute is not changed. Also, both ⟨*Term*⟩ and ⟨*BoolTerm*⟩ are evaluated before the execution of the method (in a fashion similar to that of unconditional DMAs, as explained above). By way of illustration, consider a method called make-snd-zero on pairs that sets the value of snd to zero if its current value is greater than ten but does not change it otherwise. The following declarations describe this method:

```
me make-snd-zero : Pair -> Pair .
cax snd(make-snd-zero(P)) = 0  if snd(P) > 10 .
```

Now some example evaluations:

```
eval new.Pair(p5, fst = 5, snd = 10) .
eval make-snd-zero(p5) .  ---> should be p5
eval snd(p5) .                          ---> should be 10 (no change)
eval snd(swap(incr-fst(swap(p5),2))) .  ---> should be 12
eval make-snd-zero(p5) .                ---> should be p5
eval snd(p5) .  ---> should be 0
eval fst(p5) .  ---> should be 5 (no change)
```

The second form of method axiom is the **indirect** form, which defines a method in terms of a method expression. For a standard-syntax method m, an indirect method axiom (IMA) has the following form:

```
ax m(O,args) = mexpr .
```

where `O` is a variable and `mexpr` is a method expression, which is simply a term that involves method symbols. Each IMA is interpreted as a rewrite rule, and a method defined in this way may return any value[8]. To facilitate the construction of method expressions, FOOPS provides a *method expression combinator* that composes expressions sequentially. This built-in feature has syntax `_;_`, so that if `mexpr2` and `mexpr3` are method expressions, then the following are also method expressions:

```
mexpr ; mexpr2
mexpr ; mexpr2 ; mexpr3
```

This combinator associates to the left and operates by evaluating its first argument and then its second argument; it gives as result the evaluation of its second argument. For example, assume that a method `incr-snd` (similar to `incr-fst`) has been declared. Then a method `incr-both` that adds a certain amount to each component of a pair can be declared like this:

```
me incr-both : Pair Nat Nat -> Pair .
ax incr-both(P,N1,N2) = incr-fst(P,N1) ; incr-snd(P,N2) .
```

where `N1` and `N2` are variables of sort `Nat`. Finally, IMAs may also be conditional, with the following form and usual interpretation as rewrite rules:

```
cax m(O,args) = mexpr if ⟨BoolTerm⟩ .
```

As before, the condition may not include any method symbols.

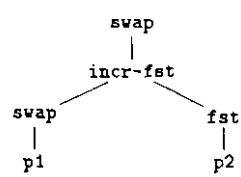
Methods defined with IMAs are also called **derived methods**.

2.2.4.1 Order of Evaluation

Method expressions are evaluated bottom-up. This may be understood by examining the **parse tree** of a term. For example, the parse tree of

```
swap(incr-fst(swap(p1),fst(p2)))
```

is:



[8]Thus, unlike the situation in some other object-oriented languages, in FOOPS there is no need for "functions" with side effects.

Bottom-up evaluation begins by first evaluating the leaves of the tree and then proceeding upwards recursively, carrying the results obtained in lower levels As methods effect state changes, each method in a method expression is (possibly) evaluated in a different context. Other evaluation strategies, such as top-down or mixed, would not be appropriate because symbolic method execution does not make sense for objects: a method must have real arguments before it can produce a real state change[9]. However, the order in which the arguments to a method are evaluated is not fixed, and any particular order may be declared as a property of the method. The syntax for this is similar to that for functions, which was given in Section 2.1.8; default evaluation orders are also similarly determined.

2.2.5 Deleting Objects

FOOPS provides every class with a method to delete objects; it gives as result the void object associated with the class. For a class `C`, its syntax is:

```
me remove_ : C -> C? .
```

If `Blackwells` is the identifier of an object of class `Bookstore`, then

```
remove Blackwells
```

yields `void-Bookstore`. In addition, `remove_` has the effect of replacing every occurrence of the supplied identifier in other (complex) objects with the same void object it returns. For example, consider a class of persons with an attribute

```
at spouse : Person -> Person .
```

and two objects, with identifiers `John` and `Susan`, such that `John` is the spouse of `Susan` and vice versa. After executing

```
remove John
```

`spouse(Susan)` yields `void-Person`.

Finally, entry-time objects cannot be deleted.

2.2.6 Invalid Object Identifiers

It is possible for methods and attributes to remain unevaluated in an expression if no rewrite rules apply to them. This might be problematic when an attribute or method is supposed to evaluate to an object identifier to be passed as argument to another attribute or method. Then what these would receive would not be a valid object identifier. By way of illustration, consider the following declarations for a class of buffers of fixed size (given by the constant `bound`):

[9] However, there exist symbolic execution systems similar to FOOPS in which method expressions remain partially unevaluated until certain conditions become true. For example, see the proposals for object-oriented concurrent execution in [9] and [76].

```
class BoundedBuffer .
at current-size : BoundedBuffer -> Nat .
me put : BoundedBuffer Nat -> BoundedBuffer .
me get : BoundedBuffer -> Nat .
var B : Buffer .  var N : Nat .
cax put(B,N) = ...  if current-size(B) < bound .
```

(The details of the axiom for put are not important here.) Now assume that Buff is the identifier of a buffer that is full. Since there are no axioms that specify the behaviour of put on full buffers, a term such as

```
put(Buff,5)
```

will not evaluate any further. If this term happens to be part of a larger term, e.g.,

```
get(put(Buff,5))
```

then unless the axioms describing get explicitly test for this kind of situation (see below), the evaluation of the term fails because there is no object with identifier put(Buff,5).

Since it may be of interest to try to do something about an invalid object identifier, FOOPS provides every class with a Bool-valued method that tests whether there exists an object with a certain identifier. For a class C, its syntax is:

```
me exists?_ : C -> Bool .
```

2.2.7 Redefinition and Dynamic Binding

Class inheritance is more flexible when the methods associated with a subclass can redefine, or override, those associated with its superclasses. FOOPS and other object-oriented languages that support redefinition come equipped with a run-time mechanism that selects the most specific version of a method, based on the class of the object to which the method is applied. This is called dynamic binding. Redefinitions in FOOPS are given just by introducing a new syntactic declaration, which must include the **redef** property, as in

```
me insert : DoublyLinkedList Elt -> DoublyLinkedList [redef] .
```

where Elt would be the class (or the sort) of the elements stored in the list. Regularity must of course still be obeyed, so that unique least parses exist. However, a method and its redefinition may behave differently. FOOPS also allows attributes to be redefined; the same syntactic restrictions apply. (Our prototype implementation issues warnings when a declaration is a redefinition attempt but does not include the **redef** property.)

Next we give a concrete example, in two parts, that involves redefining both attributes and methods. First, consider a class Linkable of objects whose state consists of a natural number and a "pointer" to another object of class Linkable, as would be used for defining linked lists:

```
omod NAT-LINKABLE is
  class Linkable .
  protecting NAT .
  at value_ : Linkable -> Nat .
  at next_  : Linkable -> Linkable .
endo
```

Module NAT-BILINKABLE below defiues a subclass of Linkable called BiLinkable whose objects store. in addition, the identifier of a "previous" bilinkable object, as would be used for defining doubly-linked lists. So that linkables and bilinkables are not mixed inadvertently, attribute next_ is redefined:

```
omod NAT-BILINKABLE is
  class BiLinkable .
  extending NAT-LINKABLE .
  subclass BiLinkable < Linkable .
  at next_ : BiLinkable -> BiLinkable [redef] .
  at prsv_ : BiLinkable -> BiLinkable .
endo
```

These declarations specify that objects of class Linkable have two attributes, value_ and next_, of coarity Nat and Linkable, respectively, while objects of class BiLinkable have three attributes, value_, next_ and prev_, of coarity Nat, BiLinkable and BiLinkable, respeetively.

Second, below we provide a versiou of NAT-LINKABLE that declares three methods, one that replaces the value stored in value_, another that replaces the value stored in next_, and a third that inserts a linkable in betweeu two others.

```
omod NAT-LINKABLE is
  class Linkable .
  protecting NAT .
  at value_ : Linkable -> Nat .
  at next_  : Linkable -> Linkable .
  me replace-value     : Linkable Nat -> Linkable .
  me replace-next      : Linkable Linkable -> Linkable .
  me put_between_and_ : Linkable Linkable Linkable -> Linkable .
  vars L L1 L2 : Linkable .  var X : Nat .
  ax value(replace-value(L,X)) = X .
  ax  next(replace-next(L,L2)) = L2 .
  ax put L between L1 and L2 = replace-next(L1,L) ;
                               replace-next(L,L2) .
endo
```

Now follows a corresponding version of NAT-BILINKABLE in which put_between_and_ and replace-next are redefined so that prev_ is also updated. Furthermore, note the

inclusion of (auxiliary) methods with a syntax similar to that for assignment in imperative languages.

```
omod NAT-BILINKABLE is
  class BiLinkable .
  extending NAT-LINKABLE .
  subclass BiLinkable < Linkable .
  at next_ : BiLinkable -> BiLinkable [redef] .
  at prev_ : BiLinkable -> BiLinkable .
  me next_:=_          : BiLinkable BiLinkable -> BiLinkable .
  me prev_:=_          : BiLinkable BiLinkable -> BiLinkable .
  me replace-next      : BiLinkable BiLinkable -> BiLinkable [redef] .
  me replace-prev      : BiLinkable BiLinkable -> BiLinkable .
  me put_between_and_  : BiLinkable BiLinkable BiLinkable
                       -> BiLinkable [redef] .
  vars L L1 L2 : BiLinkable .
  ax next(next L := L2) = L2 .
  ax prev(prev L := L2) = L2 .
  ax replace-next(L,L2) = prev L2 := L ; (next L := L2) .
  ax replace-prev(L,L2) = replace-next(L2,L) ; L .
  ax put L between L1 and L2 = replace-next(L1,L) ;
                               replace-prev(L,L2) .
endo
```

Run-time problems of the kind described in the previous section occur if no axioms are given to specify the behaviour of methods and derived attributes that redefine others.

A further possibility in FOOPS is for a derived attribute to be redefined into a stored one. This gives more flexibility to subclasses in choosing whether something that was previously computed on demand can now be stored and updated when appropriate. Note that the opposite should not be allowed: an inherited method could attempt to directly update an attribute whose value was no longer stored.

2.2.7.1 Accessing Original Versions

The redefinition of a derived attribute or a method may access the original version by using a qualification notation similar to that for disambiguating parses (presented in Section 2.1.3). For instance, a method `debit` on bank accounts (of class `Acct`) might be redefined for minimum-balance accounts (of class `MBAcct`), so that those withdrawals that would leave the balance below the required minimum are not accepted. The axiom for the redefinition would be something like this:

```
ax debit(A,AMOUNT) = (debit(A,AMOUNT)).Acct
                     if balance(A) - AMOUNT >= minbalance(A) .
```

where `A` is a variable of class `MBAcct`.

2.2.7.2 Discussion

While FOOPS adopts a so-called "variant" syntactic restriction on redefinitions (i.e., that signatures remain regular), this choice is not innocuous for type safety, in the following sense. Consider, for example, the method `replace-next` above. A call to it for an object of class `BiLinkable` selects the redefinition. But what if dynamically its second argument happens to be of class `Linkable`? The FOOPS approach is to insert a retract *at run time* on the second argument. On the other hand, langnages without retracts take this to be a fatal run-time type error. An example is Eiffel [78], although a recent proposal suggests that system-level validity checks can signal the potential occurrence of this kind of error.

An alternative syntactic restriction that does not entail retracts or possible type errors is called "contra-variant," and is different in that the new arity must be greater than the old arity except on the argument position that determines the class of the attribute or method. For example, the redefinition of `replace-next` for bilinkables would be invalid under contra-variance because the second argument goes in the opposite direction. FOOPS adopts variance for two reasons. First, it appears to capture the more common situation in practice [23, 78]. Second, it obeys the algebraic semantics of FOOPS. We only know of one language that adopts contra-variance: Trellis [102]; many others require that arities and coarities be equal except on the argument position that determines the class of the attribute or method, and this is also safe[10]. See Section 6.4 for more information about redefinition facilities in other languages.

2.2.8 Inheritance Diagrams

The diagrams of Section 2.1.4 generalise to the object level by considering sorts to be classes and functions to be either attributes or methods. Furthermore, note that it is possible for a method to be a merge and a redefinition at the same time (in fact, merging for attributes and methods implies redefinition). One additional diagram exists at the object level, and is shown in Figure 2.4. In it a method `m` of class `A` is redefined for objects of class `C`. This creates an ambiguity whenever `m` is applied to an object of class `D`, and which cannot be solved simply by qualification. This is because in a context expecting an object of class `A`, an object of class `D` can be placed, but if `m` is applied to this object, which of the two `m`'s should be chosen? It is not appropriate to require some form of qualification there, as (for example) such contexts may exist in code that does not pertain the designer of `D`. (With module importation, inheritance hierarchies may include classes from different modules; in fact, this is the typical case. See the next chapter.) Also, the qualification notation that would be needed appears to be excessively complex, as it would need to be applicable for all such possible `D`'s. Nevertheless, the worse problem is that it contradicts the goal of incremental software development. Therefore, this situation is an error in FOOPS, and some corrective action is required. The simplest option would be to merge `m` in `D`. However, if this is not appropriate then it might be possible to rename one of the `m`'s. The next chapter explains how this can be done.

[10] This option is compatible with the algebraic semantics of FOOPS

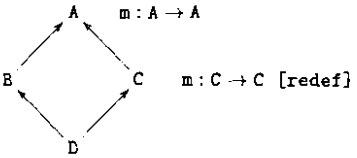


Figure 2.4: Conflicting redefinition in multiple inheritance.

2.3 Summary

This chapter has provided a detailed presentation of the main programming unit of FOOPS, the module. We explained the syntax and (informal) meaning of sorts and classes, and of functions, attributes, methods and axioms. We also gave an intuitive description of the models of computation at the functional and object levels of FOOPS, both of which are based on term rewriting. Several small examples illustrated the features of the language; more substantial examples are given in the next chapter and in one of the appendices.

FOOPS provides snpport for all the basic concepts of object-orientation; moreover, it distinguishes between values and objects and between modules and classes, and supports overloading and mixfix syntax for operations. Also, object creation is very flexible and convenient for initialising objects with complex structure.

This chapter's contributions to the development of FOOPS include derived and multi-argument attributes, explicit defaults, optional unique identifiers for objects, and in general, the detailed explication of the language features. Furthermore, based on the semantics of the language, we examined sort and class inheritance situations and conflicts, which is something that had not been done before.

The features of FOOPS presented here are further enhanced for *large-grain* programming and design by the facilities introduced in the next chapter, such as theories, views, module hierarchies, generic modules with semantic interfaces, and an information hiding mechanism.

Chapter 3

Module Reuse and Interconnection

The engineer may decide to copy as many seemingly good features as he can from existing designs that have successfully withstood the forces of man and nature, but he may also decide to improve upon those aspects of prior designs that appear to be wanting.

— Henry Petroski

Parameterised programming [35, 36] is a design technique whose aim is to increase the reusability, reliability and understandability of software modules by providing facilities for

- organising modules into hierarchies;
- declaring module properties;
- specifying how a module satisfies some properties;
- parameterising modules in order to broaden their domain of applicability;
- renaming module features so that modules can be adapted to new contexts;
- distinguishing between vertical and horizontal structuring;
- specifying module interconnections with semantic interfaces; and,
- composing modules to create new modules or actual systems.

This chapter describes the realisation of the above facilities in FOOPS and their application to (object-oriented) software design. We also discuss the capture of higher-order capabilities in a first-order setting, rules of encapsulation, and information hiding mechanisms. Additionally, several examples demonstrate the advantages of distinguishing between class and module inheritance (but Chapter 6 gives a more detailed discussion). As in the

previous chapter, we include simplified syntactic descriptions; the full syntax for the language appears in Appendix A. We believe that the combination of all these facilities for large-grain programming generalises—and in several ways clarifies—existing approaches to object-oriented programming and design.

3.1 Module Hierarchies

Modules in FOOPS can import other modules, and this gives rise to a kind of multiple inheritance at the module level that is quite different from sort or class inheritance. In simple terms, module importation is the inclusion of the declarations of one module into another. However, FOOPS provides four kinds of module importation modes for declaring how a module respects the semantics of the modules it imports. Their syntax is

Syntax 3.1 (Module Importation Modes)

{`protecting` | `extending` | `using` | `including`} ⟨*ModId*⟩ .

These modes can be abbreviated to `pr`, `ex`, `us` and `inc`, respectively. □

The **importation mode** of a module `M'` in a module `M` is

- `protecting` if `M` neither adds new items to nor identifies items of sorts or classes from `M'`;
- `extending` if `M` does not identify items of sorts or classes from `M'`; or,
- `using` or `including` otherwise.

Examples with numbers are usually the best for illustrating these modes. Consider the natural numbers in Peano notation:

```
fmod PEANO is
  sort Nat .
  fn 0 : -> Nat .
  fn s_ : Nat -> Nat .
  fn _+_ : Nat Nat -> Nat [id: (0) assoc comm] .
  fn _*_ : Nat Nat -> Nat [id: (s 0) assoc comm] .
  vars M N : Nat .
  ax (s N) + (s M) = s s (N + M) .
  ax 0 * N = 0 .
  ax (s N) * (s M) = s (N + M + (N * M)) .
endf
```

Now this module:

```
fmod PEANO-OMEGA is
  ex PEANO .
  fn omega : -> Nat .
endf
```

It extends PEANO because the constant omega is a new element of sort Nat. This other module also extends PEANO:

```
fmod PEANO-EXTRA is
  ex PEANO .
  fn times-2 : Nat -> Nat .
endf
```

because terms such as `times-2(s s 0)` denote new natural numbers (from the point of view of PEANO). However, the next module protects PEANO because it gives `times-2` a definition in terms of old elements of sort Nat:

```
fmod PEANO-EXTRA2 is
  pr PEANO .
  fn times-2 : Nat -> Nat .
  var N : Nat .
  ax times-2(N) = s s 0 * N .
endf
```

Therefore, in this module a term such as `times-2(s s 0)` does not denote a new natural number as it is equal to `s s s s 0`. A module that simply uses the "services" of another protects it too; for example:

```
fmod PEANO-LIST is
  sort List .
  pr PEANO .
  fn nil  : -> List .
  fn cons : List Nat -> List .
endf
```

The naturals modulo N can be defined by importing PEANO and adding an axiom that equates 0 to N. For `N = s s 0`, we have:

```
fmod PEANO-MOD2 is
  using PEANO .
  ax s s 0 = 0 .
endf
```

The mode is `using` because 0 and `s s 0` were not previously equal or identified.

Finally, the `including` mode is only for importing theories, and will be discussed in the sections that follow.

Of course, importation modes cannot be automatically checked for correctness; their purpose is to document the way in which modules are used.

A further way of extending a module is to declare new subsorts or subclasses, because their values and objects (respectively) can also be seen as belonging to their supersorts and superclasses. The following example illustrates this.

Example 3.2 Module ACCT below declares a class `Acct` of bank accounts, with attributes `bal_` (for balance) and `hist_` (for transaction history) and methods `debit` and `credit` for the usual operations on accounts. The module HIST declares the sort of the transaction histories, which are lists of 2-tuples whose first component is a date and whose second component is an amount of money (of sort `Money`). Lists use juxtaposition syntax (i.e., `__`) for append and the function `hd` to select the first element of a list; 2-tuples are constructed with syntax `<< _ ; _ >>`. Module HIST is extended by ACCT because of the declaration of the function `insufunds_`, which is used in transaction histories to indicate withdrawal attempts that an account could not support. Module NOW declares a class of objects with an attribute that stores a date; in particular, it declares an object with identifier `Today` which is used to hold the current date. (Modules HIST and NOW are given in Appendix B.)

```
omod ACCT is
  class Acct .
  ex HIST .
  pr NOW .
  at bal_  : Acct -> Money [default: (0)] .
  at hist_ : Acct -> Hist .
  var A : Acct .  var M : Money .
  me credit : Acct Money -> Acct .
  ax  bal credit(A,M) = bal A + M .
  ax hist credit(A,M) = << date(Today) ; M >> hist A .
  fn insufunds_ : Money -> Money? .
  me debit : Acct Money -> Acct .
  cax  bal debit(A,M) = bal A - M                          if M <= bal A .
  cax hist debit(A,M) = << date(Today) ; - M >> hist A  if M <= bal A .
  cax hist debit(A,M) = << date(Today) ; insufunds(M) >> hist A
                        if M > bal A .
endo
```

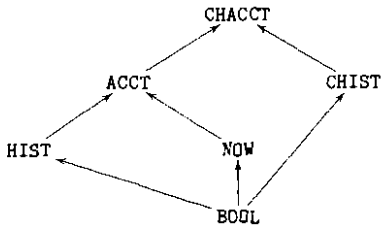
Next, module CHACCT extends ACCT by declaring a subclass `ChAcct` of `Acct`. Cheque accounts have an additioual attribute for recording the cheques written; the sort of this attribute comes from module CHIST (which is also given in Appendix B). Cheque histories are lists of triples with each entry containing a cheque number, a date and an amount; `1*_` is the selector function for the first component of a triple (whose sort is `3Tuple`). Because cheque accouuts are accounts not available from ACCT, CHACCT extends it.

```
omod CHACCT is
```

```
class ChAcct .
ex ACCT .
subclass ChAcct < Acct .
pr CHIST .
at chist_ : ChAcct -> Chist .
var C : ChAcct .  var CH : Chist .  var M : Money .
var 3TUP : 3Tuple .
--- an auxiliary to compute the next check number
fn nextchk_ : Chist -> Nat .
ax nextchk CH = if CH == emptyChist then 1 else 1 + 1st hd CH fi .
--- an auxiliary to add a new entry to the cheque history
me app-chist : ChAcct 3Tuple -> ChAcct .
ax chist app-chist(C,3TUP) = 3TUP (chist C) .
fn badch_ : Money -> Money? .
me writechk : ChAcct Money -> ChAcct .
cax writechk(C,M) = app-chist(C,<< nextchk chist C ;
                                  date(Today) ; badch(M) >>)
                    if M > bal C .
cax writechk(C,M) =
  debit(C,M); app-chist(C,<< nextchk chist C ; date(Today) ; M >>)
                    if M <= bal C .
endo
```

□

It is useful to document the **context** of a module by drawing a graph that shows all of the modules on which it relies. The following picture gives the context for CHACCT (and also for ACCT)[1]:



Here, an arrow from B to A indicates that B is imported by A. A salient characteristic of the above kind of module importation is that it is cumulative (also called transitive); for example, CHACCT also imports BOOL, HIST and NOW, although indirectly. Furthermore, it is important that modules that are multiply imported are **shared**; for example, ACCT includes

[1] Actually, the context of CHACCT would also include all the other submodules of HIST, CHIST and NOW.

only one copy of the booleans, even though it imports BOOL indirectly via two different submodules. Later we present facilities that allow modules to import others privately, thus blocking transitivity.

To allow for the graceful integration of modules that might be developed independently, FOOPS associates with each sort, class, function, attribute and method the module which declares it. For example, the full name of class Acct is Acct.ACCT, as discussed in the previous chapter. When modules are imported in protecting or extending mode, their features retain this association with their module of origin. However, certain situations call for the textual copy of the features of one module into another, and this is provided in FOOPS by the using mode of importation. (This should not really be surprising, because using makes no guarantees about preserving the semantics of imported modules.) Several other applications of using will be discussed in later sections. Here we just note that for executable modules, protecting and extending are the most common importation modes.

3.1.1 Principal Constants

In Section 2.2.3.2 we explained the role of principal constants and entry-time objects in the computation of default values for object attributes. In the presence of module inheritance, the principal constant of a sort in a module M is the first declared constant of that sort in M. If no constants are declared in M, then the principal constant is the first declared constant of that sort in the module from which M gets that sort, and so on recursively. The determination of which entry-time object to use as default for an attribute is similar.

3.2 Theories

At both the functional and the object levels, FOOPS provides special kinds of modules called **theories** whose purpose is to declare syntactic and semantic properties to be satisfied by other modules. The structure of theories is the same as for the other kind of module that we have been using, in that they may declare sorts, classes, functions and so on; also, theories can import other modules and can be parameterised, as will be discussed further below. Theories constitute the purely declarative side of FOOPS, and support high-level design and specification.

Because they classify other theories and modules by the properties that they satisfy, theories can be seen as types [39]. This view extends the previous notions of "types as sets" and "types as algebras" because theories are types that range over modules and that exist only at system or module design time.

In addition, as theories are not meant for specifying executable code, the form of their axioms is not restricted as for executable modules (see sections 2.1.5, 2.2.2 and 2.2.4). Moreover, it is not sensible to distingnisb between stored and derived attributes. In the following, we will use "module" to refer to both theories and executable modules.

Syntactically, functional theories are declared with the keyword pair **fth** ... **endfth** and object theories with the pair **oth** ... **endoth**:

Syntax 3.3 (Theories)

```
fth ⟨ModId⟩ is ⟨fModElt⟩... endfth
oth ⟨ModId⟩ is ⟨oModElt⟩... endoth
```

□

The simplest theories in FOOPS declare a single sort or class:

```
fth TRIV is
  sort Elt .
endfth

oth TRIVC is
  class C .
endoth
```

Both of these theories are automatically available in FOOPS.

The following theory describes partially ordered sets (or posets), in anti-reflexive form:

```
fth POSET is
  sort Elt .
  fn _<_ : Elt Elt -> Bool .
  vars X Y Z : Elt .
  ax X < X = false .
  ax X < Y implies not Y < X = true .
  ax (X < Y and Y < Z) implies (X < Z) = true .
endfth
```

For theories, the only importation modes that make sense are `using` and `including`, because theories do not refer to any fixed number of items or elements of sorts or classes, but to any module that satisfies their axioms and that provides the syntactic structure they declare. The next theory defines totally ordered sets, or posets in which every pair of distinct elements is related; it imports `POSET`:

```
fth TOSET is
  using POSET .
  vars X Y : Elt .
  cax X < Y or Y < X = true if X =/= Y .
endfth
```

The difference between these importation modes is that `using` copies text but `including` does not, which is useful for situations in which subtheories need to be shared. The next section provides a full example.

On the other hand, if some theory `T` is to require the inclusion of an "uncorrupted" version of the natural numbers, for example, it would import `NAT` in protecting mode:

```
oth T is
  ...
  pr NAT .
  ...
endoth
```

Because NAT is not a theory, modules satisfying T must provide the natural numbers as specified by NAT. Incidentally, BOOL is also automatically imported into every theory (in protectiug mode).

As a final example, the purpose of theory ITER-ACTIONS given further below is to declare the *variable* parts of an iteratiou construct such as the while loop, which (in general) has the following form:

initialisation;
while *test* **do**
    some-action
endwhile;
wrapup;

The text in italics denotes the variable parts; we will consider them to be methods. This example concerns loops over data structures, so each of these methods will have two arguments, one for the data structure and one for any other input. The theory is:

```
oth ITER-ACTIONS is
  classes C In Out .
  me init   : C In -> C .
  me action : C In -> C .
  me wrapup : C In -> Out .
  me test   : C In -> Bool .
endoth
```

The class Out denotes the result of the iteration. In the next section we use this theory to specify while loops in FOOPS.

3.3 Abstract Classes

Abstract classes function like templates for classes, in that they declare some methods and attributes that must be defined in their subclasses, and may also introduce some new methods defined in terms of these given ones. For example, in Eiffel [77], a class declared as abstract ("deferred") can have methods that do not have executable code; in C++ and Ada 9X [2], a class is abstract if any of its methods is not defined. Abstract classes are used for high level design. They are not generic, and are not meant to be instantiated, but rather to have their deferred methods and attributes defined in different ways by different subclasses.

In parameterised programming, we can provide this capability by defining an abstract class in a theory, and then importing that theory into executable modules, where it is enriched with subclasses that provide executable definitions for deferred methods. The advantage of this is that it does not require any new language features. For example, we could have begun specifying the bank accounts example above with an abstract class theory that captures the basic properties of accounts:

```
oth ACCT is
  class Account .
  pr MONEY .
  at balance : Account -> Money .
  me debit   : Account Money -> Account .
  me credit  : Account Money -> Account .
  me transfer_from_to_ : Money Account Account -> Account .
  vars A A' : Account .  var M : Money .
  ax transfer M from A to A' = debit(A,M); credit(A',M) .
endoth
```

Note that `debit` and `credit` are declared but not defined, while the transfer method is defined using them.

The following modules define two different subclasses of `Account`, where each would provide executable definitions for `debit` and `credit`:

```
omod SAVINGS-ACCT is
  class SavAccount .
  including ACCT .
  subclass SavAccount < Account .
  at interest-rate : SavAccount -> Float .
  me debit  : SavAccount Money -> SavAccount .
  me credit : SavAccount Money -> SavAccount .
  ... axioms for debit and credit and other declarations ...
endo

omod CHEQUE-ACCT is
  class ChAccount .
  including ACCT .
  subclase ChAccount < Account .
  me debit  : ChAccount Money -> ChAccount .
  me credit : ChAccount Money -> ChAccount .
  ... axioms for debit and credit and other declarations ...
endo
```

Note the use of the `including` mode of importation, because we do not want a different copy of `ACCOUNT` in each the last two modules; otherwise, `SavAccount` and `ChAccount` would

not have a common superclass (`SavAccount` would be a subclass of `Account.SAVINGS-ACCT` and `ChAccount` a subclass of `Account.CHEQUE-ACCT`).

Since class `Account` is abstract, it will have no objects that do not belong to a proper subclass; i.e., objects of class `Account` cannot be created directly. This is a natural consequence of the semantics of theories. Also, it is required that axioms of theories used for declaring abstract classes have an executable form.

3.4 Parameterised Modules

Besides their use for high-level specification, theories are also used to declare interface requirements for **parameterised** (or **generic**) **modules**. A parameterised module has certain parts that are fixed and certain others that are variable, and these others are specified by the **formal** parameters of the module, which themselves denote modules. Parameterisation is a way of capturing commonality and factoring out change, and allows modules to be specialised by providing different sets of **actual** parameters; because of this, parameterisation is said to **broaden** the domain of applicability of a module [114].

Because modules in FOOPS are generic over other modules, module **instantiation** (also called **actualisation**) combines not just one sort, class or operation, but logically related groups of these, yielding a kind of higher-order composition at the module level. Subsequent sections provide further discussion and examples regarding this aspect of FOOPS. In this section we are concerned with how to specify generic modules.

The following simple module for lists is generic over the kind of elements they hold:

```
fmod LIST[X :: TRIV] is
  sort List .
  protecting NAT .
  fn nil : -> List .
  fn __ : Elt List -> List .
  fn length_ : List -> Nat .
  var E : Elt .  var L : List .
  ax length nil = 0 .
  ax length(E L) = 1 + length L .
endf
```

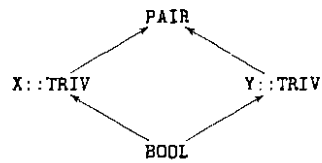
The text between `LIST` and `is` is called the module's **interface**, and indicates that valid actual arguments for `LIST` are modules that satisfy the theory `TRIV`, i.e., any module with at least one sort; `X` is a variable as in ordinary programming notation (recall the analogy between theories and types).

Generic modules can of course have more than one parameter:

```
omod PAIR[X :: TRIV, Y :: TRIV] is
  class Pair .
  at fst_ : Pair -> Elt.X .
  at snd_ : Pair -> Elt.Y .
```

```
  at equal : Pair Pair -> Bool .
  me replace-fst : Pair Elt.X -> Pair .
  me replace-snd : Pair Elt.Y -> Pair .
  vars P P2 : Pair .  var V1 : Elt.X .  var V2 : Elt.Y .
  ax equal(P,P2) = fst P == fst P2 and snd P == snd P2 .
  ax fst replace-fst(P,V1) = V1 .
  ax snd replace-snd(P,V2) = V2 .
endo
```

Parameterised modules include the theories over which they are generic; thus, their contexts are similar to those shown previously. For example, this is PAIR's context:



It is important that the two requirement theories are distinguished, because otherwise they would be shared, and this would not allow for pairs with different kinds of components. However, we will omit the qualification used in the above diagram when there is no room for confusion.

The theory ITER-ACTIONS of Section 3.2 can be used to specify the interface for a module that defines while loops :

```
omod WHILE[X :: ITER-ACTIONS] is
  mes while while-continue : C In -> Out .
  var E : C .  var I : In .
  ax while(E,I) = init(E,I); while-continue(E,I) .
  ax while-continue(E,I) =
        if test(E,I) then
            action(E,I); while-continue(E,I)
        else
            wrapup(E,I)
        fi .
endo
```

An actual argument to WHILE must provide at least all the functionality declared by ITER-ACTIONS. This module exemplifies one difference between class and module inheritance, because of the derived methods it declares; if modules were classes, WHILE would be a class and the two methods would be associated with this new class.

Theories can also be generic, as illustrated in the following example.

Example 3.4 The parameterised theory PW-ENGINE declares a class PWEngine of "password generators" and is generic over total orders. A password generator is an object which stores passwords generated by one of its methods. The axiom says that passwords must be generated in increasing order, which guarantees that they are always unique. Note the form of the axiom, which does not specify an executable pattern as defined in the previous chapter; rather, it just declares a property that any implementation must satisfy.

```
oth PW-ENGINE[X :: TOSET] is
  class PWEngine .
  at value    : PWEngine -> Elt .
  me make-pw : PWEngine -> PWEngine .
  var P : PWEngine .
  ax value(P) < value(make-pw(P)) = true .
endoth
```

□

The full syntax of parameterised modules is the following:

Syntax 3.5 (Parameterised Modules)

```
fmod ⟨ModId⟩⟨ModInterface⟩ is ⟨fModElt⟩... endf
fth ⟨ModId⟩⟨ModInterface⟩ is ⟨fModElt⟩... endfth

omod ⟨ModId⟩⟨ModInterface⟩ is ⟨oModElt⟩... endo
oth ⟨ModId⟩⟨ModInterface⟩ is ⟨oModElt⟩... endoth

⟨ModInterface⟩ ::=
     [ [ ⟨ModId⟩... :: ⟨ModExp⟩ {, ⟨ModId⟩... :: ⟨ModExp⟩}... ] ]
```

where ⟨*ModExp*⟩ denotes *module expressions*, which include module instantiations and names of non-generic modules; the sections that follow will present other kinds of module expressions that build upon these. □

3.5 Views and Instantiation

Perhaps the most important feature of the module system of FOOPS is the view, which is used to express how a module satisfies a theory. A view is a binding of the items in a module to the items in a theory, such that the theory's axioms are *behaviourally* satisfied by the module, in the sense of Section 3.5.2 below. The most immediate application of views is to instantiate generic modules, but they can also be used to express relationships of refinement between modules.

In general, there may be more than one view between a theory and a module, because modules may satisfy theories in multiple ways; also, a module may simultaneously satisfy various theories. This extends the capabilities of most other languages for associating

interfaces (also called "specifications") with actual modules (also called "bodies"); Chapter 6 provides the details of this comparison.

An **explicit view** has a name, designates its **source** theory and its **target** module, and supplies a mapping that covers every sort, class, function, attribute and method in its source. For example, there is always a view from TRIV to any non-empty module, such as BOOL:

```
view BOOL-IS-TRIV from TRIV to BOOL is
  sort Elt to Bool .
endv
```

The naturals under less-than form a poset, thus

```
view V1 from POSET to NAT is
  sort Elt to Nat .
  fn _<_ to _<_ .
endv
```

The naturals under greater-than also form a poset:

```
view V2 from POSET to NAT is
  sort Elt to Nat .
  fn _<_ to _>_ .
endv
```

To reduce tediousness and capture "obvious" mappings, FOOPS offers a set of conventions by which views can be **abbreviated**. These are:

(1) Any sort or class pair `x to x` can be omitted.

(2) A sort or class pair `x to y` can be omitted if `x` and `y` are both principal sorts or principal classes[2].

(3) Any function, attribute, or method pair `o to o` can be omitted if, under the view, the rank of the first is the same as the rank of the second.

For example, because Nat is the principal sort of NAT, V1 and V2 may be abbreviated to

```
view V1 from POSET to NAT is
  fn _<_ to _<_ .
endv

view V2 from POSET to NAT is
  fn _<_ to _>_ .
endv
```

[2]The principal sort of a module is the first sort that it mentions; the principal class of a module is the first class that it mentions.

Moreover, because of (3), V1 can be reduced to a so-called **default** or **null view**, which has an empty body:

```
view V1 from POSET to NAT is
endv
```

It is also possible to map functions, attributes and methods to expressions, providing a very flexible capability. For example, the naturals under "divides but not equal" also form a poset, but NAT does not declare a single function that tests for both of these properties. However, such a function can be constructed as part of the view:

```
view NATD from POSET to NAT is
  sort Elt to Nat .
  vars E1 E2 : Elt .
  fn E1 < E2 to (E1 divides E2 and E1 =/= E2) .
endv
```

This example shows that views can declare variables that take their values in sorts of the source theory. Note also the implicit mapping of these variables into variables of the same name and corresponding sort in the target module: in E1 < E2 the variables are of sort Elt, while in (E1 divides E2 and E1 =/= E2) they are of sort Nat. Views can declare class variables in a similar fashion.

Perhaps unexpectedly, bank accounts can be seen as implementing counters, which we define as follows:

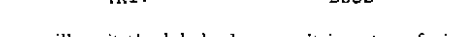
```
oth COUNTER is
  class Counter .
  pr NAT .
  at value : Counter -> Nat .
  mes inc dec : Counter -> Counter .
  var C : Counter .
  ax value(inc(C)) = s value(C) .
  ax value(dec(C)) = if value(C) == 0 then 0 else p value(C) fi .
  ax value(dec(inc(C))) = value(C) .
endoth
```

Observe that the last axiom, which expresses a fundamental property of counters, would not be valid in an executable module: it is neither a direct method axiom or an indirect method axiom. Here is the view:

```
view ACCT-IS-COUNTER from COUNTER to ACCT is
  var C : Counter .
  at value to bal_ .
  me inc(C) to credit(C,1) .
  me dec(C) to debit(C,1) .
endv
```

It is interesting to note that if the axiom `value(dec(inc(C))) = value(C)` was changed to `dec(inc(C)) = C`, then ACCT would not satisfy it, because **debit** and **credit** update not only balances but also transaction histories. In other words, a credit followed by a debit would leave an account's balance unchanged, but its transaction history is different from the original one. See Section 3.5.2 for more on the satisfaction of axioms.

We also draw context graphs for views, which we denote with labelled dashed arrows from the source to the target of the view. For example,



In future diagrams, we will omit the label whenever it is not confusing.

There can also be views between parameterised modules, or **parameterised views**. For example, consider the following theory for "container" data structures. Among others, it specifies operations for inserting and deleting elements, and for testing the presence of an element in a container

```
oth CONTAINER[X :: TRIV] is
  class Container .
  pr NAT .
  at size   : Container -> Nat .
  at member : Container Elt -> Bool .
  at empty  : Container -> Bool .
  mes insert delete : Container Elt -> Container .
  var C : Container .  vars E E' : Elt .
  ax size(insert(C,E)) = size(C) + 1 .
  ax member(insert(C,E),E') =
       if E == E' then
           true
       else if empty(C) then
           false
       else
           member(C,E')
       fi fi .
  ax size(delete(C,E)) =
       if member(C,E) then p size(C) else size(C) fi .
  ax member(delete(C,E),E') = member(C,E') .
  ax empty(C) = (size(C) == 0) .
  ax member(new.Container(),E) = false .
  ax size(new.Container()) = 0 .
endoth
```

Observe that the theory is not a full implementation because it says nothing about how containers store their elements. Therefore, well-known data structures such as lists, bags,

trees, and hash tables satisfy this theory; however, sets do not satisfy it because inserts do not always increase the size of a set.

Now assume that we have (say) a module LINKED-LIST that implements a class List of linked lists, and that is also generic over TRIV. A view from CONTAINER to LINKED-LIST would need to be parameterised:

```
view LINKED-LIST-AS-CONTAINER[X :: TRIV]
        from CONTAINER[X] to LINKED-LIST[X] is
  class Container to List .
  at size   to length .
  at insert to cons .
  ...
endv
```

The view says that for any module X that satisfies TRIV, there is a view from CONTAINER to LINKED-LIST.

The full syntax of views is the following:

Syntax 3.6 (Views)

```
view ⟨ModId⟩⟨ModInterface⟩ from ⟨ModExpr⟩ to ⟨ModExpr⟩ is
    ⟨ViewElt⟩...
endv

⟨ViewElt⟩ ::= sort ⟨Sort⟩ to ⟨Sort⟩ . | class ⟨Class⟩ to ⟨ClassRef⟩ . |
        [ fn | at | me ] ⟨OpExpr⟩ to ⟨Term⟩ . |
        fn ⟨Fn⟩ to ⟨Fn⟩ . | me ⟨Meth⟩ to ⟨Meth⟩ . |
        at ⟨Attr⟩ to ⟨Attr⟩ . | ⟨oVarDecl⟩... | ⟨fVarDecl⟩...
```

where ⟨*Sort*⟩ and ⟨*Class*⟩ denote possibly qualified sorts and classes; similarly, ⟨*Fn*⟩, ⟨*Attr*⟩ and ⟨*Meth*⟩ denote possibly qualified functions, attributes and methods. ⟨*OpExpr*⟩ is a ⟨*Term*⟩ consisting of a single operation applied to variables, and ⟨*fVarDecl*⟩ and ⟨*oVarDecl*⟩ are variable declarations as given inside modules. □

3.5.1 Module Instantiation

Parameterised modules cannot be used by themselves; rather, they need to be **instantiated** beforehand. Instantiation is the process by which actual parameters are bound to the formal parameters of a generic module and a new module is created. This binding is specified by supplying appropriate views from each formal parameter to its corresponding actual parameter. Instantiations can occur wherever modules are expected; for example, as targets of views.

One form of instantiation consists of giving view names. For example, the generic list module of the previous section can be instantiated to give lists of boolean values, as follows:

```
LIST[BOOL-IS-TRIV]
```

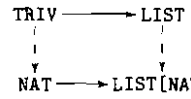
However, views not previously defined can be given "on the fly," as in

```
LIST[view to NAT is sort Elt to Nat . endv]
```

Note that this kind of view does not require a name nor does it need to mention its source, because it is implicit from the parameterised module. Furthermore, there is a convention for specifying default views on the fly, and it consists of simply giving the name of the target module. For example, the default view from TRIV to NAT is implicit in

```
LIST[NAT]
```

The contexts of instantiations include views, as in the following graph:

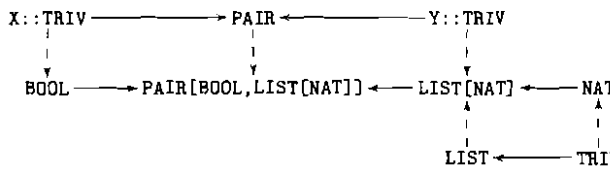


The view from LIST to LIST[NAT] maps the TRIV part of LIST to the NAT part of LIST[NAT], using the view already provided; the other parts of the two modules are mapped with default conventions. Lastly, observe that module importations can be seen as (default) views that describe inclusions.

FOOPS also supports **multi-level instantiation**, so that an actual argument to a generic module can itself be an instantiated generic. For example, the following generates a class of pairs whose first component is a boolean value and whose second component is a list of natural numbers:

```
PAIR[BOOL-IS-TRIV,LIST[NAT]]
```

Note that there are two implicit views here: one from TRIV to NAT, and another from Y::TRIV to LIST[NAT] (which maps Elt to List). The context of the above instantiation is



Views can also be deduced from module elements. For example, the default view from TRIV to NAT is implicit in the following instantiation:

```
omod NAT-LIST is
  pr NAT .
  pr LIST[Nat] .
endo
```

Or, more economically,

```
omod NAT-LIST is
  pr LIST[(Nat).NAT] .
endo
```

This kind of instantiation is particularly useful when the desired default mapping is not between principal sorts (or classes). For example, sort NzNat of NAT is not principal, but the implicit view in this instantiation is between Elt and NzNat:

```
LIST[(NzNat).NAT]
```

As a further example of this facility, consider this instantiation of the generic theory of Example 3.4 (page 61):

Example 3.7 A default view from TOSET to NAT is generated using the function _<_ of NAT:

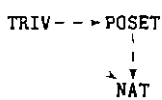
```
oth NAT-PW-GEN is
  us PW-ENGINE[(_<_).NAT] .
endoth
```

This password engine theory is used as part of a specification of resource managers that is given in Appendix B, and in which resources are locked and freed with passwords. Additionally, that appendix includes quite sophisticated instantiations of module WHILE and of other related modules that capture various kinds of iterators. □

A generic module can instantiate other generic modules using some of its own parameters. For example, a module for binary search trees could be specified as follows:

```
omod BSEARCH-TREE[X :: POSET] is
  class Tree .
  pr LIST[X] .
  ...
endo
```

Note that there is an implicit default view from TRIV (the interface theory of LIST) to POSET in the protecting declaration. Then, an instantiation of BSEARCH-TREE, say BSEARCH-TREE[NAT], can be seen as instantiating LIST by *composing* the view from TRIV to POSET with the view from POSET to NAT, which yields a view from TRIV to NAT. View composition is well-defined (see Chapter 4), and corresponds to the composition of graph arrows; for example, the dotted arrow below indicates the preceding composed view:



Alternatively, LIST[X] can be seen as creating a new LIST module whose interface theory is POSET rather than the original TRIV. The above diagram is also suggestive of another application of views: refinement. This is discussed in Section 3.10.

The various forms of instantiation have the following syntax:

Syntax 3.8 (Instantiation with Explicit Views)

⟨*ModId*⟩ [⟨*VId*⟩ {,⟨*VId*⟩}...]

where ⟨*VId*⟩ is the name of a view. □

Syntax 3.9 (Instantiation with On-the-fly Views)

⟨*ModId*⟩ [⟨*ViewArg*⟩ {, ⟨*ViewArg*⟩}...]

⟨*ViewArg*⟩ ::= view to ⟨*ModExp*⟩ is ⟨*ViewElt*⟩... endv

□

Syntax 3.10 (Instantiation with Default Views)

⟨*ModId*⟩ [⟨*Arg*⟩ {, ⟨*Arg*⟩}...] .

⟨*Arg*⟩ ::= ⟨*ModId*⟩ | sort ⟨*Sort*⟩ | class ⟨*Class*⟩ |
fn ⟨*Fn*⟩ | at ⟨*Attr*⟩ | me ⟨*Meth*⟩

where sort, class, etc. are used for disambiguation, if necessary. □

Note that it is possible for a multi-argument instantiation to combine various kinds of views.

3.5.2 Verification of Views

Up to now we have ignored the semantic aspect of views, and just used them to describe syntactic correspondences. We believe that it would be too restrictive on programming practice to always require a formal proof that a view is legitimate; therefore a practical implementation should only require syntactic validity. However, the ability to declare axioms in theories not only helps with documentation and design, it also leaves open the possibility of formal verification for critical applications. We could even use a truth management system to track the soundness of views, which might range from "mechanically verified" to "wishful-thinking," as suggested in [35].

Given a view from a theory T to a module M, the axioms in T are interpreted *behaviourally* (also called *observationally*) in M, i.e., they need only *appear* to be satisfied. For example, a module implementing stacks may satisfy the equation pop push(X,S) = S behaviourally without satisfying it literally. In fact, this happens for the traditional pointer-array implementation of stacks, in which "junk" may be left behind the pointer following a

pop. Because this junk is not reachable anymore, it does not affect the behaviour of stacks. This is illustrated in Figure 3.1, where the leftmost stack first has a 7 push'ed onto it, and then is pop'ed, yielding a new stack state that is different from the original state, but which is behaviourally equivalent to it.

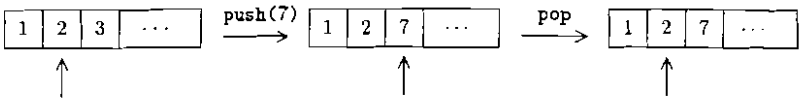


Figure 3.1: Junk after a pop.

3.6 Module Blocks and Higher-Order Composition

Module blocks allow several modules to be declared together; moreover, blocks can be parameterised, and then all modules in the block have the parameterisation of the block (in addition to their own). For example, a module MAP that defines a method map over lists could be declared in the same block as the module that defines lists:

```
block LIST-BLOCK[X :: TRIVC] is
  omod LIST is
    class List .
    at head : List -> Elt .
    ...
  endo

  oth ME is
    me m : Elt -> Elt .
  endoth

  omod MAP[M :: ME] is
    pr LIST .
    me map : List -> List .
    ...
  endo
endo
```

The theory ME declares the interface for MAP, which requires one unary method m on Elt's, and map applies m to each element of its argument. Because the block is parameterised by TRIVC, so are all its modules; additionally, a module inside a parameterised block can be parameterised over other theories, as is MAP. Instantiating LIST-BLOCK, e.g.,

as LIST-BLOCK[NAT], also instantiates all of its modules: one for lists of natural numbers, another that provides a generic map method over those lists, and a third that defines the interface to that generic. Blocks may not be nested.

A significant aspect of blocks is that private items can be used in subsequent modules within a block, but not outside of it. In the above example, this allows map to be implemented using any aspect of the LIST module, rather than just its public features, as would be required if LIST were outside of MAP's block (information hiding capabilities are explained in Section 3.9). Blocks are also useful for organising large specifications, especially if they have significant parts that are similarly parameterised.

Higher order operations (as in Smalltalk) can achieve some of the same functionality as modules that are generic over operations (such as MAP). However, such an approach is small-grained, whereas parameterised programming is large-grained, because it encapsulates operations and properties with the data that they manipulate, and abstracts over complete modules, and even blocks of modules. Moreover, the features of parameterised programming are first order, and thus simpler to reason about (see [37] and [43] for further discussion of this issue).

3.7 Module Expressions

A **module expression** specifies the design of a system (or subsystem) in terms of already given components. We have already seen some generic module instantiations, which are a simple special kind of module expression. Two further operations used to form module expressions are renaming and sum.

Renaming permits module entities to be given new names, which makes it easier to adapt modules to new contexts. For example, if a binary search tree will store indices from a database, it is more natural to call the class Index rather than Tree. This is accomplished using the "*" operator, as in

```
BSEARCH-TREE[STRING-AS-POSET] * (class Tree to Index)
```

which instantiates BSEARCH-TREE and renames the class Tree; here STRING-AS-POSET is a view of strings (the index's keys) as posets. Methods can also be renamed, as in

```
BSEARCH-TREE[STRING-AS-POSET]
    * (class Tree to Index, me insert to add-key)
```

It is possible that complex module expressions lead to the *composition* of renamings. For example, the previous module expression is equivalent to this one, which splits the renamings in two:

```
(BSEARCH-TREE[STRING-AS-POSET] * (class Tree to Index))
    * (me insert to add-key)
```

Also, the module expression

```
(BSEARCH-TREE[STRING-AS-POSET] * (class Tree to Index))
    * (class Index to Tree)
```

is the same as

```
BSEARCH-TREE[STRING-AS-POSET]
```

because the renamings in the first expression cancel each other. (A more realistic example of renaming composition appears in Section 6.3.) However, after simplification, renamings generate new modules, so that `NAT * (sort Nat to Number)` and `NAT` are different modules, and thus the sorts `Nat` and `Number` are not related, for example.

Sum, denoted "+", combines the contents of modules, taking sharing into account, as we illustrate next.

Example 3.11 This example extends Example 3.2 (page 53) by specifying savings accounts and so-called "now" accounts, which provide both savings and cheque facilities. It uses multiple inheritance for classes and modules, and module sum.

```
omod SAVACCT is
  class SavAcct .
  extending ACCT .
  subclass SavAcct < Acct .
  sort Rate .
  subsort Float < Rate .
  --- sort Float (for floating point numbers) comes from HIST
  at rate_ : SavAcct -> Rate .
  var S : SavAcct .
  me pay-interest-to_ : SavAcct -> SavAcct .
  ax pay-interest-to S = credit(S,rate S * bal S) .
endo


omod NOWACCT is
  class NowAcct .
  extending CHACCT + SAVACCT .
  subclass NowAcct < ChAcct SavAcct .
endo
```

Note that `ACCT`, imported through both `CHACCT` and `SAVACCT`, is considered as shared[3]. Another way of seeing "+" is as creating a new module that imports each of its arguments. □

[3]The reader may be wondering why we declared `Rate` to be a supersort of `Float`. The reason can be discovered by examining the typing of the last axiom together with the definition of `Money` given in Appendix B.

More complex module expressions may use multi-level instantiation, default views, renamings and sum. For example, the following module expression describes a parsing stack and a block-structured symbol table:

```
STACK[LIST[TOKEN] * (class List to Sentence)] +
  STACK[TABLE[TUPLE[STRING,TYPE]
            * (class Tuple to Variable,
               at fst to name, at snd to type)]
     * (class Table to Scope)]
   * (class Stack to SymbolTable)
```

With the **make** command, module expressions are "evaluated" (or "executed") to construct new named modules, as in

```
make PARSER is
   ... the previous module expression ...
endm
```

It is the possibility of actually building (i.e., composing) systems that distinguishes module expression evaluation from so-called "module interconnection languages," which merely provide descriptions of the structure of systems. Module composition greatly enhances the ability to reuse software.

The use of module expressions with theories and views may sometimes seem too verbose. However, a single module instantiation can compose many different functions all at once. For example, a generic complex arithmetic module **CPXA** can be easily instantiated with any of several real arithmetic modules as actual parameter [37]:

- single precision reals, `CPXA[SP-REAL]`,
- double precision reals, `CPXA[DP-REAL]`, or
- multiple precision reals, `CPXA[MP-REAL]`.

Each instantiation involves substituting dozens of functions into dozens of other functions. Furthermore, [37] suggests an abbreviated notation that is very similar to that of higher order functional programming, for those cases where one really is just composing functions (this notation uses the facilities described in Section 3.5.1).

Module expressions can be used for qualification in the same way that module identifiers have been used up to now. For example:

```
Tree.(BSEARCH-TREE[STRING-AS-POSET] * (me insert to add-key))
```

Finally, we give the formal syntax for renaming and sum:

Syntax 3.12 (Renaming)

⟨*ModExp*⟩ * (⟨*RenameElt*⟩ {, ⟨*RenameElt*⟩}...)

⟨*RenameElt*⟩ ::= sort ⟨*Sort*⟩ to ⟨*SortId*⟩ | class ⟨*Class*⟩ to ⟨*ClassId*⟩ |
fn ⟨*Fn*⟩ to ⟨*OpForm*⟩ | at ⟨*Attr*⟩ to ⟨*OpForm*⟩ |
me ⟨*Meth*⟩ to ⟨*OpForm*⟩

□

Syntax **3.13** (Sum)

⟨*ModExp*⟩ + ⟨*ModExp*⟩ {+ ⟨*ModExp*⟩}...

□

3.8 Encapsulation Rules

Now that most aspects of the module system of FOOPS have been presented, we digress to describe the encapsulation rules of the language. Encapsulation is the process of packaging information, and has to do with boundaries of definition. The encapsulation rules of the object level of FOOPS establish where the attributes and methods of a class can be declared and where their axioms can be given, and have the effect of localising the description of objects and their potential states; we believe that this makes programs easier to understand and to maintain. The rules are as follows:

(1) Except for those which are inherited, the stored attributes of a class must be declared in the same module as the class is.

(2) If A is a superclass of B, this relationship can be declared only in the same module as B is.

(3) A direct method axiom (see Section 2.2.4) for a method m of class C is valid only if both m and C are declared in the same module, and if it appears in that module.

(4) An indirect method axiom for a method m is valid only if it appears in the same module that declares m. Similarly, an axiom for a derived attribute a is valid only if it appears in the same module that declares a.

(5) Redefinitions of attributes and methods may be declared only in the same module as the class they are associated with.

(1) and (2) prevent a class from "acquiring" stored attributes in other modules; this also seems to simplify implementation aspects by allowing a modular approach to code generation and storage allocation. A salient consequence of (2) is that when a theory is used to restrict the parameters to a generic module, the body of the generic may not declare any of the theory's classes as subclasses of any others; however, any of the theory's classes can be superclasses of any classes that the generic module declares.

(3) prevents specifying direct updates to the attributes of an object of class C in a module other that that in which C is declared. (3) and (4), together, forbid scattering the definition of a method or derived attribute across various modules. But note that they allow derived attributes and methods to be declared in any module.

(5) restricts redefinitions to appear together with the class that declares them.

Because of the nature of functional-level programs, there are no corresponding rules for sorts, except those implicit in the importation modes of modules. We say more about this in the next section.

3.9 Vertical Structuring and Information Hiding

The preceding sections have illustrated how FOOPS supports *horizontal* design activities. Horizontal structuring is concerned with module aggregation, enrichment and specialisation. On the other hand, *vertical* structuring is concerned with the implementation of modules, and thus with information hiding. A clear distinction between these two activities helps separate design concerns from implementation concerns, and is better able to document the structure and dependencies in a software system [35, 50].

More formally, information hiding is the process of making certain pieces of information inaccessible. The term "information hiding" was coined by Parnas [89] to refer to the hiding of the "design details" of a module, i.e., those aspects which are purely implementational or accidental and that do not affect the module's interface. The proper use of an information hiding mechanism can have a positive impact on maintenance, because internal changes to a module many times do not affect its clients; information hiding also allows for the design of modules that can be substituted in place of others, which in FOOPS can be rigorously expressed with views (see also Section 3.10). Example applications of information hiding include hiding auxiliary operations in the definition of an abstract data type, and associating protection information with files in UNIX, whereby the contents of a file can be made read-only, executable-only or even invisible, for instance, to certain users of the system; the latter example also shows that information may sometimes be hidden in different degrees. Information hiding often appears under the headings "scoping" or "visibility" in the literature, and we also use this terminology.

In FOOPS modules are the main units of scope, which means that the visibility declarations of a module affect only other modules. This not only simplifies type checking, but is also in concert with the notion that if two features are declared together it is because they are closely related. Additionally, restricting visibility to apply at the class or at the object level, as done in several other object-oriented languages, seems overly conservative and is perhaps the reason why (at the same time) some of these languages provide special features for overcoming this limitation (see Chapter 6).

In FOOPS a module can declare that sorts and classes are private to it, i.e., invisible to other modules. Functions can also be private, and attributes and methods can be either private or *subclass-private*, which is a particularly useful mode for designing class hierarchies. Private module importation blocks the transitive visibility of module features.

In addition, verticality and module inheritance can together support so-called "private" or "implementation" class inheritance as a special case. Furthermore, a generic module can have vertical parameters, such that instantiations automatically hide the corresponding actuals; this facility can be beneficial for library design and use.

3.9.1 Attribute and Method Visibility

The attributes and methods of a class `C` in a module `M` can have one of three **visibility** levels:

- **public**, or visible in all modules that import `M`;
- **private**, or visible only in `M`; and,
- **subclass-private**, or visible in all modules that import `M` and that declare subclasses of `C`, but only when applied to objects of those subclasses.

The purpose of the last level is to allow subclasses of `C` in other modules to have special access to some of its attributes and methods. By being only applicable to subclass objects, it prevents modules that import `M` from declaring "dummy" subclasses of `C` just to be able to apply `C`'s subclass-private attributes and methods to `C` objects. For an example of the need for this kind of visibility, consider the class inheritance diagram in Figure 3.2. A point is something that has a value, a bounded point restricts the value to a certain range, a history point remembers all its previous values, and a bounded history point combines the last two.

Suppose that we want to display the value of a point on the screen, and that we define a method `display` that first clears the screen and then prints the point's value. For bounded points, we also want to print the allowable range, so the display method of `BoundedPoint` first calls `display.Point` and then prints the range; similarly for history points. Now, ideally, the display method of `BHPoint` would be defined as the sequential composition of the display methods of its immediate superclasses. But this would be incorrect, not only because the point's value would be displayed twice, but also because both methods clear the screen! A solution is to declare subclass-private methods called (say) `display-aux` in each class, to print only the additional information; their level of visibility indicates their limited purpose. Then, the (public) display methods would each clear the screen and invoke the appropriate `display-aux` methods. This example[4] generalises to any such kinds of "diamond" hierarchies in which certain initialisation actions are multiply inherited as part of methods. In addition, it illustrates one difficulty in designing reusable software and, in particular, getting inheritance hierarchies to adequately support future functionality.

Levels of visibility are declared as properties, as in

```
at contents : Set -> List [private] .
me display-aux : BoundedPoint -> BoundedPoint [subclass-private] .
```

[4] We have constructed this example by combining examples in [107] and [111].

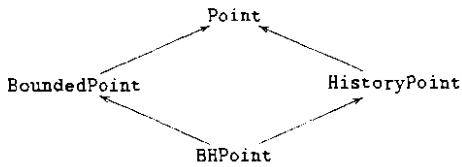


Figure 3.2: A hierarchy of points.

The default level is public.

The following is a consequence of the above definitions: a derived attribute `a` of some class `C` cannot be defined in terms of the subclass-private attributes of `C` if `a` and `C` are declared in different modules. Methods are similarly restricted.

The syntax of private and subclass-private declarations makes it clear that attributes and methods may only be privatised in the module that declares them. One reason for this is that allowing otherwise would give a way to break the conceptual unity of modules. Another reason arises after examining the meaning of multiple module importations if this were allowed. For example, assume a parameterised module `LIST` with a public method called **`reverse`**, and a parameterised module `QUEUE` that imports `LIST` but privatises **`reverse`**. The following are the possible scenarios if both `QUEUE` and `LIST` are directly imported into some other module (e.g., with "`pr QUEUE[X] + LIST[X] .`"):

(1) there are two reverses: **`reverse`**`.QUEUE` and **`reverse`**`.LIST`, where the former is private but the latter is public;

(2) **`reverse`** is private; and,

(3) **`reverse`** is public.

(1) is overly complex. (2) implies that one module may "truncate" another, and that—in this example—the full definition of lists could not be (transitively) used anymore, or at least not at the same time as that of queues. Therefore, the only reasonable option is (3). Consequently, this makes the privatisation of **`reverse`** futile, because it may be uncovered by the explicit importation of module `LIST`. However, a module can be imported vertically, which privatises all its features at the same time. Also, "vertical wrapping" supports further flexibility. These aspects are discussed in Section 3.9.4.

Finally, an attribute or method redefinition is not allowed to be less visible than what it redefines. This would give rise to so-called **non-cumulative interfaces**, where not all the attributes and methods associated with a class are also associated with its subclasses. We believe that non-cumulative interfaces are indicative of poorly designed inheritance hierarchies, and make objects look different at different levels of abstraction. Moreover, non-cumulative interfaces create dynamic binding problems [23]. For example, assume that

a class `A` has a public method `m` which is redefined and declared private for objects of a subclass `B`. Then, a call `m(X)` for `X` a variable of class `A` would type check properly everywhere. However, if `X` is at run-time bound to an object of class `B`, then the call becomes invalid, as `m` is private for objects of class `B`. If the intent is simply to use some of the implementation aspects of `A`, then vertical wrapping is the answer.

3.9.2 Function Visibility

Functions can also be public or private, but there is no option that corresponds to subclass-private. This is so because this kind of privacy does not introduce further flexibility: functional programming with pattern-matching is based on a destruct/construct model that precludes the use of inherited functions. For example, the following axiom for 2-tuples,

```
ax 1* << X ; Y >> = X .
```

would not be appropriate for 3-tuples, which would not match the left-hand side; they would require new selector axioms. This other axiom would also not be useful for 3-tuples:

```
ax rotate << X ; Y >> = << Y ; X >> .
```

not only because of the left-hand side, but also because `rotate` evaluates to a 2-tuple. In this style, constructors are always visible, and thus representation. In the object-oriented style, state is hidden behind identifiers, and no "information loss" is incurred from using inherited attributes and methods.

3.9.3 Sort and Class Visibility

Classes and sorts can also be declared public or private, with the same meaning as above. For example,

```
sort Status [private] .
class Node [private] .
```

Any functions, attributes and methods whose rank includes a private sort or class are automatically private. Also, any sort that inherits from a private one is also private; similarly for classes. This follows from their set-inclusion semantics.

For reasons similar to those given in Section 3.9.1, a sort or class can be declared private only in the module that introduces it.

3.9.4 Vertical Module Importation

Module importations also come in two varieties, public and private. When a module is imported publically (the default), its features retain their level of visibility, so that its public and subclass-private features are passed along transitively. On the other hand, private module importation is a vertical activity in that this transitivity is blocked for all the module's features; it can be seen as the conversion of all public and subclass-private features into private ones. Here are two examples:

```
pr LIST[NAT] [private] .
ex 2TUPLE[BOOL,BOOL] * (sort 2Tuple to Signal) [private] .
```

Verticality and the textual-copy semantics of the `using` mode of importation give rise to a technique called **vertical wrapping** [50]. Here some module `M` that is almost what we want is imported into a "wrapper" module `W`, from which all the functionality that we want is re-exported, but possibly slightly modified from that provided by `M`. This is achieved with a special syntax that allows `using` to redeclare visibility levels. For example,

```
omod W is
  using LIST[NAT] * (class List to Set,
                     private reverse, subclass-private head) .
endo
```

makes method `reverse` private and attribute `head` subclass-private. Thus, the `using` mode can be seen as implementing a highly stylised editor, in that the text of the module is copied, but possibly after instantiation, renaming and visibility redeclarations. Note that the cumulativeness restrictions of Section 3.9.1 must still be obeyed.

Finally, some languages support private subclass relationships, by which (for example) `B` can inherit from `A` but in a way that forbids placing objects of class `B` where objects of class `A` are expected. The purpose of this is simply to allow objects of class `B` to have access to some (or all) of the internal functionality provided for objects of class `A`; or, as Meyer [77] puts it, "to reuse a good implementation." In FOOPS, there is no direct support for this, but the `using` mode of importation can provide a similar effect as a special case, as illustrated above. In C++ [111], for instance, this example would have been constructed by declaring `List` to be a "private" superclass of `Set`. This flexibility of FOOPS illustrates another benefit of distinguishing between class and module inheritance: there is no need for dubious variants of class inheritance to provide the above functionality. In our opinion, class inheritance should be used for the hierarchical classification of objects, and not to support the reuse of code, which is the concern of module inheritance.

3.9.5 Vertical Parameterisation

Modules can also be parameterised with vertical interfaces, such that vertical actuals are automatically hidden in instantiations. This provides the ability to define modules that are generic over their internal features. For example, we can declare module `SET` to have a horizontal parameter for the elements of sets, and a vertical parameter for the underlying implementation of sets:

```
omod SET[X :: TRIV]{REP :: CONTAINER[X]} is
  class Set .
  at contents : Set -> Container .
  me insert   : Set Elt -> Set .
  ...
  ax insert(S,X) = if member(contents(S),X) then ...
```

```
    ...
endo
```

Note first that vertical parameters are specified in curly brackets immediately after the horizontal ones. (The theory CONTAINER was given on page 64.) Also, note how the horizontal parameter instantiates the vertical one. Using the parameterised view LINKED-LIST-AS-CONTAINER (given on page 65), we can instantiate SET as follows:

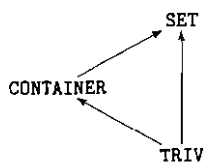
```
SET[NAT]{LINKED-LIST-AS-CONTAINER[NAT]}
```

Modules that import instantiations of SET will not have access to any code associated with the vertical actual, although all (public) features of the horizontal actual will be visible as usual. This is actually stronger than just hiding those operations of sets whose rank mentions any of the features of the vertical actual.

Because CONTAINER is fairly general, we could easily get different implementations of sets simply by providing different vertical actuals. Besides lists, we could also use trees and hash tables, for example. And the multiple implementations that are possible for all of these data structures translate into further implementation options for sets. Thus, vertical parameterisation provides a mechanism for generalising and fine-tuning library modules. For some sophisticated examples of this kind of layering see [4], in which the authors describe how vertical parameterisation ideas can be used for generating multiple implementations of database systems and communication protocols. The novelty here lies in the provision of an integrated linguistic mechanism.

The vertical importation of modules is commonplace, but this kind of vertical parameterisation and instantiation appears not to have been exploited in object-orientation, in which there is a tendency to force everything into some kind of class inheritance relationship.

The context for SET is:



In particular, it shows that there is only one copy of TRIV in SET.

3.9.6 Views

All of the previous facilities are available for theories, too. Because a generic module may in its body declare a class that inherits from one that comes from a theory, all of the subclass-private features of parameter theories are available in the generic's body, in the same way as the subclass-private features of imported modules are. Therefore, while there is no need for views to map the private features of their source, they must map the subclass-private ones.

A restriction, though, is that if `x to y` is a view element and `x` is public, then `y` must also be public. Otherwise, views could reveal some of the secret functionality of target modules.

There is a subtle point regarding the verification of views: to check that axioms hold in the target, *all* the features of the source need to be mapped. For example, if in

```
ax f(X) = g(h(X)) .
```

`h` is the only private feature and no mapping is provided for it, then there is no way to verify whether this axiom holds in the target module. This situation is actually not that surprising: in general, it is impossible to fully understand the semantics of a module without examining its internal features. See [29] for related discussion of this issue in abstract model-theoretic terms.

3.9.7 Type Checking

There are three salient aspects to type-checking this information hiding mechanism. First, a private feature of an imported module never clashes with a locally declared feature. This seems almost too trivial to mention, but in C++, for example, scoping is considered last when parsing expressions, so that clashes with "invisible" features are possible.

Second, the fact that the attributes and methods inherited by a class retain their original rank means that certain expressions that could scope correctly at run-time are statically rejected. For example, consider these modules:

```
omod A is
  class C .
  me m : C -> C [subclass-private] .
  ...
endo

omod B is
  class C2 .
  ex A .
  subclass C2 < C .
  ...
endo
```

In `B`, the expression `m(m(X))` for `X` of class `C2` would not type check. This is because in that module `m` can only be applied to objects of class `C2`, but there is no guarantee that the *outer* `m` in the expression will indeed be applied to an object of class `C2`. However, the expression `m(X); m(X)` would type check correctly.

Finally, it is possible for constants to escape visibility checks. For example, consider

```
fmod M is
  sort S .
  fn f : S -> S .
```

```
  fn c : -> S [private] .
  var X : S .
  ax f(X) = c .
endf
```

While it seems easy to reject this text because of the right-hand side of the axiom, it is in general impossible to determine whether a certain constant will be the result of some function application. Therefore, visibility declarations for constants can only be partially enforced.

3.9.8 A Note on Language Design

It is interesting that for languages which do not identify classes with modules, a subclass cannot be *safely* allowed access to all that which its superclasses have access to (when declared in separate modules). Consider this fragment:

```
omod M is
  class C .
  class Helper [private] .
  me m : C Helper -> C .
  ...
endo
```

Full access would mean that subclasses of `C` in other modules could use `Helper`. But this would also permit other modules to access `Helper` by simply declaring "dummy" subclasses of `C`, resulting in a clear violation of visibility. This is in part due to module-level scoping and to there being no boundaries between the declarations of one class and the declarations of another inside modules. When classes are modules, it is very simple to control this and give subclasses access to everything associated with their superclasses. Certainly, though, this is not always desirable and it is important to have a means for controlling it [107]. Sections 6.2 and 6.6 provide further discussion of this issue.

3.10 System Design and Prototyping

Given a system design in the form of a module expression, properties of that system can be expressed by giving views from a theory to the result of the module expression. Conversely, a view can establish the adequacy of an implementation for some specification that is given in terms of theories. If a module expression includes only executable modules, then it can be symbolically executed, as described in the previous chapter. This provides a rapid prototyping capability that we have been able to experiment with in our current FOOPS implementation.

A second approach is more straightforward: simply write the design of a system as a module expression (or a collection of module expressions defining the system, subsystems, etc.), and then either supply standard library modules, or else write rapid prototypes for

each bottom level component. It is worth noting how vertical composition can play a role in this. If some library module is close to but not actually identical with what is needed, then the library module could be vertically imported into a new "wrapper" module that provides the necessary new functionality, building on top of the old one.

Another approach uses **built-in modules** to encapsulate code written in one or more implementation languages. This can provide access to libraries in other languages, give high level structure to old code, and interface with low level facilities such as operating system routines. Built-in modules were developed for OBJ [52], where they were used to implement standard data types, such as natural numbers and Booleans. In FOOPS, built-in modules implement both standard data types and standard classes, such as arrays, and it is not difficult to write new ones[5]. Chapter 5 describes how OBJ3's built-in modules were used to develop our prototype implementation of FOOPS.

Each of these approaches can benefit from the fine-tuning capability of vertical parameterisation, because it allows replacing underlying implementation layers with new ones, so that one can configure new prototypes with varying levels of performance and resource use.

Further design support is provided by views, which as hinted earlier in this chapter, can be used to describe refinement relationships. For example, as we mentioned above, a design might begin with a module expression having certain theories in it, so that it is not really executable; call it D_0. A more developed version of D_0 might replace some of those theories with executable modules, or might replace some of the already executable modules with more efficient versions. Then this relationship between D_0 and the new version, call it D_1, can be rigorously captured with a view $D_0 \xrightarrow{v} D_1$ that would provide mappings for the replaced modules but which would be default for the others. This process could span many steps, thus $D_0 \xrightarrow{v_1} D_1 \xrightarrow{v_2} \ldots \xrightarrow{v_n} D_n$, where each arrow would denote a different view. Therefore, views can be used to express the *evolution* of designs. Furthermore, one could envisage views that included further information, such as annotations for design decisions, so that, for example, there would be a systematic way of documenting the history of a system design.

Thus, we have that a single object oriented language can support the modular expression of system designs and high level properties, as well as the modular composition and reuse of designs, specifications and code, plus prototyping by symbolic evaluation, and more efficient prototyping by vertical composition or built-in modules. At each level of abstraction, relationships of refinement and evolution can be recorded by giving suitable views and theories. This gives a very rich environment for system development.

Related discussions of the need for facilities of this kind in computer languages may be found in [28, 85] and [38]; the latter suggests a methodology called "hyperprogramming" for integrating the entire life cycle through parameterised programming, including requirements, design, specification, coding, maintenance, documentation, and version and configuration management.

[5]The built-in modules of FOOPS can use both Lisp and C code, because FOOPS is implemented in Kyoto Common Lisp, which is based on C. We acknowledge that supporting other languages for built-ins would be much more difficult, even though it is very appealing [88].

3.11 Summary

This chapter has presented in detail the facilities for reusing and interconnecting modules in FOOPS, which include module hierarchies, blocks, theories, views, horizontal and vertical parameterisation, renaming, sum, and module expressions. Parameterised modules in FOOPS use theories to specify syntactic and semantics properties expected of actual arguments, and can distinguish and document both aggregation and implementation dependencies. The purpose of this rigorous approach to module interfaces should be obvious: to increase the reliability of module interconnections and to make explicit the requirements for reusing parameterised modules. We have also discussed how these facilities support further functionality, such as abstract classes, "private" class inheritance, and higher-order composition in a first-order setting. The integration of all of the above provide a powerful environment for designing systems.

The contributions of this chapter to the development of FOOPS include the provision of abstract classes, module blocks, encapsulation rules, and information hiding and vertical structuring facilities. Moreover, we have argued that by combining views, module inheritance, and vertical structuring, FOOPS generalises orthodox approaches to object-orientation, in which classes and class inheritance are the principal mechanisms for designing, describing and putting together software architectures. Also, we hope to have shown that the object-oriented paradigm requires and admits extensions such as those provided by parameterised programming. Chapter 6 will contribnte further analysis of this by comparing FOOPS with other languages.

Chapter 4

Formal Semantics

To see what is general in what is particular and what is permanent in what is transitory is the aim of scientific thought.

— Alfred North Whitehead

This chapter provides a detailed overview of work towards a mathematical formalisation of the FOOPS language. Both syntax and semantics are fundamental for formalising programming and specification languages: syntax because we manipulate texts, diagrams, and other such descriptions; semantics because it concerns the models of those descriptions and how they are affected when the descriptions are modified. Therefore, the link between these two aspects needs to be made clear and explicit[1].

This presentation is divided in three parts. The first part explains the semantics of the functional level of FOOPS. Being a syntactic variant of OBJ, at this level denotational semantics is given by order-sorted algebras [49], while operational semantics is given by (order-sorted) term rewriting [44]. The second part discusses the semantics of the object level of FOOPS. Denotationally, a generalisation of order-sorted algebra called hidden-sorted algebra is adopted [39, 43]; this algebraic formalism takes into account that objects have internal states by generalising the satisfaction relation between sentences and models (algebras). Operationally, a form of reflection in which object level programs are reduced to functional level programs formalises object creation and destruction, and method evaluation [48]. The third part examines the semantics of parameterised programming. This semantics is grounded in the theory of institutions, which formalises the notion of "logical system" and offers a logic-independent framework for expressing how smaller specifications can be combined to form larger ones [41]. The connection with FOOPS is that both Order-Sorted Conditional Equational Logic (the logic of the functional level) and Hidden Order-Sorted Conditional Equational Logic (the logic of the object level) are institutions.

We note, however, that while the semantics of the functional level of FOOPS is fully developed, work to provide a complete formal characterisation of its object level still continues.

[1] Following this discussion, the reader is now aware that we use the word "semantics" to refer to both aspects.

The operational semantics of the object level provides a foundation, but the denotational semantics needs to be extended to capture object identifiers and creation. This is not an open problem particular to FOOPS, but a research topic being actively pursued in the formal methods community.

The simplicity of equational logic, and the amount of theory and mechanical support available for equational reasoning, is the justification for an algebraic approach to the semantics of FOOPS. Other logical systems can provide further expressiveness; however, the research programme begun by Goguen in 1978 [34] has shown that a large amount of computing science can be done with equational logic and efforts such as the present one attempt to further understand its applicability. (See [49] for more discussion about this.)

The contribution of this chapter is that it threads together the most recent work on providing a mathematical foundation for FOOPS, and can be seen as a high-level but detailed summary of the publications cited above. We assume some basic knowledge of category theory [93] (mostly for Section 4.3).

4.1 Functional-level Semantics

The following sections introduce the basic concepts of order-sorted algebra, order-sorted conditional equational logic and term rewriting. In summary, the semantics of an `fmod` is an initial algebra in the category of order-sorted algebras that satisfy its equations, while the semantics of an `fth` is any algebra in this category; term rewriting is an implementation of order-sorted equational deduction.

4.1.1 Order-Sorted Algebra

Order-sorted algebra (hereafter, OSA) constitutes an algebraic model theory and relies on the notions of order-sorted signature, algebra, specification and satisfaction. It provides a semantic domain for functional-level modules in FOOPS by interpreting a module as an order-sorted specification.

4.1.1.1 Signature

An order-sorted signature is characterised by a partially ordered set[2] of sorts S specifying a subsort relation.

Definition 4.1 An **order-sorted signature** is a triple $(S, \leq, \Sigma)$, where S is the sort set, $(S, \leq)$ is a poset, and Σ is an $S^* \times S$-sorted family $\{\Sigma_{w,s} \mid w \in S^* \text{ and } s \in S\}$, such that the following **monotonicity condition** is satisfied:

$$\text{if } \sigma \in \Sigma_{w_1,s_1} \cap \Sigma_{w_2,s_2} \text{ and } w_1 \leq w_2 \text{ then } s_1 \leq s_2$$

where $w_1 \leq w_2$ refers to the point-wise comparison of strings of sorts using the $\leq$ provided with the signature. □

[2] A partially ordered set, or poset, is a set A provided with a reflexive, antisymmetric and transitive relation $\leq$.

Elements in the sets of Σ are called **operation symbols**, or more briefly, **operators**. For $\sigma \in \Sigma_{w,s}$, we call (w, s) its **rank**, w its **arity** and s its **sort** (or **value sort** or **coarity**). When $w = \lambda$, the empty string in S^*, σ is called a **constant operator** or simply a **constant**. The monotonicity condition on the operators allows the treatment of partial functions and determines the form of polymorphic operations. Note that it implies that constants cannot be overloaded (because $w_1 = w_2$ entails both $s_1 \leq s_2$ and $s_2 \leq s_1$, and therefore $s_1 = s_2$).

In a FOOPS module the elements of an order-sorted signature can be immediatedly recognised. Indeed, the syntax of FOOPS makes explicit the binding between the langnage features and their algebraic counterparts.

Example 4.2 In the following FOOPS module specifying lists of `Nat`, `sorts` defines the set S of sorts, `subsort` defines the partial ordering on S, each `fn` introduces an operator symbol with its corresponding rank, and `NeList` is the sort of non-empty lists:

```
fmod LIST-OF-NAT is
  sorts Nat List NeList .
  subsort NeList < List .
  fn 0    : -> Nat .
  fn s_   : Nat -> Nat .
  fn nil  : -> List .
  fn cons : List Nat -> NeList .
endf
```

□

Morphisms play an essential role in the formalisation of parameterised programming. For example, morphisms between signatures describe notational changes (when bijective).

Definition 4.3 An **order-sorted signature morphism** $\phi : (S, \Sigma) \to (S', \Sigma')$ is a pair (f, g) consisting of

1. a map $f : S \to S'$ of sorts, and
2. an $S^* \times S$-indexed family of maps $g_{w,s} : \Sigma_{w,s} \to \Sigma'_{f^*(w),f(s)}$ on operation symbols, where $f^* : S^* \to S'^*$ is the extension of f to strings[3],

such that if $s_1 \leq s_2$ then $f(s_1) \leq f(s_2)$ for $s_1, s_2 \in S$. We may write $\phi(s)$ for $f(s)$, $\phi(w)$ for $f^*(w)$, and $\phi(\sigma)$ for $g_{w,s}(\sigma)$ when $\sigma \in \Sigma_{w,s}$. □

Signatures and signature morpbisms form a category.

[3]This extension is defined as follows: $f^*(\lambda) = \lambda$ and $f^*(ws) = f^*(w)f(s)$, for $w \in S^*$ and $s \in S$.

4.1.1.2 Regularity

In general, polymorphic operators bave more than one sort. In the case of subsort polymorphism, their sorts are related through the sort hierarchy. Indeed, we may have more general situations in the case of *ad hoc* polymorphism, i.e., when the same operation symbol is used for semantically unrelated operations; for example, _+_ is often used to denote both addition of natural numbers and disjunction of boolean values. Unfortunately, ambiguity can arise, as shown in the following example:

Example 4.4

```
fmod NON-REG is
  sorts S1 S2 S3 S4 S5 S6 .
  subsorts S1 < S3 S2 < S4 .
  fn a : -> S1 .
  fn b : -> S2 .
  fn f : S1 S4 -> S5 .
  fn f : S3 S2 -> S6 .
endf
```

Ambiguity arises in parsing expressions where `f` is applied to values of sorts S1 and S2, respectively. In fact both operators can be used, but they lead to different value sorts. Note that the monotonicity condition is satisfied as the two arities are not comparable in the sort ordering. □

In general, this kind of ambiguity can be removed by restricting to signatures which are "regular." Regularity guarantees the existence of a **least sort** for each term (see Section 2.1.3), which in the presence of overloading guarantees a nnique (least) parse.

Definition 4.5 An order-sorted signature $(S, \leq, \Sigma)$ is **regular** iff for each $\sigma \in \Sigma_{w_1,s_1}$ and each $w_0 \leq w_1$ there is a unique least element in the set $\{\langle w, s\rangle \mid w_0 \leq w \text{ and } \sigma \in \Sigma_{w,s}\}$. □

The signature in the previous example fails to be regular: for w_0 = S1 S2 the set of ranks $\{\langle$S1 S4, S5$\rangle, \langle$S3 S2, S6$\rangle\}$ does not admit a least element. (See Section 2.1.3 for more motivation and examples of this situation.)

In the following we restrict ourselves to signatures which are regular, although all results generalise without difficulty [42].

4.1.1.3 Algebras

An order-sorted algebra provides sets of values (called "carriers") for the sorts of an order-sorted signature and a function for each operation symbol. Inclusion on the carrier sets reflects the sort ordering, and function agreement on domain intersection reflects operator monotonicity. Thus, algebras are models.

Definition 4.6 Let $(S, \leq, \Sigma)$ be a regular order-sorted signature. Then an $(S, \leq, \Sigma)$-**algebra** A is an S-sorted family $\{A_s \mid s \in S\}$ of sets called the **carriers** of A, together with a function $A_\sigma : A_w \to A_s$ for each $\sigma \in \Sigma_{w,s}$, for $A_w = A_{s_1} \times \ldots \times A_{s_n}$ when $w = s_1 \ldots s_n$ and where A_w is a singleton set when $w = \lambda$, such that the following **monotonicity conditions** are satisfied:

1. if $s \leq s'$ in S then $A_s \subseteq A_{s'}$; and,
2. if $\sigma \in \Sigma_{w_1,s_1} \cap \Sigma_{w_2,s_2}$ and $w_1 \leq w_2$ then $A_\sigma : A_{w_1} \to A_{s_1}$ equals $A_\sigma : A_{w_2} \to A_{s_2}$ on A_{w_1}.

□

When $(S, \leq)$ is clear, we simply write Σ for $(S, \leq, \Sigma)$. Note that when $\sigma \in \Sigma_{\lambda,s}$, A_σ can be considered as an element of A_s.

Example 4.7 A possible algebra A for `LIST-OF-NAT` has as carrier sets the natural numbers in the usual notation (with *succ* the operation of successor), non-empty lists of naturals (with $\cdot$ the infix operation of concatenation) and all lists of natural numbers (with ε the empty list):

$$A_{Nat} = \{0, 1, 2, \ldots\}$$
$$A_{NeList} = \{\varepsilon \cdot 3 \cdot 10 \cdot 20 \cdot 6,\ \varepsilon \cdot 2 \cdot 4, \ldots\}$$
$$A_{List} = A_{NeList} \cup \{\varepsilon\}$$
$$A_0 = 0 \in A_{Nat}$$
$$A_{s\_} = succ : A_{Nat} \to A_{Nat}$$
$$A_{nil} = \varepsilon \in A_{List}$$
$$A_{cons} = \cdot : A_{List}\, A_{Nat} \to A_{NeList}$$

□

We will use $\mathbf{OSAlg}_\Sigma$ to denote the category of all Σ-algebras, with its arrows being Σ-algebra homomorphisms as defined further below.

4.1.1.4 Terms

A term is an expression constructed by the recursive application of the operators of a signature, starting from its constants:

Definition 4.8 Given a regular order-sorted signatnre Σ, its **terms** are the elements of the sets of an S-sorted family $\{T_{\Sigma,s} \mid s \in S\}$, where:

1. $\Sigma_{\lambda,s} \subseteq T_{\Sigma,s}$; and,
2. if $\sigma \in \Sigma_{w,s}$, where $w = s_1 \ldots s_n \neq \lambda$, and $t_i \in T_{\Sigma,s_i}$ for $i = 1, \ldots, n$, then (the striug) $\sigma(t_1, \ldots, t_n) \in T_{\Sigma,s}$.

□

The constants of a siguature play the role of **generators** of the terms in the sense that every term is built starting from them. Indeed, if there are no constants in the signature, the sets of terms are empty.

4.1.1.5 The Term Algebra

The terms of an order-sorted signature define an algebra, called the term algebra.

Definition 4.9 Given a regular order-sorted signature Σ, the order-sorted Σ-**term algebra** is the least family $\{T_{\Sigma,s} \mid s \in S\}$ of sets of terms such that:

1. if $s' \leq s$ then $T_{\Sigma,s'} \subseteq T_{\Sigma,s}$;
2. if $\sigma \in \Sigma_{w,s}$, where $w = s_1 \ldots s_n \neq \lambda$. then $T_\sigma : T_{\Sigma,w} \to T_{\Sigma,s}$ sends $t_1, \ldots, t_n$ to $\sigma(t_1, \ldots, t_n)$, where $t_i \in T_{\Sigma,s_i}$ for $i = 1, \ldots, n$.

□

Example 4.10 The term algebra T_Σ for `LIST-OF-NAT` has these carrier sets:

$T_{\Sigma,Nat} = \{$`0, s 0, s s 0,` $\ldots\}$

$T_{\Sigma,NeList} = \{$`cons(nil,0), cons(cons(nil,s 0),s s 0),` $\ldots\}$

$T_{\Sigma,List} = T_{\Sigma,NeList} \cup \{$`nil`$\}$

The function $T_{s\_}$ maps the term `t` of $T_{\Sigma,Nat}$ to the term `s t` of $T_{\Sigma,Nat}$, while T_{cons} maps the terms `t1` $\in T_{\Sigma,List}$ and `t2` $\in T_{\Sigma,Nat}$ to the term `cons(t1,t2)` of $T_{\Sigma,NeList}$. □

Given a regular order-sorted signature Σ, the term algebra T_Σ has a property, called initiality, which allows us to cousider it as the most representative algebra in $\mathbf{OSAlg}_\Sigma$. Initiality is defined in terms of the relatious between the Σ-algebras of $\mathbf{OSAlg}_\Sigma$. A Σ-algebra I is **initial** in $\mathbf{OSAlg}_\Sigma$ if and only if there exists a unique homomorphism from I to any other algebra. All initial algebras on a signature are isomorphic, or the same up to a family of hijections between their carrier sets.

A homomorphism hetweeu two order-sorted algebras is a sorted family of mappings between their carrier sets that preserves the algebraic structure, in the sense that they distribnte through the operations of the algebras and respect carrier set inclusion.

Definition 4.11 Let Σ be a regular order-sorted signature and let A and B be Σ-algebras. A Σ-**homomorphism** $h : A \to B$ is an S-sorted family $\{h_s : A_s \to B_s \mid s \in S\}$ of functions such that:

1. if $\sigma \in \Sigma_{\lambda,s}$ then $h_s(A_\sigma) = B_\sigma$;
2. if $\sigma \in \Sigma_{w,s}$ with $w = s_1, \ldots, s_n \neq \lambda$. then $h_s(A_\sigma(a)) = B_\sigma(h_w(a))$ for each $a = a_1, \ldots, a_n \in A_w$ and $h_w(a) = (h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$; and,

3. if $s \leq s'$ in $(S, \leq)$ then $h_s(a) = h_{s'}(a)$ for each $a \in A_s$.

□

Definition 4.12 Given any algebra A in $\mathbf{OSAlg}_\Sigma$, the unique homomorphism $! : T_\Sigma \to A$ is defined as follows:

1. if $\sigma \in \Sigma_{\lambda,s}$ then $!_s(T_\sigma) = A_\sigma$;
2. if $\sigma \in \Sigma_{w,s}$ with $w = s_1, \ldots, s_n \neq \lambda$, then $!_s(T_\sigma(t)) = A_\sigma(!_w(t))$ for each $t = t_1, \ldots, t_n \in T_{\Sigma,w}$ and $!_w(t) = (!_{s_1}(t_1), \ldots, !_{s_n}(t_n))$.

□

For example, the unique homomorphism between T_Σ and the algebra A in Example 4.7 maps `0` to 0, `nil` to ε, `cons` to $\cdot$, `cons(nil,0)` to $\varepsilon \cdot 0$, etc.

4.1.1.6 Equations

An equation[4] expresses equality between two terms. As terms are sorted, and can have more than one sort through the sort hierarchy, not every pair of terms can form an order-sorted equation. OSA allows a quite general form of equation not requiring the same least sort for the terms involved, but more loosely, that their sorts lie in the same connected component of the partial ordering on sorts.

Definition 4.13 Given a poset $(S, \leq)$, let $\equiv$ denote the transitive and symmetric closure of $\leq$. Then $\equiv$ is an equivalence relation whose equivalence classes are called the **connected components** of $(S, \leq)$. □

Terms which include variables are used in equations. Terms without variables are called **ground terms**. A **variable** set X is an S-sorted family $\{X_s \mid s \in S\}$ of disjoint sets. Terms with variables can be seen as a special case of ground terms by enlarging the signature with new constants that correspond to the variables; i.e., $X_s \subseteq (T_\Sigma)_s$ for $s \in S$. We will use $T_{\Sigma(X)}$ to denote a family of terms with variables taken from X.

Definition 4.14 Given a regular order-sorted signature Σ, a **Σ-equation** is a triple $\langle X, t, t' \rangle$ where X is a variable set, t and t' are in $T_{\Sigma(X)}$ and the least sorts of t and t' are in the same connected component of $(S, \leq)$. □

Variables in equations are universally quantified and an alternative notation that we use for equations is $(\forall X)\ t = t'$. In addition, equations can be conditional, with form $(\forall X)\ t = t'$ if C, where the condition C is a finite set of unquantified Σ-equations with variables in X.

We use equations to express expected properties. We could, for instance, enrich `LIST-OF-NAT` with operators to allow access to the head or the tail of a list:

[4] We use "equation" rather than "axiom" to follow the literature on semantics.

Example 4.15 Suppose the following operator declarations are added to `LIST-OF-NAT`:

```
fn hd : NeList -> Nat .
fn tl : NeList -> List .
```

Their expected properties can now be stated with equations as follows:

```
var L : List .  var N : Nat .
ax hd(cons(L,N)) = N .
ax tl(cons(L,N)) = L .
```

corresponding to the intuition that the head of a list is the last element added, while its tail is the list itself without the head. □

This gives rise to the following:

Definition 4.16 Given a regular order-sorted signature Σ and a set of Σ-equations E, the pair (Σ, E) is an order-sorted Σ-**presentation** (also called Σ-**specification**). □

A module at the functional level of FOOPS defines an order-sorted presentation. Note also that an order-sorted signature can be considered as an order-sorted presentation that has an empty set of equations.

4.1.1.7 Satisfaction

Given a signature Σ, order-sorted satisfaction expresses when a Σ-algebra satisfies a Σ-equation. Intuitively, a Σ-algebra satisfies a Σ-equation iff the two terms of the Σ-equation are *always assigned* the same element of the algebra.

In order to formalise this intuition, we need a precise definition of the notion of assignment of an element of an algebra to a term. Given a signature Σ, a term in a Σ-equation contains variables from a sorted family X. Earlier we mentioned that variables in X constitute an enlarged signature $\Sigma(X)$ where the elements of X represent constants. We can also build the term algebra $T_{\Sigma(X)}$ on $\Sigma(X)$ in the same way as on Σ. Indeed, $T_{\Sigma(X)}$ is also a Σ-algebra. In particular, it is the Σ-algebra **freely generated** from X. Freeness plays a central role in the notion of satisfaction. A Σ-term with variables in X is interpreted in a Σ-algebra A by assigning values in A to the variables in the term, and extending such an assignment to the whole term. An assignment is then a mapping from X to A. The extension of such and assignment is a homomorphism between $T_{\Sigma(X)}$ and A and its existence and uniqueness are guaranteed by the freeness of $T_{\Sigma(X)}$. (See [49] and Section 4.3.4 for more about freeness.)

Definition 4.17 Given a signature Σ, a variable set X and a Σ-algebra A, the unique homomorphism $a^* : T_{\Sigma(X)} \to A$ extending the assignment $a : X \to A$, is defined as follows:

1. if $x \in X_s$ then $a^*_s(x) = a_s(x)$;
2. if $\sigma \in \Sigma_{\lambda,s}$ then $a^*_s(T_\sigma) = A_\sigma$; and,

3. if $\sigma \in \Sigma_{w,s}$ with $w = s_1, \ldots, s_n \neq \lambda$, then $a^*(T_\sigma(t)) = A_\sigma(a^*_w(t))$ for each $t = t_1, \ldots, t_n \in T_{\Sigma,w}$ and $a^*_w(t) = (a^*_{s_1}(t_1), \ldots, a^*_{s_n}(t_n))$.

□

Example 4.18 Consider the algebra A of Example 4.7. Suppose we want to assign a value in A to the term `cons(nil,N)`. Note that `N` has sort `Nat`. We can choose, for instance, to assign the value 3 to the variable `N`. Considering that ϵ is the function of A representing the operator `nil`, and that $\cdot$ corresponds to `cons`, under this assignment the corresponding value for `cons(nil,N)` is $\epsilon \cdot 3$. □

In general, different assignments to the variables of a term yield different assignments to the term. A Σ-equation is satisfied in a Σ-algebra if and only if we can assign the same value of the algebra to the terms of the equation, for each possible assignment to their variables. As terms may have more than one sort, assignments are always applied in the least sort of the terms. In the following definition $LS(t)$ denotes the least sort of the term t.

Definition 4.19 An order-sorted Σ-algebra A **satisfies** a Σ-equation $(\forall X)\ t = t'$ iff $a^*_{LS(t)}(t) = a^*_{LS(t')}(t')$ in A for every assignment $a : X \to A$. □

The notion of satisfaction extends to sets of equations and to conditional equations. In the latter the terms of an equation $(\forall X)\ t = t'$ if C must have the same value in the algebra for all assignments that satisfy C. Satisfaction also extends to Σ-specifications:

Definition 4.20 Given an order-sorted specification $\langle \Sigma, E \rangle$, an order-sorted Σ-algebra A **satisfies** $\langle \Sigma, E \rangle$ iff it satisfies each equation in E. In this case we say that A is a $\langle \Sigma, E \rangle$-**algebra**. □

$\mathbf{OSAlg}_{\Sigma,E}$ denotes the category of Σ-algebras that satisfy $\langle \Sigma, E \rangle$; its arrows are homomorphisms between the algebras. There is also a category of presentations in which arrows are signature morphisms with the condition that equational satisfaction is preserved:

Definition 4.21 Given order-sorted presentations (Σ, E) and (Σ', E'), an order-sorted signature morphism $\Phi : \Sigma \to \Sigma'$ is a **morphism** $\Phi : (\Sigma, E) \to (\Sigma', E')$ iff

$$M' \models_{\Sigma'} E' \text{ implies } \Phi(M') \models_{\Sigma} E$$

for all Σ'-algebras M', where the **reduct** $\Phi(M')$ of M' to Σ is M' viewed as a Σ-algebra. □

4.1.1.8 Initiality

Initiality extends to the class of order-sorted algebras which satisfy an order-sorted presentation. Two properties characterise the initial algebra of an order-sorted presentation:

- every element of the (carriers of the) algebra can be named using the operators of the specification, i.e., there is *no junk* in the algebra;

- all ground equations which are satisfied in the algebra can be derived from the equations in the specification, i.e., unequal terms are *not confused* in the algebra.

The initial algebra of an order-sorted specification is related to the term algebra and, indeed, coincides with it when the set of equations is empty. The initial algebra of an order-sorted specification is a *quotient algebra* of the term algebra. This means that terms of the term algebra are considered to be the same in the initial algebra when they are made *equivalent* by the equations. To make this precise, we need some auxiliary definitions.

A Σ-equation induces a relation on each Σ-algebra:

Definition 4.22 Given an order-sorted signature Σ and a Σ-algebra B, a Σ-equation $e = \langle X, t, t' \rangle$ defines a relation R_e on B as follows:

$$b R_e b' \text{ iff } \exists a : X \to B \text{ such that } b = a^*(t) \text{ and } b' = a^*(t')$$

□

Note that R_e is an S-sorted family $\{R_{e,s} \mid s \in S\}$ of relations and that $R_{e,s} \subseteq R_{e,s'}$ when $s \leq s'$ in S. If E is a set of equations, R_E denotes the least relation which contains R_e for each $e \in E$.

Example 4.23 For the term algebra T_Σ for `LIST-OF-NAT`, the equation

```
hd(cons(L,N)) = N
```

defines, for all possible assignments to the variables `L` and `N`, a relation on $T_{\Sigma,Nat}$ which, for example, relates `hd(cons(nil,s 0))` and `s 0`. □

For our purposes, this relation is not enough. We require that the relation is also a congruence with respect to the operations and the inclusion hierarchies of the carriers of the algebra:

Definition 4.24 Given an order-sorted signature Σ and a Σ-algebra B, a **Σ-congruence** is a Σ-equivalence relation R such that:

1. if $\sigma \in \Sigma_{w,s}$ and $b, b' \in B_w$ then bRb' implies $B_\sigma(b)\ R\ B_\sigma(b')$, where $w = s_1, \ldots s_n$, $bRb' = b_1 R_{s_1} b'_1, \ldots, b_n R_{s_n} b'_n$ and $b_i, b'_i \in B_{s_i}$; and,
2. if $s \leq s'$ and $b, b' \in B_s$ then $b R_s b'$ iff $b R_{s'} b'$.

□

If e is an equation, we use $\equiv_e$ to denote the least Σ-congruence induced by R_e. By extension, if E is a set of equations we denote $\equiv_E$ the least Σ-congruence induced by R_E.

Example 4.25 The relation of the previous example extends to a congruence which, for example, relates `cons(nil,hd(cons(nil,s 0)))` and `cons(nil,s 0)`. □

We would expect the initial algebra for an order-sorted specification $\langle \Sigma, E \rangle$ to be a quotient of the term algebra T_Σ, obtained by identifying the terms related by the congruence $\equiv_E$ generated on T_Σ from the equations of the specification. However, due to sort hierarchies, we need to consider situations where equations relate terms across a connected component of the poset of sorts. For instance, we would expect the terms `tl(cons(cons(nil,0),0))` and `cons(nil,0)` to be equivalent with respect to both *List* and *NeList*, but $\equiv_E$ relates them only with respect to *List*. To this end, we will define a new congruence, derived from $\equiv_E$, on all terms with sorts in the same connected component. We present the subcase when posets and order-sorted signatures are locally filtered [49]:

Definition 4.26 A poset $(S, \leq)$ is **filtered** iff for any two elements $s, s' \in S$ there is an element $s'' \in S$ such that $s, s' \leq s''$. A poset is **locally filtered** iff each of its connected components is filtered. An order-sorted signature $(S, \leq, \Sigma)$ is **locally filtered** iff $(S, \leq)$ is locally filtered. □

Given a locally filtered order-sorted signature $(S, \leq, \Sigma)$, for each connected component C of $(S, \leq)$ we may define a congruence $\equiv_C$ on $T_{\Sigma,C} = \bigcup_{s \in C} T_{\Sigma,s}$ as follows:

$$t \equiv_C t' \quad \text{iff} \quad \exists s \in C \quad \text{such that} \quad t \equiv_{E,s} t'.$$

We can now define an initial algebra for $\langle \Sigma, E \rangle$ as a quotient algebra of the term algebra T_Σ under the congruences $\equiv_C$ generated for each connected component C. Let $T_{\Sigma,E}$ denote such an algebra. In the following definition $[t]$ denotes the equivalence class of t:

Definition 4.27 Given a locally filtered order-sorted specification $\langle \Sigma, E \rangle$, the algebra $T_{\Sigma,E}$ is defined as follows:

1. if C is a connected component of $(S, \leq)$ and $s \in C$, then the carrier set $T_{\Sigma,E,s}$ is the quotient of $T_{\Sigma,s}$ by $\equiv_C$;
2. if $\sigma \in \Sigma_{\lambda,s}$ then $T_{\Sigma,E,\sigma} = [T_\sigma]$; and,
3. if $\sigma \in \Sigma_{w,s}$ then $T_{\Sigma,E,\sigma}([t_1], \ldots, [t_n]) = [T_\sigma(t_1, \ldots, t_n)]$ for $w = s_1 \ldots s_n \neq \lambda$, $t_i \in T_{\Sigma,s_i}$ and $[t_i] \in T_{\Sigma,E,s_i}$ for $i = 1, \ldots, n$.

□

As all the initial algebras of a specification $\langle \Sigma, E \rangle$ are isomorphic, we can consider them as abstractly the same and choose $T_{\Sigma,E}$ as their representative. We can therefore refer to it simply as *the* initial algebra of the specification.

Example 4.28 The initial algebra of `LIST-OF-NAT` has the following carrier sets:

$$(T_{\Sigma,E})_{Nat} = \{[0, \texttt{hd(cons(nil,0))}, \ldots], [\texttt{s 0}, \texttt{hd(cons(nil,s 0))}, \ldots], \ldots\}$$
$$(T_{\Sigma,E})_{NeList} = \{[\texttt{cons(nil,0)}], [\texttt{cons(cons(nil,s 0),s s 0)}], \ldots\}$$
$$(T_{\Sigma,E})_{List} = (T_\Sigma)_{NeList} \cup \{[\texttt{nil}, \texttt{tl(cons(nil,0))}], \ldots\}$$

□

By this construction, we may formalise an abstract data type as a class of isomorphic initial algebras. Because a module at the functional level of FOOPS specifies an abstract data type it may be assigned as semantics the initial algebra of this specification. Additionally, this choice of the representative of the class of initial algebras allows us to apply term rewriting techniques to the specification, giving a form of computation for FOOPS programs (see sections 4.1.3 and 4.2.2).

OSA also provides a semantics for theories at the functional level of FOOPS. Because a theory has the same syntactic structure as a module, it can also be seen as representing an order-sorted specification $\langle \Sigma, E \rangle$. However its semantics is loose, in the sense that we take as its denotation the whole category $\mathbf{OSAlg}_{\Sigma,E}$ of algebras satisfying $\langle \Sigma, E \rangle$, not just those algebras which are initial in $\mathbf{OSAlg}_{\Sigma,E}$. This gives an account of the fact that a theory specifies module properties and not executable code.

4.1.2 Order-Sorted Conditional Equational Logic

The notions of order-sorted equation and order-sorted satisfaction introduced in the previous sections support equational deduction. Order-Sorted Conditional Equational Logic is a logic for equality having order-sorted signatures as vocabulary and order-sorted conditional equations as sentences. Equational deduction allows new equations to be derived from old ones by applying rules which express the properties of substituting equals for equals. In particular, they express that equality

- is *reflexive*: anything is equal to itself;
- is *symmetric*: if t is equal to t' then t' is equal to t;
- is *transitive*: if t is equal to t' and t' is equal to t'' then t is equal to t'';
- is a *congruence with respect to term substitution*: substituting equal expressions into the same expression yields equal expressions; and,
- has the property of *substitutivity*: applying the same substitution to equal expressions yields equal expressions.

Given an order-sorted specification $\langle \Sigma, E \rangle$, the equations of E together with these rules define a deduction system in which these equations represent the axioms (or assumptions)[5]. This deduction system allows new equations to be deduced; the least set of equations deducible from E is called the **deductive closure** of E and is denoted $E^\bullet$. A derivation of an equation e from E is then a proof that e belongs to $E^\bullet$ and is represented by the sequence of rule applications which actually deduce e from E. If e is deducible from E then we write $E \vdash e$.

[5] In general, a logical system can have many different and equivalent variants of its rules of deduction, and the use of a particular one is a matter of convenience.

Given an order-sorted specification (Σ, E), [49] proves that such a deduction system is **sound** and **complete** for deriving all equations which hold in $\mathbf{OSAlg}_{\Sigma,E}$. If an equation e (not belonging to E) is satisfied by each algebra in $\mathbf{OSAlg}_{\Sigma,E}$, then we write $E \models_\Sigma e$. Soundness says that truth is preserved in all algebras of $\mathbf{OSAlg}_{\Sigma,E}$, i.e., that if an equation is deducible from E then it is satisfied by each algebra in $\mathbf{OSAlg}_{\Sigma,E}$:

$$E \vdash e \quad \text{implies} \quad E \models_\Sigma e.$$

Completeness says that every equation which is true in all algebras of $\mathbf{OSAlg}_{\Sigma,E}$ can be deduced from E using the deduction system:

$$E \models_\Sigma e \quad \text{implies} \quad E \vdash e.$$

The importance of these properties is that they allow us to establish whether an equation is true in all algebras by manipulating finitary syntactic objects (the Σ-equations), while the notion of satisfaction requires manipulations of possibly non-finitary semantic objects (the algebras in $\mathbf{OSAlg}_{\Sigma,E}$). This means that deduction represents a model of computation for the functional level of FOOPS and provides an operational semantics through its specialisation to term rewriting.

Finally, the proof of completeness in [49] assumes coherent signatures. A signature is **coherent** if it is regular and locally filtered. For coherent signatures, satisfaction is closed under isomorphism, meaning that isomorphic algebras satisfy the same set of equations. One benefit of requiring signatures to be coherent is a greater simplicity and flexibility in the treatment of equality, since we can always assume that the two terms of an equation have the same sort by appealing to a common supersort.

4.1.3 Term Rewriting

Order-sorted term rewriting [40, 44, 66] is a form of equational deduction that provides a model of computation for the functional level of FOOPS, and thus an operational semantics.

Given a signature Σ, **sentences for term rewriting** are Σ-equations subjected to the restriction that variables in right-hand sides must also be present in left-hand sides. More explicitly, an equation $(\forall X)\ t_1 = t_2$ must be such that $var(t_2) \subseteq var(t_1) = X$, where $var(t)$ denotes the variables in t. Such equations are interpreted as rules for term rewriting and hence are called **Σ-rewrite rules**. A **Σ-rewrite system** is a set of Σ-rewrite rules.

The basic aspects of term rewriting were introduced in Chapter 2. In what follows we give more formal definitions. A term $t \in T_{\Sigma(X)}$ can be viewed as a **labelled tree**, or a partial function from the naturals to $\Sigma(X)$ which gives a subterm of t when provided with an index. We then use t/v to indicate the subterm of t at index v; also, we use $t[v \leftarrow t']$ to denote the result of replacing the subterm of t indicated by v with the term t'. A **substitution** is an assignment $X \to T_{\Sigma(Y)}$. If θ is an assignment, then whenever $t' = \theta(t)$ we say that t' is an **instance** of t and that θ is a **match from t to t'**. We can now define the following:

Definition 4.29 A **one-step rewrite** using an equation $(\forall X)\ \ t = t'$ in E is a triple $\langle Y, t_0, t_1\rangle$ with a match $\theta : X \to T_{\Sigma(Y)}$ from a term t to a subterm $\theta(t)$ of t_0 at index v such that $t_1 = t_0[v \leftarrow \theta(t')]$. A one-step rewrite using a set E of equations defines a binary relation $\stackrel{1}{\Rightarrow}_E$ on $T_{\Sigma(Y)}$. Whenever $t_0 \stackrel{1}{\Rightarrow}_E t_1$ we say that t_0 rewrites to t_1 in one step. □

Term rewriting is obtained by repeated application of one-step rewrites. The corresponding relation on T_Σ is the transitive reflexive closure of $\stackrel{1}{\Rightarrow}_E$ and is denoted by $\stackrel{*}{\Rightarrow}_E$; we call it the **term rewriting** relation. The following property holds:

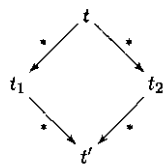
$$t \stackrel{*}{\Rightarrow}_E t' \quad \text{implies} \quad E \vdash (\forall \emptyset)\ \ t = t'$$

It says that two terms related in the term rewriting relation constitute an equation which is deducible from E. All this generalises to the case where the equations in E are conditioual (see [49] for the details).

A term that cannot be further rewritten is a **normal form** under E. If $t \stackrel{*}{\Rightarrow}_E t'$ aud t' is a normal form, t' is called a normal form of t. Using term rewriting as a model of computation, normal forms can be considered as the results of computations.

The above only explains many-sorted term rewriting. Order-sorted term rewriting is more general in that matching takes the sort hierarchy into account. For it to be safe, however, there need to be restrictions on the form of rewrite rules so that a rewrite is always *sort-decreasing*; this is guaranteed by requiring the least sort of the right-hand side of a rewrite rnle to be $\leq$ the least sort of the left-hand side. This avoids, for example, rewriting a term $f(g(x))$, where f expects an argument of sort S, to a term $f(y)$ using a rule $g(x) = y$ in which $S < LS(y)$; the problem is that $f(y)$ is an ill-formed term.

A rewrite system is **terminating** if and only if there is no infinite sequence $t_1, t_2, t_3, \ldots$ such that $t_1 \stackrel{1}{\Rightarrow}_E t_2 \stackrel{1}{\Rightarrow}_E t_3 \stackrel{1}{\Rightarrow}_E \ldots$. But even when every rewrite sequence is finite, the normal form of a term might not be unique. A rewrite system is **confluent** (or Church-Rosser) if and only if $t \stackrel{*}{\Rightarrow}_E t_1$ and $t \stackrel{*}{\Rightarrow}_E t_2$ imply the existence of a term t' such that $t_1 \stackrel{*}{\Rightarrow}_E t'$ and $t_2 \stackrel{*}{\Rightarrow}_E t'$. This property is depicted in the following diagram:



A rewrite system is **canonical** if it is terminating and confluent. In the following, we will use $[\![t]\!]$ to denote the unique normal form of t. For E a canonical system, the following property holds:

$$E \vdash (\forall \emptyset)\ \ t = t' \quad \text{iff} \quad [\![t]\!] = [\![t']\!].$$

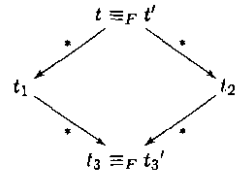
Hence, for canonical systems, we have an easy way of deciding when an equation is true: rewrite its terms and compare their normal forms. Moreover, when a set of equations forms a canonical rewriting system the operational and denotational semantics of the functional

level of FOOPS agree, in the sense that the normal forms of the terms constitute an *initial* algebra.

There is also term rewriting **modulo associativity** and **commutativity** [40]. Many commonly used operators have these properties and it is convenient to leave the system to deal with them. Furthermore, equations for associativity and commutativity destroy the property of termination of a rewrite system. Term rewriting modulo associativity and commutativity allows these properties to be considered as *built-in* to the rewrite system. This is achieved by

- considering terms as equivalent when they differ only in their parenthesisation and in the order of their factors; and by
- applying rewriting to equivalence classes of terms.

Term rewriting modulo associativity and commutativity is a particular case of the more general term rewriting **modulo a set of equations** F. All constructions and results of term rewriting generalise to this case by considering the equivalence classes of terms under the least congruence $\equiv_F$ on terms generated from F. In particular, the property of confluence of a rewrite system is defined *up to term equivalence* in the sense that different rewrites of equivalent terms must be reducible to equivalent terms, as depicted in the following diagram:



As an aside, note that because an equivalence class of terms can be infinite, the implementation of OBJ3 (on which FOOPS relies) uses a particular term as a representative of its equivalence class and applies term rewriting to it.

4.2 Object-level Semantics

In this section we describe the state of research towards providing a formal semantics for FOOPS. The operational semantics is based on a weak form of reflection [48] in which the current state of objects (the object "database") is encoded as a term in the functional level of FOOPS, and where methods are functions which take a database as argument and produce a new one: this reduces all computation to the functional level. Denotationally, an extension of OSA called hidden-sorted algebra is used [39]. However, this approach is still under development and cannot yet explain all of the features of the language.

4.2.1 Hidden-Sorted Algebra

Hidden-sorted algebra (HSA, hereafter) is a generalisation of OSA which formally captures the notion of object state and the basic information hiding concepts of the object paradigm. Its central tenet is that the states of objects are to be treated *observationally* with respect to properties: in HSA, two "object states" are equivalent if they give rise to the same behaviour, not just if they hold the same exact data. Therefore, HSA generalises satisfaction for order-sorted algebras.

This algebraic approach originated with the work of Goguen and Meseguer on abstract machines [47], and is also influenced by the work of Reichel [96, 97]. The most recent expositions are by Goguen [39], Goguen and Diaconescu [43], and Goguen and Kemp [45].

While HSA represents significant progress towards a formal denotational semantics for object orientation, it has not yet been developed to explain object creation and destruction (including identifiers). Because of this, this section uses the notation of the functional level of FOOPS, but its semantics will be given by HSA and not by OSA.

Informally, HSA: fixes a subalgebra D of data values, which would typically include the natural numbers and the booleans, for example; models states with *hidden sorts*; models data with *visible sorts*; models attributes with visible-valued functions of exactly one hidden-sorted argument; and models methods with hidden-valued functions of exactly one hidden-sorted argument.

4.2.1.1 Signature

The following definition formalises the previous informal discussion:

Definition 4.30 A **hidden-sorted signature** consists of

1. a set S of **sorts**,
2. a subset $V \subseteq S$ of **visible** sorts, where $H = S - V$ is called the set of **hidden** sorts,
3. an S-sorted signature Σ,
4. a V-sorted subsignature $\Psi \subseteq \Sigma$ called the **data signature**, and
5. a Ψ-algebra D, called the **data algebra**,

such that

1. for each $d \in D_v$ with $v \in V$ there is some $\psi \in \Psi_{\lambda,v}$ such that ψ is interpreted as d in D; for simplicity, we can assume that $D_v \subseteq \Psi_{\lambda,v}$ for each $v \in V$,
2. each $\sigma \in \Sigma_{w,s}$ with $w \in V^*$ and $s \in V$ lies in $\Psi_{w,s}$, and
3. each $\sigma \in \Sigma_{w,s}$ has at most one element of w in H.

Whenever $w \in S^*$ contains a hidden sort, we call $\sigma \in \Sigma_{w,s}$ a **method** if $s \in H$ or an **attribute** if $s \in V$. The first condition says that all data values are named by constants in Ψ. The second condition expresses the data "protection" requirement that Σ cannot add new operations on the data values of Ψ. And the final condition says that attributes and methods act on the states of single objects.

A **hidden-sorted specification** is a tuple $(S, \Sigma, V, \Psi, D, E)$, where (S, Σ, V, Ψ, D) is a hidden-sorted signature and E is a set of Σ-equations; we may abbreviate this to (Σ, D, E) or even just to (Σ, E), if the context permits. □

Example 4.31 Here we give a simple specification of stack objects; it assumes an error constant `err` of sort `Nat`[6].

```
fmod STACK is
  sort Stack .
  pr DATA .
  fn empty : -> Stack .
  fn top_  : Stack -> Nat .
  fn pop_  : Stack -> Stack .
  fn push  : Stack Nat -> Stack .
  var S : Stack .  var N : Nat .
  ax top empty = err .
  ax pop empty = empty .
  ax top push(S,N) = N .
  ax pop push(S,N) = S .
endf
```

Here we take `Stack` to be a hidden sort that denotes the state of stack objects. The constant `empty` represents the initial state of a stack, `pop_` and `push` are methods, and `top_` is an attribute.

Module `DATA` is for the data values used in `STACK`; in this case, only the natural numbers with the additional constant `err` are needed. The signature of `DATA` is given by Ψ in the preceding definition, with D some initial model for it. □

4.2.1.2 Models and Satisfaction

As already mentioned, the crucial aspect of HSA is its definition of satisfaction of an equation by an algebra, because it takes into account that objects have internal states, in the sense that only their observable properties are important. The above specification of stacks is interesting for this reason, because certain implementations, such as the one that uses an array and a pointer, do not satisfy the equation

```
ax pop push(S,X) = S .
```

[6] A more sophisticated definition of stacks would model the partiality of `top` and `pop` with subsorts, as illustrated in Chapter 2, but we choose a simpler one here for expository purposes.

literally, although they might do so *behaviourally*. In the example diagram in Figure 4.1, the pop leaves "garbage" behind, but since this garbage is not observable through any attribute, it does not affect the behaviour of the stack. Therefore, we say that the array-pointer implementation behaviourally satisfies the above equation.

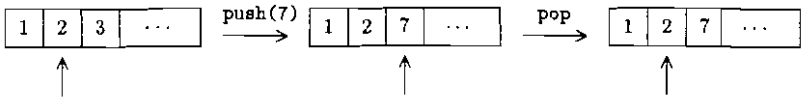


Figure 4.1: Junk after a pop.

The formal definition of behavioural satisfaction given next uses "contexts," which are terms of visible sort with a single free variable of hidden sort. The idea is that if, for each context, the same result is observed when the variable is substituted for both terms of an equation, then the equation is satisfied. The following are two example contexts for STACK, with the free variable denoted by z:

$\mathtt{top}(z)$
$\mathtt{top}(\mathtt{pop}(z))$

Definition 4.32 Given a hidden-sorted signature (S, Σ, V, Ψ, D) and an S-sorted set X of variable symbols, then a Σ-**context** is a visible sorted Σ-term having a single occurrence of a new variable symbol z. Call such a context **appropriate** for a term t iff the sort of t matches the sort of z.

A Σ-algebra A **behaviourally satisfies** a Σ-equation $(\forall X)\ t = t'$ iff A satisfies each equation $(\forall X)\ c(z \leftarrow t) = c(z \leftarrow t')$ where c is an appropriate Σ-context and $\leftarrow$ denotes substitution; in this case, we may write $A \models^D_\Sigma (\forall X)\ t = t'$.

Similarly, A **behaviourally satisfies** a conditional equation e of the form

$$(\forall X)\, t = t' \text{ if } t_1 = t'_1, ..., t_m = t'_m$$

iff for every assignment $\theta : X \to A$, we have

$$\theta^*(c(z \leftarrow t)) = \theta^*(c(z \leftarrow t'))$$

for all appropriate contexts c whenever, for $j = 1, ..., m$,

$$\theta^*(c_j(z \leftarrow t_j)) = \theta^*(c_j(z \leftarrow t'_j))$$

for all appropriate contexts c_j. As with unconditional equations, we write $A \models^D_\Sigma e$. □

In particular, note that for visible-sorted equations behavioural satisfaction reduces to ordinary satisfaction. The next step is to make the notion of model precise:

Definition 4.33 Given a hidden-sorted presentation $(S, \Sigma, V, \Psi, D, E)$, then a **model** of that presentation is a Σ-algebra A such that

1. the restriction of A to Ψ, written $A|_\Psi$, is isomorphic to D, and
2. each $e \in E$ is behaviourally satisfied.

A model of $(S, \Sigma, V, \Psi, D, E)$ is also called a $(S, \Sigma, V, \Psi, D, E)$-**algebra**, or a (Σ, D, E)-algebra. A Σ-algebra that satisfies just the first condition is called a (S, Σ, V, Ψ, D)-algebra, or a (Σ, D)-algebra, or simply a **hidden-sorted algebra**. □

Note that the intended models are not initial as in the functional level; rather, intended models are loose, so that any algebra that satisfies the above conditions for a given presentation is an acceptable model.

4.2.1.3 Extension to Order-Sortedness

The foregoing definitions have all been for the many-sorted case. We next generalise them to the order-sorted case. The principal aspect is that visible and hidden sorts form mutually exclusive inheritance hierarchies, such that a visible and a hidden sort are never related under the sort ordering.

Definition 4.34 Given an order-sorted signature $(V, \leq, \Psi)$ with a set V of visible sorts and an order-sorted Ψ-algebra D, then $(S, \leq, \Sigma, V, \Psi, D)$ is a **hidden order-sorted signature** if $(V, \leq, \Psi) \subseteq (S, \leq, \Sigma)$ as order-sorted signatures, and (S, Σ, V, Ψ, D) is a hidden many-sorted signature, such that no visible sort is related (by $\leq$) to any hidden sort. □

Definition 4.35 A **hidden order-sorted specification** is a tuple $(S, \leq, \Sigma, V, \Psi, D, E)$ where $(S, \leq, \Sigma, V, \Psi, D)$ is a hidden order-sorted signature and E is a set of Σ-equations. We may abbreviate this to just (Σ, D, E), as in the many-sorted case. □

Definition 4.36 A **hidden order-sorted** $(S, \leq, \Sigma, V, \Psi, D)$**-algebra** is an order-sorted $(S, \leq, \Sigma)$-algebra A that is also a hidden many-sorted (Σ, D)-algebra. □

4.2.1.4 Morphisms

As mentioned previously, morphisms play a central role in the formalisation of parameterised programming.

Definition 4.37 A **homomorphism** of hidden order-sorted algebras $h\colon M \to M'$ is a homomorphism of order-sorted algebras that is also a homomorphism of hidden many-sorted algebras. □

Definition 4.38 Given hidden order-sorted signatures Σ and Σ', then a **hidden order-sorted signature morphism** $\Phi\colon \Sigma \to \Sigma'$ is an order-sorted signature morphism $\Phi = (f, g)\colon \Sigma \to \Sigma'$ such that:

1. $f(v) = v$ for each $v \in V$;

2. $g(\psi) = \psi$ for each $\psi \in \Psi$;

3. $f(H) \subseteq H'$ (where $H' = S' - V$ and S' is the sort set of Σ');

4. if $\sigma' \in \Sigma'_{w',s'}$ and some sort in w' lies in $f(H)$, then $\sigma' = g(\sigma)$ for some $\sigma \in \Sigma$; and

5. for any hidden sorts h, h', if $f(h) < f(h')$ then $h < h'$.

For the many-sorted case, the last condition is omitted. □

The first three conditions say that hidden-sorted signature morphisms preserve visibility and invisibility for both sorts and operations, while the fourth and fifth conditions express the *encapsulation* of classes and subclasses, in the sense that no new methods or attributes can he defined on any imported class[7], and that any subclass relation between images of hidden sorts comes from a relation between their sources. A morphism of modules, i.e., of hidden-sorted specifications. must satisfy an additional condition:

Definition 4.39 Given hidden-sorted specifications (Σ, D, E) and (Σ', D, E'), then a hidden-sorted signature morphism $\Phi\colon (\Sigma, D) \to (\Sigma', D)$ is a **morphism** $\Phi\colon (\Sigma, D, E) \to (\Sigma', D, E')$ iff

$$M' \models^D_{\Sigma'} E' \text{ implies } \Phi(M') \models^D_{\Sigma} E$$

for all Σ'-algebras M', where the **reduct** $\Phi(M')$ of M' to Σ is M' viewed as a Σ-algebra. □

Note that the previous definition covers only the many-sorted case. The definition for the order-sorted case is given in [13], and relies on machinery whose complexity lies outside the scope of this chapter.

With these definitions, we can form categories of signatures, algebras and presentations analogous to those of the previous section.

4.2.1.5 Hidden Order-Sorted Conditional Equational Logic

The definition of Hidden Order-Sorted Conditional Equational Logic is similar to that of Order-Sorted Conditional Eqnational Logic. The vocabulary of this logic is given by hidden-sorted signatures and its sentences are hidden-sorted conditional equations; satisfaction is behavioural. No rules of deduction have been developed for this logic[8], but verification techniques are discussed in [43], based on work reported in [13]. They involve producing an ordinary order-sorted algebra from a hidden-sorted algebra, and using induction over all possible contexts to prove hidden-sorted equations; the advantage is that then ordinary deduction can be used.

[7] As will be explained in the next section, module importation is formalised in terms of (inclusion) morphisms.

[8] According to the theory of institutions, rules of deduction are not necessary for a system to qualify as a logic. The satisfaction relation is more important.

4.2.2 Reflection

In this section we show how to encode the object modules of FOOPS as programs in its functional level, thereby reducing all computation to this level. In addition to summarising [48], we show how to encode dynamic binding by collapsing all of the different versions of an operation into one, and give a deduction rule for sequential composition. For a different but related approach to this operational semantics see [9].

4.2.2.1 The Functional Level of FOOPS as a FOOPS Program

Let FL be the functional sub-language of FOOPS, and let FLAT-FL be a sub-language of FL that does not support module importation nor parameterisation; any FL program can be converted to an FLAT-FL program by simply copying modules inside each other. In FL we can define the syntax of FLAT-FL programs by giving the appropriate declarations for sorts, subsorts, functions, axioms, terms and so on, along with their constructors, and then supplying a sort `FlatFl` with constructor `fmod`. For example, (part of) a module for natural numbers could then be encoded as the following term of sort `FlatFl`:

```
fmod(NAT, sorts(Nat),
          fns((0 : -> Nat),(+ : Nat Nat -> Nat)),
          axs(+(0,N) = N))
```

For a complete description of FLAT-FL, we need axioms to describe the behaviour of `eval` to compute normal forms; it would accept as arguments a term of sort `FlatFl` and a term to normalise. But notice that the description of FLAT-FL programs is simply a module in FL, and therefore could be included as part of the FL library. Then, we do not really need to provide a definition of `eval` for FLAT-FL programs, as we could use the `eval` already available in the FL system.

This semantics uses FLAT-FL programs to describe states of the FOOPS database. Here is where the reflection comes in: the underlying equational logic will *change* with computations, to reflect how the database evolves as methods act upon objects. This is achieved by encoding the database as an `FlatFl` term, and encoding methods as functions on databases.

4.2.2.2 Representing the Database

Now we show how a FOOPS database can be described as a set of axioms written in FLAT-FL. First, we convert all class and subclass declarations to sort and subsort declarations, and all attribute declarations to functions (methods are left out for the moment). At any particular instance, all object identifiers are simply constants of the appropriate sort, and all attributes are just functions on these identifiers. To illustrate, we will use the classes `Acct` and `SavAcct` of examples 3.2 and 3.11 (see pages 53 and 71). After

```
eval new.Acct(JohnAcct, bal_ = 500, hist_ = emptyHist) .
eval new.SavAcct(MarySavAcct, bal_ = 750,
                 hist_ = emptyHist, rate_ = .05) .
```

the FOOPS database (as a term of sort FlatFl) would be

```
fmod FOOPS-DB is
  ...
  fn JohnAcct    : -> Acct .
  fn MarySavAcct : -> SavAcct .
  ax bal JohnAcct  = 500 .
  ax hist JohnAcct = emptyHist .
  ax bal MarySavAcct  = 750 .
  ax hist MarySavAcct = emptyHist .
  ax rate MarySavAcct = .05 .
endf
```

where "..." stands for all the sort and function declarations as explained above.

Since there is an order-sorted algebra associated with every FLAT-FL program, and every state of a FOOPS database can be described as an FLAT-FL program, every FOOPS database has an associated order-sorted algebra. In the above state of the database, the algebra has a carrier Acct with element JohnAcct and functions bal_ and hist_, and a carrier SavAcct with element MarySavAcct and functions bal_, hist_ and rate_; also included are the carriers and functions associated with imported sorts such as Money and Hist. Hence models: every FOOPS database has as model an initial algebra.

4.2.2.3 Evaluating Method Expressions

Methods are encoded as functions on FlatFl terms, such that they take a FOOPS database and produce a new, "updated" database. For example, the expression

```
credit(JohnAcct, 100)
```

would replace the axioms

```
ax bal JohnAcct  = 500
ax hist JohnAcct = emptyHist
```

with the axioms

```
ax bal JohnAcct  = 600
ax hist JohnAcct = << d ; 100 >>
```

where d would be the current date. This operational semantics views credit, as well as all other methods, as "edit" functions on terms of sort FlatFl.

Methods new.Acct and remove would work as expected, by adding or removing constants of the appropriate sort. Creation would also need to take care of the appropriate initialisations (including defaults), and removal of subtracting the axioms associated with its argument.

For every complex method expression, such as

```
transfer 100 from credit(JohnAcct,50) to debit(MarySavAcct,125)
```

there is a parse tree, where method invocations in nodes take the database from one state to another, and therefore each node is possibly evaluated in a different state. The evaluation is bottom-up, but there is no fixed evaluation order horizontally, as methods may be annotated with evaluation strategies for their arguments. Then, as the nodes in the tree are evaluated, database states are propagated upward. Denotationally, for each node in this tree there is a corresponding initial algebra.

Let us now clearly outline the steps to carry out the transformation of a FOOPS program P. We begin by defining A, the FL module containing all functional level definitions in P. Then we extend A to include all classes, subclasses and attributes in P, duly converted to sorts, subsorts and functions, respectively. We call this module $D_{0,P}$, the initial database of P. As was explained earlier, A and $D_{0,P}$ can be defined as terms of sort `FlatFl`. Now we can define a module P' as an extension of the FL module for FLAT-FL programs, and include in it the methods of P as the appropriate functions on `FlatFl` terms (for parsing, P' will also need to include a copy of the declarations of $D_{0,P}$). Then P' is used to compute new databases, beginning with $D_{0,P}$.

4.2.2.4 Dynamic Binding

The machinery described above provides a simple way of handling dynamically bound methods: use the database to check the sort of the object identifier, and make all the methods be conditional on this check. Those methods not redefined in all subclasses would need a disjunction of predicates to indicate that they applied to objects of more than one class. An alternative approach would involve changing the way rewrite rules are matched, as illustrated in our prototype implementation of FOOPS (see Chapter 5).

4.2.2.5 Rules of deduction

Deduction in FOOPS is viewed as computation in an equational logic whose axioms (i.e., the database) change as expressions are evaluated. Below we present the rules of deduction via a relation $\longrightarrow_P$ on pairs $\langle e, D\rangle$, where P is a FOOPS program, e a method expression and D is a reachable database described as an `FlatFl` term (i.e., reachable from the initial database associated with P). For notation, let A be the specification of all the functional level components in P, $e_1, \ldots, e_n$ method expressions, $v_1, \ldots, v_n$ functional level values, m a method symbol, f a function symbol, and o either a method or a function symbol. The rules are (dropping P as a subscript):

(1) $\langle o(e_1, \ldots, e_n), D\rangle \longrightarrow \langle o(v_1, \ldots, v_n), D_n\rangle$, where

$$\begin{aligned}
&\langle e_1, D\rangle \longrightarrow \langle v_1, D_1\rangle\\
&\langle e_2, D_1\rangle \longrightarrow \langle v_2, D_2\rangle\\
&\ldots\\
&\langle e_n, D_{n-1}\rangle \longrightarrow \langle v_n, D_n\rangle
\end{aligned}$$

(2) $\langle f(v_1, \ldots, v_n), D \rangle \longrightarrow \langle v, D \rangle$, where v is the normal form of $f(v_1, \ldots, v_n)$ under $A \cup D$.

(3) $\langle m(v_1, \ldots, v_k), D \rangle \longrightarrow \langle v, D' \rangle$, where D' is the normal form of $\mathtt{m}(v_1, \ldots, v_n, D)$, and $\mathtt{m}$ is the corresponding function on FlatFl terms defined in P'.

Finally, we give the deduction rule for evaluating the built-in sequential composition operator:

$$\langle e_1; e_2, D \rangle \longrightarrow \langle v_1; e_2, D_1 \rangle \longrightarrow \langle e_2, D_1 \rangle$$

where $\langle e_1, D \rangle \longrightarrow \langle v_1, D_1 \rangle$.

4.3 Semantics of Parameterised Programming

A semantics for parameterised programming in FOOPS is given in the framework of the theory of institutions, which formally captures the notion of "logical system" [41]. Institutions demonstrate that the most important aspect of specification and large-grain programming—namely, combining components to form larger ones—can be formalised *independently* of the underlying logical system of a particnlar langnage. By showing that a logical system is an institution, a number of important results regarding this and other aspects apply automatically, thus simplifying the task of giving a formal semantics to a language. Both Order-Sorted Conditional Equational Logic and Hidden Order-Sorted Conditional Equational Logic are institutions [13, 41, 43]. Various other logical systems have been shown to be institutions, inciuding First Order Logic and Horn Clause Logic [41].

In this section we define institntions and show how module reuse and interconnection in FOOPS are thus formalised. We assume knowledge of some basic category theory, including functors and colimits. As for notation, we nse boldface for the name of categories, $|\mathbf{C}|$ for the objects of category $\mathbf{C}$, 1_A to denote identity at A, $E^\bullet$ to denote the closure of a set of sentences E, and $f; g$ to denote the composition of arrows f and g, in diagrammatic order. This presentation is based on [39, 41].

4.3.1 Institutions

The essence of an institution is the relationship between syntax (i.e., sentences and signatures) and semantics (i.e., models). This relationship is embodied in the so-called Satisfaction Condition, which states that a change in syntax induces a corresponding change in semantics, such that *truth is invariant under change of notation*. An institution consists of

- a collection of signatures and signature morphisms, such that for each signature Σ there is
- a collection of Σ-sentences,
- a collection of Σ-models, and
- a satisfaction relation of Σ-sentences by Σ-models,

such that when a signature changes (by a signature morphism), satisfaction of sentences by models changes consistently. Here is the formal definition:

Definition 4.40 An **institution** $\mathcal{I}$ consists of

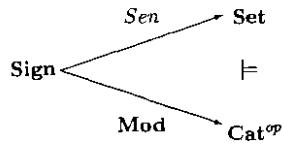
1. a category **Sign**, whose objects are called **signatures**,
2. a functor Sen : **Sign** $\to$ **Set**, giving for each signature a set whose elements are called **sentences** over that signature,
3. a functor **Mod** : **Sign** $\to$ **Cat**^{op} giving for each signature Σ a category whose objects are called Σ-**models**, and whose arrows are called Σ-(model) **morphisms**, and
4. a relation $\models_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times Sen(\Sigma)$ for each $\Sigma \in |\mathbf{Sign}|$, called Σ-**satisfaction**,

such that for each morphism $\phi : \Sigma \to \Sigma'$ in **Sign**, the **Satisfaction Condition**

$$m' \models_{\Sigma'} Sen(\phi)(e) \text{ iff } \mathbf{Mod}(\phi)(m') \models_\Sigma e$$

holds for each $m' \in |\mathbf{Mod}(\Sigma')|$ and each $e \in Sen(\Sigma)$. We will drop the signature subscripts on the satisfaction relation when it is not confusing. □

The following picture[9] may help visualise the above relationships:



$$\begin{array}{ccccc} \Sigma & & \mathbf{Mod}(\Sigma) & \models_\Sigma & Sen(\Sigma) \\ \phi\downarrow & & \mathbf{Mod}(\phi)\uparrow & & \downarrow Sen(\phi) \\ \Sigma' & & \mathbf{Mod}(\Sigma') & \models_{\Sigma'} & Sen(\Sigma') \end{array}$$

In the ensuing exposition, we assume a fixed but arbitrary institution $\mathcal{I}$.

Fact 4.41 Order-Sorted Conditional Equational Logic is an institution [41]. We will denote this institution by $\mathcal{OSL}$. □

Fact 4.42 Hidden Order-Sorted Conditional Equational Logic is an institution [13]. We will denote this institution by $\mathcal{HSL}$. □

The proofs of many of the facts and theorems that follow depend crucially on the Satisfaction Condition.

[9] We thank the authors of [41] for letting us borrow the LaTeX code for this picture.

4.3.2 Theories

Definition 4.43 A **theory** is a presentation (Σ, E) such that E is closed. We may also call (Σ, E) a Σ-theory. □

Given a theory $T = (\Sigma, E)$ and a Σ-model M, we say that M satisfies T iff $M \models P$ for each $P \in E$, written $M \models E$; we write $M \models T$ whenever M satisfies T. Therefore, a Σ-theory classifies Σ-models by whether or not they satisfy it. In particular, if we let E^* be the collection of all Σ-models that satisfy each sentence in E, then we have that E^* is a full subcategory of $\mathbf{Mod}(\Sigma)$.

Definition 4.44 The **denotation** T^* of a Σ-theory T is the set of models that satisfy it:

$$T^* = \{M \mid M \models T\}$$

□

Definition 4.45 The theory M^* of a Σ-model M is the set of all the sentences it satisfies:

$$M^* = \{E \mid M \models E\}$$

□

For example, for the FOOPS theory `TRIV`,

```
fth TRIV is
  sort Elt .
endfth
```

$TRIV^* = \mathbf{Set}$, the category of all sets. (Here and onwards, we use typewriter mode for module names and italics for the associated theories, so that $TRIV$ is the theory of module `TRIV`.) For the theory

```
fth MONDID is
  sort Elt .
  fn e : -> Elt .
  fn _·_ : Elt Elt -> Elt .
  vars A B C : Elt .
  ax A · e = A .
  ax e · A = A .
  ax A · (B · C) = (A · B) · C .
endfth
```

$MONOID^* = \mathbf{Monoid}$, the category of all monoids.

4.3.3 Dependent Theories

Relationships between theories will help describe phenomena such as module inheritance and parameterisation.

Definition 4.46 A **view** or **theory morphism** is a siguature morphism $\phi : \langle \Sigma, E \rangle \to \langle \Sigma', E' \rangle$ such that $P \in E$ implies $\phi(P) \in E'^{\bullet}$. This defines a category of theories over an institution. □

Definition 4.47 A **subtheory** is a view $\langle \Sigma, E \rangle \to \langle \Sigma', E' \rangle$ where $\Sigma \subseteq \Sigma'$ and $E \subseteq E'$. □

The following is entailed by the definition of theory morphism:

Fact 4.48 $\phi : \langle \Sigma, E \rangle \to \langle \Sigma', E' \rangle$ is a theory morphism iff $M' \models E'$ implies $M' \models \phi(E)$ for all Σ' models M'. Here $\phi(E) = \{\phi(P) \mid P \in E\}$. □

The practical consequence of the last fact is that to check whether a signature morphism is a theory morphism there is no need to compute closures (which might be infinite).

Definition 4.49 The **denotation** of a theory morphism $\phi : T \to T'$ is its **reduct** or forgetful functor $\phi^* : T'^* \to T^*$, which sends a T'-model M' to the T-model $\mathbf{Mod}(\phi)(M')$ and sends a T'-model morphism $f' : M' \to N'$ to $\phi^*(f') = \mathbf{Mod}(\phi)(f') : \mathbf{Mod}(\phi)(M') \to \mathbf{Mod}(\phi)(N')$. □

For example, *TRIV* is a subtheory of every theory with at least one sort, and thus $TRIV \subseteq MONOID$. This induces a forgetful functor from **Monoid to Set** that sends a monoid to its underlying set. In particular, there is a subtheory relation between the theory of a generic module and the theory of its parameters. For example,

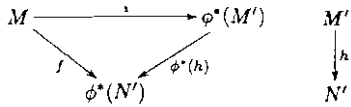
```
fmod B-SEARCH-TREE[X :: POSET] is
  sort Tree .
  ...
endf
```

is associated with a theory *B-SEARCH-TREE* which includes *POSET*. Its denotation is a forgetful functor from **B-Search-Tree** to **Poset**.

4.3.4 Constraints

A coustraiut is a special kind of sentence that expresses things such as initiality requirements on models, or that a model is freely generated by another. In this sense, for example, constraints help characterise the various modes of module inheritance in FOOPS, including parameterisation. The notion of freeness is essential here, and we begin with its definition:

Definition 4.50 Given a theory morphism $\phi : T \to T'$, let $\phi^* : T'^* \to T^*$ he its forgetful functor, and let M be a T-model. Then a T'-model M' is **free** over M (with respeet to ϕ^*) iff there is some morphism $\imath : M \to \phi^*(M')$ such that given any other T'-model N' and morphism $f : M \to \phi^*(N')$, there is a unique T'-model morphism $h : M' \to N'$ such that $\imath ; \phi^*(h) = f$. The following diagrams illustrate this (the triangle commutes):



□

Intuitively, M' is the "best" T'-model that extends M along ϕ. (Moreover, free models are unique up to isomorphism.) A common example of this situation is a free algebra over some set X of variables.

Definition 4.51 An institution is **liberal** iff for every theory morphism $\phi : T \to T'$ and T-model M, there is a T'-model M' such that M' is free over M with respect to ϕ^*. □

Fact 4.52 $\mathcal{OSL}$ is a liberal institution. □

Fact 4.53 $\mathcal{HSL}$ is not a liberal institution [45]. □

Definition 4.54 Given a theory morphism $\phi : T \to T'$, the function $M \mapsto M'$ for $M \in T^*$ and $M' \in T'^*$, where M' is free over M with respect to ϕ^*, extends uniquely to a functor, denoted $\phi^\$: T^* \to T'^*$, and called a **free functor**. □

In the following, without loss of generality, we will use the notation $\phi^\$(M)$ even if the institution involved is not liberal.

To explain parameterisation we need a way to state that a model is free over *part* of a theory. In particular, we want models of generic modules to be free over the parameters. The next definition formalises this.

Definition 4.55 Given a theory morphism $\phi : T \to T'$ and a T'-model M', we say that M' is **ϕ-free** iff $\phi^\$(\phi^*(M'))$ is free over $\phi^*(M')$ and the morphism $1_{\phi^*(M')} : \phi^\$(\phi^*(M')) \to M'$ is an isomorphism. □

Now the definition of the new kind of sentence:

Definition 4.56 Given a signature Σ, a **Σ-constraint** is a pair $\langle \phi : T \to T', \theta : Sign(T') \to \Sigma \rangle$ where T and T' are theories and $Sign(T')$ is the signature of T'. Given a Σ-model M and a Σ-constraint $c = \langle \phi : T \to T', \theta : Sign(T') \to \Sigma \rangle$, we say that M satisfies c iff $\theta(M)$ satisfies T' and is ϕ-free. The **ψ-translation** of a constraint $c = \langle \phi, \theta \rangle$ is the constraint $\langle \phi, \theta; \psi \rangle$, denoted $\psi(c)$. □

The next fact establishes that constraints can be treated like regular sentences:

Fact 4.57 Given an institution $\mathcal{I}$, a new institution $\mathcal{C}(\mathcal{I})$ is obtained from $\mathcal{I}$ as follows: its signatures are those of $\mathcal{I}$, and its sentences are those of $\mathcal{I}$ plus the constraints as new sentences, with satisfaction and translation as in the previous definition. □

Some examples at the functional level of FOOPS might help clarify matters. An `fmod` declaration indicates that only initial models are accepted. This is captured by a constraint in which the source of the theory morphism is the empty theory $\emptyset$ (which has only one model with no sorts or operations)[10]. Constraints of this kind are called **initiality constraints**. For example, the equations of the theory of

```
fmod NAT is
  sort Nat .
  fn 0   : -> Nat .
  fn s_  : Nat -> Nat .
  fn _+_ : Nat Nat -> Nat [comm] .
  vars M N : Nat .
  ax 0 + N = N .
  ax s M + N = s(M + N) .
endf
```

include the constraint $\langle \phi : \emptyset \to NAT, 1_\Sigma \rangle$, which says that any model of NAT must be free over the empty theory.

Parameterised theories have as their denotation a forgetful functor to the class of models of the parameter. For example,

```
fth VECTOR[X :: FIELD] is
  sort Vector .
  ...
endfth
```

has as its denotation the functor $F^* : VECTOR^* \to FIELD^*$. Note here that no constraints are necessary: any model that satisfies the equations of $VECTOR$ (which includes $FIELD$) is accepted.

On the other hand, for parameterised (executable) modules the intention is that their models be free over the parameters; i.e., the rest is to be interpreted initially. For example, the theory of

```
fmod LIST[X :: TRIV] is
  sort List .
  ...
endf
```

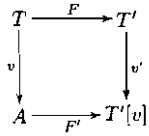
will also include the constraint $\langle \phi : TRIV \hookrightarrow LIST, 1_{Sign(LIST)} \rangle$, which says that any model of $LIST$ must be free over $TRIV$. Hereafter, we will call the theory of either generic (including those at the object level) a **parameterised theory**.

[10] To see this, substitute $\emptyset$ for M in the diagram of Definition 4.50.

4.3.5 Instantiation

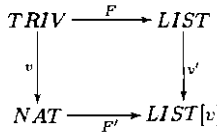
The semantics of instantiating generics is based on the categorical concept of pushout (colimit). It originated with Clear [14], based on ideas in [33].

Definition 4.58 Given a parameterised theory $F : T \to T'$ and a view $v : T \to A$ of an "actual" theory A as an "instance" of T, the result of "applying" F to v is the pushout diagram shown below. The pushout object is $T'[v]$, and its denotation is $(T'[v])^*$. The morphisms F' and v' are derived from F and v, respectively.



□

Example 4.59 Consider the generic module LIST above. Assuming a view v from $TRIV$ to NAT, we derive the pushout diagram below, where the F in the previous definition is the inclusion $TRIV \hookrightarrow LIST$:



□

But these constructions are only significant if colimits always exist for the institutions of interest.

Fact 4.60 $\mathcal{OSL}$ and $\mathcal{HSL}$ have finite colimits for all diagrams over the category of theories. □

4.3.6 Module Hierarchies

The semantics of module importation is simply an application of the concepts that we have developed so far, and is based on subtheories and on constraints to characterise the mode of importation. For example, the theory of module BOOL includes the initiality constraint $\langle \phi : \emptyset \to BOOL, 1_{Sign(BOOL)} \rangle$. Importing a module in protecting mode means that its denotation is preserved; i.e., all its properties must be satisfied by models of the importing module. The constraints of the imported modules need to have their signatures translated in order for this to make sense. For example, importing BOOL into STACK,

```
fmod STACK[X :: TRIV] is
  sort Stack .
  pr BOOL .
  ...
endf
```

means that $STACK$ includes all of the declarations and equations of $BOOL$, and in particular these constraints:

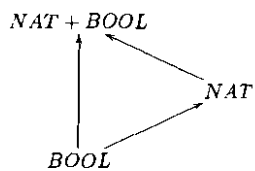
- $\langle \phi : TRIV \hookrightarrow STACK, 1_{Sign(STACK)} \rangle$, which says that a model of $STACK$ must freely extend $TRIV$; and,
- $\langle \phi : \emptyset \to BOOL, \theta : Sign(BOOL) \to Sign(STACK) \rangle$, which says that the reduct of a $STACK$ model M to $BOOL$ must be an initial model.

Note how $BOOL$'s constraint was translated to fit its new context. Observe also that by requiring initiality of the reduct, we obtain the so-called "no junk, no confusion" property.

The remaining two importation modes, extending and using, can be explained by relaxing the interpretation of constraints to allow arrows that are not bijective for the morphism $\phi^{\$}(\phi^*(M')) \to M'$ (see Definition 4.50). For extending, the arrow is relaxed to injectiveness ("no confusion"): for using, any arrow is acceptable.

4.3.7 Module Expressions

The semantics of the rest of the module expression language of FOOPS also falls out of the above presentation. A renaming is just a bijective morphism in the category of signatures. Sums are colimits, such that modules imported multiply are shared; this is a natural consequence of inclusions. For example, the sum `NAT + BOOL` has only one copy of `BOOL`, even though in FOOPS `BOOL` would be automatically imported into `NAT`. The following diagram, in which arrows are theory inclusions, may help visualise this:



The denotation of the sum is $(NAT + BOOL)^*$, where $NAT + BOOL$ is the colimit of the diagram. Given that pushouts explain instantiation, we have that colimits explain module expressions in general. Finally, note the correspondence of these diagrams to those for contexts given in Chapter 3.

4.4 Summary

This chapter introduced the basic aspects of order-sorted algebra, hidden order-sorted algebra and institutions, which, together with order-sorted term rewriting, provide a semantic foundation for FOOPS. Further work remains for hidden order-sorted algebra, so that features such as object creation, object deletion and method redefinition are formally captured. Additionally, the information hiding facilities of FOOPS need to be formalised; research reported in [29] could provide a starting point in this direction.

Chapter 5

Implementation

The reward of effort is not only the goal but also the struggle itself.

— Fortune cookie

Our prototype implementation of FOOPS is a translator using OBJ3+ as its target language. OBJ3+ is OBJ3 [53] extended with data structnres for manipnlating persistent entities; subsequently, we may also refer to these structures with their stored state as the *FOOPS database*. Access to the FOOPS database from an OBJ3+ program is possible by using **built-in axioms**, a special kind of axiom provided by OBJ3, having the following syntax:

```
bq  ⟨Term⟩ = ⟨Lisp⟩ .
cbq ⟨Term⟩ = ⟨Lisp⟩ if ⟨Term⟩ .
```

where ⟨*Lisp*⟩ is any expression in Kyoto Common Lisp. How these axioms are implemented as rewrite rules will be explained with examples in this chapter. (There can also be built-in sorts aud one other kind of built-in axiom, but this prototype does uot employ either.)

In general, the translator works by converting class declarations to sort declarations, and attribute and method declarations to function declarations; also, some syntactic conversions are performed (see the chart in Appendix A). Direct method axioms are translated into built-in axioms whose right-hand sides invoke operations that update the FOOPS database; axioms for derived attributes and methods can be interpreted directly as rewrite rules. Also, built-in axioms are generated to fetch the valne of stored attributes. To implement dynamic binding, we made a small change to OBJ3+'s term rewriting engine.

The system architecture in terms of data flow is shown in Figure 5.1. From a FOOPS text, the translator generates an OBJ3 text, and this is evalnated by the OBJ3 interpreter, which interacts with the object database. The box labelled "FOOPS PRELUDE" stands for some object level modules that are automatically available, like those for lists and sets; similarly for the box labelled "OBJ3 PRELUDE", but at the functional level. The box labelled "RESULT EXPRESSIONS" is for the output of the interpreter. OBJ3+ comprises the boxes labelled "OBJ3 INTERPRETER", "OBJ3 PRELUDE", and "FOOPS DATABASE".

To give a very rough indication of the effort involved in creating this implementation, the size of the code added to OBJ3 is about 10,000 lines.

This prototype implementation supports most of the features of FOOPS. Missing are vertical structuring facilities and module blocks. Minor omissions and special cases are carefully documented in [95], the reference manual for the system.

This chapter describes the information stored for each module, how classes are translated and the information stored for each one, the representation of objects and how creation and destruction are implemented, the generation of attribute and method axioms, the implementation of redefinitions and of dynamic binding, and how theories, views and module expressions are processed (on top of any processing done by OBJ3+). We also provide ideas for improving this prototype implementation and for realising some of the features that it does not currently support.

5.1 Modules

The translator keeps information about object-level modules in the **module table**. Each entry in this table consists of:

- a pointer to an OBJ3+ module structure, which is a record that holds information about a module's sorts, variables, axioms, etc.; and,
- a list of the classes that a module declares. Each element in this list is a reference to another table that stores information about classes (see below).

Information regarding functional-level modules is not recorded in this table, as all that FOOPS needs to know about them is stored in OBJ3+'s data structures.

5.2 Sorts and Classes

Sort declarations in FOOPS do not require translation, but support for error supersorts, voids and default values for attributes requires that some extra declarations be included for each sort. For a sort S, these are:

```
sort S? .
subsort S < S? .
op void-S : -> S? .
op S-pc : -> S .
bq S-pc = (get-principal-constant (term$sort self)) .
```

where `get-principal-constant` is an internal routine that returns a term whose top symbol is the principal constant of a given sort, in this case the sort of the term stored in variable `self`[1]. This variable is automatically provided by OBJ3, and during evaluation its value is always the term on the left-hand side of the rule being applied. Consequently, the

[1] "op" is OBJ3's syntax for function declarations.

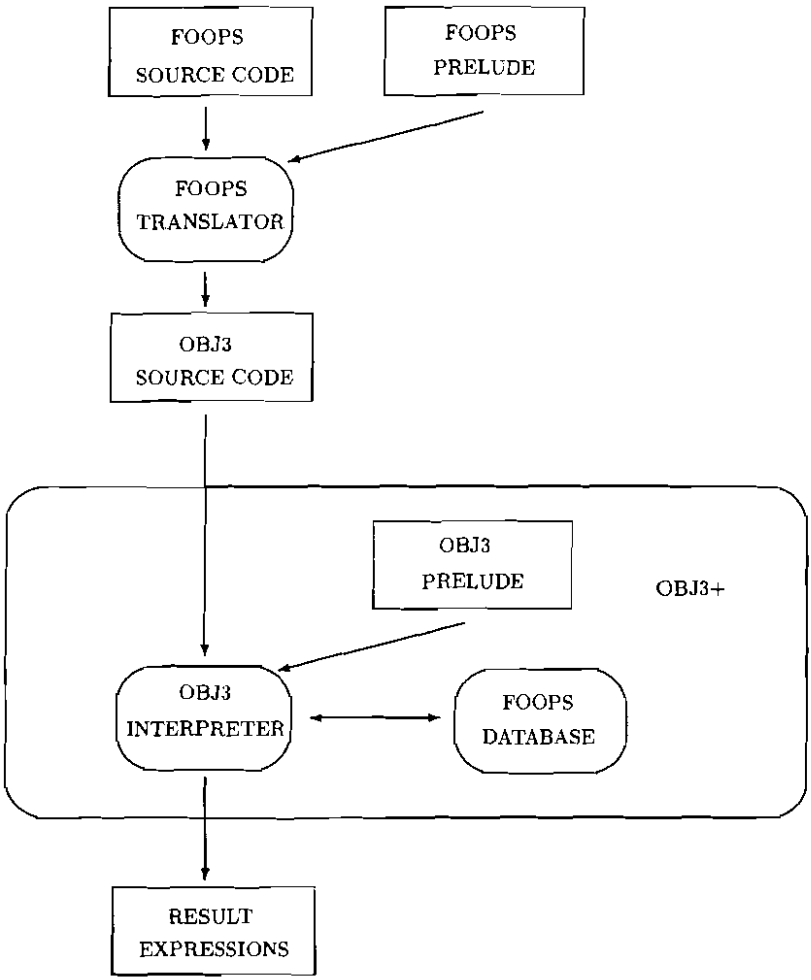


Figure 5.1: Data-flow diagram of the FOOPS system.

evaluation of the term S-pc yields the principal constant of sort S (of course, S-pc is not considered). The translator uses this operation to implement object creation.

Classes are mapped onto sorts, and in addition to declarations similar to the above, the translator generates several others. For a class C, the following operators are automatically generated:

- `invent-C-id`, which creates a unique object identifier for an object of class C;
- `make-C-id`, which takes a string and makes it into an object identifier;
- `make-C-object`, which creates an object of class C and assigns defaults to all of its attribntes;
- `remove`, which destroys objects:
- `exists`, which tests for the presence of objects; and,
- `all-C`, which creates a list of all the objects of class C.

Only the last three of these operators are accessible to users of FOOPS. Also generated is the sequential composition operator. It is defined by the following declarations:

```
op _ ; _ : Universal Univeraal
           -> Universal [strat (1 2 0) gather (E e)] .
vars M M' : Universal .
eq M ; M' = M' .
```

where `Universal` is a built-in supersort of all sorts[2]. The items in brackets that follow the operator's signature specify its *evaluation strategy* and its *gathering pattern*, respectively. The first, `strat (1 2 0)`, gives the order in which argnments to it are to be evaluated; in this case, it specifies left-to-right evaluation. `gather (E e)` is more complicated to explain and we refer those interested in its details to [53]; here it will snffice to say that it specifies that the operator is left associative. The effect of the above declarations is that expressions such as E1 ; E2 are evaluated as follows: first, E1 is evaluated; then. E2 is evaluated; and finally, the entire expression is rewritten to whatever E2 evaluated to.

5.2.1 The Class Table

Information about classes is kept in the **class table**. Each entry in this table corresponds to a class, and contains the following information:

- the name of the sort to which the class is mapped. In OBJ3+, sorts are represented internally as record structures with information about their name, module of origin, etc.

[2] "eq" and "ceq" is OBJ3 syntax for unconditional and conditional axioms.

- the attributes associated with the class. These are split into three lists: the stored attributes, the derived attributes, and the undefined attributes. The stored attributes is a list of pairs, with first component an OBJ3+ operator structure (which stores the name, rank, etc. of an operator) and second component a term structure (i.e., an expression), which holds the default value (if any) for the attribute. The derived and undefined attributes are simply lists of operator structures. This last kind of attribute only exists internally, and the list represents those attributes which, in the midst of processing an object module, are not yet known to be either stored or derived (because this classification is determined by the way an attribute is used, as explained in Section 2.2.2).

- the methods associated with the class. They are stored as a list of OBJ3+ operator structures, and are used for type checking axioms.

- the attributes and the methods that the class redefines. Both are lists of operator structures. The attributes are used to determine the structure of objects and for type checking axioms. The methods are only used for type checking axioms.

- the `set-C` operator. For a class `C`, this is a built-in operator for updating objects of class `C` (Section 5.5 describes it in detail).

- the list of identifiers of the objects of the class. This list is used by `all-C` to compute metaclasses.

For each class, this table does not contain any data related to its superclasses, except that `set-C` must account for inherited stored attributes; not copying any other information about superclasses saves space and simplifies module instantiations and renamings (see Section 5.7). A class inheritance hierarchy is not explicitly kept by the translator because this is already done in OBJ3+ for sorts, and classes are mapped onto sorts; i.e., the translator simply interfaces the underlying OBJ3+ implementation of the sort inheritance hierarchy. However, the class table helps prevent users from merging the sort and class hierarchies.

5.3 Objects

Object identifiers are represented as nullary operators in OBJ3. The state of an object is a list of pairs, with first component the string name of an attribute and second component the attribute's current value. This information is stored in the **object table**, which is a hash table indexed by object identifiers. For example, in the context of

```
class Pair .
ats (fst_) (snd_) : Pair -> Nat .
```

the entry for an object with identifier `P`, attribute `fst_` equal to `0` and attribute `snd_` equal to `1` is something of the form $\langle \texttt{P}, (\langle \texttt{fst\_}, 0\rangle, \langle \texttt{snd\_}, 1\rangle)\rangle$.

5.3.1 Object Creation

Given the flexibility for creating objects in FOOPS, their implementation is quite involved. First, because the creation method has optional arguments and does not restrain their positions, it cannot be directly mapped onto an operator in OBJ3+. Also, evaluating an invocation of this method may require the creation of many objects and the assignment of default values to attributes. To parse a call to `new`, a small recursive descent parser was built, and it is used to process the right-hand side of axioms and the arguments to `eval` commands. Our approach to the translation of `new` is to convert a call to it to a call to the appropriate `set-C` operator. For example, in the context of

```
class ColourPair .
sort Colour .
fns red blue : -> Colour .
ats (fst_) (snd_) : ColourPair -> Colour .
```

the term

```
new.ColourPair(P, fst_ = blue)
```

is translated to

```
set-ColourPair(make-ColourPair-id('P),blue,Colour-pc)
```

where the operation `make-ColourPair-id` interns `P` as an object identifier of class `ColourPair` and the nullary operator `Colour-pc` retrieves the principal constant of sort `Colour`; `'P` is OBJ3+'s notation for a string consisting of character `P`. Interning a string as an object identifier amounts to making its contents into a nullary operator of the appropriate sort and entering it into the object and class tables; the first step will later on permit the proper parsing (by OBJ3+) of expressions involving identifiers. For example, after the above expression is evaluated, OBJ3+ will be able to parse terms such as `fst P`.

Principal constants are not directly inserted in the call to `set-C` because they may change depending on module importations. For example, if the above creation expression is given on the right-hand side of some axiom in a module with no principal constant of sort `Colour`, and this module is later extended by some other module that declares a constant `X` of sort `Colour`, then `X` is the default for `snd_` in the context of this other module. Explicit default expressions, however, are always inserted directly in calls to `set-C`.

When `new` is invoked but an identifier is not provided, the `set-C` expression will involve the operation `invent-C-id` to generate and intern a fresh object identifier. For example,

```
new.ColourPair(fst_ = red, snd_ = blue)
```

translates to

```
set-ColourPair(invent-ColourPair-id,red,blue)
```

A fresh identifier is not inserted directly because in general it would not be correct to do so. For instance, if the previous creation expression appears in the right-hand side of some axiom, and this axiom is used repeatedly for rewriting terms, then all its uses after the first one would result in errors regarding the uniqueness of object identifiers.

To explain the computation of defaults for complex attributes, consider the following fragment:

```
omod CAR is
  classes Car Motor .
  sort Colour .
  ...
  at colour : Car -> Colour .
  at motor  : Car -> Motor .
  ...
endo
```

An expression such as

```
new.Car(TheMachine, colour = red)
```

translates to

```
set-Car(make-Car-id('TheMachine),red,make-Motor-object('Car))
```

in which the operation `make-Motor-object` creates an object of class `Motor` and assigns defaults to all of its attributes. The argument `'Car` is used to help detect cycles and avoid infinite loops, as was explained in Chapter 2.

5.3.2 Entry-time Objects

The declaration of an entry-time object is translated into an operator declaration in OBJ3+; also, the identifier is inserted into the object table and into its class' entry in the class table. Each axiom that gives an initial value to an attribute of an entry-time object is interpreted as an update to the corresponding object. However, this implementation does not compute defaults for those attributes which are not initialised (but this would be easy to add—the underlying machinery required is already there).

5.3.3 Object Destruction

The destruction of an object (with the `remove` method) requires updates to various internal data structures. The nullary operator that corresponds to the object's identifier must be removed from the current module, from OBJ3+'s parsing dictionaries, and from the class' entry in the class table. Moreover, the object must be removed from the object table.

The present implementation of `remove` creates dangling references, because other objects may still refer to an object that has been destroyed. This means that something must be done when such locations are de-referenced and about the possibility of dangling references

being "undangled," a situation that would arise if the identifier of a deleted object could be reused. The first concern is addressed in the next section. The reuse of identifiers is averted by using the **deleted object table**, which stores the identifiers of the objects that have been removed. Therefore, object creation also entails verifying that identifiers are unique over this table too.

5.4 Attribute Axioms

Axioms for derived attributes do not require translation: they are directly executable. However, for each stored attribute, a built-in axiom that queries the FOOPS database for its value is generated. Because whether an attribute is stored or derived depends on how it is used in axioms, this generation does not take place until all the axioms declared in the module have been analysed. For a stored attribute such as

```
at length : List -> Nat .
```

the corresponding built-in axiom is

```
[get-attr] beq length(L) = (get-attribute-value L 'length) .
```

The bracketed string "`get-attr`" that precedes the axiom is its **label**. This is a feature of OBJ3+ that the translator employs to facilitate module expression evaluation, which may require that some of these axioms be regenerated (this is explained further below). The right-hand side of the axiom is a call to the FOOPS database routine `get-attribute-value`, of two arguments: an object identifier and the name of an attribute. It retrieves the corresponding object from the object table and from it the named attribute's value; in this case, `'length` is Lisp's notation for symbol `length`. If `get-attribute-value` cannot find an object with the given identifier, its stops the evaluation process and displays an error message. A sequential name search is used to extract the value of an attribute from an object.

The generation of axioms of this kind may call for new variables to be declared. If a variable of the required sort is available in the module, then it is used; otherwise, one is created.

In a rewrite that involves a built-in axiom, the variables on the right-hand side of the axiom are bound to the *internal* representation of the matched values of corresponding variables on its left-hand side. For instance, in a rewrite of the term

```
length(X)
```

the first argument to `get-attribute-value` will be a Lisp data structure that represents the term `X`, and which therefore carries sort information, etc.

5.4.1 Dangling References

Since FOOPS supports explicit object destruction, its storage system must deal with dangling references. To illustrate how they are handled, we will use the specification shown in Figure 5.2.

```
omod PERSON is
  class Person .
  protecting NAME .
  at name_    : Person -> Name .
  at spouse_ : Person -> Person .
  me marry_and_ : Person Person -> Person .
endo
```

Figure 5.2: A partial specification of a class of persons.

Assume that two objects of class `Person`, with identifiers `David` and `Agnes`, are married, and thus `David`'s spouse attribute stores `Agnes` and vice versa. Now consider the following evaluations:

```
eval remove Agnes .
eval spouse David .
```

After the first `eval`, `Agnes` is a dangling reference, because the object associated with it has been removed but `David` still refers to it. Therefore, the result of the second evaluation should be `void-Person` (as explained in Chapter 2). To accomplish this, the axiom generated for stored complex attributes is slightly different from the one shown previously. For `spouse`, it is:

```
[get-attr] beq spouse P = (get-complex-attribute-value P 'spouse_) .
```

where the database routine `get-complex-attribute-value` works as follows. If the object identifier stored by the given attribute is in the object table, then it is returned; otherwise, the corresponding `void-C` is returned. For the latter case, it may be necessary to "wrap" the void in a retract so that the attribute's coarity is respected (recall that OBJ3 inserts retracts only during parsing). The evaluation of `spouse David` thus yields `r:Person?>Person(void-Person)`, as desired.

5.5 Method Axioms

While indirect method axioms can be executed verbatim, the DMAs that define a method are grouped and converted into a single axiom whose right-hand side interfaces the FOOPS database. Since DMAs need not be given in any particular order, the generation of this axiom occurs only when an entire module has been analysed.

The grouping of DMAs is done using the **method axiom table** (MAT). Each entry in this table indicates how a method updates attributes. Specifically, each entry stores:

- a method pattern that is common to a group of DMAs. For example, for the DMAs

```
ax a(m(X)) = expr1 .
ax b(m(X)) = expr2  if C .
```

```
omod PAIR is
  class Pair .
  pr NAT .
  ats (fst_) (snd_) : Pair -> Nat .
  me incr-fst : Pair -> Pair .
  me swap     : Pair -> Pair .
  me make-snd-zero : Pair -> Pair .
  var P : Pair .  vars N1 N2 : Nat .
  ax fst incr-fst(P) = fst(P) + 1 .
  ax fst swap(P) = snd(P) .
  ax snd swap(P) = fst(P) .
  cax snd make-snd-zero(P) = 0  if snd(P) > 0 .
endo
```

Figure 5.3: A specification of a class of pairs of natural numbers.

it is the term `m(X)`; and

- a list of updates, each of which is a pair with first component an attribute name and second component another pair: a term that represents a new value for the attribute and the condition under which this value is to be assigned to the attribute. For unconditional DMAs, the condition is recorded as `true`. By way of illustration, the above DMAs would be stored in the MAT as something of the form

  `(m(X), ((´a,((expr1,true))), (´b,((expr2,C)))))`

The MAT is implemented as a hash table, with its key being the method pattern.

Once all of the DMAs in a module have been analysed, an axiom is generated for each entry in the MAT. For a method associated with a class `C`, the right-hand side of the axiom is a call to the custom-built operator `set-C`, which updates all of the attributes of an object. To exemplify this process, consider the specification of pairs in Figure 5.3. Operator `set-Pair` is defined by the following declarations:

```
op set-Pair : Pair Nat Nat -> Pair .
[set] beq set-Pair(P,N1,N2) = (let ((objaddr (get-object-address P)))
                                 (write-attribute objaddr ´fst_ N1)
                                 (write-attribute objaddr ´snd_ N2)
                                 P) .
```

In the axiom, the FOOPS database routine `get-object-address` accepts as argument an object identifier and returns a pointer to its entry in the object table. With this pointer, an attribute name and a value, `write-attribute` updates the object's state with this value. If `get-object-address` cannot find the object, it stops the evaluation process and displays an error message. Note that the left-hand side of the axiom must not repeat variables.

because otherwise it would not provide matches unless all variables with the same name could be bound to the same term.

Next we show the resulting axioms for the methods in Figure 5.3. The axiom for `incr-fst` is

```
eq incr-fst(P) = set-Pair(P,fst(P) + 1,snd(P)) .
```

The one for `swap` is

```
eq swap(P) = set-Pair(P,snd(P),fst(P)) .
```

Finally, the axiom for `make-snd-zero` is

```
eq make-snd-zero(P) = set-Pair(P,fst(P),
                               if snd(P) > 10 then 0 else snd(P) fi) .
```

This scheme is simple but may involve unnecessary updates to attributes which do not change, such as `fst_` in the last axiom; see Section 5.8 for some ideas on how to improve upon this.

5.6 Class Inheritance and Redefinitions

Chapter 2 described several situations in which a class definition can be erroneous because of inheritance clashes. Detecting this kind of error involves traversing the class inheritance hierarchy and simultaneously accessing the class table to fetch information about attributes and methods. An analogous algorithm is used to check whether an attribute or method redefines another.

When an attribute or a method is a redefinition, adherence to various rules must be ascertained. These rules are:

- **encapsulation rule**: an attribute `a` may be redefined by an attribute `a'` associated with class `C` provided `a'` and `C` are declared in the same module; similarly for methods. (See Section 3.8.)
- **variance rule**: as described in Section 2.2.7.
- **attribute-redefinition rule**: a derived attribute `a` may be redefined by an attribute `a'` that can be either stored or derived; but if `a` is a stored attribute then `a'` must also be a stored attribute. (See Section 2.2.7.)

A limitation of this prototype implementation is that the attribute-redefinition rule is checked at declaration time and not after an attribute is fixed as either stored or derived. Therefore, the translator rejects the following fragment:

```
classes A B .
subclass B < A .
at a : A -> A .
at a : B -> B [redef] .
```

because the validity of the redefinition requires knowing whether the attributes are stored or derived.

5.6.1 Dynamic Binding

The implementation of dynamic binding required a small adaptation of OBJ3's rewriting engine. First, OBJ3 does order-sorted matching and rewriting. This means that when searching for axioms to rewrite a term with, the engine will consider as candidates all those whose left-hand side matches the term, modulo any subsort relationships. For example, given the declarations

```
sorts C D .
subsort D < C .
op f : C -> C .
op f : D -> D .
op z : -> D .
var Xc : C .  var Xd : D .
eq f(Xc) = EXP1 .
eq f(Xd) = EXP2 .
```

to rewrite `f(z)` the engine considers both axioms as equally good candidates. For object-oriented dynamic binding, however, we are only interested in the *exact* sort matches; for example, if `f` was a method redefined for objects of class `D`, then the first axiom should not be considered in the evaluation of `f(z)`. Above, only the second axiom provides an exact match for the `f` associated with `D`. This simple exact-matching scheme works properly because OBJ3's parser always assigns the lowest possible sort to an expression, and at every step in the rewriting process it makes sure that this remains the case. For instance, assume a further operator `g : C -> C`, and the axiom

```
ax g(Xc) = f(Xc) .
```

The parser will assign sort `C` to both terms in this axiom. After rewriting `g(y)` to `f(y)` for some `y` of sort `D`, the engine will immediately re-parse `f(y)` (and assign it sort `D`), thus allowing the second axiom to be considered as a candidate for rewriting this term, as desired for FOOPS. Because dynamic binding only exists at the object level of FOOPS, exact matching is exclusively done for operators that correspond to attributes and methods.

5.7 Theories, Views and Module Expressions

Given that it is meaningless to create objects of classes declared in theories, and because axioms declared in theories may have free form, object-level theories require much less processing by the translator. Basically, all that is needed is to record the theory in the module table, to record the classes, attributes and methods in the class table (to use for certain consistency checks; see below) and to include the declaration of the sequential composition operator.

Renamings and instantiations require similar processing by the translator. First, recall that modules `M` and `M * (sort S to S')` are semantically different, and that `LIST[X :: TRIV]` and `LIST[NAT]` are also different. OBJ3 evaluates `M * (sort S to S')` by making a copy of `M` but recreating all of its elements so as to reflect the renamings. `LIST[NAT]` is also a copy of `LIST[X :: TRIV]` but with recreations being dictated by the default view from `TRIV` to `NAT`, which maps `Elt` to `Nat`. Additionally, the other declarations of `NAT` must be included in the result.

This creation of modules through mappings requires that the module and class tables of the translator be extended with new entries that correspond to the new modules. Furthermore, some extra processing of renamings and instantiations is necessary. This is because the name of several of the operators automatically generated by the translator are derived from the name of the sort or class that they are associated with; for example, `void-C` and `all-C` need to be renamed if `C` is renamed. Also, several of the axioms that the translator generates need to be recreated. In particular, the built-in axioms for `set-C` and for fetching attribute values in the FOOPS database need to have their right-hand sides recreated whenever attributes are renamed. This recreation is simplified by the labels attached to these axioms, and which free the translator from having to examine the structure of each axiom to determine whether it needs regeneration.

Two further constructs remain: sum and `make`. Because in OBJ3 importing `A + B` is the same as importing `A` and `B` independently, the FOOPS translator does nothing extra for sums. For `make`, on the other hand, it needs to determine whether the module expression results in a functional or in an object-level module. This is achieved with some simple "type" inference: if any of the modules involved is an object module, then so is the result; otherwise, the result is a functional module.

Lastly, various consistency checks are required for semantic validity. For example, a view may not map a sort to a class, nor an attribute to a method, and a functional-level module may not import object-level modules. However, our prototype implementation does not attempt to verify any of these conditions.

5.8 Further Work

The quality of this prototype implementation of FOOPS can be enhanced in several ways. We consider the most worthwhile to be:

- type-checking views and module importations, as described in the previous section; and,
- detecting run-time violations to variance (discussed in Chapter 2).

As regards to efficiency, our personal experience is that the translation and interpretation of FOOPS code compares favourably to that of OBJ3+ code. Since FOOPS is built on top of OBJ3, the best way to make this prototype implementation more efficient is to make OBJ3 more efficient; particular areas to focus on are OBJ3's rewrite engine and its module expression evaluator (a notably slow component). Nevertheless, various aspects of this

prototype implementation could be optimised. The most promising ones seem to be the translation of DMAs and access to objects.

It is not difficult to see how to optimise the translation of DMAs: use the FOOPS database routine `write-attribute` to update the attributes affected by a method, and translate inline rather than to a call to `set-C` (see Section 5.5). This would not only economise updates to attributes whose value does not change, but would also save looking up this value in the first place. However, it would slightly slow down the evaluation of module expressions, because method axioms would then require a regeneration similar to that now done for each `set-C` axiom.

A different storage layout for objects could speed up access to attributes. What may appear to be a simple implementation aspect is not so because of features such as multiple inheritance [22, 105]. Our implementation stores attribute names along with their values in each object. Connor et. al [22] suggest that this could be improved by associating a table of *(attribute-name,offset-into-object)* pairs with each class, and having each object point to its corresponding table. This table could be a linked list or, if the number of attributes is large, a hash table. As a further improvement, attribute names can be hashed onto integers, saving some space. However, neither of these schemes admits overloaded attribute names.

We also note that the distinction between values and objects in FOOPS is itself an "implementation optimisation," a direct benefit of its semantics. Languages that consider everything to be an object often take advantage of this distinction as part of a compiler optimisation phase (e.g., Eiffel [77] and Gemstone [75]), and then only for built-in types; user-declared types are always regarded as defining objects.

Lastly, we provide ideas for implementing two features that this prototype does not support:

- **vertical parameterisation**: The implementation of vertical parameterisation could be achieved by mapping vertical interfaces to horizontal ones. For example,

```
omod SET[X :: TRIV]{REP :: CONTAINER[X]} is
  ...
endo
```

would be translated to

```
omod SET[X :: TRIV, REP :: CONTAINER[X]] is
  ...
endo
```

Instantiations would need to be mapped similarly. Furthermore, the visibility restrictions associated with vertical composition must be checked.

- **abstract classes**: Our implementation "almost" supports abstract classes, as they are based on theories and their importation. Two crucial aspects of abstract classes are that they cannot be instantiated, and that axioms must define executable patterns. As

noted above, the first aspect is already present in the implementation. The second only requires that the form of axioms be verified by the same machinery that establishes whether an axiom declared in an executable module is valid.

- **module blocks**: These should also be relatively easy to implement given all of the facilities already present in the language.

5.9 Summary and Conclusions

This chapter has given a fairly detailed overview of a prototype implementation of a translator for FOOPS. As its target language, this translator uses OBJ3+, an extension of OBJ3 that includes data structures for managing persistent entities. These data structures and those internal to the translator were explained, as were also the most important translation schemes, strategies and trade-offs. Furthermore, we provided ideas on how to implement some of the features of FOOPS not covered by our prototype implementation, and on how to enhance the implementation of some existing features.

The development of this prototype has been of great help with design matters in FOOPS. An implementation forces one to be very specific about details, which in turn lead to insights about the language itself and the various design alternatives. Also, the availability of a sufficiently sophisticated tool for testing complex examples is invaluable, especially for a language such as FOOPS, which includes many novel features. Finally, we were fortunate to have available the implementation of OBJ3 to build upon.

Chapter 6

Evaluation and Comparison with other Languages

Even though most programming languages technically have the same expressive power, differences among languages can significantly affect their practical utility.

— Mary Shaw

This chapter aims to provide an evaluation of FOOPS by comparing it to other object-oriented programming languages. In previous chapters we have offered some discussion as to what we believe distinguishes FOOPS; here we provide a more in-depth analysis and reflect upon the design decisions that characterise various language efforts.

We investigate modelling facilities and mechanisms for reuse, including values, objects, classes, modules, inheritance, information hiding, genericity and renaming. We also attend to the process of software design and development, and in particular to any language features that seem to facilitate (or hinder) it. Our intention is not to select a "best" language—no language has yet proven optimal for all programming tasks—but to discuss those features which in our opinion are important for large-grain programming.

While around fifteen languages are mentioned in this survey, closer attention is paid to Ada 9X, C++, Eiffel, Oberon-2 and Smalltalk-80:

Ada 9X [2] is the official object-oriented extension of Ada. As its predecessor, it was designed by committee. In Ada 9X classes are declared as special kinds of records, and subclasses are record extensions. It supports modules, and is strongly typed.

C++ [111] is probably the most widespread object-oriented language, mainly because of its link to C and therefore to UNIX. Its chief designer was Bjarne Stroustrup of AT&T, and C++ was influenced by Simula, CLU, Ada and ML. It enhances C by providing higher-level structuring facilities (such as classes) and by being more strict about typing. However, C++ retains the spirit of C-style programming, which is very low level, but nevertheless suited for systems programming.

Eiffel [77, 78] was designed by Bertrand Meyer of Interactive Software Engineering, and contrasts with C++ by incorporating ideas from formal methods, such as invariants and pre-

and post-conditions. Also, it has full static typing. Eiffel is probably the fastest emerging object-oriented language.

Oberon-2 [81, 82] is the invention of Professor Niklaus Wirth, and descends from Modula-2. While Oberon-2 is not as well known as the other languages, we study it because (like Ada 9X) it does not identify modules with classes. Oberon-2 is strongly typed.

Smalltalk-80 [56] is the latest version of the widely-used language which sparked object-orientation. Smalltalk-80 is perhaps most noted for its graphical programming environment and for its uniformity: everything is an object (including classes). It supports a free, typeless style of programming similar to that of Lisp. In what follows, we will just call it Smalltalk.

6.1 Objects and Values

The distinction between objects and values is fundamental in FOOPS [54], but largely ignored in other object-oriented programming languages. Given that these two kinds of entity have completely different semantics, we are in agreement with MacLennan [74] that much confusion can be avoided if they are kept separate in a language. In fact, we believe that the distinction clarifies many situations and examples. Furthermore, optimising compilers can take advantage of it. We begin by defining what values and objects are.

The main characteristics of values are that they are atemporal, immutable and referentially transparent. In more concrete terms, a value is never created or destroyed, it simply exists; a value is stateless, so it can never change or be changed; and expressions involving only values give the same result regardless of context. A value is an ideal entity. The prime example of a collection of values is the set of numbers, including constants such as the complex number i. Other examples include the colours and mathematical constructions such as categories and graphs (see [53] for an encoding of the former in OBJ3). Most languages support enumerated types, which denote finite sets of values, but not as generally as might be wished; often, these types cannot be organised in inheritance hierarchies.

Objects, on the other hand, are entities that are created, destroyed, and that change. In addition, they can be shared, in the sense that two objects sharing a third one see any changes made to it. In this regard, it seems that misunderstanding is due to identifying a value with the storage location that holds it; what can be "shared" is the location, not the value itself[1]. Furthermore, the uniqueness of an object is externally determined; for example, by assigning a unique identifier to it. More important, however, is that the principal motivation for having objects—and this goes back to Simula—is to simulate real-world processes, such as the behaviour of nuclear reactors.

Some object-oriented languages distinguish between an object structure and a pointer to it, and this is one way that so-called complex and composite objects can arise. A **complex object** is one that refers to another object; for example, an employee object with an attribute that stores the identity of the employee's manager. A **composite object** is one that *incorporates* another; for example, a car has doors and not just references to them, which would (potentially) allow for them to be shared with other cars. In turn,

[1] Perhaps this is also due to computers only being able to deal with objects; values must be faked.

these languages also have different semantics for assignments between object structures and pointers to them. The two kinds of assignments are respectively called projective and polymorphic. A **projective assignment** is one in which some structure is forgotten, such as when assigning a triple indexed by components x, y and z to a tuple variable indexed by x and y, where the z component is left ont. A **polymorphic assignment** manipulates pointers, such that there is no information loss. (As usual, the validity of these assignments depends on there being an appropriate relationship—commonly inheritance—between the underlying types of the two entities.) Note the impact upon dynamic binding, which crucially depends on polymorpbic assignment.

Next we examine how various languages treat objects and values.

6.1.1 C++

C++ supports both object structures and pointers to them, and by default variables and attributes store structures. If the target of an assignment is a structure, the assignment is projective: if the target is a pointer, the assignment is polymorphic. The only support for values is the enumerated type; the different kinds of numbers are built-in.

6.1.2 Eiffel

By default, variables and attributes in Eiffel store references to objects. However, so called "expanded classes" are available, and as the name suggests, objects of these classes are expanded in place, i.e., never referred to. Thus they give rise to composite objects. According to [78], the main reason for including expanded classes in Eiffel was to give more natural support for classes such as `BOOL` and `INTEGER`; in other words, from our point of view, to support values. However, the definition of an expanded class is still made in terms of object concepts such as attributes and methods and, in fact, Eiffel anyway gives special treatment to objects of classes `BOOL` and `INTEGER`: **true** and **false** are predefined keywords and 1, 2, 3, ... are automatically recognised as integers. In opposition to other languages that distinguish object structures from pointers to them, Eiffel forbids projective assignments, so that `X := Y` is valid only if `X` and `Y` are of the same exact class.

A further feature in Eiffel is the **unique attribute**, which is used to declare class-wide constants and (especially) to make up for the lack of enumerated types and values. For example,

```
Red, Blue, Yellow : INTEGER is unique;
```

declares three unique attributes, `Red`, `Blue` and `Yellow`, which are automatically assigned increasing and distinct integers. This seems a very roundabout way to specify constants, especially because access to them can only be achieved indirectly through objects. (This issue is also related to the distinction between classes and modules, to be discussed in the next section.) Lastly, we note the specification of complex numbers given in [77], in which the constant `i` is represented as a (constant) function on a complex number. To access `i`, then, an object of class `COMPLEX` is needed, say `X`, so that the value of `i` is given by the expression `X.i`, which seems counter-intuitive.

6.1.3 Oberon-2

Oberon-2 also omits enumerated types, and builds in the numbers and the booleans. It distinguishes between object structures and pointers to them, but the way this is done is similar to Pascal and may seem verbose to some. For example,

```
TYPE
    Node = POINTER TO NodeDesc;
    NodeDesc = RECORD ... END;
```

(Compare this to C++'s asterisk notation: `NodeDesc*` would denote a type whose elements are pointers to elements of `NodeDesc`, and would be equivalent to the above definition of `Node`.) Many other object-oriented languages default to pointers, which appear to be more common. Projective assignment is allowed.

6.1.4 Smalltalk

In Smalltalk, all types are classes and all references to objects are via pointers. `BOOL`, `INTEGER`, etc. are classes that receive special treatment; for example, their instances are fixed.

6.1.5 FOOPS

Values in FOOPS are user-definable, and this eliminates the need for special built-in support for them; this is a consequence of the language reflecting its underlying semantics. Additionally, FOOPS supports inheritance at both the level of values and the level of objects (i.e., for sorts and classes), and this further enhances expressive power. However, FOOPS does not provide for composite objects, although complex objects can be defined.

6.1.6 Conclusions

The main benefit of identifying values and objects is uniformity and economy of constructs. However, it seems to us that separating them so as to respect their distinct semantics is more valuable. For example, there is no need for awkward classes such as `BOOL` and `INTEGER` (in Eiffel and Smalltalk, for example), which do not really define objects, and it allows us to maintain a clear distinction between attributes, functions and methods: attributes describe object storage, functions are as in mathematics, and methods update objects. Moreover, users can declare their own values, and can organise them in inheritance hierarchies. The example of a class of complex numbers that appears in several introductory books on object-oriented programming [64, 77, 111] is a telling example of a common misunderstanding; a mathematician would be shocked to read about creating and destroying complex numbers!

A further advantage of this separation is that it benefits code optimisation, because knowing that something is a value can be used to save space and pointer dereferences. For example, the Gemstone [75] compiler has an optimisation step based on this (for numbers and booleans), and [78] suggests similar optimisations for unique attributes in Eiffel. In

```
omod PRIVATE-INSTRUCTION is
  classes Student Teacher .
  ...
  at teachers : Student -> SetOfTeachers .
  at students : Teacher -> SetOfStudents .
  ...
endm
```

Figure 6.1: Classes with mutually recursive definitions.

FOOPS, this kind of optimisation can be driven by program texts, and need not be based on some specially-treated classes.

Finally, the ability to define composite objects would increase the expressive power of FOOPS; this needs further consideration.

6.2 Classes and Modules

Another fundamental distinction that FOOPS makes is between classes and modules. In FOOPS, the main unit of encapsulation and scope is the module, which can declare several classes and their associated attributes and methods. Most other object-oriented languages equate modules with classes, so that there is exactly one class per module. While this view presents certain benefits, the following argues that there are compelling reasons for distinguishing between classes and modules.

First, it is possible to package together classes that mutually refer to each other, as in the example in Figure 6.1, or as in the more subtle example in Figure 6.2. It is very natural to allow for this kind of logical relationship to be captured directly and reflected in the physical structure of the system; in both examples, neither class could be used in isolation. This distinction also appears to simplify compiler design; for example, Meyer [77] reports significant initial difficulties implementing mutually recursive class definitions in Eiffel's compiler, and that the resulting algorithm was surprisingly complex. Also, note that using some kind of "forward" declaration syntax to overcome this is not a modular solution.

Second, it is common for sub-system requirements to indicate that several classes are needed. In FOOPS, theory modules express requirements. For example, this theory specifies the concept of a graph, which includes vertices and edges, among other things:

```
oth GRAPH is
  classes Graph Vertex Edge .
  ats source target : Edge -> Vertex .
  me is-incident : Graph Vertex Edge -> Bool .
  me make-edge   : Graph Vertex Vertex -> Edge .
  ...
endoth
```

```
omod SALES-PEOPLE is
  classes SalesPerson SalesManager .
  ex MANAGER .
  subclass SalesPerson  < Employee .
  subclass SalesManager < SalesPerson Manager .
  at manager : SalesPerson -> SalesManager .
  ...
endo
```

Figure 6.2: Classes `SalesPerson` and `SalesManager` need each other.

And the following theory is used further below as the interface for a module that implements `while` loops; `C` is the class of the structure to be iterated over, and classes `In` and `Out` are for the input and output of the iteration:

```
oth ITER-ACTIONS is
  classes C In Out .
  me init   : C In -> C .
  me action : C In -> C .
  me wrapup : C In -> Out .
  me test   : C In -> Bool .
endoth
```

(This theory also appeared in Section 3.2.) Without modules, these examples would be much more verbose and less natural.

Third, because the main unit of scope in FOOPS is the module, there is no need for special features that declare that one class can access the internals of another. Such features, which are present in Eiffel and C++, lead to what Szyperski [113] calls "spaghetti scoping;" furthermore, they increase the coupling between software components [15]. With modules, no such *ad hoc* visibility features are required: if two classes are closely related they can be packaged together into one unit. This is particularly relevant for so-called local classes. (See Section 6.6 for further discussion of scoping.)

Next, the distinction between class and module relationships permits declaring derived operations and mnemonics for constants. Derived operations (i.e., those defined in terms of others) are necessary for the development of components such as `math` packages, where the relationship with the original numeric types is neither inheritance nor clientship; in the following example, adapted from [100], the module `FLOAT` is extended with further trigonometric functions:

```
fmod TRIG-FUNCTIONS is
  pr FLOAT .
  fn tan   : Float -> Float .
  fn cot   : Float -> Float .
  fn sec   : Float -> Float .
```

```
omod WHILE[X :: ITER-ACTIONS] is
  --- the next two methods are derived
  mes while while-continue : C In -> Out .
  var E : C .  var I : In .
  ax while(E,I) = init(E,I); while-continue(E,I) .
  ax while-continue(E,I) =
        if test(E,I) then
            action(E,I); while-continue(E,I)
        else
            wrapup(E,I)
        fi .
endo
```

Figure 6.3: A generic definition of `while` loops.

```
  fn cosec : Float -> Float .
  ... axioms ...
endf
```

In conjunction with generic modules, derived operations can also be used to specify general classes of iterators over data structures, as Figure 6.3 illustrates. Wirth [117] makes the more general comment that derived operations are desirable because it is impossible to predict all of the useful operations associated with a certain type, and because changing the source code that declares the type would most likely require recompiling its clients; moreover, such a change might even be impossible to effect if the source code is not available. Another case in point here arises from our experience with Common Lisp: it is customary to find useful, new ways to manipulate lists, even though Common Lisp comes equipped with an extensive library of list functions. A further application of modules regards the packaging of sets of related constants, as the modules in Figures 6.4 and 6.5 show (these are also adapted from [100]). As before, the relationship with the original numeric types is neither inheritance nor clientship.

An alternative way to define iterators is used in the Eiffel Libraries [79]. In FOOPS, for example, this technique would be realised by having module `WHILE` declare a new class called `While` as client of the data structure to be iterated over, and have the iteration methods (i.e., `action`, `test`, etc.) be associated with this new class. The flexibility that this allows is that particular iterations could be defined by subclassing `While` and overriding its iteration methods; moreover, its subclasses could declare attributes for temporary storage. This approach actually highlights another advantage of the distinction between classes and modules, because these subclasses would mostly arise as auxiliaries to other classes needing looping constructs to define their methods. With modules, these iterator classes are naturally defined as local classes.

Also, separating class and module inheritance leads naturally to features such as "private" or "implementation" class inheritance [111]. With this kind of inheritance, a class

```
fmod ISD is
  pr NAT .
  let NUL = 0 .   let SOH = 1 .   let STX = 2 .   let ETX = 3 .
  let EOT = 4 .   let ENQ = 5 .   let ACK = 6 .   let BEL = 7 .
  let BS  = 10 .  let HT  = 11 .  let LF  = 12 .  let VT  = 13 .
  let FF  = 14 .  let CR  = 15 .  let SO  = 16 .  let SI  = 17 .
  let DLE = 20 .  let DC1 = 21 .  let DC2 = 22 .  let DC3 = 23 .
  let DC4 = 24 .  let NAK = 25 .  let SYN = 26 .  let ETB = 27 .
  let CAN = 30 .  let EM  = 31 .  let SUB = 32 .  let ESC = 33 .
  let FS  = 34 .  let GS  = 35 .  let RS  = 36 .  let US  = 37 .
  let SP  = 40 .  let DEL = 177 .
endf
```

Figure 6.4: Mnemonics for the ISO control codes.

```
fmod MATH-CONSTANTS is
  pr FLOAT .
  let e            = 2.7182 .
  let golden-ratio = 1.6180 .
  let loge10       = 2.3025 .
  let log10e       = 0.4342 .
  let sqrt-of-2    = 1.4142 .
  let sqrt-of-10   = 3.1622 .
endf
```

Figure 6.5: Some common mathematical constants.

B can inherit from a class A but in a way that forbids placing objects of class B where objects of class A are expected. The purpose of this is simply to allow objects of class B to have access to some (or all) of the internal functionality provided for objects of class A. In FOOPS, there is no direct support for this, but the `using` mode of module importation can provide a similar effect as a special case (see Chapter 3). To us, it seems dubious to so radically adapt class inheritance to support a feature that is more appropriately supported by module inheritance, which deals with code reuse.

Finally, at present it is simpler for compilers to optimise code when there are no private declarations involved. That modules allow for scoping units larger than classes therefore has implementation advantages too. Nonetheless, we note that recent work attempts to close this performance gap (see, for example, [21, 58]).

6.2.1 Other Languages

Except for Oberon-2, Modula-3 [83] and Ada 9X, all the other languages that we are aware of identify modules with classes. One that stands out as slightly different is C++, which allows scoping to occur at both the level of classes and the level of files. Still, C++ files are not really modules as in FOOPS and the others, because while files may have private classes, the features of these classes are not freely accessible to other classes in the same file; furthermore, there is no notion of parameterisation at the file level. C++ also supports nested classes, which are class definitions given inside others. This still does not quite correspond to modules, because the enclosing classes do not have full access to the features of the nested ones (and vice versa). Also, nested classes would be unnecessary if C++ had modules, because they could then be declared locally.

6.2.2 Summary and Conclusions

We have compared two notions of encapsulation and scope in object-oriented languages that can package together several classes, and in languages that allow exactly one class declaration. We noted that the latter notion allows class relationships to be more precisely controlled, but that the former notion (modules) appears to be more convenient with regards to logical and physical encapsulation, as well as with regards to some forms of scoping. Also, we discussed how differentiating module importation from class relationships gives rise to derived operations, permits the orderly declaration of mnemonics for constants, and naturally supports private class inheritance. A further advantage of modules is that a renaming mechanism can allow classes to be given new names (see the next section).

While the above provides a strong case for differentiating between modules and classes, the modules-as-classes notion also has some advantages. The most important one seems to be that visibility relationships between objects of different classes can be more precisely controlled (Section 6.6 gives more details about this), and that it is possible to allow a class to make its definition fully visible to its subclasses (this was discussed in Section 3.9). But these reasons do not seem sufficient to identify the two concepts, and probably there are ways to achieve these advantages, if they are needed, without the *ad hoc* modes of class

inheritance and scoping that we will later on examine.

Consequently, there seems to be sufficient motivation, with regards to both clarity of concepts and functionality, for considering classes as collections of objects and class inheritance as a mechanism for the hierarchical classification of these, and to distinguish these concepts from modules (which can declare many classes) and from module inheritance, which concerns code reuse and large-grain structuring.

6.3 Renaming

Renaming is a mechanism that attempts to make component reuse more flexible by allowing features such as attributes, methods and classes to be given new names. Not many langnages provide support for renaming; in fact, it appears that by far the most sophisticated renaming mechanisms are those of Eiffel and FOOPS. This section will thus concentrate on comparing these two.

We will examine what each language allows to be renamed, the relationship between new and old names, and the impact npon readability and reuse of the renaming mechanism that is provided.

6.3.1 Eiffel

The renaming mechanism of Eiffel allows classes to give new names to inherited attributes, functions and methods. Except for a special situation discussed below, a rename introduces a kind of alias, in the following sense. Consider a class `DRIVER` with an attribute called `tvs` (for traffic violations)[2]. A client of this class could declare a method `m` that takes a driver object as its argument[3], and in its body ask for the driver's `tvs`:

```
m(d : DRIVER) is
  do
    ...
    d.tvs
    ...
  end;
```

Subsequently, we may declare a snbclass of `DRIVER` called `BRITISH-DRIVER` that renames `tvs`:

```
class BRITISH-DRIVER is
  inherit
    DRIVER rename tvs to br_tvs
    end
  ...
```

[2]This example is adapted from [77].

[3]In Eiffel, as in several other object-oriented programming languages, methods receive the object they operate on as an implicit argument. Therefore, `m` really has two arguments, but we adopt Eiffel's terminology for the present discussion.

```
end;
```

This means that BRITISH-DRIVER objects cannot be *directly* asked for their tvs, because BRITISH-DRIVER calls that attribute by another name. For example,

```
bd : BRITISH-DRIVER;
!!bd;                -- create an object and attach it to variable bd
... bd.tvs ...;      -- error!
... bd.br_tvs ...;   -- ok!
```

However, a BRITISH-DRIVER object could be an argument to method m, where asking for its tvs is fine, as in that context it is viewed as a DRIVER. There, the expression d.tvs refers to attribute br_tvs for objects of class BRITISH-DRIVER.

We now extend this example to illustrate bow renamings are not treated as aliases when a class inherits from two otbers, each of which has a feature that renames the same one. Consider the class FRENCH-DRIVER, which is similar to BRITISH-DRIVER:

```
class FRENCH-DRIVER is
  inherit
    DRIVER rename tvs to fr_tvs
    end
  ...
end;
```

and the class FR-BR-DRIVER that inherits from both FRENCH-DRIVER and BRITISH-DRIVER:

```
class FR-BR-DRIVER is
  inherit
    FRENCH-DRIVER  end;
    BRITISH-DRIVER end
  ...
end;
```

In this case, Eiffel considers fr_tvs and br_tvs as two different attributes of FR-BR-DRIVER, even though they rename the same attribute of DRIVER. (But as will be seen in more detail in Section 6.4, FR-BR-DRIVER has only one copy of those attributes and methods of DRIVER that were not renamed.)

This new class would introduce a conflict in method m, because for FR-BR-DRIVER objects the expression d.tvs is now ambiguous: does it refer to fr_tvs or to br_tvs? Therefore, the Eiffel compiler rejects the above definition of FR-BR-DRIVER. To make it valid, a **select** clause can be used to indicate how to solve the ambiguity. For example,

```
class FR-BR-DRIVER is
  inherit
    FRENCH-DRIVER end;
    BRITISH-DRIVER
```

```
        select fr_tvs
      end
    ...
end;
```

This version of FR-BR-DRIVER specifies that in m (and in any other similar functions and methods that manipulate drivers) the expression d.tvs refers to attribute fr_tvs whenever d is bound to an object of class FR-BR-DRIVER.

To summarise, when single inheritance is used, renaming is analogous to introducing aliases. But for situations in which there is multiple inheritance with a common superclass, and a feature of that class is renamed, the mechanism generates new, independent features.

One unfortunate consequence of this kind of renaming is that it can make programs harder to read, as a feature can have different names in different contexts. For example, consider the class hierarchy in Figure 6.6, where the x's are (say) methods, and the arrows labelled with "*" indicate renamings. Now consider the following methods p and q declared in a client of A and a client of B, respectively:

```
class CLIENT-OF-A is
feature
  p(a : A) is
    do
      ...
      a.x;
      ...
  end
end;

class CLIENT-OF-B is
feature
  q(b : B) is
    local
      ca : CLIENT-OF-A;
    do
      ...
       b.x'; ca.p(b);
      ...
  end
end;
```

If method q is invoked with an object of class C as argument, then to trace what the program does one must be aware of all the previous names that the x'' method had (and similarly for any other features that were renamed). For example, in the context of the initial call to q, the object of class C had a method called x''. However, in the context of q itself, this method is known as x'. And finally, in the context of method p (which q invokes), x''

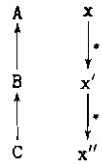


Figure 6.6: Renaming along a class hierarchy.

is known as x. Thus, tracing calls to methods may demand namespace conversions under Eiffel's approach to renaming. We suspect that this also forces the run-time system to keep track of all the previous names of an object's features.

6.3.2 FOOPS

In FOOPS the renaming mechanism works at the level of modules, and this allows classes (and sorts) to be renamed too. It also permits further precision of names for expressions such as

```
STACK[LIST[TUPLE[SYMBOL,SYMBOL]]]
```

because the class `Stack` of `STACK` can be renamed to something more meaningful; in this case we mean to refer to block-structured symbol tables, and instead we could have written

```
STACK[LIST[TUPLE[SYMBOL,SYMBOL]]] * (class Stack to SymbolTable)
```

In Eiffel, one would have to write the entire expression every time a reference was made to the class of a symbol table, because classes cannot be given new names; or perhaps inheritance could be used to create a new class name, but this seems *ad hoc*. It should be observed that in FOOPS the above expression would denote a module name, so we could still not avoid it unless a **make** was used, for example. The point is that this is more flexible because it occurs at a higher level of abstraction.

The most significant difference between the two approaches, however, is that in FOOPS renaming is an operation that takes a module and generates a new one, and not a way of introducing aliases or duplicating particular functions, attributes or methods. Therefore, the classes `Stack` and `SymbolTable` from the above module expressions (respectively) are not related. This distinction comes from the semantics of module renaming: the * operator is interpreted as a morphism in the category of signatures, as explained in Chapter 4.

This approach avoids the above difficulties of renaming-by-aliases, and seems to be flexible enough to capture other common situations. For example, a module that implements a **map** method on lists is more readable if the source and target classes are given mnemonic names:

```
oth ME is
  classes C D .
  me m : C -> D .
endo

omod MAP[X :: ME] is
  pr LIST[C] * (class List to SourceList) .
  pr LIST[D] * (class List to TargetList) .
  me map : SourceList -> TargetList .
  ...
endo
```

(Without renaming, the two list classes would have to be differentiated with qualifications, i.e., `List.LIST[C]` and `List.LIST[D]`.) However, this setup does not appear to be appropriate for a potential client of `MAP` dealing with lists of (say) employees obtained from

```
LIST[EMPLOYEE] * (class List to EmployeeList)
```

because an instantiation such as

```
MAP[view to EMPLOYEE is
      class C to Employee .
      class D to String .
      me m to name .
    endv]
```

generates a method map with interface

```
me map : SourceList -> TargetList .
```

Consequently, for an object `el` of class `EmployeeList` the expression `map(el)` does not type check. Fortunately, the same renaming mechanism allows `MAP` to be tailored to the above situation by applying renaming to it:

```
MAP[view to EMPLOYEE is
      class C to Employee .
      class D to String .
      me m to name .
    endv]
* (class SourceList to EmployeeList)
```

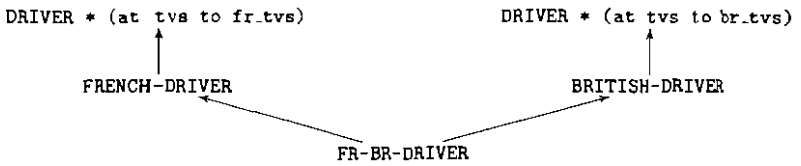
Then the resulting method `map` is

```
me map : EmployeeList -> TargetList .
```

as required. In sum, the compositional nature of morphisms—and thus renamings—allowed `MAP` to be adapted properly. Still, we note that this approach can become tedious when

many features are renamed at the same time, or when different modules need to be adapted in different ways to fit a given context.

Finally, observe that the renaming mechanism of FOOPS gives the drivers' example a different "topology," because the resulting module inheritance diagram is



and not as in Eiffel, where DRIVER is (partially) "shared" by FRENCH-DRIVER and BRITISH-DRIVER. This means that those features of drivers that are not renamed, such as **eyesight**, for example, are in FOOPS "duplicated" in FR-BR-DRIVER. To prevent this, those features that should not appear twice need to be promoted to a sub-module of DRIVER, which would then be shared. For the present example this seems appropriate, because it highlights that a higher-level abstraction such as PERSON might be missing. On the other hand, FR-BR-DRIVER would no longer be related to DRIVER, and this may or may not be desirable. We will have more to say about this in the next section.

6.3.3 Other Languages

We are aware of only one other language with support for renaming: Ada. Its renaming mechanism introduces true aliases, in the sense that if some context renames f to g, both are valid names in that context. Because Ada supports modules in a way similar to FOOPS, types can also be renamed (and thus "classes" in Ada 9X). A further difference with FOOPS is that Ada renamings are not part of a module (or "package") expression sublanguage.

6.3.4 Summary and Conclusions

We have compared two sophisticated approaches to renaming in object-oriented programming languages, those of Eiffel and FOOPS. Eiffel's approach is based on a form of aliasing, although for multiple inheritance with common superclasses renaming actually duplicates features. The approach of FOOPS is that renaming is an operator that generates new modules, and thus avoids a readability problem associated with the aliases approach. However, this seems to be the cost of renaming within a class inheritance hierarchy. In FOOPS, a module cannot inherit another and at the same time rename some of its features. The module that is inherited is actually that which the renaming generated; for certain situations, this might prove inappropriate, as the next section discusses. Also, observe that renaming in Eiffel is tied to class inheritance, and that classes cannot be renamed as in FOOPS.

Languages that Support Only Single Inheritance	Languages that Support Multiple Inheritance
Ada 9X	C++
Modula-3	CLOS
Oberon-2	Eiffel
Objective-C	FOOPS
Object-Pascal	LOOPS
Simula	Trellis
Smalltalk	TROLL

Fignre 6.7: Support for class inheritance in various languages.

6.4 Class Inheritance

This section compares the mechanisms for class inheritance and redefinition in several langnages. Class inheritance is a classification tool, but it is also used as a mechanism for code rense. Redefinition allows subclasses to override some of what they inherit. thus reconciling reusability with extendibility. We begin by detailing the kind of inheritance supported, then we explain the way redefinition and dynamic binding operate, and finally we discnss inheritance conflicts and their resolution. This last topic is very important because of its implications for reusing old classes.

6.4.1 Kinds of Inheritance

A basic question concerning class inheritance is whether a class can inherit from more than one other class. While some might expect that at this level of maturity all object-oriented languages would support mnltiple inheritance, this is not so, as Figure 6.7 illustrates. Note that of the languages that do not identify modnles with classes the only one that supports multiple inheritance is FOOPS.

6.4.2 Redefinition and Dynamic Binding

There are many aspects to a redefinition facility in an object-oriented language; we summarise those that we deem most important in Figure 6.8. The purpose of the first column of the table is self-explanatory, and we use the terminology of each language. Ada 9X, Modula-3 and Oberon-2 use "procedures" for what we call "methods." In C++, attributes are called "data members" and methods are called "member functions;" there can also be fnnctions that are not "members," and these are similar to derived operations.

The second column asks whether a subclass can always redefine what it inherits. In C++, a member function can only be redefined if it was declared "virtual." This approach is motivated by efficiency reasons; the extensibility associated with object-oriented software wonld call for all member functions to be virtual by default [61, 99] (but there is some disagreement regarding this). In Eiffel, the reverse occnrs: redefinition is allowed provided

	What can be redefined?	Can always redefine?	Need tag?	Compatibility	Can access originals? Which?	Can merge?	Computed → stored?
Ada 9X	procedures	Yes	No	fixed	Yes/all	N/A	No
C++	member functions	if virtual	No	fixed	Yes/all	Yes	No
Eiffel	atts, fns & mes	if not frozen	Yes	variant	Yes/all (w/select)	if deferred	Yes
FOOPS	atts & mes	Yes	Yes	variant	Yes/all	Yes	Yes
Modula-3	methods	Yes	Yes	fixed	Yes/all	N/A	No
Oberon-2	procedures	if type-bound	No	fixed	Yes/ immediate	N/A	No
Smalltalk	methods	Yes	No	by name	Yes/ immediate	N/A	No
Trellis	atts & mes	Yes	No	contra-variant	Yes/all	Yes	Yes

Figure 6.8: Comparison of redefinition facilities.

the attribute, function or method was not originally designated as "frozen." In Oberon-2, "type-bound" procedures are analogues to member functions in C++; those which are not type-bound are similar to derived operations.

The third column asks whether the new version of an operation needs to be tagged in some way. For example, in FOOPS, redefinitions must include **redef** as a property. We think that requiring a tag makes programs easier to read and can avoid unintended clashes or redefinitions.

The next column compares the syntactic requirements on the new version. The difference between variant and contra-variant compatibility was discussed in Section 2.2.7; Trellis [65, 102] is the only language that we know of which has a contra-variant compatibility rule. Fixed compatibility means that no difference in rank is allowed except at the argument position for the object which is to be updated or queried. These concepts do not apply to untyped languages; therefore, Smalltalk only requires that the new and old versions have the same name.

The fifth column indicates whether a redefinition can access any of the previous versions of what it redefines. Here "all" means that it can refer to all of them, while "immediate" means that it can only access that (or *those*, if merging is allowed; see below) which it directly redefines. Note the entry for Eiffel, which specifies that **select** must be used in order to access previous versions. This is because in Eiffel if a class `B` wishes to inherit a class `A` and redefine one of its methods, say `m`, and at the same time refer to the original `m`, then it must inherit `A` twice and rename one of the `m`'s, so that they become two different methods; then the new `m` can refer to the old one but using a different name; this requires non-cumulative interfaces (see Section 3.9). Thus, an ambiguity similar to that discussed in the section on renaming occurs, and `B` is required to select between the new `m` and the old, renamed version. Most other languages in the table provide some kind of qualification notation for referring to original versions.

We also note that Snyder [106] warns about the potential problems of allowing a class to refer explicitly to its non-immediate superclasses. He argues that if a class `C` is allowed to do this and then one of its immediate superclasses decides to change its own superclasses, then `C` can become invalid, even if its superclasses continue to provide the same attributes

and methods. But this only applies to situations in which class inheritance is used for implementation reasons and not for classification. When the purpose is to classify, the knowledge that a subclass relation exists is used throughout a whole system: a piece of code that works for a class C also works for any of its subclasses.

The sixth column refers to whether a class can have a feature that redefines two or more others (from different, immediate superclasses) at the same time. In Eiffel, two methods or functions can only be merged if they are both **deferred**, or not implemented by their classes. To get around this, a facility called **undefinition** can be used to make a method or function deferred. The need for such a facility is not clear to us, and unfortunately [78] does not provide (in our opinion) sufficient motivation for it.

Finally, the last column asks whether a method (or function or procedure or derived attribute, depending on the language) can be redefined into an attribute. In Eiffel, only functions can be redefined into attributes. In FOOPS, the only possibility is for derived attributes to be redefined into stored attributes.

6.4.3 Inheritance Conflicts

This section discusses various kinds of inheritance conflicts and compares how some languages approach their resolution; studying this gives us a way to clearly understand the inheritance mechanism of a language.

For single inheritance, the only possible conflict occurs when a class declares a feature with the same name as one it inherits. In typeless languages such as Smalltalk this is considered redefinition, as was explained above. For typed languages, the rank of the feature must also be taken into account, and the validity of the new declaration depends on whether the two ranks are related. We say that rank $\langle S_1\ S_2\ \ldots\ S_{N-1}, S_N\rangle$ is **related** to rank $\langle T_1\ T_2\ \ldots\ T_{N-1}, T_N\rangle$ if and only if for $i = 1..N$, either $S_i \leq T_i$ or $T_i \leq S_i$. (Note that both variance and contra-variance are special cases of relatedness.) If they are related, then the new feature is considered a redefinition attempt. If they are not related, then it depends on whether the language permits overloaded names; for example, Ada 9X, C++ and FOOPS do but Eiffel and Oberon-2 do not.

For multiple inheritance there are several other more complex cases to consider. We will examine these others and their resolution by studying four inheritance diagrams. The first is given in Figure 6.9, and depicts a class D that inherits a feature called a directly from classes B and C. In Eiffel, overloading is not permitted, and so one of the a's must be renamed, or they could be merged if their ranks are compatible. In FOOPS, if having both a's in D violates regularity then it is an error, and to solve it renaming or merging must be used; however, renaming would be quite dangerous, as it generates new modules, and thus new classes. C++ permits both a's to be inherited and leaves it up to D and its clients to resolve any ambiguities. This, however, is not very convenient as they both now have longer names (in C++ notation, `B::a` and `C::a`); but as C++ does not support renaming, qualification is more reasonable than having to change either B or C, which is not always possible and might wreck other classes.

The diagram in Figure 6.10 illustrates a common situation: inheriting a feature from

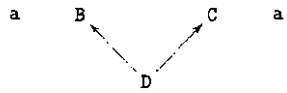


Figure 6.9: Inheritance of similar feature from different parents.

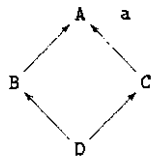


Figure 6.10: Inheritance via distinct paths.

the same class via distinct paths. In both Eiffel and FOOPS, D is associated with only one a. In C++, the number of a's in D depends on whether B and C declared that they were inheriting A "virtually." If they both did, then there is only one a in D. If at least one did not, then there are two a's in D. and as indicated above qualifications must be provided to distinguish the two. This state of affairs in C++ has been criticised (for example, see [61]), because it is believed that it should he the business of the designer of D to choose whether it inherits the features of A once or twice, not the business of the designers of B and C, who cannot be expected to predict all their possible subclasses.

Figure 6.11 shows a diagram in which B and C both inherit from A but where C renames a to c. In Eiffel this is fine provided D selects either a or c. In FOOPS, the diagram would be different because renaming generates new modules. Figure 6.12 gives the corresponding picture in FOOPS at the module level (assuming a is an attribute).

Finally, Figure 6.13 presents a diagram similar to that in Figure 6.11. but now C redefines

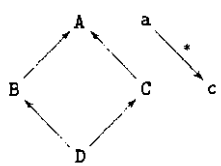


Figure 6.11: Conflicting rename in multiple inheritance.

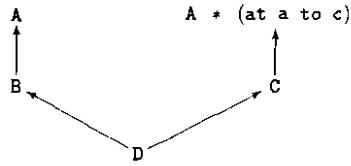


Figure 6.12: Multiple inheritance rename in FOOPS.

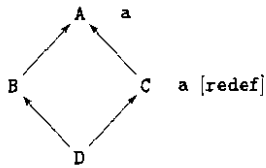


Figure 6.13: Conflicting redefinition in multiple inheritance.

a. In C++, the ambiguity in D is solved by so-called "domination," whereby, for classes Z1 and Z2 that declare (or redefine) some member function f, Z1::f is chosen over Z2::f if and only if Z1 is a subclass of Z2. For the present case, this means that if d is bound to an object of class D, then d.a refers to C's version of a. If B had also redefined a, then D is invalid, because there is no way in C++ to choose between the two a's. For Eiffel and FOOPS, on the other hand, the situation is erroneous regardless of whether B also redefines a. As before, Eiffel requires either renaming one of the a's and then **select**ing, or undefining both and then merging. In FOOPS, either merging or renaming would do; but it is possible that neither option is suitable, and one of the original classes will need modification, which might prove undesirable.

A further issue that affects conflict resolution is the visibility of a feature. That is, whether a private feature of a superclass can clash with a feature in any of its subclasses. This will be discussed in Section 6.6.

Linearisation. There is a completely different approach to conflict resolution for multiple inheritance that is adopted by some object-oriented extensions of Lisp, such as CLOS [32] and its predecessors LOOPS [110] and Flavors [80]. In these languages, a class hierarchy graph is converted to a single inheritance chain, and then conflicts are dealt with in that setting.

For more discussion about conflict resolution see also [106].

6.4.4 Summary and Conclusions

We have compared class inheritance and redefinition in several object-oriented languages. Concerning redefinition, FOOPS matches favourably with the others; in fact, FOOPS and Eiffel were fonnd to be the most flexible in this aspect. Regarding inheritance, we noted that among the languages that do not identify modules and classes FOOPS is the only one that provides multiple inheritance. We also discussed several kinds of multiple inheritance conflicts and how languages snch as C++, Eiffel and FOOPS deal with them. Here we found Eiffel and FOOPS agreeing most of the time on what is a conflict; key distinctions between the two included that only FOOPS supports overloading, and that renaming, a common way to resolve conflicts, has a different meaning in each. Of the two, Eiffel offers the most sophisticated facilities for deriving suhclasses. Also, it may he that in FOOPS some conflicts can only be resolved in ways that reqnire modifying the original classes, which is not always possible or desirable. C++ seemed to be less straightforward with respect to multiple inheritance, possibly because of its emphasis on efficiency (see [111]), which (among other things) might have climinated the option of having a renaming facility that could be used to simplify conflict resolntion.

Lastly, it appears obvious that while multiple inheritance is a powerful classification tool, it is much harder to design, understand and implement than single inheritance. For these reasons. it has been omitted from several object-oriented languages.

6.5 Genericity

Genericity is an abstraction mechanism for capturing commonality, and the instantiation of generic classes or modules is a powerful way to reuse specifications and code. Here arises anotber point of comparison between encapsulation units: with generic modules, reuse occnrs in larger steps than with generic classes. In the comparison and evaluation that follows, one of onr main concerns is support for expressing requirements on actual arguments to generic classes or modnles, facilities for structuring these requirements, and instantiation mechanisms. We also discuss conpling and higher-order operations.

6.5.1 Ada and Ada 9X

Genericity in Ada occurs at the level of modules (called "packages") and requirements on actual arguments can only be syntactic. These requirements are not themselves expressed using modnles. but given as prefixes to generic modules, as in

```
generic
  type Elt;
  with function "<" (X, Y : Elt) return Boolean is <>;
package B-SEARCH-TREE is
  ...
end B-SEARCH-TREE;
```

which says that valid arguments for instantiating B-SEARCH-TREE are a type and an ordering function on elements of that type. In Ada, procedures can also be generic over types and other procedures, and Ada 9X extends Ada with higher-order procedures.

Ada does not allow multi-level module instantiation, as in B-SEARCH-TREE[LIST[NAT]], but Ada 9X provides some support for this, although still not in its fully general form.

6.5.2 C++

Classes in C++ can be generic over strings, constants, functions and other classes. However, it is not possible to express things such as the syntactic requirement that a certain actual argument class must provide some member function f. Also, there are no means for expressing semantic requirements on actual arguments. C++ supports higher-order operations through so-called "function pointers." As mentioned previously, "modules" (files) cannot be generic. Multi-level instantiation is provided.

6.5.3 CLU

The language CLU was the first to implement generic modules [70, 71]. The modules of CLU, called "clusters," are identified with types and can be generic in a way similar to packages in Ada. (In fact, CLU influenced the design of Ada.) Multi-level instantiation is possible.

6.5.4 Eiffel

Eiffel [78] has an interesting feature called **constrained genericity**, which allows a generic class to specify that certain properties should be satisfied by actual arguments, through the existence of a suitable inheritance relation. For example, an Eiffel class for binary search trees with the header

```
B-SEARCH-TREE[X -> POSET]
```

indicates that valid instantiations must bind X to a subclass of POSET (partially ordered sets)[4]. Further semantic requirements may be expressed in Eiffel using its assertions facility, which permits attaching pre- and post-conditions to functions and methods, and invariants to classes.

In Eiffel, if C′ is a subclass of C, then P[C′] is automatically a subclass of P[C]. No other language that we are aware of is like this. Multi-level instantiation is supported.

6.5.5 ML

While ML [92] is not an object-oriented language, its module system is quite powerful and deserves mention. (In fact, ML's module system was inspired by the module facilities of the specification language Clear, from which OBJ and FOOPS derive some of theirs.) The

[4] Dee [57] has a similar facility. Also, this is what Cardelli and Wegner [17] call "bounded parametric polymorphism."

"signatures" of ML are similar to the theories of FOOPS, except that they may only declare syntactic information. For example, the following signature declares a type T with a binary function called `less`:

```
signature POSET =
  sig
  type T
  val less: T * T -> bool
  end;
```

ML's "structures" correspond to non-generic modules in FOOPS, while "functors" are generic structures. Both need to always be associated with a signature, giving way to a many-one relationship. For the binary search trees example above, we would first declare the signature, i.e.,

```
signature B-SEARCH-TREE =
  sig
  ...
  end;
```

and then the functor:

```
functor B-Search-Tree (structure Elt: POSET) : B-SEARCH-TREE =
  struct
  ...
  end;
```

In contrast with FOOPS, ML does not have views. Rather, structures are used to provide the bindings necessary to make other structures match signatures. For example, the type `int` is built-in in ML. To instantiate `B-Search-Tree` with integers, we need to declare a structure with signature `POSET` such that `T` is "bound" to `int` and `less` is "bound" to the desired ordering. The following structure associates `less` with the built-in less-than function on integers:

```
structure IntPoset: POSET =
  struct
  type T = int;
  val less: int * int -> bool = op <
end;
```

An instantiation then proceeds as expected:

```
structure IntTree = B-Search-Tree(structure Elt = IntPoset);
```

It would have been possible to create the structure on the fly, as in

```
structure IntTree =
    B-Search-Tree(struct
                    type T = int;
                    val less: int * int -> bool = op <
                  end);
```

ML's signatures can be generic, but this is not declared with the expected syntax. For example, the following signature for priority queues is satisfied by any structure that includes a substructure with signature POSET (for the elements of queues):

```
signature PQUEUE =
  sig
  structure Poset: POSET
  ...
  end;
```

An interesting feature of ML's module system is **sharing constraints**, by which it may be required that certain substructures or types of actual arguments to functors be the same. An example given in [92] illustrates this with a functor that takes as parameters a priority queue and an indexed table, and which requires that the type of the queue's elements is the same as the type of the table's key. Thus:

```
functor SharFUN(structure Pqueue: PQUEUE and Table: TABLE
                sharing type Pqueue.Poset.T = Table.key) =
  struct
  ...
  end;
```

In FOOPS, we would have specified this situation by making PQUEUE and TABLE generic, such that the sharing would be implicit by instantiating both with similar arguments. However, ML's facility covers situations in which one did not anticipate that a theory was better rendered as generic.

Lastly, there is the language Extended ML [101] which enriches ML with non-executable axioms that may be used for documentation and perhaps to support formal development (in much the same way as axioms are used in FOOPS theories). Axioms may also be declared inside structures and functors, to indicate intermediate development stages; that is, a structure or functor is executable only if it does not include any axioms.

6.5.6 Modula-3

Every module in Modula-3 has an "interface," which declares syntactic properties [16]. The association between modules and interfaces is one-many, but fixed (i.e., a new interface for a module cannot be introduced without changing the module).

Both modules and interfaces can be generic. However, parameters are only implicitly restricted. For example, in

```
generic interface STACK(Elem);
  type Stack;
  procedure push(s : Stack; x : Elem.Item);
  ...
end STACK.
```

Elem is just the name of some interface that must provide a type called Item; but one can only know this requirement by examining the generic's body. Instantiations require identities on names; i.e., a valid argument to STACK must provide a type called Item (as in ML). Therefore, to create an interface for (say) stacks of integers we need an intermediate step to bind Item:

```
interface INT;
  type Item = INTEGER;
end INT.

interface INT-STACK = STACK(INT)
end INT-STACK.
```

Multi-level instantiation using STACK is not really possible because of the bindings required for type Item. For example, to create an interface for stacks of stacks of integers, INT-STACK must be changed to include a binding for type Item:

```
interface INT-STACK = STACK(INT);
  type Item = Stack;
end INT-STACK.

interface STACK-STACK-INT = STACK(INT-STACK)
end STACK-STACK-INT.
```

Alternatively, a completely new interface could have been declared.

6.5.7 Oberon-2

Oberon-2 does not support generic modules or classes, nor higher-order operations.

6.5.8 P++

Each component in P++ [104] is associated with one "realm." Realm interfaces correspond to theories in parameterised programming, although without any semantic constraints. Also, there are no views, so techniques such as those of ML need to be used for bindings. This language is primarily based on vertical parameterisation, although a limited form of horizontal parameterisation allows constants and types, without any horizontal composition.

6.5.9 Smalltalk

Smalltalk does not support generic classes, but higher-order methods help provide some of the missing functionality. For example, the above B-SEARCH-TREE class can be implemented in Smalltalk as SortedCollection in the Smalltalk library: the comparison method is given as an argument to the method that creates binary search trees, each of which would store it in an attribute. In this sense, generic class instantiation in C++ and Eiffel (for example) would be equivalent to object creation in Smalltalk.

6.5.10 FOOPS

In FOOPS, theory modules are used to specify syntactic and semantic requirements on actual arguments to generic modules, and views express how actuals satisfy theories. Theories can be generic and can be combined with sum and can have their features renamed with "*", as they are bona fide modules. FOOPS modules can also be generic over vertical components. Here we would also like to mention closely related work by Tracz [114], whose LILEANNA system implements the horizontal and vertical composition ideas of LIL [35] for the Ada language, using ANNA [73] as its specification language.

6.5.11 Comparison with Constrained Genericity

Although Eiffel's constrained genericity is a significant advance in generic classes, it still has some drawbacks for reusability. For example, to create a binary search tree for employees, EMPLOYEE must be a subclass of POSET. But employees can be partially ordered in many different ways, e.g., by age, name, salary, department number, seniority, rank, employee number, etc. These relationships could be obtained by creating a new subclass for each one, e.g., EMPLOYEE-AS-POSET-BY-AGE and EMPLOYEE-AS-POSET-BY-SALARY, but this *ad hoc* use of inheritance would produce an awkward plethora of mystifying subclasses.

Another way to specify such relationships is to do so at design time. For example, the Eiffel Libraries [79] contain a class TRAVERSABLE, and data structures for lists and chains are given as subclasses of it. The classes HASHABLE and ADDABLE with their descendants are similar. However, this approach not only produces awkward inheritance relations (e.g., consider how many times EMPLOYEE would have to inherit POSET), but it also requires foreknowledge of all relevant properties and potential uses of a software component, which seems unrealistic.

Structuring by libraries exacerbates this problem. For example, if POSET and B-SEARCH-TREE belong to library L_1 and EMPLOYEE to library L_2, and we want to have a binary-search tree of employees, then we have two choices. The first is to change EMPLOYEE so that it is a subclass of POSET. This is not only dangerous because of possible name clashes with entities in POSET, but it may even be impossible if the source code of EMPLOYEE is not available. The second choice is to create a new class that captures the relationship, but as discussed above, this leads to a proliferation of *ad hoc* subclasses.

In summary, class inheritance works best for simple tree (or lattice) structures, but in many applications, a given class may satisfy many different interfaces, and may satisfy some

of these in several different ways; furthermore, a given interface may be satisfied by many different classes, sometimes in multiple ways. Moreover, interfaces may have multiple classes and complex properties that involve several classes. See also [3] for some discussion about why class inheritance is inappropriate for large-scale software reuse; in particular, they have found that generic modules are more powerful in that they induce less relationships among components and reduce the total number of components required to build similar abstractions.

Meyer's pioneering comparison of inheritance and composition [77] argued that genericity and inheritance could simulate each other, and also argued that simulating inheritance by genericity was unsatisfactory, because the structures needed for dynamic binding tend to obstruct reuse and maintenance. However, the above difficulties with the use of class inheritance for reusing of generic software components suggests reconsidering Meyer's claim that inheritance is more powerful than genericity: these difficulties also motivate the investigation of alternative mechanisms for composing software components. It may seem that FOOPS solved some of these problems before they were widely discussed!

In FOOPS, the problem of viewing modules differently in different contexts is solved by theories and views, without requiring any additional special-purpose classes or modules. Because the source and target of a view are independent of the view itself, views can express relationships that have not already been expressed at design or coding time; this answers the "foreknowledge" problem. More generally, views can assert that a given module satisfies many different specifications, or that it satisfies the same specification in different ways; they can also assert that a given specification is satisfied by many different modules. Views and theories also solve the library problem, because previously fixed inheritance relationships are not needed for module composition. Moreover, theories can involve multiple classes and complex properties of these classes.

Rosen [98] also indicates some difficulties with class inheritance, and advocates an approach inspired by Ada that emphasises composition; he also argues that good language design should emphasise either inheritance or (module) composition, but not both. However, we think that one can have the best of both worlds, and that this gives rise to some useful new capabilities, such as those provided by parameterised programming.

6.5.12 Comparison with Higher-Order Capabilities

An alternative framework (such as Smalltalk's) might use higher-order operations to achieve some of the functionality of instantiation by views in parameterised programming. However, the difference between these two approaches can be seen as the difference between programming-in-the-small and programming-in-the-large. In parameterised programming operations are considered in association with the kind of data they manipulate, not in isolation; and by allowing abstraction over entire logical units (including module blocks), it realises *large-grain* composition. Additionally, theories and views support further design activities and give rise to other useful features, as discussed in Chapter 3; see also Section 6.7 below.

Moreover, abbreviations suggested in [37] approximate the more economical notation of

higher-order operations in simple situations, and the features of FOOPS are first-order, and thus simpler to reasou about. (See [37] for fnrther discussion of these issues.)

Finally, in parameterised programming several semantic checks can be carried ont at the module composition level, which is more abstract. For example, no errors due to pointers to operations being `nil` can occur (as is possible in Lisp and in Smalltalk). Also, we consider it valuable that the semantics of compositious can be documented.

6.5.13 Summary and Conclusions

We have discussed mechauisms for declaring and instantiatiug generic components iu several languages. Along with FOOPS, the most powerful seem to be Eiffel, ML, Modula-3 and P++, but note that Eiffel's main programming unit is the class aud that ML is uot an object-oriented language. They all provide some form of reusable construct for specifying requirements on arguments, but only Eiffel, FOOPS and Modula-3 can express requirements with compouents that can be composed in the same manner as the generic components that use them (ML's signatures cannot be composed). However, we sbowed that Eiffel's binding mechanism is much less general than tbe views of FOOPS.

Views are also absent froiu the other languages, and many use modules to create bindings, such that all "views" are default. We believe that the advantage of views lies in that tbere is no ueed for modules whose purpose is to provide bindings and not to capture useful data or algorithmic abstractions, that views are self documenting, and that views also serve to express and record refinement and evolution relationships, as discussed in Chapter 3.

We also considered higher-order capabilities, concluding that parameterised programmiug already provides similar functionality. Nevertheless, we note that the appeal of higher-order operatious is difficult to match, perhaps because they seem more economic in terms of uotation.

Vertical parameterisation is a further aspect of generic modules, but we defer its discussiou to the next section.

6.6 Information Hiding

An information hiding mechanism is one of the most complex language aspects to design. The design space is enormous, and the trade-offs difficult to measure. This section examines a variety of approacbes to hiding (and sbaring) information in object-oriented languages.

Languages differ as to what can be hidden; the options include attributes, methods, classes and inheritance relationships. There are also degrees of hiding. For example, some entity might be declared visible to some classes but not to others, or might be visible in a special mode; for instance, an attribute could be invisible, read-only, or read/write.

There are basically two ways to describe visibility. The first is to separate the specification (or interface) of a class or module from its full implementation (also called "body"). The second is to have just one text that is annotated with visibility information, as in FOOPS.

Finally, an information hiding mechanism must also be analysed from a type-checking and security viewpoint, so as to assess the confidence it gives designers that visibility declarations are respected or not easily circumvented.

6.6.1 Ada and Ada 9X

Packages are Ada's main mechanism for encapsulating a set of related variables, routines (procedures and functions) and types; they are also the main unit of scope. Packages have two parts: a specification and a body. The specification lists the types and routines that are visible to clients of the package, and this information is sufficient for clients to be compiled. The body of the package gives the implementation of the items listed in the specification part. With this separation, a very straightforward form of information hiding is achieved by omiting from the specification any auxiliaries that are solely needed for implementation purposes. These would be declared in the body of the package, and available only there.

Ada, however, provides an intermediate level between visible and invisible types. For this purpose, it allows package specifications to be divided into public and private sections. All that is mentioned in the public section is visible to clients, even, for example, the fields of a record type declared there. However, the representation of a type may be made invisible to clients by placing it in the private section of the specification. Such types are of two kinds, **private** and **limited private**, and must be so labelled in the public section of the specification. The reason why the representation of these types must be in the specification is purely implementational: the compiler needs the information in order to allocate space for variables and parameters of these types in clients of the package.

The difference between private and limited private types is the following. With private types, the operations assignment, equality, and inequality are automatically visible to clients of the package, in addition to any others mentioned in the public section of the specification. Types that are limited private, by contrast, only have visible those operations in the public section; assignment, equality, and inequality are automatically hidden to clients; furthermore, not even the body of the package has access to built-in equality and inequality predicates on elements of these types. Types that are limited private are useful for defining data structures that, for example, only tag deleted elements, and these must be ignored when testing for equality (the built-in literal equality would not be appropriate). Note also that types defined in terms of limited private types are automatically limited private.

Ada 9X provides **tagged records** ("classes") to support single class inheritance. As with plain records, it is not possible to hide just some fields: either all are read/write, or none is visible.

A further feature in the new Ada is the **child unit**, which allows a set of related packages to be hierarchically organised. A child unit is also a package, but with declarations stating its connection to another package (its "parent"). It seems that one of the most relevant aspects of this new feature regards visibility: a child unit has full access to the private entities of its parent. According to Barnes [2], one of the principal motivations for child units was for subclasses declared in them to have full visibility of the declarations associated with superclasses in their parents. However, we believe that this gives too much freedom,

as it permits unrestricted manipulation of objects of inherited classes. As will be seen below, other object-oriented languages allow a subclass to share the implementations of its superclasses, but not to apply private operations to objects of those classes. (This was also discussed in Section 3.9.)

6.6.2 C++

A class definition in C++ is divided into interface and implementation parts, allowing other clients and subclasses to be compiled independently of the implementation part. The interface of a class declares its attributes (called "data members") and methods (called "member functions") and whether they are **public**, **protected** or **private**. Public means visible to clients and subclasses; protected means visible only to subclasses (but can only be applied to objects of subclasses; cf. Ada); and private means not visible to any other client or subclass. Visibility restrictions do not apply between objects of the same class, which are free to access each others' internals. Members are private by default, and public data members are read/write.

The implementation part of a class in C++ is not encapsulated, as is the interface. The implementation of a member function is provided by prefixing the class name to the method's header and giving its body, and it does not have to be placed anywhere in particular. As in Ada, the interface/body separation is not as flexible as might be desired. For example, if after describing an interface the need arises for an extra member to help implement a member function, then the interface must be changed to include the new member. In fact, this is more restrictive than in Ada, for there a body can include a procedure that was not even mentioned in the interface, and which belongs to the same scope.

Class inheritance in C++ can also be public, protected or private. If class `B` publicly inherits from class `A`, then an object of `B` can be placed anywhere an object of `A` is expected. If protected, this can only happen within the definition of `B` and its subclasses. And if private, it can only happen within the definition of `B` itself. When inheritance is public, the subclass cannot restrict the visibility of any member more than its parents do.

One salient characteristic of C++'s private members is that they can clash with members in subclasses. For example, if class `A` has a private member function `m` and a subclass `B` of `A` declares a member function `m` too, then the application of `m` to objects of class `B` will be ambiguous. This seems counter-intuitive, and as first indicated (in general) by Snyder [106], it means that changes to the private part of a class can invalidate its subclasses.

A class can declare that other classes and functions are its **friends**, and these have unrestricted access to its private and protected members; also revealed to friends are any inheritance relationships. As in real life, friendship is neither inherited nor transitive.

As mentioned in Section 6.2, classes can be nested within each other, but no special scoping rules apply between them (unless friends); that is, they act as clients of each other. However, a class can declare whether its nested classes are public, protected or private.

It was also mentioned previously that in C++ files can also hide information: entities declared "static" are not visible in any other files. Moreover, it is common practice in the C/C++ community to have "header" files, which act as interface descriptions to files.

However, this is only a convention.

Finally, we note that the pointer-level programming facilities of C++ allow certain visibility "restrictions" to be side-stepped. This is because objects are simply regions of storage which can be indirectly manipulated with appropriate hackery [12, 99, 111]. Thus, C++'s type-checker verifies access, not visibility. (This has much to do with C's "unsafe" programming facilities and tactics having migrated to C++.)

6.6.3 Eiffel

Unlike in the previous languages, there is no separation of specification and implementation for Eiffel classes. Designers annotate class texts to declare which methods are private and which are public. This only affects clients, though, because subclasses can access everything they inherit (but again, only to be applied to objects of their own). Public attributes are read-only, as they can only be directly updated by methods associated with their own class. Communication between objects of the same class can only occur through public attributes and methods (cf. C++). Inherited methods and attributes are private by default, possibly because Eiffel does not require a subclass to export at least the same features as each of its superclasses do.

A facility called **selective exports** allows fine-grain control over visibility by specifying that some features are available only to certain classes. A feature selectively exported to a class C is by default selectively exported to subclasses of C, but not to C's clients. (Note here the difference with friendship in C++, which is not inherited.) Eiffel is like this because of its view that a class should have available the same features used to implement its superclasses.

6.6.4 Oberon-2

Oberon-2 modules also come in one piece, and visibility restrictions are given with annotations. As in Ada, the main unit of scope is the module, and thus types may also be hidden. Entities are private by default, and records ("classes") can specify whether fields are invisible, read-only or read/write. (Note how this is more flexible than in Ada.) Private entities are never visible outside the module that declares them, and they never clash with entities in other modules. A recent proposal by Nigro [86] to extend the information hiding facilities of Oberon-2 suffers from the same disadvantages as Ada 9X's child units.

6.6.5 Smalltalk

In Smalltalk, information hiding is not under the programmer's control. Clients of a class have access to all of its methods, but never direct access to attributes. Subclasses, on the other hand, have full access to inherited features. Communication between objects of the same class can only be achieved through methods.

6.6.6 FOOPS

Except when theories describe generic-module interfaces, FOOPS modules come in one piece and may be annotated with visibility information. The main unit of scope is the module, and classes and sorts can he private to their module or visible in importing modules. Attributes and methods can be private, subclass-private or public. If private, they are only visible in the module that declares them. If subclass-private, they are only visible in modules that declare subclasses (and then only if applied to objects of these classes). And if pnblic (the default) they are visible everywhere. Functions, on the other hand, can only be private or public. Furthermore, private features uever clash with others in other modules. The only possible visibility difference between a class and its superclasses is that a redefinition can be more visible than what it redefines. FOOPS also supports vertical module parameterisation. and module blocks for larger scoping units.

6.6.7 Summary and Conclusions

This section has surveyed the information hiding facilities of various languages. Because information hiding is snch a difficult aspect, the wealth of features and approaches is not surprising With regards to units of scope, there are three broad kinds of language: those with module scope in which many classes have fnll access to each others' features (Ada 9X, Oberon-2, FOOPS); those with class scope in which objects of the same class have full access to each others' internals (C++); and those with object scope in which objects communicate with others via visible operations (Eiffel, Smalltalk). Some of those in the last two categories provide special facilities for classes to reveal otherwise private information to certain classes; namely, C++'s friends and Eiffel's selective exports. While this gives designers fine-grain control, it also creates so-called "spaghetti scoping" [113], which seems to indicate that a higher-level scoping unit might be nissing (e.g., modules). Additionally, these facilities increase the coupling between software components [15].

Of the languages surveyed, C++ is the only one that directly supports hiding inheritance relationships, and in various degrees. (A similar but more restricted capability is fonnd in the language Dee [57].) In FOOPS, there is a more satisfactory way of achieving some[5] of C++'s functionality with the `using` importation mode, as discussed in Chapter 3.

The separation of specifications and implementations in Ada 9X and C++ was shown not to be as convenient as one might wish, because specifications also need to mention private information, which in theory does not belong there. This separation also entails duplicating declarations, which requires more work from both programmers and compilers, especially in ascertaining that the two parts are in agreement [117]. While this has been considered important for allowing separate compilation and thus more effective for software development in teams, it is possible for tools to generate "specifications" from single texts. For example, the Eiffel system has facilities for this; furthermore, compilers can do it automatically [57]. However, there remains a tension between abstract interfaces and the ability to obtain efficient implementations; see [21, 111] for further discussion about this.

[5] It is not possible to simulate protected inheritance in FOOPS.

Eiffel was the only surveyed langnage to permit a subclass to provide fewer public attribntes and methods than its superclasses. In Section 3.9 we mentioned that this creates problems for dynamic binding and that it makes objects look different at different levels of (class) abstraction. However, we note that Eiffel's argument is that this prevents constant rearrangements of inheritance hierarchies when more subclasses need to be accommodated, and that it is feasible to implement an incremental type checker that verifies dynamic binding safety [78].

FOOPS and P++ appear to be the only languages that support the explicit vertical parameterisation of components. Some languages allow actual parameters to generic modnles to be hidden after instantiation, by simply omitting their elements from the module's interface (e.g., this is possible in Modula-3 and ML). The difference is then with regards to how things look in the end (i.e., with the implicit documentation from the genetic's header). Also, this technique might incur in the problems about partial visibility of features that was discnssed in Section 3.9. In several other languages all actual parameters are always visible from instantiations; Ada and Eiffel are two examples.

Lastly, among languages with modules FOOPS offers the most flexible facilities without compromising the encapsulation of data in objects.

6.7 System Design and Development

The pnrpose of programming language notations is to support the design and development of systems that simplify or enbance some task. In order for this to happen, some systematic method is reqnired [8, 77]. Obviously, the languages we have discussed here support object-oriented design and programming as summarised in Chapter 1. But some languages are hetter than others in supporting both activities. We think that Eiffel and FOOPS go beyond other ohject-oriented langnages is this aspect.

Eiffel allows methods and functions to have pre- and post-conditions, and invariants can he attached to classes; all of these are executable, and compiler settings determine whether they are tested at run-time. Also, Eiffel's compiler automatically ANDs the invariant of a class with those of its superclasses, and similarly combines the pre- and post-conditions of methods and functions with those of their redefinitions. These capabilities, together with Eiffel's exception-handling mechanism, constitute a design technique called **programming by contract**.

FOOPS provides theories and views to express relationships of refinement and evolution at different levels of design and programming abstraction, as was discussed in Chapter 3. This approach generalises others based on classes and class inheritance, because it works at the level of systems (modules), and because a view is a completely general mapping between independent modules. It is also valuable that executable and non-executable specifications can be mixed in the same language, that there exists support for various kinds of vertical activities, and that module expressions to describe system designs are provided.

While FOOPS does not offer the pre- and post-condition support of Eiffel, pre-conditions conld he provided by generalising "sort constraints" [42]; it may also be possible to include

class invariants in FOOPS by generalising "sort assertions" [42]; both of these aspects require further research. Note, on the other hand, that theories and views serve to specify syntactic and semantic properties of modules, and thus on groups of classes, which is something not found (in its full generality) in any other object-oriented language.

A further promising line of research in object-oriented design is exemplified by the "roles" of TROLL [62] and the "predicate classes" of Cecil [18], by which an object may belong to different classes at different times based on some conditions or explicitly induced situations (see also [119]). For example, an instance of a class `Tree` may be regarded as an instance of a class `BalancedTree` whenever it satisfies the `balanced` predicate on trees; or a person that finds a job becomes an employee, then perhaps a manager, and so on. In fact, this kind of modelling was one of the motivations for the sort assertions facility mentioned above; see also [50] for some ideas on how to capture this using views dynamically.

Languages such as Ada 9X, C++, Modula-3 and Oberon-2, which descend from older imperative languages, are much less uniform than Eiffel. Smalltalk and FOOPS in their support for the object paradigm. From a theoretical viewpoint, they carry excess baggage; for example, records that are not tagged (Ada 9X), "structs" (C++), global data, and in general, they provide different syntax for expressions involving entities of class types versus those involving entities of "non-class" types. A consequence of this is that software development with these languages does not *naturally* lead to object-oriented architectures, as systems can be built using either functional or object-oriented decomposition. Because the latter is believed superior, having language support for both seems unnecessary. Furthermore, it is becoming increasingly common for a language to be recommended and used for both design and programming, and sometimes even for analysis; in fact, software shops that concentrate on one or two languages will usually design and develop with them. It thus seems advantageous to have "clean" object-oriented languages. Nevertheless, from a practical viewpoint, it is desirable that languages evolve such that existing systems built using earlier versions of the language can be combined with those built with revamped versions; to a large extent, this is true for Ada 9X and C++, for example.

A good language and development environment depends on many things: documentation, debugging facilities, schematic editors, code browsers, compilers, component library and retrieval system, portability, wide applicability, vendor support, integration with other tools (such as operating systems), etc. But while these aspects provide significant leverage, the success of a large software project unfortunately hinges upon less tangible phenomena: the complexity of the problem domain; the fidelity of the requirements document; the effectiveness of communication between the people involved; and the care put into analysing the social context into which the resulting system will be integrated. (These issues lie beyond the scope of this thesis; see [8, 10, 20, 27, 111] for more information and discussion.)

6.8 Summary

This chapter has surveyed and analysed the aspects of object-oriented programming languages that we consider the most important for capturing abstractions and facilitating

software reuse, and has compared them to those present in FOOPS. We began by looking at values and objects, and concluded that it is beneficial for both to be first-class citizens in a language; similarly, we argued that separating the notion of class from the notion of module offered several important advantages. Next we discussed renaming facilities. FOOPS and Eiffel were compared directly as they are the most advanced in this regard. FOOPS showed advantages in code readability and in allowing not only attributes, functions and methods but also classes to be renamed; Eiffel excelled accordingly in providing a sophisticated mechanism for manipulating class texts when a class inherits from another.

Class inheritance and redefinitions were then examined, and here language details were found to vary widely; no language was a clearly superior to others but those with multiple inheritance appear to be more flexible for design (although single inheritance is a simpler concept); also, FOOPS is the only language that separates modules from classes and that includes multiple class inheritance. Support for genericity was considered next, and we found FOOPS to be more powerful than other languages due to its theories and views; various other languages offer higher-order functions and methods to achieve some (but not all) of the same functionality. Vertical parameterisation was another aspect that distinguished FOOPS. We then proceeded to information hiding, and here also language details vary widely; C++ was notable for the various ways in which inheritance relationships can be concealed. However, we have argued in several places against the tendency of trying to make structuring mechanisms for programming-in-the-large fit into some form of class inheritance relationship. Among the languages with modules, FOOPS offers the most flexible information hiding facilities.

Finally, we discussed aspects of system design and development and support for integrating design and programming activities; again, the theories and views of FOOPS appear to provided added flexibility (this was more fully discussed in Chapter 3). We also argued that languages that provide first-class support for both object-oriented and functional decomposition are less uniform and perhaps more confusing. To conclude, we noted that although language issues are very important, the success of a large software project depends on many factors other than the particular programming language used.

Chapter 7

Summary and Further Work

I dread success. To have succeeded is to have finished one's business on earth, like the male spider, who is killed by the female the moment he has succeeded in his courtship. I like a state of continual becoming, with a goal in front and not behind.

— George Bernard Shaw

This thesis gives a detailed study of FOOPS, a wide-spectrum object-oriented langnage. We began by providing pragmatic and economic motivations for the object paradigm and for software reuse, and discussed how FOOPS represents an effort to extend the featnres of object-orientation with novel snpport for the reuse and compositiou (also called interconnection) of modules.

We then described the internal structure of modules, inclnding sort and class declarations; functions, attributes and methods; inheritance of sorts and of classes; and redefinition and dynamic binding for class inheritance. Of particnlar interest here were the declarative style of specification, which involves the use of axioms to express the properties of functions, attributes and methods, and the support for object creation, especially the mechanism for determining and assigning default valnes to attributes.

Third, we examined the capabilities of FOOPS for the reuse and interconnection of modules. An important aspect of FOOPS is its support for desigu in the same framework as specification and coding. Module expressions represent designs, and when they consist of executable modules only, can be symbolically evaluated to prodnce a prototype for the system. The key features here are theories, which are modules that declare properties, and views, which are bindings that express how modnles satisfy theories; views permit many-many relationships between theories and modules. In addition, modules can be composed both horizontally and vertically, allowing designs to be structured in both directions at the same time: horizontally to express module aggregations and specialisations, and vertically to describe layers of abstraction (or stacks of abstract machines). We also discussed information hiding facilities, support for capturing refinement and evolutions relationships between systems, and built-in modules, which cau be used to interface code written in other

languages, and in a way that connects seamlessly with code written in FOOPS[1]. Several examples motivated our discussions; more appear in an appendix.

Next, we supplied a fairly in-depth view of current work towards a semantics for FOOPS, including order-sorted algebra, a logic of inheritance; hidden order-sorted algebra, which formalises basic intuitions about information hiding and the encapsulation of object states; and the theory of institutions, which provides a framework that formally captures features for putting modules together, but which is independent of the logic used for the declarations that modules encapsulate.

In addition, we examined the prototype implementation of FOOPS that we built using facilities given by the implementation of OBJ3. It supports most of the features of FOOPS described here; a notable exception is vertical parameterisation. We discussed its overall design, and also translation and data structnre decisions. Furthermore, we suggested how to improve certain aspects and how to implement some of those features not currently available.

Also, we evaluated FOOPS by comparing it with several other languages. We concentrated on support for large-grain issues such as system design and module reuse and composition. We considered the distinctions between sorts, classes and modules, mechanisms for renaming module features, facilities for class inheritance and redefinition, generic modules and their iustantiation, and information hiding capabilities. We concluded that what distinguishes FOOPS from other languages, and what gives it most of its power, are the separation of modules from classes, including the different kinds of inheritance for each, and its first-class support for theories and views, not only for parameterising and instantiating modnles, hut also for high-level design aud for recording design and historical information. Additionally, it is important that modules can have horizontal as well as vertical parameters.

Even though this thesis has focused on practical aspects, the formal semantics for many of the facilities of FOOPS has been a valuable tool for us throughout the development of this thesis. We used it to uncover and explain the details of several features, to propose new features and applications, to guide our prototype implementation, and to conduct the evaluation and comparison.

It seems to us that the object community may not have paid sufficient attention to large-grain phenomena such as generic architectures (i.e., designs) for large systems, the global properties of snch designs, the compatibility of sub-components, the integration of these capabilities with configuration and version management, and the recording of system development information. We suggest that adding features like those discussed in this thesis to existiug object oriented languages, even to those that identify classes and modules, could enhance their capabilities for design and reuse.

[1]Currently, only Kyoto Common Lisp and, to a certain extent, C can be used quite naturally. Interfacing other languages requires mnch more effort.

7.1 Further Work

There are various directions that seem worth pursuing from the basis provided by this thesis. First, we would like to extend our prototype implementation of FOOPS with the information hiding facilities that we have proposed.

Second, we would like to undertake more comprehensive case studies, to gain a deeper understanding of the language. A method or guidelines for designing systems with FOOPS would be a desirable output of this process. To provide some graphical support, perhaps some existing object-oriented design notation (such as Booch's [8]) could be extended to take into consideration aspects such as theories, views, module expressions, and vertical and horizontal interfaces.

Also, there are extensions to FOOPS that seem to be wanting. For example, "`let`" expressions for declaring local variables in axioms, and user-defined object creation methods that override the automatically provided ones. Also, a notation for pattern-matching objects and succinctly describing updates to them, such as that proposed by Meseguer for his language Maude [76], could reduce the amount of text in many of the FOOPS modules that we have written [108].

A further project is a more detailed comparison of programs at the object level of FOOPS with those at its functional level. In sections 2.2.3 and 3.9 we discussed some characteristic distinctions in the form of axioms, but there seem to be some interesting lessons to be learned from doing this more thoroughly.

Finally, we would also like to study how a program could manage different implementations of objects of the same class. As a start, we believe that it is reasonable to consider that the expressions

```
SET[NAT]{MY-LIST-HACK}
```

and

```
SET[NAT]{MY-OTHER-LIST-HACK}
```

are equivalent, even though they give rise to `Set` classes with different implementations, because the external behaviour of their objects is indistinguishable. It should therefore be possible to interchange them, in the sense of identifying the two classes. The implementation of this appears to be straightforward; its semantics needs to be more carefully considered, and we suspect that behavioural satisfaction and the model-theory work reported in [29] could be relevant to this problem.

Appendix A

Formal Syntax for FOOPS

All you have to do is close your eyes and wait for the symbols.

— Tennessee Williams

This appendix gives a syntax for FOOPS, in five sub-sections. The first describes lexical analysis very briefly; the second presents the syntax for the functional sub-language of FOOPS; the third describes the syntax of the object-oriented sub-language of FOOPS; the fourth gives the syntax for views and module expressions; and the last subsection gives the syntax for the top-level of FOOPS. Each subsection builds upon the previous one. Syntax is described in the extended BNF notation given on page 12; in addition, we use --- to indicate comments in the syntactic description (as opposed to comments in FOOPS code).

As mentioned earlier, the functional level of FOOPS is a syntactic variant of OBJ3. For easy cross-reference, the table below gives the actual syntactic correspondence:

OBJ3 Syntax	*FOOPS Syntax*
`obj`	`fmod`
`endobj`	`endfmod`
`theory`	`ftheory`
`endtheory`	`endftheory`
`eq`	`ax`
`ceq`	`cax`
`beq`	`bax`
`cbeq`	`cbax`
`op`	`fn`
`ops`	`fns`
`ev`	`lisp`
`red`	`eval`

A.1 Lexical Analysis

Tokens are sequences of characters delimited by blanks. The characters "(", ")", and "," are always treated as single character symbols, while tabs and returns are equivalent to

blanks (except inside comments). In many contexts, "[", "]", and "_" are also treated as single character symbols (e.g., in terms).

A.2 Functional-level Modules

```
⟨fMod⟩ ::= fmod ⟨ModInterface⟩ is {⟨fModElt⟩ | ⟨Builtins⟩}... endfmod

⟨fTheory⟩ ::= fth ⟨ModInterface⟩ is ⟨fModElt⟩... endfth

⟨ModInterface⟩ ::= ⟨ModId⟩
      [ [ ⟨ModId⟩... :: ⟨ModExp⟩ {, ⟨ModId⟩... :: ⟨ModExp⟩}... ] ]
      [ "{" ⟨ModId⟩... :: ⟨ModExp⟩ {, ⟨ModId⟩... :: ⟨ModExp⟩}... "}" ]

⟨fModElt⟩ ::=
    {extending | including | protecting} ⟨ModExp⟩ [[private]] . |
    using ⟨ModExp⟩ [⟨Overrides⟩]
        [with ⟨ModExp⟩ [⟨Overrides⟩] {and ⟨ModExp⟩ [⟨Overrides⟩]}...] .
    define ⟨SortId⟩ is ⟨ModExp⟩ [[private]] . |
    sort ⟨SortId⟩ ⟨SortId⟩... [[private]] . |
    principal-sort ⟨Sort⟩ [[private]] . |
    subsort ⟨Sort⟩ ⟨Sort⟩... < ⟨Sort⟩ ⟨Sort⟩... {< ⟨Sort⟩ ⟨Sort⟩...}... . |
    fn ⟨OpForm⟩ : ⟨Sort⟩... -> ⟨Sort⟩ [ [ ⟨fPropList⟩ ] ] . |
    fns ⟨OpForm⟩ ⟨OpForm⟩... : ⟨Sort⟩... -> ⟨Sort⟩ [ [ ⟨fPropList⟩ ] ] . |
    fn-as ⟨OpForm⟩ : ⟨Sort⟩... -> ⟨Sort⟩ for ⟨Term⟩
                                        if ⟨Term⟩ [ [ ⟨fPropList⟩ ] ] . |
    let ⟨Sym⟩ [ : ⟨Sort⟩ ] = ⟨Term⟩ . |
    var ⟨VarId⟩ ⟨VarId⟩... : ⟨Sort⟩ . |
    ax ⟨Term⟩ = ⟨Term⟩ . |
    cax ⟨Term⟩ = ⟨Term⟩ if ⟨Term⟩ . |
    ⟨Misc⟩
--- the definition of ⟨Misc⟩ appears in Section A.5

⟨fPropList⟩ ::= ⟨fProp⟩ ⟨fProp⟩...

⟨fProp⟩ ::= {assoc | comm | {id: | idr:} ⟨Term⟩ | idem | memo |
        strat (⟨Int⟩ ⟨Int⟩...) | prec ⟨Nat⟩ | gather ({e | E | &}...) |
        private | polymorphic ⟨Lisp⟩ | intrinsic}

⟨Builtins⟩ ::=
      bsort ⟨SortId⟩ ⟨Lisp⟩ . |
      bq ⟨Term⟩ = ⟨Lisp⟩ . | bax ⟨Term⟩ = ⟨Lisp⟩ . |
      cbq ⟨Term⟩ = ⟨Lisp⟩ if ⟨Term⟩ . | cbax ⟨Term⟩ = ⟨Lisp⟩ if ⟨Term⟩ .
```

```
⟨ModId⟩   --- simple identifier, by convention all caps
⟨SortId⟩  --- simple identifier, by convention capitalised
⟨VarId⟩   --- simple identifier, typically capitalised
⟨OpName⟩ ::= ⟨Sym⟩ {"_" | " " | ⟨Sym⟩}...
⟨Sym⟩ --- any symbol (blank delimited)
⟨OpForm⟩ ::= ⟨OpName⟩ | (⟨OpName⟩)
⟨Sort⟩   ::= ⟨SortId⟩ | ⟨SortId⟩.⟨Qual⟩
⟨Qual⟩   ::= ⟨ModId⟩ | (⟨ModExp⟩)
⟨Lisp⟩ --- a Kyoto Common Lisp expression

--- the definition of ⟨Overrides⟩ is given in the following section


--- equivalent forms ---

endf = endfmod                  endv = weiv
fth = ftheory                   endfth = endftheory
dfn = define                    us = using
ex = extending                  pr = protecting
sort = sorts                    subsort = subsorts
psort = principal-sort          var = vars
assoc = associative             comm = commutative
id: = identity:                 idr: = identity-rules:
idem = idempotent               prec = precedence
gather = gathering              strat = strategy
poly = polymorphic
```

A.3 Object-level Modules

```
⟨oMod⟩ ::= omod ⟨ModInterface⟩ is {⟨oModElt⟩ | ⟨Builtins⟩}... endomod

⟨oTheory⟩ ::= oth ⟨ModInterface⟩ is ⟨oModElt⟩... endoth

⟨Kind⟩ ::= ⟨Sort⟩ | ⟨Class⟩

⟨oModElt⟩ ::= ⟨fModElt⟩ |
    define ⟨ClassId⟩ is ⟨ModExp⟩ [[private]] . |
    class ⟨ClassId⟩ ⟨ClassId⟩... . |
    principal-class ⟨Class⟩ . |
    subclass ⟨Class⟩ ⟨Class⟩... < ⟨Class⟩ ⟨Class⟩...
                                {< ⟨Class⟩ ⟨Class⟩...}... . |
```

```
    me ⟨OpForm⟩ : ⟨Kind⟩... -> ⟨Kind⟩ [ [ ⟨oPropList⟩ ] ] . |
    mes ⟨OpForm⟩ ⟨OpForm⟩... : ⟨Kind⟩... -> ⟨Kind⟩ [ [ ⟨oProp⟩ ] ] . |
    at ⟨OpForm⟩ : ⟨Kind⟩... -> ⟨Kind⟩ [ [ ⟨oPropList⟩ ] ] . |
    ats ⟨OpForm⟩ ⟨OpForm⟩... : ⟨Kind⟩ -> ⟨Kind⟩ [ [ ⟨oPropList⟩ ] ] . |
    var ⟨VarId⟩ ⟨VarId⟩... : ⟨Class⟩ .
--- attributee and methods must mention at least one
--- class in their arities (except for methods with null
--- aritiee, in which case the coarity must be a class)

⟨oPropList⟩ ::= ⟨oProp⟩ ⟨oProp⟩...

⟨oProp⟩ ::= {assoc | idem | strat (⟨Int⟩ ⟨Int⟩...) | prec ⟨Nat⟩ |
        gather ({e | E | &}...) | polymorphic ⟨Lisp⟩ | intrinsic |
        redef | default: (⟨Term⟩) | {private | subclass-private}}
--- "default:" option only applies to attributes

⟨ClassId⟩ --- simple identifier, by convention capitalised
⟨Class⟩ ::= ⟨ClassId⟩ | ⟨ClassId⟩.⟨Qual⟩

⟨Overrides⟩ ::=
    [ [private ⟨KindRef⟩ ⟨KindRef⟩... ,]
      [subclase-private ⟨OpRef⟩ ⟨OpRef⟩... ,]
      [public ⟨KindRef⟩ ⟨KindRef⟩...] ]
--- for functional modulee, the first two options are not valid
--- the definition of ⟨OpRef⟩ appears towards the end of the next section

--- object creation ---

⟨NewObj⟩ ::= new.⟨ClassId⟩([⟨NewObjArgs⟩])

⟨NewObjArgs⟩ ::= ⟨ObjectId⟩ {, ⟨AttrInit⟩}...

⟨AttrInit⟩ ::= ⟨OpName⟩ = ⟨Term⟩

--- ⟨NewObj⟩ can be put anywhere a ⟨Term⟩ is
--- expected, except on the left-hand sides of axioms

⟨ObjectId⟩ --- eimple identifier


--- equivalent forms ---
```

```
dfn = define                    endo = endomod
oth = otheory                   endoth = endotheory
class = classes                 subclass = subclasses
pclass = principal-class        subclass-private = sc-private
```

A.4 Views and Module Expressions

The syntax for module expressions and views is similar for the functional and the object levels. There is an important semantic restriction, however. in that views may not map modules (or module elements) from one level to the other. But in order to avoid repetition we glance over this restriction in the BNF specification given below.

```
--- views ---

⟨View⟩ ::= view ⟨ModId⟩ from ⟨ModExp⟩ to ⟨ModExp⟩ is ⟨ViewElt⟩... endv

⟨ViewElt⟩ ::= sort ⟨SortRef⟩ to ⟨SortRef⟩ . | class ⟨ClassRef⟩ to ⟨ClassRef⟩ . |
        fn ⟨OpExpr⟩ to ⟨Term⟩ . | fn ⟨FnRef⟩ to ⟨FnRef⟩ . |
        me ⟨MethRef⟩ to ⟨MethRef⟩ | at ⟨AttrRef⟩ to ⟨AttrRef⟩ |
        ⟨oVarDecl⟩... | ⟨fVarDecl⟩...
--- priority given to ⟨OpExpr⟩ case
--- vars are declared with sorts or classes from source of view (a theory)

--- terms ---

⟨Term⟩ ::= ⟨Mixfix⟩ | ⟨VarId⟩ | (⟨Term⟩) |
    ⟨OpName⟩ (⟨Term⟩ {, ⟨Term⟩}...) | (⟨Term⟩).⟨OpQual⟩
--- precedence and gathering rules used to eliminate ambiguity

⟨OpQual⟩ ::= ⟨Sort⟩ | ⟨Class⟩ | ⟨ModId⟩ | ⟨ModExp⟩

⟨Mixfix⟩ --- mixfix operation applied to arguments

--- module expressions ---

⟨ModExp⟩ ::= ⟨ModId⟩ | ⟨ModId⟩ is ⟨ModExpRename⟩ |
        ⟨ModExpRename⟩ * ⟨ModExp⟩ | ⟨ModExpRename⟩

⟨ModExpRename⟩ ::= ⟨ModExpInst⟩ [ * (⟨RenameElt⟩ {, ⟨RenameElt⟩}...) ]

⟨ModExpInst⟩ ::= ⟨ParamModExp⟩ ⟨HorParams⟩ ⟨VertParams⟩ |
                 ⟨ParamModExp⟩ {⟨HorParams⟩ | ⟨VertParams⟩} |
                 (⟨ModExp⟩)
```

```
⟨HorParams⟩ ::= [ ⟨Arg⟩ {, ⟨Arg⟩}... ]
⟨VertParams⟩ ::= "{" ⟨Arg⟩ {, ⟨Arg⟩}... "}"

⟨ParamModExp⟩ ::= ⟨ModId⟩ | (⟨ModId⟩ * (⟨RenameElt⟩ {, ⟨RenameElt⟩}...))

⟨RenameElt⟩ ::= sort ⟨SortRef⟩ to ⟨SortId⟩ | class ⟨ClassRef⟩ to ⟨ClassId⟩ |
          fn ⟨FnRef⟩ to ⟨OpForm⟩ | at ⟨AttrRef⟩ to ⟨OpForm⟩ |
          me ⟨MethRef⟩ to ⟨OpForm⟩

⟨Arg⟩ ::= ⟨ViewArg⟩ | ⟨ModExp⟩ |
    ⟨SortRef⟩ | sort ⟨SortRef⟩ | ⟨ClassRef⟩ | class ⟨ClassRef⟩ |
    ⟨FnRef⟩ | fn ⟨FnRef⟩ | ⟨AttrRef⟩ | at ⟨AttrRef⟩ |
    ⟨MethRef⟩ | me ⟨MethRef⟩
--- may need to precede ⟨FnRef⟩ by "fn", for example, to distinguish
--- from the general case (i.e., from a module name)

⟨ViewArg⟩ ::= view [ from ⟨ModExp⟩ ] to ⟨ModExp⟩ is ⟨ViewElt⟩... endv

⟨SortRef⟩  ::= ⟨Sort⟩ | (⟨Sort⟩)
⟨ClassRef⟩ ::= ⟨Class⟩ | (⟨Class⟩)
⟨FnRef⟩    ::= ⟨FnSpec⟩ | (⟨FnSpec⟩) | ((⟨FnSpec⟩).⟨OpQual⟩)
⟨AttrRef⟩  ::= ⟨AttrSpec⟩ | (⟨AttrSpce⟩) | ((⟨AttrSpec⟩).⟨OpQual⟩)
⟨MethRef⟩  ::= ⟨MethSpec⟩ | (⟨MethSpec⟩) | ((⟨MethSpec⟩).⟨OpQual⟩)
--- in views, (op).(M) must be enclosed in parenthesis, i.e. ((op).(M))
⟨OpRef⟩ ::= ⟨FnRef⟩ | ⟨AttrRef⟩ | ⟨MethRef⟩
⟨FnSpec⟩   ::= ⟨OpName⟩ | ⟨OpName⟩ : ⟨SortId⟩... -> ⟨SortId⟩
⟨AttrSpec⟩ ::= ⟨OpName⟩ | ⟨OpName⟩ : ⟨KindId⟩ -> ⟨KindId⟩
⟨MethSpec⟩ ::= ⟨OpName⟩ | ⟨OpName⟩ : ⟨KindId⟩... -> ⟨KindId⟩
⟨KindId⟩   ::= ⟨SortId⟩ | ⟨ClassId⟩
⟨OpExpr⟩ --- a ⟨Term⟩ that is a single operation applied to variables
```

A.5 The Top Level

```
⟨FOOPS-Top⟩ ::= {⟨fMod⟩ | ⟨fTheory⟩ | ⟨oMod⟩ | ⟨oTheory⟩ |
          ⟨View⟩ | ⟨Make⟩ | ⟨Evaluation⟩ |
          input ⟨FileName⟩ | quit | eof |
          start ⟨Term⟩ . | start-term ⟨Term⟩ . |
          open [⟨ModExp⟩] . | openr [⟨ModExp⟩] . | close |
          ⟨OtherTop⟩}...

⟨Make⟩ ::= make ⟨ModInterface⟩ is ⟨ModExp⟩ endm
```

```
⟨Evaluation⟩ ::= eval [ in ⟨ModExp⟩ : ] ⟨Term⟩ .

⟨Apply⟩ ::=
  apply { reduction | red | print | retr |
    -retr with sort ⟨Sort⟩ |
    ⟨RuleSpec⟩ [ with ⟨VarId⟩ = ⟨Term⟩ { , ⟨VarId⟩ = ⟨Term⟩ }... ]}
  { at | within }
  ⟨Selector⟩ { of ⟨Selector⟩ } ... .

⟨RuleSpec⟩ ::= [-]⟨ModId⟩.⟨RuleId⟩ | [-].⟨RuleId⟩
⟨RuleId⟩ ::= ⟨Natural⟩ | ⟨Id⟩

⟨Selector⟩ ::= that | top |
     (⟨Natural⟩...) |
    [ ⟨Natural⟩ { .. ⟨Natural⟩ ] ] |
    "{" ⟨Natural⟩ {, ⟨Natural⟩}... "}"
--- note that "()" is a valid selector

⟨OtherTop⟩ ::= ⟨EvalLoop⟩ | ⟨Commands⟩ | call-that ⟨Id⟩ [ ⟨ModId⟩ ] . |
      test evaluation [ in ⟨ModExp⟩ : ] ⟨Term⟩ expect: ⟨Term⟩ . | ⟨Misc⟩

⟨EvalLoop⟩ ::= eval-loop {. | ⟨ModId⟩} {⟨Term⟩ .}...

⟨Commands⟩ ::= cd ⟨Sym⟩ | pwd | ls |
    do ⟨DoOption⟩ . |
    select [⟨ModExp⟩] . |
    set ⟨SetOption⟩ . |
    show [⟨ShowOption⟩] .
--- in select, can use "open" to refer to the open module

⟨DoOption⟩ ::= clear memo | gc | save ⟨Sym⟩... | restore ⟨Sym⟩... | ?

⟨SetOption⟩ ::= {abbrev quals | all eqns | all rules | blips |
       clear memo | gc show | include BOOL | obj2 |
       print with parens | reduce conditions | show retracts |
       show var sorts | stats | trace | trace whole} ⟨Polarity⟩
    | ?

⟨Polarity⟩ ::= on | off

⟨ShowOption⟩ ::=
```

```
    {abbrev | all | eqs | mod | name | ops | params | principal-sort |
       rules | select | sign | sorts | subs | vars}
       |⟨ParamSpec⟩ | ⟨SubmodSpec⟩| [⟨ModExp⟩] |
    [all] modes | modules | pending | op ⟨OpRef⟩ | rule ⟨RuleSpec⟩ |
    sort ⟨SortRef⟩ | term | that | time | verbose | ⟨ModExp⟩ |
    ⟨ParamSpec⟩ | ⟨SubmodSpec⟩ | ?
--- can use "open" to refer to the open module

⟨ParamSpec⟩  ::= param ⟨NaturalNumber⟩
⟨SubmodSpec⟩ ::= sub   ⟨NaturalNumber⟩

⟨Misc⟩ ::= lisp ⟨Lisp⟩ | lisp-quiet ⟨Lisp⟩ | parse ⟨Term⟩ . | ⟨Comment⟩

⟨Comment⟩ ::= *** ⟨Rest-of-line⟩ | ***> ⟨Rest-of-line⟩ |
    *** (⟨Text-with-balanced-parentheses⟩)
⟨Rest-of-line⟩ --- the remaining text of the current line


--- equivalent forms ---

el = eval-loop      in = input      q = quit      *** = ---
```

Appendix B

More Examples

The more, the merrier.

— Author unknown

This appendix provides more details and further illustration of examples given in Chapter 3. First, we show the auxiliary modules of the bank accounts example, and also give a module that defines minimum balance accounts. In addition, we include examples involving computations with metaclasses. Second, we complete the generic resource manager example. Finally, we give some example uses of the generic WHILE module.

All of these examples run on the prototype implementation of FOOPS that we have built.

B.1 Bank Accounts

First we present auxiliary modules that define money, dates and transaction histories. The functional module MONEY is:

```
fmod MONEY is
  sort Money .
  pr FLOAT .
  subsort Float < Money .
endf
```

These two modules define the basics of dates:

```
fmod MONTH is
  sort Month .
  ops Jan Feb Mar Apr May Jun : -> Month .
  ops Jul Aug Sep Oct Nov Dec : -> Month .
  op next_ : Month -> Month .
  ax next Jan = Feb .   ax next Feb = Mar .
```

```
  ax next Mar = Apr .   ax next Apr = May .
  ax next May = Jun .   ax next Jun = Jul .
  ax next Jul = Aug .   ax next Aug = Sep .
  ax next Sep = Oct .   ax next Oct = Nov .
  ax next Nov = Dec .   ax next Dec = Jan .

  pr NAT .
  op #daye_ : Month -> Nat .
  ax #days Jan = 31 .   ax #days Feb = 28 .
  ax #days Mar = 31 .   ax #days Apr = 30 .
  ax #days May = 31 .   ax #days Jun = 30 .
  ax #days Jul = 31 .   ax #days Aug = 31 .
  ax #days Sep = 30 .   ax #days Oct = 31 .
  ax #days Nov = 30 .   ax #days Dec = 31 .
endf

fmod DATE is
  sort Date .
  pr MONTH .
  op [_-_-_] : NzNat Month NzNat -> Date .
  op next_ : Date -> Date .
  op day_ : Date -> NzNat .
  op month_ : Date -> Month .
  op year_ : Date -> NzNat .
  var DT : Date .
  var D : NzNat .
  var M : Month .
  var Y : NzNat .
  ax day [D - M - Y] = D .
  ax month [D - M - Y] = M .
  ax year [D - M - Y] = Y .

  cax next DT = [(1 + day DT) - (month DT) - (year DT)]
    if day DT < #days (month DT) .

  cax next DT = [1 - (next month DT) - (year DT)]
    if month DT =/= Dec and day DT == #days (month DT) .

  cax next DT = [1 - (next month DT) - (1 + year DT)]
    if month DT == Dec and day DT == 31 .
endo
```

Module NOW declares a class of objects with one attribute, which gives a date. It also declares an entry-time object to hold the current date.

```
omod NOW is
  class Day .
  pr DATE .
  at date  : Day -> Date .
  me Today : -> Day .
  me next  : Day -> Day .
  var D : Day .
  ax date(next(D)) = next date(D) .
  ax date(Today) =  [ 23 - Aug - 1993 ] .
endo
```

The transaction history of an account is a list of 2-tuples whose first component is a date and whose second component is an amount of money. Module 2TUPLE is [1]:

```
fmod 2TUPLE[C1 :: TRIV, C2 :: TRIV] is
  sort 2Tuple .
  fn <<_;_>> : Elt.C1 Elt.C2 -> 2Tuple .
  fn 1*_ : 2Tuple -> Elt.C1 .
  fn 2*_ : 2Tuple -> Elt.C2 .
  var e1 : Elt.C1 .
  var e2 : Elt.C2 .
  ax 1* << e1 ; e2 >> = e1 .
  ax 2* << e1 ; e2 >> = e2 .
endo
```

Module LIST is:

```
fmod LIST[X :: TRIV] is
  sorts List NeList .
  subsort NeList < List .
  fn nil : -> List .
  fn _ _ : Elt List -> NeList .
  fn hd _ : NeList -> Elt .
  fn tl _ : NeList -> List .
  var E : Elt .  var L : List .
  ax hd(E L) = E .
  ax tl(E L) = L .
endo
```

Combining all of the above modules, we get transaction histories:

[1] This module is part of the default environment for our prototype implementation, which also includes modules for 3-, and 4-tuples.

```
fmod HIST is
  define Hist is LIST[2TUPLE[DATE,MONEY]]
                    * (sort NeList to NeHist,
                       fn nil to emptyHist) .
endf
```

(A `define` declaration renames the principal sort or class of the module expression to the sort or class given.)

Next the definition of cheque histories:

```
fmod CHIST is
  define Chist is LIST[3TUPLE[NAT,DATE,MONEY]]
                  * (sort NeList to NeChist,
                     fn nil to emptyChist) .
endf
```

Lastly, we define minimum balauce savings accounts. These accounts have a minimum balance requirement that must be respected by debits. Therefore the debit method is redefined:

```
omod MBSAVACCT is
  class MBSavAcct .
  ex SAVACCT .
  subclass MBSavAcct < SavAcct .
  at minbal_ : MBSavAcct -> Money .
  var MBSA : MBSavAcct .  var M : Money .
  me debit : MBSavAcct Money -> MBSavAcct [redef] .
  cax  bal debit(MBSA,M) = bal MBSA - M
                if minbal MBSA <= bal MBSA - M .
  cax hist debit(MBSA,M) = << date(Today) ; - M >> hist MBSA
                if minbal MBSA <= bal MBSA - M .
  cax hist debit(MBSA,M) = << date(Today) ; insufunds(M) >> hist MBSA
                if minbal MBSA > bal MBSA - M .
endo
```

B.1.1 Computing with Metaclasses

This section defines a parameterised module that is generic over a binary method `m`, and which defines a method `iter` that applies `m` to each existing object of the class of `m`. Subsequently, this module is instantiated twice to define the methods `iter-credit` and `iter-debit`, which change all of the current objects of class `Acct` by applying `credit` and `debit`, respectively, to each object. First, the theory for the binary method:

```
oth ME is
```

```
  class C . sort Param .
  me m : C Param -> C .
endoth
```

Second, the parameterised module that defines the iteration. A metaclass is defined as an instance of a class called `IdList`, which is declared in the built-in module `OBJ-IDLIST`. This class has two associated attributes, `hd_` and `tl_`, which help define the iteration.

```
omod ITER[M :: ME] is
  pr OBJ-IDLIST .
  me iter : IdList Param -> IdList .
  var X : C . var P : Param . var L : IdList .
  ax iter(L,P) = if L == nil then
                     nil
                 else
                     m(hd(L),P); iter(tl(L),P)
                 fi .
endo
```

The next two `make` commands instantiate `ITER` by viewing `iter` as `debit` and as `credit`, respectively.

```
make ITER-CREDIT is
  ITER[view to ACCT is
         class C to Acct .
         sort Param to Money .
         me m to credit .
       endv] * (me iter to iter-credit) .
endm

make ITER-DEBIT is
  ITER[view to ACCT is
         class C to Acct .
         sort Param to Money .
         me m to debit .
       endv] * (me iter to iter-debit) .
endm
```

This `make` just combines the two previous modules:

```
make ITER-ACCT is ITER-CREDIT + ITER-DEBIT . endm
```

Now some example evaluations:

```
eval new.Acct(A, bal_ = 100) .
eval new.Acct(B, bal_ = 50) .

eval iter-credit(all-Acct,100) .
eval bal A .  ---> should be 200
eval bal B .  ---> should be 150

eval iter-debit(all-Acct,25) .
eval bal A .  ---> should be 175
eval bal B .  ---> should be 125
```

B.2 A Resource Manager

For the specification of the resource manager, we begin by specifying password engines: objects of one attribute, the value of which is to be used as a password. An engine needs to support one method to generate new passwords. A requirement on this method is that the password it generates must be different from those that it had generated previously. This may be accomplished in at least two ways. One is to remember all previous passwords and ensure that new ones are not in this set. The other, which is the option that we have chosen, is to require that the set of possible passwords form a total order, so that a new password may be generated simply by remembering the last one and choosing as the next password a "greater" one. The theory of strict total orders and the object-level theory that expresses the uniqueness of passwords were given in Chapter 3 (see pages 56 and 61).

A model of the total order theory is the natural numbers. They can be used to define the actual engines as a class in which each object stores a natural number that can only be replaced by a larger one, regardless of the initial value stored.

```
omod NAT-NUMBER-PW-ENGINE is
    class PWEngine .
    pr NAT * (sort Nat to Password) .
    at value    : PWEngine -> Password [default: (0)] .
    me make-pw : PWEngine -> PWEngine .
    var P : PWEngine .
    ax value(make-pw(P)) = value(P) + 1 .
  endo
```

We may assert the validity of this implementation with a view, which in this case happens to be empty:

```
view IS-PWE from PW-ENGINE[NAT * (sort Nat to Password)]
            to    NAT-NUMBER-PW-ENGINE is
endv
```

The class of resource managers may now be defined. It is given in a generic module of one parameter, the kind of resource to be managed. A manager stores the current set of free resources, the current set of locked resources, a password engine and a status code that describes the outcome of the last request or release. The `request` method returns a password to the desired resource if it is free; otherwise, `void-Password` is returned. The `release` method frees the resource associated with the password it receives as parameter. The auxiliary modules `SET` and `MAP` are shown further below.

```
omod RESOURCE-MANAGER[RSC :: TRIV * (sort Elt to Resource)] is
    class ResourceMgr .
    sort MgrStatus .
    fns granted released unavailable unknown : -> MgrStatus .
    pr SET[RSC] * (class Set to Resources) .
    pr NAT-NUMBER-PW-ENGINE .
    pr MAP[Password,RSC] .  --- association between locked resources
                            --- and passwords
    at free-resources   : ResourceMgr -> Resources .
    at locked-reeources : ResourceMgr -> Map .
    at pw-engine        : ResourceMgr -> PWEngine .
    --- status of last request or release
    at status           : ResourceMgr -> MgrStatus .
    var P  : Password .
    var R  : Resource .
    var RM : ResourceMgr .

    --- methods to toggle the status attribute
    mes granted released     : ResourceMgr -> ResourceMgr .
    mes unavailable unknown : ResourceMgr -> ResourceMgr .
    ax status(granted(RM))     = granted .
    ax status(released(RM))    = released .
    ax status(unavailable(RM)) = unavailable .
    ax status(unknown(RM))     = unknown .

    --- auxiliary to place a new password in the engine,
    --- and then return it
    me make-pw : ResourceMgr -> Password .
    ax make-pw(RM) = make-pw(pw-engine(RM)); value(pw-engine(RM)) .

    --- is this resource free?
    me is-free : ResourceMgr Resource -> Bool .
    ax is-free(RM,R) = member(free-resources(RM),R) .

    --- is this resource locked?
```

```
    me is-locked : ResourceMgr Resource -> Bool .
    ax is-locked(RM,R) = is-data(locked-resources(RM),R) .

    --- is this a resource the manager knows of?
    me is-resource : ResourceMgr Resource -> Bool .
    ax is-resource(RM,R) = is-free(RM,R) or is-locked(RM,R) .

    --- accept a new resource if it is not a repeat
    me add-resource : ResourceMgr Resource -> ResourceMgr .
    ax add-resource(RM,R) = insert(free-resources(RM),R); RM .

    --- returns a password to the resource if it is free;
    --- otherwise returns void-Password; sets the status
    --- attribute accordingly
    me request : ResourceMgr Resource -> Password? .
    ax request(RM,R) =
          if is-free(RM,R) then
              delete(free-resources(RM),R);
              make-pw(RM);
              insert(locked-resources(RM),value(pw-engine(RM)),R);
              granted(RM); value(pw-engine(RM))
          else if is-resource(RM,R) then
              unavailable(RM); void-Password
          else
              unknown(RM); void-Password
          fi fi .

    --- if there is a resource locked with the given password
    --- then unlock it and insert it into the free-resource pool;
    --- otherwise do nothing;
    --- also, set the status attribute accordingly
    me release : ResourceMgr Password -> ResourceMgr .
    ax release(RM,P) =
          if is-key(locked-resources(RM),P) then
              insert(free-resources(RM),
                     get-data(locked-resources(RM),P));
              delete(locked-resources(RM),P);
              released(RM)
          else
              unknown(RM)
          fi .
endo
```

By defining a module that declares various resources, such as:

```
fmod RESOURCES is
  sort Resource .
  fns diskA diskB diskC : -> Resource .
  fns printerA printerB : -> Resource .
endf
```

we may instantiate the resource manager module, like this:

```
make TEST-RESOURCE-MGR is RESOURCE-MANAGER[RESOURCES] . endm
```

Then, the following evaluation creates a resource manager with all its internal state initialised:

```
eval new.ResourceMgr(Mgr) .
```

And these other evaluations show the defaults that were computed:

```
eval empty(free-resources(Mgr)) .     ---> should be true
eval empty(locked-resources(Mgr)) .   ---> should be true
eval value(pw-engine(Mgr)) .          ---> should be 0
```

B.2.1 Auxiliaries

The first auxiliary module specifies sets by interfacing the underlying Lisp system. We will not explain this in any more detail; it is just an application of a facility described in [53], and also discussed in Chapter 3. The module after that defines sets at the object level by using this initial description. There, each set is associated with a "cursor" that may be used for iterating over its elements.

```
--- this routine prints sets with the usual curly-bracket notation
    lisp-quiet
    (defun set$print (s)
      (princ "")
      (dotimes (i (length s))
        (when (< 0 i) (princ ","))
        (print$check)
        (term$print (elt s i))
        )
      (princ "")
      )

    fmod BUILT-IN-SET[X :: TRIV] is
      --- built-in sorts are defined using lisp
      bsort BISet ((lambda (x) nil) (lambda (x) (break))
```

```
                set$print (lambda (x) t)) .
  pr NAT .
  fn make-set : -> BISet .
  fn insert : BISet Elt -> BISet .
  fn delete : BISet Elt -> BISet .
  --- cardinality
  fn nb-elts : BISet -> Nat .
  --- membership test
  fn member  : BISet Elt -> Bool .
  --- emptyness
  fn empty : BISet -> Bool .
  --- iteration help
  fn ith : BISet Nat -> Elt .
  fn position : BISet Elt -> Nat .
  var S : BISet .  var E : Elt .  var NZ : NzNat .
  bq make-set = (progn nil) .
  beq insert(S,E) = (term$make_built_in_constant (term$sort self)
     (adjoin E (term$built_in_value S) :test #'term$similar2)) .
   beq delete(S,E) = (term$make_built_in_constant (term$sort self)
      (remove-if #'(lambda (x) (term$similar2 x E))
                   (term$built_in_value S))) .
  bq nb-elts(S) = (length S) .
  beq member(S,E) =
      (obj_BOOL$coerce_to_Bool
        (find-if #'(lambda (x) (term$similar2 x E))
          (term$built_in_value s))) .
  ax empty(S) = nb-elts(S) == 0 .
  cbeq ith(S,NZ) = (elt (term$built_in_value S)
                        (1- (term$built_in_value NZ)))
                   if NZ <= nb-elts(S) .
  cbeq position(S,E) =
    (term$make_built_in_constant (term$sort self)
      (1+ (position-if #'(lambda (x) (term$similar2 x E))
                       (term$built_in_value S))))
    if member(S,E) .
  cbeq position(S,E) = 0  if not member(S,E) .
endf

omod SET[X :: TRIV] is
  class Set .
  pr BUILT-IN-SET[X] .
```

```
--- aids for iterating over
--- cursor: index of current element
at current : Set -> Nat [default: (0)] .
--- place cursor on firet element
me start : Set -> Set .
--- move cursor to next element
me forth : Set -> Set .
--- value of current element
at value : Set -> Elt? .
--- is cursor out of bounds?
at finished : Set -> Bool .

at contents : Set -> BISet [default: (make-set)] .
me insert : Set Elt -> Set .
me delete : Set Elt -> Set .
--- cardinality
at nb-elts : Set -> Nat .
--- membership test
at member  : Set Elt -> Bool .
--- emptyness
at empty : Set -> Bool .
var S : Set .  var E : Elt .
cax contents(insert(S,E)) = insert(contents(S),E) if not member(S,E) .
cax contents(insert(S,E)) = contents(S)           if member(S,E) .
ax contents(delete(S,E)) = delete(contents(S),E) .
ax current(delete(S,E)) =
      if position(contents(S),E) >= current(S) then
          current(S)
      else
          p current(S)
      fi .
ax nb-elts(S)  = nb-elts(contents(S)) .
ax member(S,E) = member(contente(S),E) .
ax empty(S)    = empty(contents(S)) .
ax finished(S) = current(S) > nb-elts(S) .
ax value(S) =  if finished(S) then
                   void-Elt
               else
                   ith(contents(S),current(S))
               fi .
ax current(start(S)) = 1 .
ax current(forth(S)) = current(S) + 1 .
```

```
endo
```

Module MULTI-MAP declares a class of objects whose state is a set of pairs, where the first component of the pair is called its **key** and the second component is called its **data**. Keys may be associated with more than one data. (See page 59 for the definition of pairs.)

```
omod MULTI-MAP[KEY  :: TRIV * (sort Elt to Key),
               DATA :: TRIV * (sort Elt to Data)] is
  class Multimap .
  ex SET[PAIR[KEY,DATA] * (at fst_ to key, at snd_ to data,
                           me replace-fst to replace-key,
                           me replace-snd to replace-data)]
     * (me insert to set-insert) .
  subclass Multimap < Set .

  pr SET[KEY]  * (class Set to KeySet) .
  pr SET[DATA] * (class Set to DataSet) .

  var M : Multimap .
  var K : Key .        var D : Data .
  var DS : DataSet .   var KS : KeySet .

  --- number of elements with given key
  me count-key  : Multimap Key -> Nat .
  me count-key2 : Multimap Key -> Nat [private] .
  ax count-key(M,K) = count-key2(start(M),K) .
  ax count-key2(M,K) =
       if finished(M) then
           0
       else if key(value(M)) == K then
           1 + count-key2(forth(M),K)
       else
           count-key2(forth(M),K)
       fi fi .

  --- number of elements with given data
  me count-data  : Multimap Data -> Nat .
  me count-data2 : Multimap Data -> Nat [private] .
  ax count-data(M,D) = count-data2(start(M),D) .
  ax count-data2(M,D) =
       if finished(M) then
           0
       else if data(value(M)) == D then
```

```
            1 + count-data2(forth(M),D)
        else
            count-data2(forth(M),D)
        fi fi .

--- is there an element with the given key?
me is-key : Multimap Key -> Bool .
ax is-key(M,K) = count-key(M,K) > 0 .

--- is there an element with the given data?
me is-data : Multimap Data -> Bool .
ax is-data(M,D) = count-data(M,D) > 0 .

--- test for presence of a given (key,data) pair
me member  : Multimap Key Data -> Bool .
me member2 : Multimap Key Data -> Bool [private] .
ax member(M,K,D) = member2(start(M),K,D) .
ax member2(M,K,D) =
        if finished(M) then
            false
        else if key(value(M)) == K and data(value(M)) == D then
            true
        else
            member2(forth(M),K,D)
        fi fi .

--- nothing happens for duplicate pairs
me insert : Multimap Key Data -> Multimap .
ax insert(M,K,D) =
        if member(M,K,D) then
            M
        else
            set-insert(M,new.Pair(key = K, data = D))
        fi .

me keyset : Multimap -> KeySet .
me keyset2 : Multimap KeySet -> KeySet [private] .
ax keyset(M) = keyset2(start(M),new.KeySet()) .
ax keyset2(M,KS) =
        if finished(M) then
            KS
        else
```

```
            insert(KS,key(value(M)));
            keyset2(forth(M),KS)
        fi .

  me dataset : Multimap -> DataSet .
  me dataset2 : Multimap DataSet -> DataSet [private] .
  ax dataset(M) = dataset2(start(M),new.DataSet()) .
  ax dataset2(M,DS) =
        if finished(M) then
            DS
        else
            insert(DS,data(value(M)));
            dataset2(forth(M),DS)
        fi .

  --- remove all entries with the given key
  --- no effect if no such key
  me delete  : Multimap Key -> Multimap .
  me delete2 : Multimap Key -> Multimap [private] .
  ax delete(M,K) = delete2(start(M),K) .
  ax delete2(M,K) =
        if finished(M) then
            M
        else if key(value(M)) == K then
            delete(M,value(M)); delete2(M,K)
        else
            delete2(forth(M),K)
        fi fi .
endo
```

Finally, module MAP declares a *subclass of* Multimap *in* which keys are unique. This requires the redefinition of method insert.

```
omod MAP[KEY  :: TRIV * (sort Elt to Key),
         DATA :: TRIV * (sort Elt to Data)] is
  class Map .
  ex MULTI-MAP[KEY,DATA] .
  subclass Map < Multimap .

  var M : Map .  var K : Key .  var D : Data .

  --- return data with the given key
```

```
  me get-data  : Map Key -> Data? .
  me get-data2 : Map Key -> Data? [private] .
  ax get-data(M,K) = get-data2(start(M),K) .
  ax get-data2(M,K) =
        if finished(M) then
            void-Data
        else if key(value(M)) == K then
            data(value(M))
        else
            get-data2(forth(M),K)
        fi fi .

  --- redefinition: don't allow two elements with the same key
  me insert : Map Key Data -> Map [redef] .
  ax insert(M,K,D) =
        if is-key(M,K) then
            M
        else
            set-insert(M,new.Pair(key = K, data = D))
        fi .
endo
```

B.3 Iterators

We restrict our attention to linear iteration, and in particular to linear iteration over traversable structures. **Linear iteration** is that in which the loop proceeds over the structure in one direction only. A **traversable structure** is one that may be looped over with a cursor, and that supports the following operations:

- **item**, which returns the value of the item where the cursor is at;
- **start**, which places that cursor on the first element;
- **forth**, which moves the cursor to the next element; and,
- **finished**, which tests whether the cursor is on any item. It is used as a termination test.

The particular kind of linear iteration that we consider is the **while** loop, which (in general) has the form of this Pascal-like fragment:

initialisation;
while *test* **do**

some-action
endwhile;
wrapup;

For traversable structures (or traversables, for short), the above fragment specialises to the following one:

initialisation; *start*;
while not *(finished)* **and** *test* **do**
some-action; *forth*
endwhile;
wrapup;

Other forms of iteration, such as two-way traversals, and other kinds of loop structure, such as **repeat-until**, follow a similar pattern.

To realise these kinds of iteration we use the following setup. Theory `ITER-ACTIONS` (see page 57) describes a class with the methods `init`, `action`, `test` and `wrapup`. It serves to express the minimal requirements on actual arguments to the parameterised module `WHILE` (see page 60), which declares a method `while` that is defined in terms of these other methods. Theory `TRAVERSABLE` describes traversable structures, and is extended by theory `TRAVERSABLE-WITH-ACTIONS`, which describes a class of traversable structures which also has methods that correspond to `init`, `action`, etc. This theory is in turn used to describe the interface to module `TRAV-WHILE`, which specialises method `while` for traversable structures.

```
oth TRAVERSABLE[X :: TRIV] is
  class C .
  me item_  : C -> Elt? .
  me start_ : C -> C .
  me forth_ : C -> C .
  me finished_   : C -> Bool .
endoth

--- "using TRAVERSABLE[X]" would be ideal, but
--- this importation mode is not yet implemented in
--- FOOPS (but OBJ3 supports it)
oth TRAVERSABLE-WITH-ACTIONS[X :: TRIV] is
  class C .
  sorts In Out .
  me item_     : C -> Elt? .
  me start_    : C -> C .
  me forth_    : C -> C .
  me finished_ : C -> Bool .
  me init      : C In -> C .
```

```
  me action    : C In -> C .
  me test      : C In -> Bool .
  me wrapup    : C In -> Out .
endoth
```

Module TRAV-WHILE is defined as a parameterised module of two arguments. The first is for the kind of data stored in the traversable structure, and the second is for the structure itself. The body of TRAV-WHILE is an instantiation of WHILE, and a view describes the binding from the elements ITER-ACTIONS (the interface theory of WHILE) to the elements TRAVERSABLE-WITH-ACTIONS. This view adds-in the code to move along the traversable structure:

```
make TRAV-WHILE[DATA :: TRIV, X :: TRAVERSABLE-WITH-ACTIONS[DATA]] is
  WHILE[view to X is
          class C  to C .
          sort In  to In .
          sort Out to Out .
          var E : C .  var I : In .
          me init(E,I)   to init(E,I); start(E) .
          me action(E,I) to action(E,I) ; forth(E) .
          me test(E,I)   to if not finished(E)
                              then test(E,I)
                              else false
                            fi .
          me wrapup(E,I) to wrapup(E,I) .
        endv] .
endm
```

The resulting axioms that describe methods while and while-continue are then:

```
ax while(E,I) = start(E); init(E,I); while-continue(E,I) .
ax while-continue(E,I) =
       if (if not finished(E) then test(E,I) else false fi) then
           action(E,I); forth(E); while-continue(E,I)
       else
           wrapup(E,I)
       fi .
```

Incidentally, we could have relied on default-view conventions and omitted these view elements:

```
class C   to C .
sort In  to In .
sort Out to Out .
me wrapup(E,I) to wrapup(E,I) .
```

Now a simple instantiation. First we present the basic parts of a library module that implements linked lists (its full definition is available with the implementation of the system).

```
omod LINKED-LIST[X :: TRIV] is
  class List .
  pr LINKABLE[X] .   --- private
  pr INT .
  at nb-elts_  : List -> Nat [default: (0)] .
  --- first element at position 1
  at position_ : List -> Nat [default: (0)] .

  --- identity method
  me id : List -> List .

  --- value of element at cursor position
  at item_ : List -> Elt? .
  --- is list empty?
  at empty_ : List -> Bool .
  --- is cursor off right edge?
  at offright_ : List -> Bool .
  --- is cursor off left edge?
  at offleft_ : List -> Bool .

  at finished_ : List -> Bool .

  --- does element i exist?
  at valid-position : List Nat -> Bool .

  --- removes all elements
  me clear : List -> List .

  --- change value of element at cursor position
  me replace-value : List Elt -> List .

  --- move cursor to first element; no effect if list is empty
  me start_ : List -> List .

  --- move cursor off left edge; private
  me go-offleft_ : List -> List .
  --- move cursor off right edge; private
  me go-offright_ : List -> List .
```

```
  --- move cursor to next element
  me forth_ : List -> List .

  --- insert an element to the right of cursor position.
  --- Do not move cursor. Position does not change.
  --- if list is empty, it is left offleft
  --- if list is offleft, element is inserted at the beginning
  --- if list is offright, element is inserted at the tail
  me insert-right : List Elt -> List .

  --- insert an element to the left of cursor position.
  --- Do not move cursor. Position increases by 1.
  --- if list is empty, it is left offleft
  --- if list is offleft, element is inserted at the beginning
  --- if list is offright, elemsnt is inserted at the tail
  me insert-left : List Elt -> List .
endo
```

The following **make** generates a (module that includes a) method for clearing a list and inserting into it a value a certain number of times. The value and the number of times is given in a pair.

```
make INIT-LIST[X :: TRIV] is
  WHILE[view to LINKED-LIST[X] *
                PAIR[X,NAT] * (at fst_ to value, at snd_ to length) is
        class C   to List .
        class In  to Pair .
        class Out to List .
        var L : C .   var I : In .
        me init(L,I)   to clear(L) .
        me action(L,I) to insert-right(L,value(I)) .
        me test(L,I)   to nb-elts(L) < length(I) .
        me wrapup(L,I) to id(L) .
      endv]
  * (me while to init-list) .
endm
```

The initialisation is to clear the list, the action is an insert, the test checks the number of elements in the list, and the wrapup just returns the list[2]. An example use of the method would be `init-list(l,p)`, where `l` is a list and `p` a pair with the required values for its components. Note that this instantiation of `WHILE` is multilevel (i.e., of the form `F[G[H] * P[H,R]]`).

Now an example that involves copying lists with the following methods:

[2]The current implementation does not allow the target of **wrapup** to simply be L, as would be desired.

- **copy** l_1 **to** l_2, which clears l_2 and then copies the contents of l_1 onto it, and
- **make-copy**(l), which creates a new list whose elements are the same as those of l.

Module COPY-LIST below instantiates TRAV-WHILE by providing a view from TRAVERSABLE-WITH-ACTIONS to module LIST. The fact that there is a default view from TRAVERSABLE (a subtheory of TRAVERSABLE-WITH-ACTIONS) to LIST helps here, as we are able to omit from the view elements such as

```
me start(L) to start(L) .
```

The module is:

```
omod COPY-LIST[X :: TRIV] is
  pr TRAV-WHILE[X,
                view to LINKED-LIST[X] is
                  class C   to List .
                  class In  to List .
                  class Out to List .
                  var L : C .  var I : In .
                  me init(L,I)   to start(clear(I)); L .
                  me action(L,I) to insert-right(I,item(L));
                                    forth(I); L .
                  me test(L,I)   to true .
                  me wrapup(L,I) to id(L) .
                endv]
    * (me while to copy_to_, me while-continue to continue-copy_to_) .

  me make-copy : List -> List .
  var L : List .
  ax make-copy(L) = copy L to new.List() .
endo
```

In particular, observe that we use the extra parameters to `init`, `action`, etc. to carry around the target list. Also, note how `make-copy` uses `copy_to_` and object creation to generate a fresh copy of its argument.

Finally, the example below is derived from one that appears in [78], pages 174–177. It consists of adding the speed of the first n particles in a list. The top-level method is:

add-speeds(l,n).

This example follows a pattern similar to the previous one, and again the extra input argument to `init`, `action`, etc. serves to carry around temporary data. For this, module TRIPLE is used:

First we need definitions for speed and particles.

```
make SPEED is FLOAT * (sort Float to Speed) . endm

omod PARTICLE is
  class Particle .
  pr FLOAT + SPEED .
  at mass  : Particle -> Float .
  at speed : Particle -> Speed .
  at positively_charged : Particle -> Bool .
  --- etc.
endo
```

Below we use 3-tuples for storing the number of elements examined so far (`counted`), the number of particles to be examined (`threshold`), and the accumulated result (`sum`).

```
omod TRIPLE[X :: TRIV, Y :: TRIV, Z :: TRIV] is
  class Triple .
  ex PAIR[X,Y] .
  subclass Triple < Pair .
  at trd_ : Triple -> Elt.Z .
  vars T T2 : Triple .  var V : Elt.Z .
  at equal : Triple Triple -> Bool [redef] .
  ax equal(T,T2) = fst T == fst T2  and
                   snd T == snd T2  and
                   trd T == trd T2 .
  me replace-trd : Triple Elt.Z -> Triple .
  ax trd replace-trd(T,V) = V .
endo

omod PARTICLE-SPEEDS is
  pr TRAV-WHILE[PARTICLE,
                view to LINKED-LIST[PARTICLE]
                        + TRIPLE[INT,INT,SPEED]
                          * (at fst_ to counted,
                             me replace-fst to set-counted,
                             at snd_ to threshold,
                             me replace-snd to set-threshold,
                             at trd_ to sum,
                             me replace-trd to set-sum) is
                  class C   to List .
                  class In  to Triple .
                  sort Elt to Particle .
                  sort Elt? to Particle? .
                  sort  Out to Speed .
```

```
    var L : C .  var T : In .
    me init(L,T)   to set-sum(T,0); set-counted(T,0); L .
    me action(L,T) to set-sum(T,sum(T) + speed(item(L)));
                      set-counted(T,counted(T) + 1); L .
    me test(L,T)   to threshold(T) > counted(T) .
    me wrapup(L,T) to sum(T) .
  endv] .

  me add-speeds : List Nat -> Speed .
  var L : List .   var N : Nat .
  ax add-speeds(L,N) = while(L,new.Triple(threshold = N)) .
endo
```

It seems interesting that this way of structuring iterators differs from Meyer's in that we avoid the use of constrained genericity. Section 6.5 discussed the advantages of modules being *generic* over *theories* and instantiated with *views*.

Finally, we note that it would also be possible to have module `WHILE` declare a new class called `While` as client of the data structure to be iterated over, and have the iteration methods be associated with this new class. (Incidentally, `While` would be a natural candidate for an abstract class.) The flexibility that this allows is that particular iterations could be defined by subclassing `While` and overriding its iteration methods; moreover, its subclasses could declare attributes for temporary storage. This alternative approach exemplifies another advantage of the distinction between classes and modules, because these subclasses would arise mostly as auxiliaries to other classes (our experiments confirm this).

Bibliography

What is so wounderful about great literature is that it transforms the man who reads it towards the condition of the man who wrote, and brings to birth in us also the creative impulse.

— E.M. Forster

Note: references [5], [72] and [91] are not cited in the text.

[1] Antonio Alencar. *OOZE: An Object Oriented Z Environment.* PhD thesis, Oxford University, 1994 (to appear).

[2] John Barnes. Introducing Ada 9X. Technical report, Intermetrics, Inc., February 1993. Ada 9X Project Report.

[3] Don Batory, Vivek Singhal, and Jeff Thomas. Scalable Software Libraries. In *Proceedings of the ACM Symposium on the Foundation of Software Engineering*, 1993 (to appear).

[4] Don S. Batory and Sean O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[5] Edward Berard. Abstraction, Encapsulation, and Information Hiding. *USENET newsgroup* `comp.object`. November 1991.

[6] Gleun Bergland. A Guided Tour of Programming Methodologies. *Computer*, 14(10):13–37, October 1981.

[7] Barry W. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[8] Grady Booch. *Object Oriented Design, with Applications.* Benjamin/Cummings, 1991.

[9] Paulo Borba. A Proof System for FOOPS, 1993. Programming Research Group, University of Oxford.

[10] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, 1975.

[11] Frederick P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, April 1987.

[12] Timothy Budd. *An Introduction to Object-Oriented Programming.* Addison-Wesley, 1991.

[13] Rod Burstall and Răzvan Diaconescu. Hiding and Behaviour: an Institutional Approach. Technical Report ECS-LFCS-8892-253, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.

[14] Rod Burstall and Joseph Goguen. The Semantics of Clear, a Specification Language. In Dines Bjørner, editor, *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer-Verlag, 1980. Lecture Notes in Computer Science, Volume 86.

[15] Frank W. Calliss. A Comparison of Module Constructs in Programming Languages. *ACM SIGPLAN Notices*, 26(1):38–46, January 1991.

[16] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 Language Definition. *ACM SIGPLAN Notices*, 27(8):15–42, August 1992.

[17] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[18] Craig Chambers. Predicate Classes. In *Proceedings of the European Conference on Object-Oriented Programming, Kaiserslautern, Germany*, 1993. To appear.

[19] J. C. Cleaveland. Building an Application Generator. *IEEE Software*, 5(4):25–33, July 1988.

[20] Peter Coad and Edward Yourdon. *Object-Oriented Analysis.* Yourdon Press, 1991.

[21] Christian S. Collberg. *Flexible Encapsulation.* PhD thesis, Lund University, Sweden, 1992.

[22] R.C.H. Connor, A. Dearle, R. Morrison, and A.L. Brown. An Object Addressing Mechanism for Statically Typed Languages with Multiple Inheritance. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, Orlando, Florida*, pages 279–285, 1989.

[23] William R. Cook. A Proposal for making Eiffel Type-safe. *Computer Journal*, 32(4):305–311, 1989.

[24] Fernando J. Corbató. On Building Systems That Will Fail. *Communications of the ACM*, 34(9):72–81, September 1991. Turing Award Lecture.

[25] Brad J. Cox and Andrew J. Novobilski. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1991.

[26] Clive Davidson. The man who made computers personal. *New Scientist*, pages 32–35, 19 June 1993.

[27] Peter J. Denning. Beyond Formalism. *American Scientist*, 79(1):8–10, January-February 1991.

[28] L. Peter Deutsch. Posted comments. *USENET newsgroup* `comp.object`, May 1992.

[29] Răzvan Diaconescu, Joseph Goguen, and Petros Stefaneas. Logical Support for Modularisation. In Gordon Plotkin and Gérard Huet, editors, *Proceedings of the Workshop on Types and Logical Frameworks, Edinburgh, Scotland*. Cambridge University Press, 1992.

[30] Liesbeth Dusink. fntroduction to Re-use. In Liesbeth Dusink and Patrick Hall, editors, *Proceedings of the Software Re-use Workshop, Utrecht, The Netherlands 1989*, Workshops in Computing, pages 1–6. Springer-Verlag, 1991.

[31] Guy Fitzgerald. Implications of Strategic Information Technology for Requirements Engineering. Lecture given at Oxford University, England, 14 May 1992.

[32] Richard Gabriel, Jon L. White, and Daniel G. Bobrow. CLOS: Integrating Object-Oriented and Functional Programming. *Communications of the ACM*, 34(9):29–38, September 1991.

[33] Joseph Goguen. Mathematical Representation of Hierarchically Organized Systems. In E. Attinger, editor, *Global Systems Dynamics*, pages 112–128. S. Karger, 1971.

[34] Joseph Goguen. Order Sorted Algebra. Technical Report 14, UCLA Computer Science Department, 1978. Semantics and Theory of Computation Series.

[35] Joseph Goguen. Reusing and Interconnecting Software Components. *Computer*, 19(2):16–28, February 1986.

[36] Joseph Goguen. Principles of Parameterized Programming. In Ted Biggerstaff and Alan Perlis, editors, *Software Reusability, Volume I: Concepts and Models*, pages 159–225. Addison-Wesley, 1989.

[37] Joseph Goguen. Higher-Order Functions Considered Unnecessary for Higher-Order Programming. In David Turner, editor, *Research Topics in Functional Programming*, pages 309–351. Addison-Wesley, 1990.

[38] Joseph Goguen. Hyperprogramming: A Formal Approach to Software Environments. In *Proceedings of the Symposium on Formal Approaches to Software Environment Technology*. Joint System Development Corporation, Tokyo, Japan, 1990.

[39] Joseph Goguen. Types as Theories. In George Michael Reed, Andrew William Roscoe, and Ralph F. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford University Press, 1991.

[40] Joseph Goguen. *Theorem Proving and Algebra*. M.I.T. Press, to appear.

[41] Joseph Goguen and Rod Burstall. Institutions: Abstract Model Theory for Specification and Programming. *Journal of the ACM*, 39(1):95–146, January 1992.

[42] Joseph Goguen and Răzvan Diaconescu. A Survey of Order Sorted Algebra. To appear in *Bulletin of the European Association for Theoretical Computer Science*, 1993.

[43] Joseph Goguen and Răzvan Diaconescu. Towards an Algebraic Semantics for the Object Paradigm. In *Proceedings, Tenth Workshop on Abstract Data Types*. Springer, to appear 1993.

[44] Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Operational Semantics of Order-Sorted Algebra. In W. Brauer, editor, *Proceedings, 1985 International Conference on Automata, Languages and Programming*. Springer-Verlag, 1985. Lecture Notes in Computer Science, Volume 194.

[45] Joseph Goguen and Tom Kemp. A Hidden Herbrand Theorem, 1993. Submitted to special issue of *Theoretical Computer Science*, edited by Andrew William Roscoe and Michael W. Mislove.

[46] Joseph Goguen, Claude Kirchner, and José Meseguer. Concurrent Term Rewriting as a Model of Computation. In Joseph H. Fasel and Robert M. Keller, editors, *Graph Reduction, Lecture Notes in Computer Science, No. 279*, pages 53–93. Springer-Verlag, 1986.

[47] Joseph Goguen and José Meseguer. Universal Realization, Persistent Interconnection and Implementation of Abstract Modules. In M. Nielsen and E.M. Schmidt, editors, *Proceedings, 9th International Conference on Automata, Languages and Programming*, pages 265–281. Springer, 1982. Lecture Notes in Computer Science, Volume 140.

[48] Joseph Goguen and José Meseguer. Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics. In *Research Directions in Object-Oriented Programming*, pages 417–477. M.I.T. Press, 1987.

[49] Joseph Goguen and José Meseguer. Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

[50] Joseph Goguen and Adolfo Socorro. Module Composition and System Design for the Object Paradigm. *Journal of Object-Oriented Programming*, August 1994 (to appear).

[51] Joseph Goguen, James Thatcher, and Eric Wagner. An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. Technical Report RC 6487, IBM T.J. Watson Research Center, October 1976. In *Current Trends in Programming Methodology, IV*, Raymond Yeh, editor, Prentice-Hall, 1978, pages 80–149.

[52] Joseph Goguen and Timothy Winkler. Introducing OBJ3. Technical report, Computer Science Laboratory, SRI International, August 1988. SRI-CSL-88-9.

[53] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannand. Introducing OBJ. Technical report, SRI International, 1993. To appear.

[54] Joseph Goguen and David Wolfram. On Types and FOOPS. In Robert Meersman, William Kent, and Samit Khosla, editors, *Proceedings of the IFIP TC2 Working Conference on Database Semantics: Object-Oriented Databases: Analysis, Design and Construction, Windermere, United Kingdom*, July 1990.

[55] Joseph A. Goguen. Modular Algebraic Specification of Some Basic Geometrical Constructions. *Artificial Intelligence*, 37:123–153, 1988.

[56] Adele Goldberg and David Robson. *Smalltalk-80: The Language.* Addison-Wesley, 1989.

[57] Peter Grogono. Issues in the Design of an Object-Oriented Programming Language. *Structured Programming*, 12:1–15, 1991.

[58] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. *Proceedings of the European Conference on Object-Oriented Programming, Geneva, Switzerland.* July 1991. Lecture Notes in Compnter Science 512.

[59] Jean D. Ichbiah. Rationale for the Design of the Ada Programming Language. *ACM SIGPLAN Notices*, 14(6), Juue 1979.

[60] Michael A. Jackson. *System Development.* Prentice-Hall International, 1983.

[61] Ian Joyner. A Criticism of C++. *Journal of Object-Oriented Programming*, 1993. To appear.

[62] Ralf Jnngclaus, Gunter Saake, Thorsten Hartmann, and Cristina Sernadas. Object-Oriented Specification of Information Systems: The TROLL Language. Technical report, Technische Universität Braunschweig, December 1991. Report 91-04.

[63] Alan C. Kay. The Early History of Smalltalk. In *History of Programming Languages Conferenee (HOPL-II)*, 1993. *ACM SIGPLAN Notices*, 28(3):69-95, March.

[64] Setrag Khoshafian and Razunik Abnous. *Object Orientation: Concepts, Languages, Databases, User Interfaces.* John Wiley and Sons, 1990.

[65] Michael Kilian. Trellis: Turning Designs Into Programs. *Communications of the ACM*, 33(9):65–67, September 1990.

[66] Claude Kirchner, Hélène Kirchner, and José Meseguer. Operational Semantics of OBJ3. Technical report, CRIN, France, August 1988. Rapport CRIN 87-R-87.

[67] Jan Willem Klop. Term Rewriting Systems: From Church-Rosser To Knuth-Bendix and Beyond. Technical report, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1990. Report CS-R9013.

[68] Charles W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[69] Henry Lieberman. Using Prototypical Objects to Implement Shared Behaviour in Object-Oriented Systems. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, Portland, Oregon*, pages 11–22, 1986.

[70] Barbara Liskov. A History of CLU. In *History of Programming Languages Conference (HOPL-II)*, 1993. *ACM SIGPLAN Notices*, 28(3):133-147, March.

[71] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual.* Lecture Notes in Computer Science, Volume 114. Springer-Verlag, 1981.

[72] Barbara H. Liskov. A Design Methodology for Reliable Software Systems. In *Proceedings of the Fall Joint Computer Conference 41, Part 1*, 1972. Also in Peter Freeman and Anthony Wasserman, editors, *Tutorial on Software Design Techniques*, pages 53-61, IEEE Computer Society, 1977.

[73] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe. ANNA: *A Language for Annotating Ada Programs.* Springer-Verlag, 1987. Lecture Notes in Computer Science, Volume 260.

[74] B. J. MacLennan. Values and Objects in Programming Languages. *ACM SIGPLAN Notices*, 17(12):70–79, December 1982.

[75] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. Development of an Object-Oriented DBMS. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, Portland, Oregon*, 1986.

[76] José Meseguer. A Logical Theory of Concurrent Objects. In *Proceedings of the Joint OOPSLA/ECOOP Conference, Ottawa, Canada*, pages 101–115, 1990.

[77] Bertrand Meyer. *Object-oriented Software Construction.* Prentice-Hall, 1988.

[78] Bertrand Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

[79] Bertrand Meyer and Jean-Marc Nerson. Eiffel: The Libraries. Technical report, Interactive Software Engineering, October 1990. Report TR-EI-7/LI.

[80] David A. Moon. Object-Oriented Programming with Flavors. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, Portland, Oregon*, pages 1–8, 1986.

[81] Hanspeter Mössenböck. Object-Oriented Programming in Oberon-2. Technical report, Institut für Computersysteme, ETH Zürich, 1992.

[82] Hanspeter Mössenböck and Niklaus Wirth. The Programming Language Oberon-2. Technical report, Institut für Compntersysteme, ETH Zürich, March 1992.

[83] Greg Nelson, editor. *Systems Programming in Modula-3*. Prentice-Hall, 1991.

[84] Peter G. Neumann. The Role of Software Engineering. *Communications of the ACM*, 36(5):114, May 1993.

[85] Oscar M. Nierstrasz. A Survey of Object-Oriented Concepts. In Won Kim and Frederick Lochovski, editors, *Object-Oriented Concepts and Applications*. Addison-Wesley, 1988.

[86] Libero Nigro. On the Type Extensions of Oberon-2. *ACM SIGPLAN Notices*, 28(2):41–44, February 1993.

[87] Kristen Nygaard and Ole-Johan Dahl. The Development of the Simula Languages. In *History of Programming Languages Conference (HOPL-I)*, 1978. *ACM SIGPLAN Notices*, 13(8):245-272, August.

[88] Critical Research Directions in Programming Languages. Report on a workshop sponsored by the U.S. Office of Naval Research in Miami Beach, Florida. *ACM SIGPLAN Notices*, 24(11):10–25, November 1989.

[89] David L. Parnas. Information Distribution Aspects of Software Methodology. Technical report, Carnegie-Mellon University, February 1971.

[90] David L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):220–225, December 1972.

[91] David L. Parnas. A Technique for Software Module Specification with Examples. *Communications of the ACM*, 15(5):330–336, May 1972.

[92] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[93] Benjamin Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[94] Rubén Prieto-Díaz and Peter Freeman. Classifying Software for Reusability. *IEEE Software*, 4(1):6–16, January 1987.

[95] Lucia Rapanotti and Adolfo Socorro. Introducing FOOPS. Technical report, Oxford University Computing Laboratory, November 1992. PRG-TR-28-92.

[96] Horst Reichel. Behavioural Equivalence – A Unifying Concept for Initial and Final Specifications. In *Proceeedings, Third Hungarian Computer Seience Conference.* Akademiai Kiado, 1981. Budapest.

[97] Horst Reichel. Behavioural Validity of Conditional Equations in Abstract Data Types. In *Contributions to General Algebra 3.* Teubner, 1985. Proceedings of the Vienna Conference, June 21-24, 1984.

[98] J. P. Rosen. What Orientation Should Ada Objects Take? *Communications of the ACM*, 35(11):71–76, November 1992.

[99] Markku Sakkinen. The Darker Side of C++ Revisited. *Structured Programming*, 13:155–177, 1992.

[100] Arthur Sale. *Modula-2: Discipline and Design.* Addison-Wesley, 1986.

[101] Donald Sannella and Andrzej Tarlecki. Extended ML: Past, Present and Future. In H. Ehrig, K. P. Jantke, F. Orejas, and H. Reichel, editors, *Proceedings of the 7th International Workshop on Specification of Abstract Data Types, Wusterhausen/Dosse, Germany*, pages 297–322, 1990. Lecture Notes in Computer Science, Volume 534.

[102] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpot. An Introduction to Trellis/Owl. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, Portland, Oregon*, pages 9–16, 1986.

[103] Mary Shaw. Abstraction Techniques in Modern Programming Languages. *IEEE Software*, 1(4):10–26, October 1984.

[104] Vivek Singhal and Don Batory. P++: A Language for Large-Scale Reusable Software Components. Technical report, University of Texas at Austin, 1993 (to appear).

[105] Alan Snyder. CommonObjects: An Overview. *ACM SIGPLAN Notiees*, 21(10):19–28, 1986.

[106] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference. Portland, Oregon.* pages 38–45, 1986.

[107] Alan Snyder. Inheritance and the Development of Encapsulated Software Components. In *Research Directions in Object-Oriented Programming*, pages 165–188. M.I.T. Press, 1987.

[108] Adolfo Socorro. Pattern Matching for Objects in FOOPS. Programming Research Group, Oxford University Computing Laboratory, May 1991.

[109] Thomas A. Standish. An Essay on Software Reuse. *IEEE Transaetions on Software Engineering*, 10(5):494–497, September 1984.

[110] Mark Stefik and Daniel G. Bobrow. Object-Oriented Programming: Themes and Variations. *The AI Magazine*, 6(4):40–62, 1986.

[111] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, second edition, 1991.

[112] Bjarne Stroustrup. A History of C++: 1979-1991. In *History of Programming Languages Conference (HOPL-II)*, 1993. *ACM SIGPLAN Notices*, 28(3):271-298, March.

[113] Clemens A. Szyperski. Import is Not Inheritance: Why We Need both Modules and Classes. *Proceedings of the European Conference on Object-Oriented Programming*, pages 19–32, 1992.

[114] William Joseph Tracz. *Formal Specification of Parameterized Programs in LILEANNA.* PhD thesis, Stanford University, 1993 (to appear).

[115] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference, Orlando, Florida*, pages 227–242, 1987.

[116] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software.* Prentice-Hall, 1990.

[117] Nicklaus Wirth. From Modula to Oberon. *Software—Practice and Experience*, 18(7):661–670, July 1988.

[118] Niklaus Wirth. Modula: A Language for Modular Multi-programming. *Software—Practice and Experience*, 7(1):3–35, 1977.

[119] Stanley Zdonik. Can Objects Change Types? Can Type Objects Change? In *Proceedings of the Workshop on Database Programming Languages. Roscoff, Finistere, France*, 1987.