Models of Modularity:
A Study of Object-Oriented Programming

Phillip M. Yelland
Trinity College

A dissertation submitted for the degree of
Doctor of Philosophy in the University of Cambridge

March 1991

Abstract

This dissertation sets out an examination of mathematical models for object-oriented programming languages. It differs from prior investigations of the subject in the particular emphasis which is placed upon the derivation of *abstract denotational models* for "realistic" object-oriented languages. Abstract models express only sufficient of the internal details of objects to account for their externally observed behaviour; it is generally felt that such models are particularly suited to the study of the fundamentals of programming languages. Most of the models produced heretofore either fail to be abstract, or choose to eschew certain features of object-oriented languages in the interests of mathematical tractability.

A general mathematical framework is set out within which *modular* programming languages (of which object-oriented languages are particular examples) may be scrutinized. It is shown how certain qualities of such languages and their models may be expressed formally within this framework. It is also established that under certain specified conditions, techniques of universal algebra traditionally used in the treatment of abstract data-types may be employed to produce abstract denotational models for them.

The results of this theoretical development are applied to a sequence of four languages, each of which expresses a combination of features drawn from widely-used object-oriented languages. This exercise seems to indicate that some of the specification methods put forward for use with object-oriented languages are based upon unrealistic assumptions, and that certain features of these languages significantly complicate their semantics and (it may be reasonably assumed) undermine their efficacy.

After detailing a series of simple experiments designed to confirm that the linguistic features studied are indeed frequently employed in everyday programming, a number of suggestions are made concerning the future development of object-oriented programming languages and techniques.

Acknowlegements

I would like to acknowledge the contributions of many, many people (too many to mention here in total) to the production of this dissertation. My supervisor Alan Mycroft advanced clear and contructive criticism throughout the research. Members of the Department of Computer Science at Manchester University—amongst them Cliff Jones, Mario Wolczko, Trevor Hopkins, Michael Fisher, Ifor Williams and the rest of the mushroom-persons—gave generous help, encouragement and advice (and made available test harnesses used in the experiments of chapter 15). Stimulating and enlightening discussions were provided by Andrew Pitts, Tobias Nipkow and William Cook. The staff and management of Harlequin Ltd. and the Active Book Company Ltd. tolerated many of the eccentricities of a woolly-headed post-graduate. Tim Rowledge and Smalltalk Express Ltd. kindly furnished a Smalltalk interpreter meet for the investigations of chapter 15. The organizers of the ECOOP '89 and REX '90 conferences provided invaluable opportunities to present elements of this work at academic convocations. Friends, and my family especially, sustained me through many a dark night of the soul. And my wife, Chu Yuen-Ling, made it all worthwhile.

The work described herein was supported by a studentship from the Science and Engineering Research Council.

This dissertation is a result of my own work and is not the outcome of any work done in collaboration.

I hereby state that this dissertation is not substantially the same as any I have submitted for a degree, diploma or other qualification at any other university. Furthermore, no part of this dissertation has already been or is being concurrently submitted for any such degree, diploma or other qualification.

Contents

Chapter 1

Introduction

1.1 Deliverance or Chimera?

Until the early 1980's, "object-oriented programming" was largely the concern of academics, researchers and other such impractical folks. Few people disputed that object-oriented systems such as Smalltalk might significantly increase the productivity of programmers, but it was generally felt that they were intimidatingly esoteric and inherently inefficient [FS83]. The intervening decade has seen many changes.

Vast improvements in available computing resources have further exposed the embarrassing state of under-development of common programming practice [Man89]. The gains in programmer-productivity promised by the object-oriented approach have become increasingly desirable, while any lack of run-time performance has been ameliorated by the power of modern computing platforms. In addition, the development of "hybrid object-oriented languages" such as Objective-C [Cox86] and C++ [SE90] has established object-oriented programming in more familiar contexts and extended its appeal to a broader section of the programming populace.

Suddenly, advocates of object-oriented programming are ubiquitous. Introductions to object orientation abound in the "trade press" [Men90][Lew90], claiming that object-oriented programming will substantially reduce software backlogs, and usher in a new era of sophisticated applications, punctually delivered. Any new product worth its price is described as object-oriented; applications range from fractal modeling [SC89] to intelligent tutoring systems [BCS86].

Unfortunately, the embrace of object-oriented programming has been so enthusiastic that it seems to have been subject to very little in the way of dispassionate review. There is not even a precise and universally accepted definition of object-oriented programming—some products described as "object-oriented" seem to have only the most tenuous of claims on such a description. An invidious observer might suggest that in a few years, "object-oriented" will join those other great buzz-words of popular technology, such as "user-friendly," or "integrated". Maybe we will see object-oriented hair preparations in high-streets and shopping malls.

Tim Rensch summed this up in a prophetic article written in 1982 [Ren82]:[1]

> "My guess is that object-oriented programming will be in the 1980's what structured programming was in the 1970's. Everyone will be in favour of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip-service to it. Every programmer will practise it—differently. And no-one will know just what it is."

The work detailed in this dissertation is an attempt to address certain aspects of this problem.

1.2 The Study of Object-Oriented Programming

For our purposes, the keys to object-oriented programming are *object-oriented programming languages*—computer languages specifically designed to facilitate object-oriented programming. (We will often abbreviate the term "object-oriented language" to *OOL*.) Some researchers claim that object-oriented programming may be carried out in more conventional languages (the most popular in this regard is Ada [Boo86]), and might insist that studying object-oriented languages in particular pays attention only to one expression of the object-oriented style. We would counter that prior to the widespread use of OOL's, very little in the way of object-oriented programming was undertaken. In addition, most of the object-oriented programming carried out in non-object-oriented languages starts off by simulating their facilities.

Unfortunately, finding a truly synoptic collection of object-oriented languages is not by any means straightforward—a host of such languages have been designed, many with unique combinations of features. In the next chapter, we will attempt to draw out a set of salient features common to most OOL's. These are the features we study in later chapters.

The main vehicles for our study of OOL's are so-called *denotational* mathematical models.[2] This not the first time such techniques have been applied to object-oriented

---

[1] Via [Wol88].

[2] We will often use the words "model" and "semantics" as synonyms.

languages, as we shall see in chapter 7. The models presented in this dissertation, though, have certain characteristics which set them apart from those produced heretofore. For the most part, existing denotational models of object-oriented languages fall into one of two categories:

Models in the first category describe languages in everyday use, and constitute "implementations" of object-oriented languages in formal notations, such as VDM or the λ-calculus. While they are usually intuitive, and very useful in proving programs correct, they also tend to incorporate "implementation details", which are required for the correct functioning of the models, but are not strictly necessary to describe the languages. These extraneous details can make it difficult to derive fundamental insights into the basic qualities which make a language "object-oriented".

Models of the second kind rest on the appealing simile (illustrated in later chapters) which may be drawn between the programming concept of *data-abstractions* and the mathematical notion of *algebras*. Proponents of this sort of model argue that since algebras may be used as elegant and informative representations of data-abstractions, and since the prime concern of object-oriented languages is the constitution of data-abstractions (as we will see presently), *ergo* algebras also capture the semantics of object-oriented languages in a straightforward and informative way. Unfortunately, there have been few attempts to validate this assertion comprehensively, and it is often treated more-or-less as an article of faith.

The research described here draws on elements of both schools of semantics. Our intention is to produce models for object-oriented languages with features (most notably references and cyclic structures of objects) explained by the first school of model, but largely neglected by the second school. However, we insist that our models are *abstract*, in that they avoid the implementation details evidenced by the first type of model. We are able to fulfill this requirement by using the algebraic techniques of the second school. In doing so, we are also able to cast light on the applicability of algebraically-based specification methods to object-oriented programming languages. In addition, the exercise permits a sound assessment to be made of potential shortcomings of current object-oriented languages as regards disciplined software engineering.

Our first task is to delineate the languages we intend to study.

Chapter 2

Discovering Object-Oriented Programming Languages

Nowadays, object-oriented languages abound; one recent survey lists eighty-eight [Sau89]. These languages display a great diversity—C++ [SE90] and the Actor languages [Agh86], for example, appear to share little more than the title "object-oriented." In spite of this, we will attempt circumscribe a set of features shared by many, if not all, object-oriented languages. Our departure point in this exercise is Wegner's investigation in [Weg87]. Following the broad outline of his treatise, we will introduce several topics (or "dimensions", as Wegner would have it) of primary concern to the designers of object-oriented languages. In later chapters, we will fashion a series of languages which embody certain design decisions in these areas. Here, we describe briefly some of the directions in which designers of existing languages have proceeded, and our motives in selecting the particular interpretations we have chosen to study.

2.1 Elements of Object-Oriented Languages

2.1.1 Objects

The concept of the object is, of course, fundamental to object-oriented systems. Almost all such systems—from knowledge-representation languages such as KEE [KKW84] and programming languages like Object Forth [Pou86], to distributed operating systems such as MACH [Mar85]—embody identifiably similar ideas in this respect.

Basically, an *object* is a parcel of memory, usually constructed by delimiting an area of computer store. An object's *state* is determined by the configuration of its memory. In general, the state of an object may change over time. In many OOLs, including those studied here, the memory of an object comprises a collection of named variables, which we call *private variables*.

In general, the private variables of an object may have "primitive" values, such as integers and booleans, or they may contain other objects, or *references* (see next section) to objects.

2.1.2 Object-Identities and References

In most object-oriented languages, an object has an *identity*, as well as a state. The identity of an object distinguishes it from other objects; many objects with different identities may have the same state, but only one object may possess a particular identity. The identity of an object has often been neglected; many theoretical or philosophical descriptions of object-oriented programming acknowledge the existence of state in an object, but until recently, few have ascribed much significance to an its identity (Wegner's treatise, for example, makes no mention of identities).[1] This is unfortunate, since the use of identities may have a significant effect upon the semantics of an OOL.

This is especially true of languages with *references*. A reference may be thought of as an object identity which is shared by two or more objects. Shared references usually allow the state of an object to be inspected and/or changed by more than one other object, and this may have profound consequences for the way in which objects interact in a language. We will meet the implications of the use of references several times during the course of the our investigations.

2.1.3 Encapsulation

Object-oriented languages almost invariably associate *operations* with objects. These may be applied to objects to change or inspect certain aspects of its internal memory. Some OOL's, such as Simula [B+84], or CLOS [BDG+87], also allow the private variables of an object to be accessed or changed directly from outside it. In other languages, like Smalltalk [GR83] or Trellis/Owl [SCBK86], access to private variables is provided by operations alone. Objects in these latter languages are termed *encapsulated*.

All of the languages studied in the dissertation feature encapsulated objects.

[1] Notable early exceptions to this are [Wol88], [Sak89] and [AK90].

2.1.4 Object Creation

In order to make use of objects in a programming language, one must clearly have some means of introducing them. The most direct approach is simply to present each object using an individual declaration. Such a declaration might define the names and initial values of an object's private variables, and the operations which may be applied to it. This *static* style of presentation is reminiscent of the way in which systems are defined in so-called "modular" languages like Modula-2 [Wir82], or in abstract process notations such as CSP [Hoa85]. It is fairly uncommon in mainstream OOL's.

Much more common is the *dynamic* creation of objects. In most OOL's, dynamic object-creation essentially amounts to copying objects. In languages with *exemplars*, no distinction is made between the sorts of objects copied and those created by copying. *Class-based* languages, on the other hand, call objects of the former sort *classes*, and those of the latter sort *instances*. Classes may be thought of as template objects, with a fixed state and globally accessible names.[2] Usually, we speak of "creating an instance of a class," rather than of copying it. Note that the effect of applying an operation to an object depends (amongst other things) upon its class or exemplar.

In this dissertation, we will study OOL's with static object-declarations and with dynamic, class-based object-creation. Unless confusion would result, we will refer to definitions of both objects (in static languages) and classes (in dynamic ones) as "object-declarations".

2.1.5 Typing: Dynamic and Static

In common with other kinds of programming language, OOL's come in statically and dynamically typed varieties. Dynamically-typed languages attach types to objects themselves, rather than to the variables used to manipulate them—in implementation terms, this normally means tagging objects. Languages with dynamic typing determine types at run-time, when operations are actually applied to objects; since objects may be passed between variables, compile-time type-checking of such languages is extremely difficult. Examples of dynamically-typed OOL's include Smalltalk [GR83], CLOS [BDG+87] and Actor [Duf86]. Statically-typed languages label variables with types. In the main, types in these languages are checked at compile-time. Prominent examples are C++[SE90], Trellis/Owl [SCBK86] and Eiffel [Mey88].

[2] Admittedly, this is a fairly idiosyncratic interpretation of object-creation in class-based languages, and very few actually implement object creation as copying. Nonetheless, from the point of view of models in this dissertation, the notion is both adequate and useful.

In this dissertation, we concentrate exclusively on dynamically-typed languages. Statically-typed OOL's present rather more of a problem than dynamically-typed ones, in that creating a safe but flexible type system for OOL's is in itself a challenging task, and still the topic of much research (see, for example, [DT88] [CW85] [Coo89b] [AGOO86]); even so apparently finely wrought a type system as Eiffel's has been demonstrated to be seriously deficient in some respects [Coo89b].

2.1.6 Inheritance

Inheritance has long been associated with OOL's. Unfortunately, confusion has surrounded the subject, and only relatively recently has significant progress been made towards fully understanding it [Sak89] [Coo89a] [Hen90]. It has become increasingly apparent that inheritance as it figures in most OOL's actually combines two separate concepts: *sub-typing* and *code-sharing*.[3]

Sub-typing relates to the way in which objects interact with each other; briefly, a type A is a *sub-type* of type B if objects of type A may replace those of type B in all contexts in which they are used. We have omitted subtyping from our discussion for two reasons:

1. Sub-typing is of most interest in relation to statically-typed languages, where it may be used to increase the flexibility of type-systems. As we pointed out above, we have chosen not to discuss statically-typed languages.

2. To be truly effective, sub-typing requires some approximate notion of the *behaviour* objects in an OOL [Ame90a]. Since one major aim of our research is to establish correct definitions of behaviour for real OOL's, prior introduction of sub-typing would be premature.[4]

Code-sharing permits elements of an object's definition—such as the declarations of its private variables or operations—to be re-used when constructing other objects. Researchers have pointed out that the code-sharing facilities of many OOL's frequently contravene encapsulation, and have proposed more restricted facilities to rectify this [Sny87]. In fact, all OOL's provide a rather more prosaic means of re-using the definition of an

[3] Sub-typing and code-sharing correspond respectively to *essential* and *accidental* inheritance in the discussion of [Sak89]. Certain OOL's do achieve a partial or a complete separation of these concepts—C++ [SE90], for example, allows the specification of "private superclasses" allowing code-sharing without sub-typing, and in POOL-I [Ame90b] sub-typing and code-sharing facilities are quite distinct.

[4] In fact it should be possible to draw out a definition of subtyping from our discussion; this may be done by applying the development in [Ame90a] to the results in chapter 10. We will return to this topic in the conclusion.

object, simply by allowing one object to apply operations to another. As Sakkinen has recently indicated [Sak89], the modified code-sharing schemes mentioned above have come increasingly to resemble this sort of *object-composition*. We share Sakkinen's view that the logical conclusion of this trend is the abolition of code-sharing in favour of (possibly modified) composition facilities. Therefore, since object-composition figures in all of the languages studied here, a discussion of code-sharing is not included.[5]

2.1.7 Concurrency

Objects are often regarded as natural units of concurrency or pseudo-concurrency. This view was manifest in the design of Simula, and has underlain the design of many object-based operating systems, such as Mach [Mar85]. In a concurrent object-based system, an object which invokes an operation on another object does not generally wait until that operation has been completed before continuing with its own task. This contrasts with sequential systems, in which the execution of an operation causes the object which applied it to wait until completion. Many OOL's have been designed to support concurrency. These include the Actor languages [Agh86], POOL [AdBKR86], Beta [KMMPN85], Concurrent Smalltalk [YT86] and ABCL/1 [YBS86].

Concurrency does not figure amongst the features we examine, largely due to the complications which its incorporation would entail.

2.1.8 Other Features

In the above, we have covered only the most common and least esoteric features of OOL's. Omissions include multiple inheritance (as exhibited by Eiffel and CLOS, for example), metaclasses (as in Smalltalk, CLOS, ObjvList) and logical programming facilities (YaFool, Object Prolog).[6] Treatment of many of them presents no particular technical difficulty, but would unduly lengthen the dissertation—others pose great technical problems, and must remain the subject of future research. We return to such questions in our final chapter.

---

[5] A number of researchers—most notably [Coo89b], [Kam88] and [Hen90]—have pointed out interesting self-referential aspects of code-sharing in OOL's. As we point out in the conclusion, these develop the characteristics of cyclic object-structures examined in chapter 12.

[6] For a discussion of these features and the languages in which they occur, see [Sau89].

2.2 A Programming Example

To illustrate the preceding discussion, we will now see how some of the language features selected combine in a simple object-oriented programming exercise. We will use a fairly simple object-oriented programming language with encapsulated objects, class-based dynamic object creation and dynamic typing. Elements of this language will figure again in the language studies of chapters 10 through 13.

The object of this exercise is a simulation of a simple 3-bit binary counter which counts from 0 to 7, and then "rolls over" to 0 again. We begin with the definition of a basic component of such a counter, introduced by the class "Bit". Bits are objects representing simple two-state switches. They have a single private variable, "state". They are subject to operators "value", which returns the state of a Bit, "init", which forces a Bit into an initial state where its state is 0, and "tick", which flips its state between the values 0 and 1.

From the semantic point of view, it is convenient to separate operators into *functions*, such as value, which interrogate but do not affect the memory of an object, and *procedures*, such as init or tick, which change the contents of its memory but do not return a value. We might define Bits using the following class-definition:

class Bit **has variables** state

  functions value = state

  procedures init: state $\leftarrow$ 0

    tick: **if** state = 0 **then** state $\leftarrow$ 1 **else** state $\leftarrow$ 0

The definition supplies the name of the class, and the private variables and operations possessed by its instances. We have used simple imperative language to define the effects of procedures on objects. Note that the statement "state $\leftarrow$ 1" in the definition of tick assigns the integer 1 to the private variable of the object to which the operation is applied.

The simulation of binary counters is introduced by the class "Counter". The state of a Counter is composed of three Bits, held in private variables "bit_1", "bit_2" and "bit_3". We define a function "count" on Counters, which returns the present value of the count by summing the appropriately weighted values of the three Bits. To initialize a Counter, we apply the operation "init", which places an initialized Bit in each variable. (Note that the effect of applying init to an instance of class Counter accords with its definition in that class, and is not the same as applying init to a Bit.) A new instance of class Bit is denoted by the expression "**new** Bit"—the *state* of a new instance is undefined, and it must be properly initialized by application of the operation init. The "tick" operation advances the count by flipping bits in the Counter and performing the necessary carry propagation. We use the notation "var.op" to apply the operation "op" to the object in private variable "var". Nesting in conditional statements is indicated by indentation.

```
class Counter has variables bit₁, bit₂, bit₄
          functions value = bit₁.value+2*bit₂.value+ 4*bit₄.value
          procedures init: bit₁ ← new Bit; bit₁.init;
                           bit₂ ← new Bit; bit₂.init;
                           bit₄ ← new Bit; bit₄.init;
                     tick: bit₁.tick;
                           if bit₁.value = 0 then
                               bit₂.tick;
                               if bit₂.value = 0 then
                                   bit₄.tick
```

It is important to note that the simulation requires *both* classes be supplied; class Counter on its own is not complete. In fact, both classes together comprise a simple example of a *system*—a concept we explore in greater length later later on.

We have now outlined the aspects of object-oriented programming languages which we intend to cover. These features are all embodied in the languages defined in chapters 10 to 13. Before we attempt to formulate semantic models for such features, however, we must examine in detail a software design technique popularly associated with OOL's, viz. *modular programming*.

Chapter 3

Modular Programming and Object-Oriented Languages

3.1 Modular Programming

Modular programming is motivated by the view that a large and complex program constructed by connecting smaller simpler components is significantly easier to design, test and modify than one constructed as a single indivisible unit.[1] The components of such a composite program are commonly termed *modules*, and they are connected by means of *interfaces*.

Central to the efficacity of this approach to programming is the observation that in general, many details of a module's make-up are immaterial to the rôle which it plays in a program.[2] In most cases, for example, the particular algorithm (Quicksort, heapsort, bubble-sort, Shellsort, etc.) employed by a module which sorts integers is irrelevant to the correct functioning of a program in which it is used. By suitably adjusting the interface of the module, such details may be rendered invisible to its fellow components. This technique—usually known as *information hiding*—gives the designer latitude to change certain aspects of the module without concern for possible implications for other modules, provided that the consequences of those changes have no effects upon its interface characteristics. Thus, the designer of the sort module mentioned above would be free in the interests of efficiency to substitute a Shellsort algorithm for a bubble-sort, for example.

Information hiding also has another benefit. Removing extraneous details from a mod-

[1]Similar ideas have long been held in engineering circles—their origins have even been traced to Aristotle [Kap58] by some researchers.

[2]Some authors choose to distinguish between a *module* and its *implementation*. For them, a module is simply a "unit of functionality", which offers a given service to other modules. The implementation of such a module relates the particular means by which that service is provided. In the interests of simplicity, we regard a module and its implementation as synonymous.

ule's interface normally allows it to be described in simpler terms to programmers making use of it. The use of such simplified module descriptions can significantly reduce the burden on programmers implementing a modular system, since they are spared the necessity of examining in detail the modules they use. In our example, a sort module might be simply described "sorts an array of integers in place", which is likely to be much better received by its users than an account of the mechanics of the sorting procedure. The descriptions of modules in a modular program are termed *specifications*.[3]

In this chapter we will informally examine aspects of modules and their specifications. We will also look in a very general way at programming languages designed to facilitate modular programming, of which the OOL's are particular examples. First of all, we investigate the form of modular programming promoted by OOL's.

3.2 Data-Abstractions

The key to modular programming in object-oriented languages is the use of *data-abstractions*. A data-abstraction consists of a body of data and a collection of operations acting upon that data. The operations of a data-abstraction comprise its interface, and they provide the only means of access to its data—the data itself is excluded from the interface. This means that the specification of a data-abstraction need make no mention of actual data, concentrating instead on the operations in the abstraction's interface. It is plain that the objects described in the previous chapter are a form of data-abstraction, but the precise relationship between an object and the data-abstraction which it represents is not always straightforward, as we will demonstrate later.

First explored in the early 1970's [Par72], [Sha81],[LAB+81] [LZ74], data-abstractions have proved an increasingly popular basis for modular program construction. There is a vast literature on the theoretical aspects of data-abstractions, much of it concerned with producing specifications for them—see, for example [Gut75] [EFH83] [Jon86] [Spi88]. In chapter 5, we will illustrate some of the basic theoretical techniques used to specify data-abstractions.

Just as data-abstractions parallel objects in OOL's, so *abstract data-types* are analogues

[3]They are two main approaches to the use of specifications in modular programming: in the *top-down* approach—popularized in the "software methodologies" of the 1970's [YC79]—the specification of a module is fixed before the module itself is designed. The *bottom-up* approach—of great currency among advocates of re-usable software components [Mey88]—imposes a specification on an existing module. The terminology used here tends to favour the bottom-up approach (for example, strictly speaking, specifications in the top-down approach would be *pre*scriptions, rather than *de*scriptions as we have them here.) This is because it is usually given greater prominence in discussions of object-oriented programming. Nonetheless, most of the our conclusions apply equally to both programming strategies.

of classes. Like classes, abstract data-types are capable of generating an indefinite number of data-abstractions.[4] It is commonly the case that all of the data-abstractions produced by an abstract data-type have the same set of operations comprising their interfaces, but each will normally have its own unique body of data (and hence its own individual state). So as to avoid needless repetition, we will not normally distinguish between data-abstractions and abstract data-types, referring to both simply as "data-abstractions".

Time-honoured examples of data-abstractions are last-in-first-out stacks. We will use such stacks as exemplary data-abstractions in this and following chapters. The particular stacks used store natural numbers, and have four operations in their interfaces; "push", which adds a natural number to a stack, "pop", which removes the most recently-added natural number from a stack, and "top", which returns the most recently-added element of a stack.[5] We will assume that the operation "newStack" creates an empty stack.

3.3 OOL's: Modules and Systems

As we said above, OOL's may be thought of as modular programming languages facilitating the definition of data-abstractions. The modules of a program written using an OOL are the class- or object-declarations in it, and the data-abstractions in question are the classes or objects they define. It is important to note however, that in general, a module in an OOL does not by itself define a data-abstraction. This is because such a module may make use of other objects or classes defined in the program. We saw one example of this in the programming exercise at the end of chapter 2; the declaration of class Counter does not define a data-abstraction, since the bits of a Counter are defined by the separate class Bit.

One way of tackling this problem is to use the concept of the *parameterized data-abstraction* (or *parameterized abstract data-type*). A parameterized data-abstraction is simply a function which accepts data-abstractions (or even other parameterized data-abstractions) and produces new data-abstractions (or parameterized data-abstractions). Parameterized data-abstractions have long been subjects of study, especially by those investigating formal specifications [Gan83]. Unfortunately, as we will show in chapter 7, their application in this particular context is fraught with technical difficulties.

Instead, we will specialize our treatment of modular programming a little by con-

[4]Some writers use the term "abstract data-type" in a more specialized context to describe a module which creates *immutable* data-abstractions (i.e. abstractions whose data is not subject to change). Abstract data-types in our terminology may generate both immutable and mutable abstractions.

[5]Operation pop has no effect when applied to an empty stack, and value of top applied to an empty stack is undefined.

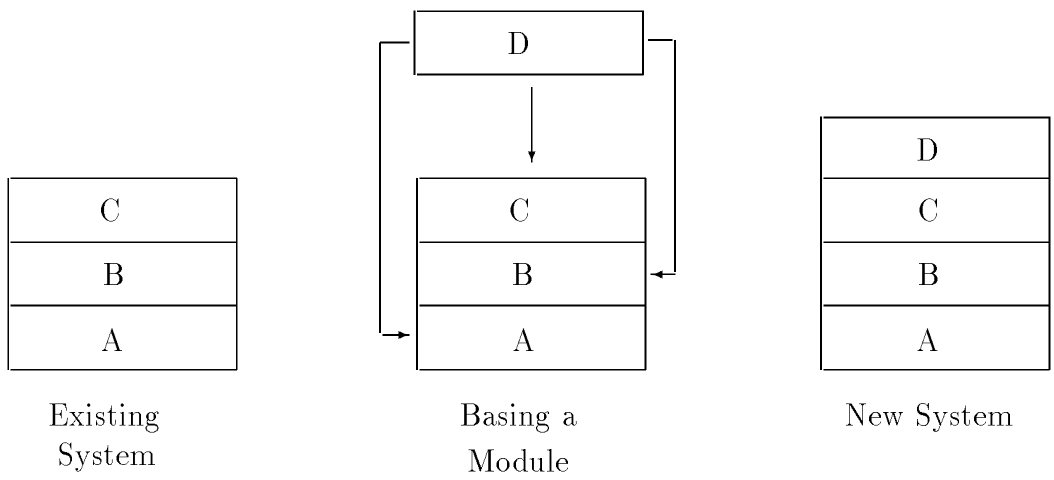

Figure 3.1: Building Systems from Modules

centrating on self-contained assemblies of modules, rather than on modules themselves. Such an assembly—which we term a *system*—should contain sufficient modules to define completely a data-abstraction or data-abstractions. Provided all of the data-abstractions required by a new module are to be found in a particular system, a new system may be produced simply by adding the module to it. In such an instance, we say that the new module is *based on* the system. Rather than thinking of a modular program simply as a set of connected modules, we will picture it as an evolving series of systems, produced successively by basing modules on the systems produced by other modules in the program. (See the figure for an illustration.) To get the whole process off the ground, we require a system which contains very basic data-abstractions, such as the natural numbers, or booleans.[6] We will assume that such a system is given *ab origine*. We may discern three systems arising in the programming exercise at the end of chapter 2: The first of these is a basic system, defining the integers and so on. Upon this is based the declaration of the class Bit, producing a second system defining class Bit. The third system results from basing the declaration of Counter on the second system, and defines two classes, Bit and Counter.

Note that the concepts of information hiding and specifications apply equally as well to systems as they do to the modules they contain. In particular, the specification of a system simply amounts to the specifications of each of the data-abstractions it defines.[7]

[6]For readers to whom the prospect of defining as data-abstractions such fundamental structures as the integers or booleans seems a little perverse in practice, we would point out Smalltalk-80 [GR83] gains great flexibility with just such a tactic.

[7]While we have used data-abstractions to motivate our definition of systems, we should point out that the notion is more generally applicable—taking in languages (such as Scheme [SS75]) which define

3.4 Properties of Systems and Specifications

3.4.1 Observable Behaviour

A pivotal quality of system is its *observable behaviour* (or simply *behaviour*). This describes precisely the way in which it interacts with modules based upon it. For example, a system containing a definition of stacks would relate the results of creating stacks and pushing and popping elements. In essence, the observable behaviour of a system summarizes the data-abstractions which it defines for the purposes of programmers basing modules upon it. We will encounter the notion of observable behaviour in a number of more formal guises later on.

A concept which follows directly from that of observable behaviour is *behavioural equivalence*; two systems are behaviourally equivalent when they have the same observable behaviour. This means that they may be interchanged without any effect on modules based upon them. As an example, consider two systems which define stacks. In one of these systems, a stack might use the familiar array-and-pointer implementation, in which an index into an array of elements is used to indicate the element currently "on top" of a stack. In the other system, a stack might be represented by a linked list of elements; elements are pushed by inserting them at the head of the list, and the stack is popped by removing the element at the head. It may be shown that provided access to stacks is restricted solely to their operations (and other abstractions in the systems are equal) both systems are behaviourally equivalent.

3.4.2 Correctness

The user of a system is guided in that use by the system's interface specification. For example, a module-designer making use of one of the systems in the previous section would use the system's specification to predict the effect manipulating stacks. A *correct* specification of a system accurately describes (at least) the observable behaviour of that system. By the definition of observable behaviour, a correct specification contains sufficient information to enable the results of interacting with that system to be predicted.

3.4.3 Abstractness

Note that the above definition does not prevent a correct specification from containing extraneous details of the kind discussed at the beginning of the chapter. In principle,

functions, for example, rather than data-abstractions.

providing a correct specification for a system is simple—one need merely publish the text of the system itself. This, of course, would largely eliminate the benefits of modular programming; such a wealth of information in a system's specification would unduly burden the user of a system and restrict the system's designer. It is preferable, as we pointed out, to remove unnecessary details from a system's specification. The ideal in this regard, as Parnas maintained, would be a specification which contained precisely enough information to describe accurately the observable behaviour of a system. Such specifications have long been of interest to researchers investigating modular programming—see [Jon86] for example.

An *abstract* specification is one which contains no more information than is required to describe its observable behaviour. By this definition, of course, some trivial specification ("this system may do anything") qualifies as abstract. Parnas's ideal specification is therefore one which is both *abstract* and *correct*.[8]

3.5 Stability and Modular Programming Languages

Almost invariably in the construction of modular programs, programmers believe that they can anticipate the way in which the systems they produce will interact with other modules. This is to say that they have some pre-conceived notion of their systems' observable behaviour. For example, the designer of class Counter would normally assume that all modules using Counters affected their state only by application of operations init and tick. He or she would not expect individual Bits of a Counter to be changed arbitrarily. As a consequence, he or she is confident that in all modules in which they are used as specified, Counters will simulate 3-bit binary counters.

In making such judgements, the programmer is actually making an assessment of the programming language in which modules are written. The assumption that Bits are not changed arbitrarily, for example, follows from the fact that objects in the programming language used are encapsulated. If such an assessment is correct, the programming language is *stable* with respect to the notion of observable behaviour used.

A precise definition of stability is as follows: a modular programming language is *stable* with respect to a given notion of observable behaviour if and only if any any two systems which are behaviourally equivalent (with respect to that notion of behaviour) may be interchanged without affecting the observable behaviour of any system produced by basing modules upon them. As a consequence, changes may be made to the modules

[8] Many writers, including Parnas, have seen fit to conflate the definitions of abstractness and correctness—we find it more convenient to keep them separate.

of a system in a stable programming language without changing any correct abstract specification for it, provided those changes do not affect its perceived behaviour.

Concepts related to stability have surfaced many times in research into specification languages for abstract data-types—c.f. [BG80] and [ST88]. Very thorough-going examinations of stability in programming languages are to be found in the theses of Nipkow [Nip87] and Schoett [Sch87].

There are two ways of bringing about stability in programming languages. The first approach is to take an existing idea of observable behaviour and design a programming language which is stable with respect to it. As might be expected, this approach underlies the design of many modular and object-oriented languages. Based as they are on data-abstractions, most of these languages try to restrict access to systems to declared sets of operations associated with the data-abstractions they define. The intention is to express the behaviour of systems in such languages entirely in terms of these declared operations. This tactic has resulted in the "interface specifications" of languages such as Modula-2, Ada and Smalltalk. Some designers [Sha81] [Mey88] have even endeavoured to formalize such notions using algebraic techniques similar to those described in following chapters. The OOL in chapter 2 is one example of this approach to stability.

The second approach entails taking an existing programming language and trying to formulate an notion of observable behaviour such that the language is stable with respect to that notion. This is how we will proceed in the remainder of this dissertation; we will distill a series of object-oriented programming languages from those in actual use, and in each case will attempt to devise a definition of observable behaviour with respect to which the language is stable.

In the next few chapters, we will begin to establish the formal framework for such investigations.

Chapter 4

Mathematical Preliminaries

This chapter summarizes (mainly for reference) the basic mathematical definitions and results used in later discussions. (Much of the material presented is standard, and the reader may wish to skip this chapter at first reading.)

4.1 Sets

The foundation for all of the mathematical definitions in this dissertation is an (informal) version of naïve set-theory. We assume the existence of a universe [Mac72], providing an approximation to a "set of all sets".[1] Words like "term", "expression", and "meaning" are used fairly informally, in line with their conventional usage in logic and set theory [Joh87]. (Later on, some are defined formally with regard to algebras.)

Below is an outline of the set-related notation and definitions we will use. The symbol "$\stackrel{\text{def}}{=}$" stands for "definitional equality", and means that the expression on the left of the symbol is to be interpreted identically to that on the right. The analogous symbol for predicates is "$\stackrel{\text{def}}{\Longleftrightarrow}$".

[1] For the reader to whom the use of universes is unfamiliar, it is summarized in appendix A.

$\emptyset$	$\stackrel{\text{def}}{=}$	$\{\}$	—the empty set
$\mathbf{N}$	$\stackrel{\text{def}}{=}$	$\{0, 1, 2, 3, \ldots\}$	—the natural numbers
$\mathbf{N}^+$	$\stackrel{\text{def}}{=}$	$\{1, 2, 3, \ldots\}$	—the strictly positive natural numbers
$\mathbf{B}$	$\stackrel{\text{def}}{=}$	$\{\mathbf{true}, \mathbf{false}\}$	—the boolean values
(a, b)	$\stackrel{\text{def}}{=}$	$\{\{a\}, \{a, b\}\}$	—ordered pairs
$A \times B$	$\stackrel{\text{def}}{=}$	$\{(a, b) \mid a \in A, b \in B\}$	—cartesian product
$A \cup B$	$\stackrel{\text{def}}{=}$	$\{e \mid e \in A or e \in B\}$	—set union
$A \setminus B$	$\stackrel{\text{def}}{=}$	$\{a \mid a \in A \wedge a \notin B\}$	—set difference
$A \subseteq B$	$\stackrel{\text{def}}{\iff}$	$a \in A \Rightarrow a \in B$	—set inclusion
$A = B$	$\stackrel{\text{def}}{\iff}$	$A \subseteq B \wedge B \subseteq A$	—set equality
$\mathcal{P}(A)$	$\stackrel{\text{def}}{=}$	$\{s \mid s \subseteq A\}$	—powerset construction

4.2 Relations

Definition 1 *A* relation *between sets A and B is a set $R \subseteq A \times B$. Write:*

$$a\,R\,b \stackrel{\text{def}}{\iff} (a, b) \in R$$

The domain *of R,* dom(R)*, is A, and its* codomain*,* cod(R)*, is B.*

Definition 2 *If R and S are two relations, the (diagrammatic)* composition *of R and S is defined:*

$$R; S \stackrel{\text{def}}{=} \{(a, c) \mid \exists b.\ (a, b) \in R \wedge (b, c) \in S\}$$

Definition 3 *The* restriction *of a relation R to a set A is the relation $R|_A$, defined:*

$$R|_A \stackrel{\text{def}}{=} \{(a, b) \in R \mid a \in A\}$$

4.3 Partial Functions

Definition 4 *A* partial function *f from set A to set B, (written $f : A \rightarrow B$), is a relation $f \subseteq A \times B$, such that for any $a \in A$, there is at most one $b \in B$, such that $(a, b) \in f$. That is:*

$$(a, b) \in f \wedge (a, c) \in f \Rightarrow b = c$$

The term $f(a)$ denotes the (unique) $b \in B$ such that $(a, b) \in f$. If no such member of B exists, then we say that $f(a)$ is undefined, *written $f(a) = \perp$. (Otherwise, $f(a)$ is* defined.*)*

In the interest of clarity, we will occasionally write fa instead of $f(a)$. In case of ambiguity, function application associates to the left, so that $fab \stackrel{\text{def}}{=} (f(a))(b)$.

Definition 5 *A partial function $f : A \to B$ is* total *iff for all $a \in A$, $f(a)$ is defined.*

We will always use the term "total function" explicitly—the word "function" by itself stands for "partial function." Note that all the definitions concerning relations also apply to functions. One particular consequence of these definitions is the *strictness* of partial functions; for functions $f : A \to B$, $g : B \to C$ and element $a \in A$, $f(a) = \perp \Rightarrow g(f(a)) = \perp$.

In order to deal with partial functions with the minimum of fuss, it is common to specify a particular form of the equality predicate. There are a number of suitable candidates, of which we have chosen one commonly known as *strong equality* [BW82]. It may be defined informally as follows:

Definition 6 *Let t_1 and t_2 be terms. Write $t_1 =_\perp t_2$ iff the meanings of t_1 and t_2 are both undefined, or if t_1 and t_2 both have defined meanings, and they denote the same value. In symbols:*

$$t_1 =_\perp t_2 \stackrel{\text{def}}{\iff} \begin{cases} t_1 = t_2 = \perp & or \\ t_1 = t_2 \neq \perp \end{cases}$$

We will frequently use the ordinary equality symbol "$=$", rather than "$=_\perp$", where the context allows.

Functions enjoy the following fundamental property, known as *extensionality*:

Fact 1 *If $f : A \to B$ and $g : A \to B$ are functions, and for all $a \in A$, $f(a) =_\perp g(a)$, then $f = g$.*

We will occasionally have recourse to a simple first-order substitutive notation for defining partial functions. Let x be a variable denoting some element of a set A, t a term, and B a set such that for all $a \in A$, the value $t[a/x]$, which results from substituting a for x throughout t, is either undefined, or is a member of B. Then the term $\lambda x : A.\, t : B$ denotes the function $f : A \to B$, such that for all $a \in A$, $f(a) =_\perp t[a/x]$. The variable x

is known as the *bound variable* in such a definition. Note that x must not occur as the bound variable of a function definition in t. The notation generalizes straightforwardly for more than one bound variable. We will use the abbreviated notation $\lambda x.\, t$, where domain and codomain are understood.[2]

The following notation is also useful:

Definition 7 *If $f : A \to B$ is a function and $a \in A$, $b \in B$ are members of A and B respectively, then $f[b/a]$ denotes the function such that for any $x \in A$:*

$$f[b/a](x) =_{\perp} \begin{cases} b & \text{if } x = a, \\ f(x) & \text{otherwise} \end{cases}$$

Definition 8 *The* wholly undefined function $\Omega : A \to B$ *is simply the empty set, $\emptyset$, regarded as a partial function from set A to set B. It follows that for all $a \in A$, $\Omega(a) = \perp$.* [3]

Definition 9 *Given sets A, B and an element $b \in B$, $\mathbf{const}(b)$ is the constant function:*

$$\mathbf{const}(b) \stackrel{\text{def}}{=} \lambda a{:}A.\, b{:}B$$

4.4 Families and Sequences

We will often have cause to collect sets into *families*:

Definition 10 *Let I and A be sets. An* I-indexed family of sets *(in A) is a total function $f : I \to \mathcal{P}(A)$. Write such a family $(A_i)_{i \in I}$, where $A_i = f(i)$, for $i \in I$.*

In most situations, we will give each A_i, for $i \in I$, and regard the set A implicitly as the union of all the A_i's. Note that if I is the empty set, then so is $(A_i)_{i \in I}$. The standard constructions on sets generalize straightforwardly to families. For example, if A is a family $(A_i)_{i \in I}$ of sets:

$$\bigcup A \stackrel{\text{def}}{=} \bigcup_{i \in I} A_i \stackrel{\text{def}}{=} \{x \mid \exists i.\, i \in I \ \wedge \ x \in A_i\}$$

[2] This notation is sometimes extended to facilitate the definition of functions whose domains are cartesian products. For example, "$\lambda(x, y).\, t$" denotes a function whose argument is a pair; the first element of this pair is accorded the name x, while the second component is named y.

[3] If we were to be strictly rigorous, we would decorate names like Ω so as to identify unambiguously their domains and co-domains. This would eliminate potential confusion, for example, between the undefined functions on unequal sets A and B. Rather than risk cluttering the exposition, however, we will normally rely on context to eliminate such problems.

Definition 11 *A* sequence *of length n over a given set A is a member of $\prod_{\{1,\ldots,n\}} A$. The empty set, regarded as the unique sequence of length 0 over A is written Λ_A, often abbreviated to Λ where the context allows. The expression $|s|$ denotes the length of sequence s. The set A^* is the set of all sequences over A, so that $A^* = \bigcup_{n \in \mathbf{N}} \prod_{\{1,\ldots,n\}} A$.*

We will sometimes refer to a sequence of length n as an "n-tuple".

Chapter 5

Data-Abstractions and Algebras

The mathematical constructions of *universal algebra* have had an influence on many investigations of data-abstractions. This application of universal algebra evolved not from programming practice, but from the algebraic machine models of automata theory [Hol82], later generalized by Eilenberg and Wright [EW67] and Give'on , Arbib and Manes [GA68a] [GA68b] [AM74]. These concepts were adapted to study data-abstractions by Guttag [Gut75], Zilles and Liskov [LZ74] [Zil73] and the ADJ group at IBM [GTW78]. In spite of their origins, algebraic tools have proven so well fitted to the discussion of data-abstractions in programming that they are used in almost all research in this area.

This chapter introduces the most basic concept in the field—the *heterogeneous* or *many-sorted algebra*—and looks briefly at the way in which it may used to model data-abstractions. In later chapters, we will see how algebras may be used to describe the objects and classes of OOL's.

5.1 Signatures and Algebras

Broadly speaking, a many-sorted algebra is simply a family of sets, together with a collection of functions on those sets. The sets of an algebra are referred to as its *carriers*, and the functions are known as *operations*. An algebra is defined syntactically by its *signature*, which contains a collection of *sorts* naming the carriers of the algebra, and a number of *operators*, which name its operations. The formal definition of a signature is set out below.[1]

Definition 12 *Assume a given set S of* sorts. *An S-*sorted set *is an S-indexed family $(A_s)_{s\in S}$.*

[1]By necessity, the definitions in this chapter are a little terse. Much fuller accounts may be found in [Gra69] or [BW82].

We will use the notation s_A to name a component of an S-sorted set A.

If w is a sequence (of sorts) $s_1 s_2 \ldots s_n$ over S, then $w_A \stackrel{\text{def}}{=} (s_1)_A \times (s_2)_A \times \ldots \times (s_n)_A$.

Definition 13 *An S-*sorted signature *is a pair (S, Σ), where S is a given set of sorts, and Σ is a $S^* \times S$-sorted family of sets $(\Sigma_{w,s})_{S^* \times S}$.*

The members of $\Sigma_{w,s}$, for $w \in S^*, s \in S$ are known as operators *of* arity *w, and* sort *s. Such an operator is often written $\sigma : w \to s$. Members of $\Sigma_{\Lambda,s}$ have empty arities, and are referred to as* constants. *The constant $\sigma \in \Sigma_{\Lambda,s}$ is written $\sigma : s$. If $\sigma : w \to s$ is an operator, define* $\mathrm{arity}(\sigma) = w$ *and* $\mathrm{sort}(\sigma) = s$.

By abuse of notation, a signature (S, Σ) is denoted by Σ; if this is the case, then $\mathrm{Sorts}(\Sigma) \stackrel{\text{def}}{=} S$, *and* $\mathrm{Ops}(\Sigma) \stackrel{\text{def}}{=} \bigcup_{S^* \times S} \Sigma_{w,s}$.

In this dissertation, we make use of *partial* many-sorted algebras, whose operations are partial functions. While less familiar than *total* algebras (whose operations must be total functions), partial algebras enable us to give a much more convenient account of errors in programs.

Definition 14 *Let Σ be an S-sorted signature. A* partial Σ-algebra, *A, consists of an S-sorted set $(A_s)_{s \in S}$, and for each operator $\sigma : w \to s$ in Σ, a partial function $\sigma_A : w_A \to s_A$.*

If A is a partial Σ-algebra, and Σ an S-sorted signature, define $\mathrm{Sig}(A) \stackrel{\text{def}}{=} \Sigma$, $\mathrm{Sorts}(A) \stackrel{\text{def}}{=} S$, *and* $\mathrm{Ops}(A) \stackrel{\text{def}}{=} \mathrm{Ops}(\Sigma)$.

The class of all partial Σ-algebras is denoted by $\mathbf{Alg}(\Sigma)$.

We will usually abbreviate "partial Σ-algebra" to "Σ-algebra."

5.2 Describing Data-Abstractions

In principle, describing a data-abstraction using algebraic methods is straightforward. Loosely speaking, one simply identifies the data of the abstraction with the elements of the carriers of an algebra, and the operations of the abstraction with those of the algebra. To illustrate more fully the way in which this is done, we will show how the stacks of natural numbers introduced in section 3.2 may be modelled.

5.2.1 Signatures

The first step in this process is to fix the signature of the stack model. Difficulty arises from the mutable nature of stacks. Given that the state of a stack may change (as it is created, pushed and popped), how might it be represented by the immutable mathematical entities which constitute an algebra? The established method of doing this is to be found in automata theory; instead of using one element of an algebra, we use a collection of them, each describing a particular state which the stack may adopt. The stack is represented by a member of this collection which describes its current state. A new member is substituted as the state of the stack changes. In the signature of the stack model, sort "StackState" indicates a carrier which contains representations of all the possible states of a stack. The signature includes one other sort, "Nat", which names the set of natural numbers, constituting the elements of stacks.

The first operators required in the signature of the stack model are the numerals 0, 1, 2, 3, and so on, used to name the elements of stacks. These are simply constants of sort Nat. The rest of the operators of the signature follow from the definition of stacks given in section 3.2. As you will recall, operation push pushes a natural number onto a stack, pop pops the top element off a stack, and top returns a stack's top element; newStack creates an empty stack. Since the first two operations may change the state of a stack, their analogues in the signature accept an element of sort StackState (representing the state of the stack before the operation) and return a StackState (which represents the state of the stack after the operation). The operator corresponding to top takes a StackState (the state of the stack interrogated) and returns an element of sort Nat (the stack's topmost element). Finally, the operator for newStack is actually a constant of sort StackState, which denotes the state of a stack when first it is created.

To summarize, the signature of the stack-model is given:

$$
\begin{array}{lll}
\Sigma_{\text{Stack}} = & \textbf{sorts} & \mathsf{Nat} \\
 & & \mathsf{StackState} \\
 & & \\
 & \textbf{operators} & 0, 1, 2, 3, \ldots : \mathsf{Nat} \\
 & & \mathsf{newStack} : \mathsf{StackState} \\
 & & \mathsf{push} : \mathsf{StackState} \times \mathsf{Nat} \to \mathsf{StackState} \\
 & & \mathsf{pop} : \mathsf{StackState} \to \mathsf{StackState} \\
 & & \mathsf{top} : \mathsf{StackState} \to \mathsf{Nat}
\end{array}
$$

5.2.2 Algebras

The signature Σ_{Stack} describes syntactically the operations of a stack abstraction. To complete the specification of the abstraction, we need to provide a particular Σ_{Stack}-algebra which interprets the sorts and operators of the signature appropriately.

For example, consider a stack which uses the array-and-pointer representation described in section 3.4.1. The state of such a stack could be specified by a pair, consisting of a natural number (the index of the stack) and a function from natural numbers to natural numbers (the array holding the elements of the stack). A set of such pairs would compose the carrier of sort $\mathsf{StackState}$ in the model. The obvious candidate for the carrier Nat is the set $\mathbf{N}$. The operations of the model are derived fairly straightforwardly from the representation of states. For example, the state which results from pushing a number n onto a stack whose state is $(index, array)$ may be described $(index + 1, array[n/index + 1])$—increasing the index by one, and associating the result with the element supplied. The full development is embodied in the algebra A_{Stack} defined below (assume that the function "max" returns the greater of its two arguments):

carriers

$$\mathsf{Nat}_{A_{\mathsf{Stack}}} = \mathbf{N}$$
$$\mathsf{StackState}_{A_{\mathsf{Stack}}} = \mathbf{N} \times (\mathbf{N} \to \mathbf{N})$$

operations

$$\mathsf{0} = 0,\ \mathsf{1}_{A_{\mathsf{Stack}}} = 1,\ \mathsf{2}_{A_{\mathsf{Stack}}} = 2,\ \mathsf{3}_{A_{\mathsf{Stack}}} = 3, \ldots$$
$$\mathsf{newStack}_{A_{\mathsf{Stack}}} = (0, \Omega)$$
$$\mathsf{push}_{A_{\mathsf{Stack}}} = \lambda(index, array), n.\ (index + 1, array[n/index + 1])$$
$$\mathsf{pop}_{A_{\mathsf{Stack}}} = \lambda(index, array).\ (\max(index - 1, 0), array)$$
$$\mathsf{top}_{A_{\mathsf{Stack}}} = \lambda(index, array).\ array(index)$$

5.2.3 An Algebraic Definition of Behaviour

Motivation

Though the algebra A_{Stack} given above may be used as a reasonable specification of one particular kind of stack abstraction (with array-and-pointer representation), it is by no means an *abstract* specification of stacks. It is hard to see, for example, how a stack whose representation is a linked-list could be related directly to the elements and operations of the algebra. Fortunately, there is a means of deriving mechanically from A_{Stack} an algebra which is a truly abstract specification of stacks. This derivation rests upon an algebraic formulation of observable behaviour.

We require a few more definitions:

Definition 15 *Given an S-sorted signature Σ, define the S-sorted set $(T_\Sigma)_S$ of Σ* (ground) terms *as follows:*

- $\Sigma_{\Lambda,s} \subseteq (T_\Sigma)_s$, *for all* $s \in S$.
- $t_1 \in (T_\Sigma)_{s_1}, \ldots, t_n \in (T_\Sigma)_{s_n} \Rightarrow \sigma(t_1, \ldots, t_n) \in (T_\Sigma)_s$,
  for $s_1, \ldots, s_n, s \in S$, $\sigma : s_1 \ldots s_n \to s \in \Sigma$.

Given a term $t \in T_\Sigma$, *its* interpretation, t_A, *with respect to a* Σ*-algebra* A *is defined inductively:*

- *If* $t = \sigma : s \in \Sigma$, *for* $s \in S$, *then* $t_A = \sigma_A$.
- *If* $t = \sigma(t_1, \ldots, t_n)$, *for* $\sigma : s_1 \ldots s_n \to s \in \Sigma$, $t_1 \in (T_\Sigma)_{s_1}, \ldots, t_n \in (T_\Sigma)_{s_n}$,
  then $t_A \stackrel{\text{def}}{=} \sigma_A((t_1)_A, \ldots, (t_n)_A)$.[2]

For a Σ*-algebra* A, *let* $\text{ev}_A : T_\Sigma \to A$ *be defined such that for all* $t \in T_\Sigma$, $\text{ev}_A(t) = t_A$.

Definition 16 *The* sort, $\text{sort}(t)$, *of a term* $t \in T_\Sigma$, *for* S*-sorted signature* Σ *is that* $s \in S$ *such that* $t \in (T_\Sigma)_s$.

The terms of an algebra's signature represent sequences of operations which it is possible to carry out in the algebra. If the algebra represents a data-abstraction, then these operations may be regarded as manipulations of that data-abstraction. In the stack example, for instance, pushing the numbers 1 and 2 onto a new stack might be represented by the term "$\mathsf{push}(\mathsf{push}(\mathsf{newStack}, 1), 2)$", and the topmost element of the stack so resulting would be given by "$\mathsf{top}(\mathsf{push}(\mathsf{push}(\mathsf{newStack}, 1), 2))$". The interpretation of a term with respect to an algebra representing a data-type gives the result of the sequence of operations which the term describes. Thus the interpretation of $\mathsf{push}(\mathsf{push}(\mathsf{newStack}, 1), 2)$ with respect to A_{Stack} is $(2, \Omega[1/1][2/2])$ and that of $\mathsf{top}(\mathsf{push}(\mathsf{push}(\mathsf{newStack}, 1), 2))$ is 2.

The motivation behind the algebraic definition of behaviour is the observation that the results of only certain terms over an algebra's signature are actually of interest to the user of the data-abstraction which the algebra represents. For example, the user of a stack need have no regard for the actual result of interpreting $\mathsf{push}(\mathsf{push}(\mathsf{newStack}, 1), 2)$, because it represents the state of a stack which—since stacks are data-abstractions—is hidden from the user; while it happens to be the pair $(2, \Omega[1/1][2/2])$ in the particular representation of stacks we have chosen, it could equally as well have been a description of a linked list, for example, had we chosen a behaviourally equivalent representation. On

[2]One consequence of the definitions given in chapter 4 is that $(\exists i \in \{1, \ldots, n\}.\ (t_i)_A = \perp) \Rightarrow t_A = \perp$.

the other hand, it is of interest that the result of $\mathsf{top}(\mathsf{push}(\mathsf{push}(\mathsf{newStack}, 1), 2))$ is 2, since one assumes that the user supplied the number 2 pushed onto the stack, and needs to be assured that immediately after the push it may be retrieved from the top of the stack.

A convenient way of distinguishing terms of interest is to divide the sorts of a signature into *visible* and *hidden* sorts. Loosely speaking, the visible sorts in the signature of an algebra representing a data-abstraction name the sets from which the inputs and outputs of the abstraction are drawn. In the stack example, the only visible sort is Nat, which names the natural numbers. The hidden sorts usually contain representations of the states of the abstraction. In our example, $\mathsf{StackState}$ is an hidden sort, since it contains the possible states of a stack. Dividing the sorts of a signature in this way also divides the terms over that signature according to their sort; terms of a visible sort are *visible terms*, while those of hidden sort are (naturally enough) *hidden terms*. In general, it is only visible terms—such as $\mathsf{top}(\mathsf{push}(\mathsf{push}(\mathsf{newStack}, 1), 2))$—which are interesting to users.

The observable behaviour of an algebra is simply determined by the results of interpreting a given set of visible terms over its signature. Early works on the subject (such as [GM82]) used the *entire* set of visible term's over an algebra's signature to determine its behaviour. In the case studies presented later on, we will find that in some situations the sequences of operations described by certain visible terms will never in practice be applied to an abstraction. Therefore, and in accordance with more recent research on the subject (such as [ST87]), we will allow behaviour to be fixed using a *specified subset* of the set of visible terms over a signature.

Persistent Extensions with No Inputs

In order to compare meaningfully the behaviour of two algebras, it is necessary that they have the same signature (with the same visible and hidden sorts). Additionally, because input and output will be interpreted by the users of the abstractions the algebras represent, both are required to have precisely the same interpretations of the visible sorts. (Σ_{Stack}-algebras whose behaviour is to be compared with that of A_{Stack}, for example, should all interpret sort Nat as the set of natural numbers, $\mathbf{N}$.) It is also convenient to have them agree on given operations whose domains and codomains are visible. Obvious candidates in the case of stack are the numeral constants $\mathsf{0}, \mathsf{1}, \mathsf{2}, \mathsf{3}$ and so on. The most commodious way of ensuring agreement on both sorts and operations is to collect those sorts and operations into a *subsignature* of the common signature of both algebras. One would then insist that the algebras to be compared be *persistent extensions* of a given algebra whose signature is that subsignature. The requisite formal definitions are given below:

Definition 17 *A signature Σ' is a* subsignature *of signature Σ, written $\Sigma' \subseteq \Sigma$, iff:*

- $\mathrm{Sorts}(\Sigma') \subseteq \mathrm{Sorts}(\Sigma)$
- $\Sigma'_{w,s} \subseteq \Sigma_{w,s}$, for $w \in \mathrm{Sorts}(\Sigma')^*, s \in \mathrm{Sorts}(\Sigma')$

Definition 18 *Let A be a Σ-algebra, and Σ', Σ two signatures such that $\Sigma' \subseteq \Sigma$. The* Σ'-reduct *of A, $A|_{\Sigma'}$, is the Σ'-algebra such that:*

- $s_{A|_{\Sigma'}} = s_A$, *for* $s \in \mathrm{Sorts}(\Sigma')$,
- $\sigma_{A|_{\Sigma'}} = \sigma_A$, *for* $\sigma \in \Sigma'_{w,s}, w \in \mathrm{Sorts}(\Sigma')^*, s \in \mathrm{Sorts}(\Sigma')$

Definition 19 *Let V be a Σ'-algebra, and $\Sigma' \subseteq \Sigma$. A Σ-algebra A is a* persistent extension *of V iff $A|_{\Sigma'} = V$.*

In the stack example, the most reasonable subsignature to select in the above regard is that describing the natural numbers:

$$\begin{aligned}\Sigma_{\mathrm{Nat}} = \ & \textbf{sorts} \ \ \mathsf{Nat} \\ & \textbf{operators} \ \ \mathsf{0} : \mathsf{Nat}, \mathsf{1} : \mathsf{Nat}, \mathsf{2} : \mathsf{Nat}, \ldots\end{aligned}$$

For which one would supply the obvious algebra:

$$\begin{aligned}A_{\mathrm{Nat}} = \ & \textbf{carriers} \ \ \mathsf{Nat}_{A_{\mathrm{Nat}}} = \mathbf{N} \\ & \textbf{operations} \ \ \mathsf{0}_{A_{\mathrm{Nat}}} = 0, \mathsf{1}_{A_{\mathrm{Nat}}} = 1, \mathsf{2}_{A_{\mathrm{Nat}}}, \ldots\end{aligned}$$

One would then only consider comparing the behaviour of Σ_{Stack}-algebras which are persistent extensions of A_{Nat}. (The example A_{Stack}, of course, is one such algebra.)

A few definitions are useful when discussing persistent extensions:

Definition 20 *For signature Σ and algebra V, such that $\mathrm{Sig}(V) \subseteq \Sigma$, define:*

- $\mathrm{VSorts}(\Sigma, V) \stackrel{\mathrm{def}}{=} \mathrm{Sorts}(V)$,
- $\mathrm{HSorts}(\Sigma, V) \stackrel{\mathrm{def}}{=} \mathrm{Sorts}(\Sigma) \setminus \mathrm{Sorts}(V)$,
- $\mathrm{VOps}(\Sigma, V) \stackrel{\mathrm{def}}{=} \mathrm{Ops}(V)$,
- $\mathrm{HOps}(\Sigma, V) \stackrel{\mathrm{def}}{=} \mathrm{Ops}(\Sigma) \setminus \mathrm{Ops}(V)$.

Sets $\mathrm{VSorts}(\Sigma, V)$ and $\mathrm{VOps}(\Sigma, V)$ are the *visible sorts* and *visible operations*, respectively, of Σ. Similarly, $\mathrm{HSorts}(\Sigma, V)$ and $\mathrm{HOps}(\Sigma, V)$ are the *hidden sorts* and *hidden operations* of Σ.

In this paper, we will actually deal only with a restricted form of persistent extension:

Definition 21 *A pair (Σ, V), such that $\mathrm{Sig}(V) \subseteq \Sigma$,* has no inputs *iff, for all $\sigma : w \rightarrow s \in \mathrm{HOps}(\Sigma, V)$, $w \in \mathrm{HSorts}(\Sigma, V)^*$ (i.e. there is no visible sort in the arity of a hidden operator). A Σ-algebra which is a persistent extension of V has no inputs iff (Σ, V) has no inputs.*

This restriction results in great technical advantage, as established by the following lemma:

Lemma 1 *If (Σ, V) has no inputs, then all operators in Σ have arities and sorts of the form:*

1) $\sigma : h_1, \ldots, h_n \rightarrow h$, for $h_1, \ldots, h_n, h \in \mathrm{HSorts}(\Sigma, V)$,

2) $\sigma : v_1, \ldots, v_n \rightarrow v$, for $v_1, \ldots, v_n, v \in \mathrm{VSorts}(\Sigma, V)$,

3) $\sigma : h_1, \ldots, h_n \rightarrow v$, for $h_1, \ldots, h_n \in \mathrm{HSorts}(\Sigma, V), v \in \mathrm{VSorts}(\Sigma, V)$.

We will collect together algebras of interest into classes as follows:

Definition 22 *For signature Σ and algebra V, such that $\mathrm{Sig}(V) \subseteq \Sigma$ and (Σ, V) has no inputs, $\mathbf{Alg}(\Sigma, V)$ is the class of Σ-algebras which are persistent extensions of V. The members of $\mathbf{Alg}(\Sigma, V)$ are known as* (Σ, V)-algebras.

Note that by definition, (Σ, V)-algebras have no inputs.

Algebraic Behaviour

We can now begin formalizing the algebraic definition of observable behaviour introduced at the beginning of the section. We start by defining visible terms:

Definition 23 *A* visible term *on a signature Σ is a term $v \in (T_\Sigma)_v$, for $v \in \mathrm{VSorts}(\Sigma, V)$. Let $T^{\mathrm{v}}_{\Sigma,V}$ be the set of all visible terms on Σ.*

As we said above, the behaviour of an algebra is determined by a set of visible terms on its signature. We can establish this formally by saying that two (Σ, V)-algebras have the *same* observable behaviour with respect to a set of visible terms when they produce the same results for all of those visible terms. It should be stressed that the observational equivalence of *algebras* introduced here is a relation quite different in general from the observational equivalence of *systems* which was introduced informally in chapter 3. To make clear this distinction, we will henceforth refer to the former relation as "a-behavioural equivalence" (*a*lgebraic behavioural equivalence) and the latter as "l-behavioural equivalence" (*l*inguistic behavioural equivalence).

Definition 24 *Let W be a set of visible terms, $W \subseteq T^{\mathrm{v}}_{\Sigma,V}$. Two (Σ, V)-algebras are* a-behaviourally equivalent *with respect to W, written $A \overset{\mathrm{abeh}}{=}_W B$, iff for any visible term, $t \in W$, $t_A =_\perp t_B$.*

In order to associate conveniently a particular notion of observation with an algebra, the following definitions are useful:

Definition 25 *For algebra V and $W \subseteq \mathcal{C}_{\Sigma,V}$, the set $\mathbf{Alg}(\Sigma, V, W)$ of* (Σ, V, W)-algebras *is defined as $\mathbf{Alg}(\Sigma, V) \times \{W\}$.*

(Σ, V, W)-algebra $B = (C, W)$ inherits all relevant properties of algebra $C \in \mathbf{Alg}(\Sigma, V)$, so that for example, $\mathrm{Sig}(B) = \mathrm{Sig}(C)$, for any $t \in T_{\mathrm{Sig}(C)}$, $t_B = t_C$, and so on.

(Σ, V, W)-algebras A and B are a-behaviourally equivalent*, written $A \overset{\mathrm{abeh}}{=} B$ iff $A \overset{\mathrm{abeh}}{=}_W B$.*

We round off with the following results, which will be of use later on:

Lemma 2 *For $A, B \in \mathbf{Alg}(\Sigma, V, W)$, $A \overset{\mathrm{beh}}{=} B \iff \mathrm{ev}_A|_W = \mathrm{ev}_B|_W$*

Lemma 3 *For signature Σ and algebra V, such that $\mathrm{Sig}(V) \subseteq \Sigma$ and (Σ, V) has no inputs, if $t \in T^{\mathrm{v}}_{\Sigma,V}$, then either $t \in T_{\mathrm{Sig}(V)}$, or t is of the form $\sigma(t_1, \ldots, t_n)$, for $\sigma \in \mathrm{HOps}(\Sigma, V)$, $\mathrm{sort}(\sigma) \in \mathrm{VSorts}(\Sigma, V)$, and $t_1, \ldots, t_n \in T_{\Sigma \setminus \mathrm{Sig}(V)}$.*

5.3 Term Realizations

At the start of the last section, we stated that there was a mechanical procedure by which a correct abstract specification may be derived from an algebraic specification which was not necessarily abstract itself. This is based upon the use of the *term realization*, defined as follows:

Definition 26 *For* (Σ, V, W)*-algebra* A*, define the* term realization of A, $\mathcal{I}(A)$*, as follows:*

$$
\begin{array}{lll}
\textbf{carriers} & h_{\mathcal{I}(A)} = h_{T_\Sigma}, & \text{for } h \in \mathrm{HSorts}(\Sigma, V), \\
 & v_{\mathcal{I}(A)} = v_V, & \text{for } v \in \mathrm{VSorts}(\Sigma, V), \\
\textbf{operations} & \sigma_{\mathcal{I}(A)} = \sigma_V, & \text{for } \sigma \in \mathrm{VOps}(\Sigma, V), \\
 & \sigma_{\mathcal{I}(A)} = \sigma_{T_\Sigma}, & \text{for } \sigma \in \mathrm{HOps}(\Sigma, V), \mathrm{Sort}(\sigma) \in \mathrm{HSorts}(\Sigma, V), \\
 & \sigma_{\mathcal{I}(A)} = \sigma_{T_\Sigma}; \mathrm{ev}_A|_W, & \text{for } \sigma \in \mathrm{HOps}(\Sigma, V), \mathrm{Sort}(\sigma) \in \mathrm{VSorts}(\Sigma, V).
\end{array}
$$

It is easy, by recourse to the definitions, to verify that $\mathcal{I}(A)$ is a (Σ, V)-algebra—that the above definition uniquely defines a (Σ, V, W)-algebra follows from lemma 1.

Intuitively, the term realization of an algebra is no more than a table of its observable behaviour, expressed itself as an algebra. Interpreting a term with respect to the term realization amounts to "looking up" the visible result (if there is one) of the operations it represents in that table. Since a hidden term has no visible result, looking up its value in the term realization simply yields the term itself. Looking up the value of a visible term, however, returns the visible value which would result if that term were interpreted with respect to the original algebra. The latter point is the intuitive justification for the following lemma:

Theorem 1 *For* (Σ, V, W)*-algebra* A*,* $\mathcal{I}(A) \stackrel{\text{abeh}}{=} A$*.*

Proof:

We need to show that, for all $t \in T^{\mathrm{v}}_{\Sigma,V}$, $t_{\mathcal{I}(A)} = t_A$. From lemma 3, we have two cases to consider:

If, in the first case, $t \in T_{\mathrm{Sig}(V)}$, then the required equality holds, since both algebras are persistent extensions of V.

For the second case, we have $\sigma(t_1, \ldots, t_n)$, for $\sigma \in \mathrm{HOps}(\Sigma, V)$, $\mathrm{sort}(\sigma) \in \mathrm{VSorts}(\Sigma, V)$, and $t_1, \ldots, t_n \in T_{\Sigma \setminus \mathrm{Sig}(V)}$. In this case, from the definitions:

$$
\begin{array}{rcl}
t_{\mathcal{I}(A)} & = & (\sigma(t_1, \ldots, t_n))_{\mathcal{I}(A)} \\
 & = & \sigma_{\mathcal{I}(A)}((t_1)_{\mathcal{I}(A)}, \ldots, (t_n)_{\mathcal{I}(A)}) \\
 & = & \mathrm{ev}_A(\sigma(t_1, \ldots, t_n)) \\
 & = & t_A
\end{array}
$$

□

The term realization of an algebra discloses no real information about the structure of that algebra's hidden carriers and operations, since its hidden carriers are merely sets of terms, and its hidden operations simply term-building functions. Hence it is not difficult to show that it is an algebra's behaviour alone which determines its term realization. Put another way, two algebras have the same observable behaviour precisely when they have the same term realization:

Theorem 2 *For (Σ, V, W)-algebras A and B, $A \stackrel{\text{beh}}{=} B \iff \mathcal{I}(A) = \mathcal{I}(B)$.*

Proof:

($\Rightarrow$) That $\mathcal{I}(A)$ and $\mathcal{I}(B)$ have equal carriers follows from the fact that they are determined in both cases by T_Σ and V alone. To show the equality of operators, consider the three cases of lemma 1. Cases 1) and 2) are established by observing that they are determined by T_Σ and V respectively. In case 3), we have that $\mathrm{ev}_A|_W = \mathrm{ev}_B|_W$, by lemma 2. Thus $\sigma_{T_\Sigma}; \mathrm{ev}_A|_W = \sigma_{T_\Sigma}; \mathrm{ev}_B|_W$, so that $\sigma_{\mathcal{I}(A)} = \sigma_{\mathcal{I}(B)}$.

($\Leftarrow$) We have:

$$\begin{array}{rcl}
\mathcal{I}(A) = \mathcal{I}(B) & \Rightarrow & \sigma_{\mathcal{I}(A)} = \sigma_{\mathcal{I}(B)} \qquad \text{for } \sigma : h_1, \ldots, h_n \to v, \\
& & h_1, \ldots, h_n, h \in \mathrm{HSorts}(\Sigma, V), v \in \mathrm{VSorts}(\Sigma, V) \\
& \Rightarrow & \sigma_{T_\Sigma}; \mathrm{ev}_A|_W = \sigma_{T_\Sigma}; \mathrm{ev}_B|_W \\
& \Rightarrow & \mathrm{ev}_A|_W = \mathrm{ev}_B|_W \\
& \Rightarrow & A \stackrel{\text{beh}}{=} B
\end{array}$$

$\square$

Thus the term realization is the correct abstract specification which we sought earlier.

5.4 Algebraic Techniques and OOL's

Algebras constitute eminently suitable models of data-abstractions. And as we pointed out in chapter 3, the objects and classes defined in OOL's may be construed as data-abstractions. One is thus compelled to the conclusion that algebraic methods may be usefully applied to the task of modelling OOL's. Indeed, this has become an increasingly popular theme of recent research into object-oriented programming. However, such an exercise should be undertaken with a good deal of circumspection.

The naive approach to the problem of specifying an object algebraically is simply to take the operations declared in the object's definition and form them into the operations of an algebra. Representations of the state of the object become hidden carriers of the algebra, and input/output values (such as integers, booleans and so on) comprise the algebra's visible carriers. The behaviour of the algebra is fixed by the set of visible terms over its signature. The suggestion is that the algebra so formed models the object in question; by constructing the algebra's term realization an abstract specification of the model may be produced. This is the approach which has been advocated by many writers on the subject of object-oriented programming [GM86].

Unfortunately, while it is a valid means of modelling some very simple objects, certain characteristics of almost all OOL's currently in use (springing in the main, as we shall see, from the presence of shared objects and references) mean that this way of modelling objects is flawed in general. To indicate why this is so, and to show how objects may be correctly modelled by algebras, the next chapters establish a framework for describing mathematically modular languages in general and object-oriented languages in particular.

Chapter 6

A Framework for Modular Languages

Having laid the requisite notational groundwork, we can now begin to express formally the discussion of modular programming and modular languages set out in chapter 3. The mathematical framework proposed here is used in later chapters in detailed studies of OOL's. Keeping this framework fairly general (and the mathematical concepts in it relatively weak), means that it adapts readily—as we will see—to a wide variety of object-oriented languages. Elements of this presentation are derived from research by Fisher [Fis87] and Back and Mannila [BM84].

6.1 Pre-Modular Languages and Systems

Fundamentally, a modular programming language is merely a notation in which modules are written. On a very abstract level, we can represent such a modular language simply by the set of all possible modules which may be written in it.

It is useful to impose a little structure on this set of modules. Recall that in chapter 3, we depicted a modular program evolving from a series of *systems*, or self-contained collections of modules. According to this picture, the programmer charged with designing a particular module is provided with a system upon which he or she *bases* that module. The result is a new system including the new module, and this passed on for use by other modules. It turns out to be technically very convenient to associate with any module an index—actually a strictly positive integer—which determines the number of modules which are required in any system upon which the module may be based. Thus a module with index i may only be based upon a system with $i-1$ modules in it. This divides the set of modules written in a language into a *family* of sets, indexed by the positive integers, whose i-th component contains those modules with index i.

We incorporate these ideas into the mathematical definition of a *pre-modular language*:

Definition 27 *A* pre-modular language *is an* $\mathbf{N}^+$*-indexed family* $(\mathrm{Mod}_i)_{i\in\mathbf{N}^+}$ *of sets of* modules.

Though the definition of a pre-modular language appears a little trivial, we will see how expressing an OOL as a pre-modular language gives a helpful degree of control over the the way in which its class- or object-declarations refer to each other. For example, by indexing the class-declarations of an OOL and forming from them the sets of modules of a pre-modular language, we are assured that all systems of class-definitions in the language must contain classes with indices $\{1, \ldots, n\}$, for some $n \in \mathbf{N}^+$ (this is established in detail below). If we stipulate that class-declarations may only refer to declarations with strictly lesser indices, cyclic class-references are prohibited, and systems are guaranteed to contain all the declarations referenced by their members. This facilitates the use of simple mathematical induction to establish properties of systems.[1]

The next definition describes formally how systems are constructed in pre-modular languages. Such systems must comply with the requirement outlined above that modules in component Mod_i may be based only upon systems containing $i-1$ objects. Observing this requirement is made easier by splitting the systems of a pre-modular language themselves into a family of sets, depending upon the number of modules they contain.

One possible representation of a system is simply a set of modules, drawn from the appropriate components of a pre-modular language. But such a representation would not clearly indicate the order in which modules were composed to construct the system—this is particularly true if the components of the modular language are not distinct. Instead, a system is represented as a *sequence* of modules which occur in the order they were used in constructing the system.[2] (For example, the representation of the system defining Bits and $\mathsf{Counters}$ in the exercise at the and of chapter 2 is simply a sequence containing the declarations of classes Bit and $\mathsf{Counter}$, in that order.) In order to satisfy the constraints set out for modules in a pre-modular language, the i-th component of a system must be drawn from the i-th component of the language in which it is defined.

To embody this idea mathematically, we need the following definition:

[1]We should point out that the use of pre-modular languages is not incompatible with the treatment of cyclic references in OOL's, as is shown in chapter 11.

[2]This form of representation, of course, accords with way in which class-declarations occur sequentially in the text of a program in some OOL's.

Definition 28 *If $(A_i)_{i\in\mathbf{N}^+}$ is an $\mathbf{N}^+$-indexed family and Λ some fixed element not in A, define an $\mathbf{N}$-indexed family $A^\#$, such that:*

$$\begin{array}{rcl} A_0^\# & \stackrel{\text{def}}{=} & \{\Lambda\} \\ A_n^\# & \stackrel{\text{def}}{=} & \prod_{i\in\{1,\ldots,n\}} A_i, \text{ for } n \in \mathbf{N} \end{array}$$

If $s = (a_1, \ldots, a_n) \in A_n^\#$, and $a_{n+1} \in A_{n+1}$, then by (s, a_{n+1}) we denote $(a_1, \ldots, a_n, a_{n+1}) \in A_{n+1}^\#$.

Now we can formally define a *system* in a pre-modular language Mod as an element of $\mathrm{Mod}^\#$. The construction of $\mathrm{Mod}^\#$ ensures that a system defined in this way satisfies the restrictions we gave above.

6.2 Modular Languages and Observations

Pre-modular languages provide a set of modules and guidelines for assembling them into systems. In chapter 3, we pointed out another aspect of modular programming languages, involving a notion of *observable behaviour* which relates the way in which systems in a language may interact with modules based upon them. Clearly, we some notion of observable behaviour in our mathematical framework. For the sake of succinctness, rather than keeping observable behaviour and modular languages separate, the formal definition of a modular language given below actually combines a pre-modular language with a particular conception of observable behaviour.

Observable behaviour in the mathematical definition of a modular language is determined by a set of *observations*. Each observation corresponds to an interaction between a system and a module based on it. Just as it was convenient to index the modules of a pre-modular language, so the observations of a modular language are divided into an $\mathbf{N}^+$-indexed family of sets, $(\mathrm{Obs}_i)_{i\in\mathbf{N}^+}$. The observations in Obs_i correspond to those made by modules in Mod_i and so may only be applied to systems in $\mathrm{Mod}_{i-1}^\#$.

The results of applying observations in a modular language are collected into an algebra, V. This algebra defines input and output values, such as the booleans, or natural numbers, as discussed in section 3.3. In order actually to apply the observations, a modular language has an $\mathbf{N}^+$-indexed family of *evaluation functions*, $(\mathrm{eval}_i)_{i\in\mathbf{N}^+}$. Each evaluation function eval_i applies an observation with index i to a system with $i-1$ modules and returns a visible value.

In summary, we have the following definition:

Definition 29 *A* modular language*, $\mathcal{L}$, is a 4-tuple* $((\mathrm{Mod}_i)_{i \in \mathbf{N}^+}, (\mathrm{Obs}_i)_{i \in \mathbf{N}^+}, V, (\mathrm{eval}_i)_{i \in \mathbf{N}^+})$*, where:*

1. $(\mathrm{Mod}_i)_{i \in \mathbf{N}^+}$ *is a pre-modular language,*
2. $(\mathrm{Obs}_i)_{i \in \mathbf{N}^+}$ *is an* $\mathbf{N}$*-indexed family of sets of* observations*,*
3. V *is an algebra of* visible values*,*
4. $(\mathrm{eval}_i)_{i \in \mathbf{N}^+}$ *is a family of* evaluation functions*, such that for all* $i \in \mathbf{N}$*,* $\mathrm{eval}_{i+1} : \mathrm{Obs}_{i+1} \times \mathrm{Mod}_i^{\#} \to V$.

6.3 Properties of Systems and Modular Languages

The last definition allows us to express mathematically some of the concepts associated with modular languages and systems in chapter 3. The first of these is behavioural equivalence.

6.3.1 Behavioural Equivalence

Recall that we want to keep distinct the algebraic notion of a-behavioural equivalence (discussed in chapter 5) and the linguistic notion—formally expressed here—of *l-behavioural equivalence* of systems in modular languages.

Since the behaviour of a system in a modular language is fixed by the set of observations which may be applied to it, two systems are l-behaviourally equivalent when the results of applying all possible observations to them are the same. Formally:

Definition 30 *Let* $\mathcal{L} = (\mathrm{Mod}, \mathrm{Obs}, V, \mathrm{eval})$ *be a modular language. For all* $i \in \mathbf{N}$*, systems* $s_1, s_2 \in \mathrm{Mod}_i^{\#}$ *are* l-behaviourally equivalent *wrt.* $\mathcal{L}$*, written* $s_1 \overset{\mathrm{lbeh}}{=}_{\mathcal{L}} s_2$*, iff:*

$$\forall o \in \mathrm{Obs}_{i+1}.\ \mathrm{eval}_{i+1}(o, s_1) = \mathrm{eval}_{i+1}(o, s_2)$$

We will often abbreviate $\overset{\mathrm{lbeh}}{=}_{\mathcal{L}}$ to $\overset{\mathrm{lbeh}}{=}$ where the context permits.

To summarize: *a*-behavioural equivalence is a relation between *algebras*, induced by sets of visible terms, whereas *l*-behavioural equivalence is a relation between *systems*, induced by sets of observations and evaluation functions.

6.3.2 Stability

Recall that informally, a modular language is *stable* with respect to a given notion of observable behaviour if and only if any two systems which are behaviourally equivalent (with respect to the given notion of behaviour) may be interchanged without affecting the observable behaviour of any system produced by basing modules upon them. (Note that the mathematical definition of a modular language includes a notion of behaviour, so henceforth we talk of a modular language simply as "stable", rather than stable with respect to a given notion of behaviour.)

The mathematical definition of stability may be transcribed fairly directly from this informal definition:

Definition 31 *A modular language* $\mathcal{L} = (\mathrm{Mod}, \mathrm{Obs}, V, \mathrm{eval})$ *is* stable *iff for all* $i \in \mathbf{N}$ *and* $s_1, s_2 \in \mathrm{Mod}_i^{\#}$*:*

$$s_1 \stackrel{\mathrm{lbeh}}{=} s_2 \;\Rightarrow\; \forall m \in \mathrm{Mod}_{i+1}.\, (s_1, m) \stackrel{\mathrm{lbeh}}{=} (s_2, m)$$

Now that we have a generalized, formal definition of modular languages, we can begin in the next chapter to examine their mathematical semantics.

Chapter 7

Denotational Models for Modular Languages

We are ready at last to begin investigating denotational semantics for object-oriented languages, as we undertook to do in chapter 1. These investigations are guided by the discussion of modular languages set out in the preceding chapters. In this chapter, we will introduce a blueprint for the denotational semantics of such modular languages which is used to shape the models for OOL's given in the case studies later on. We begin by examining the link between denotational semantics and specifications.

7.1 Denotations and Specifications

As its name suggests, a denotational semantics ascribes *denotations* to the constructs or phrases of a programming language. According to the precise definition of the word, the "denotation" of a language construct is a "[mathematical] term symbolizing or signifying" it [Irv72]. Construing this fairly liberally, we can regard the denotation of a construct as a sort of mathematical specification for it.[1] Now the central construct of modular languages is the system, and for our purposes, the main function of a denotational semantics for a modular language is to deliver the denotations of systems written in that language. Thus we may consider that a denotational semantics for a modular language provides mathematical specifications for its systems.

This view of denotational semantics is informative in two respects:

- It allows a denotational semantics to be used to investigate how the systems of a modular language may be formally described. We will return to this point in later chapters.

[1] Similar sentiments have been expressed on previous occasions—see [OH84], for example.

- It enables us to characterize denotational semantics in the light of the properties of specifications discussed in chapter 3. This is taken up in more detail below.

7.2 Semantics and Modular Languages

First, we need to describe formally a semantics which provides denotations for the systems of a modular language. Recall that a modular language is represented by a sequence $((\mathrm{Mod}_i)_{i\in\mathbf{N}^+}, (\mathrm{Obs}_i)_{i\in\mathbf{N}^+}, V, (\mathrm{eval}_i)_{i\in\mathbf{N}^+})$. The systems in such a language are drawn from the $\mathbf{N}$-indexed family $\mathrm{Mod}^{\#}$.

If a semantics is to provide denotations for systems, we require at least a set "DSys" containing those denotations, and a function S—usually termed a *semantic function*—which maps a system to its denotation. Since $\mathrm{Mod}^{\#}$ is $\mathbf{N}$-indexed, we will actually divide DSys into an $\mathbf{N}$-indexed family of sets, and S into an $\mathbf{N}$-indexed family of functions. Each set DSys_i contains the denotations of systems in $\mathrm{Mod}_i^{\#}$ (which contain i modules), ascribed by the semantic function S_i.

Thus we have:

Definition 32 *A* denotational semantics *for a modular language* $\mathcal{L} = ((\mathrm{Mod}_i)_{i\in\mathbf{N}^+}, (\mathrm{Obs}_i)_{i\in\mathbf{N}^+}, V, (\mathrm{eval}_i)_{i\in\mathbf{N}^+})$ *is a pair* $((\mathrm{DSys}_i)_{i\in\mathbf{N}}, (S_i)_{i\in\mathbf{N}})$, *where:*

- $(\mathrm{DSys}_i)_{i\in\mathbf{N}}$ *is a family of sets of* system denotations.
- $(S_i)_{i\in\mathbf{N}}$ *is a family of* semantic functions, *such that for all* $i \in \mathbf{N}$, $S_i : \mathrm{Mod}_i^{\#} \to \mathrm{DSys}_i$.

In keeping with standard (Strachey-style) notation, we will normally write $S_i[\![s]\!]$, rather than $S_i(s)$. Also, in order to avoid extraneous subscripts, we will omit indices on S (writing $S[\![s]\!]$, for example, rather than $S_i[\![s]\!]$) where convenient.

We will assume $\mathcal{L}$ is a fixed modular language $((\mathrm{Mod}_i)_{i\in\mathbf{N}^+}, (\mathrm{Obs}_i)_{i\in\mathbf{N}^+}, V, (\mathrm{eval}_i)_{i\in\mathbf{N}^+})$ throughout the remainder of the chapter.

Continuing the theme of denotations as specifications, there are two properties of denotational semantics which we can derive from the above definition.

7.2.1 Correctness

Recall that a correct specification for a system accurately describes that system's observable behaviour. A consequence is that two systems with different observable behaviour cannot have the same correct specification. If a denotational semantics were to produce denotations which were correct specifications of the systems they represented, then this implies that such a semantics would assign equal denotations to two systems only if they had the same observable behaviour. Therefore, such a denotational semantics is termed *correct*. The formal definition is as follows:

Definition 33 *A denotational semantics $\mathcal{S}$ for $\mathcal{L}$ is* correct *iff for all $i \in \mathbf{N}$:*

$$\forall s_1, s_2 \in \mathrm{Mod}_i^{\#}.\ S[\![s_1]\!] = S[\![s_2]\!] \ \Rightarrow\ s_1 \overset{\mathrm{lbeh}}{=}_{\mathcal{L}} s_2$$

7.2.2 Abstractness

Consider a denotational semantics for a language which assigns *equal* denotations to all systems in the language which are l-behaviourally equivalent. This means that no aspect of a system's denotation may depend upon factors not reflected in its observable behaviour. In other words, the denotation is an abstract specification of the system, following the definition in section 3.4.3. For this reason, such a semantics is itself known as *abstract*. Formally:

Definition 34 *A denotational semantics $\mathcal{S}$ for $\mathcal{L}$ is* abstract *iff for all $i \in \mathbf{N}$:*

$$\forall s_1, s_2 \in \mathrm{Mod}_i^{\#}.\ s_1 \overset{\mathrm{lbeh}}{=}_{\mathcal{L}} s_2 \ \Rightarrow\ S[\![s_1]\!] = S[\![s_2]\!]$$

Note that abstractness does not imply the correctness of a denotational semantics; a semantics which gave some fixed, trivial denotation to any system would be abstract. However, a corollary of the comments in section 3.4.3 is that a denotational semantics which is *both* correct and abstract delivers an "ideal" specification (in the sense of Parnas) for all the systems of a modular language.

7.3 Compositional Semantics

Let us return to the picture of the programmer basing a new module upon a supplied system, so as to produce a new system supplied to others. It is reasonable to expect that given a specification for the original system, and knowing the details of his or her own module, the programmer should be able to fashion a specification for the new system. Reading "denotation" for "specification", this implies that given the denotation of a system and a module, it should be possible to derive the denotation of the new system which results from basing that module on the original one. To do this, we require a new kind of model, known as a *compositional* denotational semantics. This is defined formally as follows:

Definition 35 *A* compositional *denotational semantics for $\mathcal{L}$ is a triple* $((\mathrm{DSys}_i)_{i\in\mathbf{N}}, (M_i)_{i\in\mathbf{N}^+}, \beta)$*, where:*

- $(\mathrm{DSys}_i)_{i\in\mathbf{N}}$ *is a family of sets of* system denotations.
- $(M_i)_{i\in\mathbf{N}}$ *is a family of* semantic functions *(for modules), such that for all* $i \in \mathbf{N}$*,* $M_i : \mathrm{Mod}_i \to \mathrm{DSys}_{i-1} \to \mathrm{DSys}_i$.
- $\beta \in \mathrm{DSys}_0$ *is the* base system denotation.

Again, we will omit the subscript on M where judicious.

A compositional semantics allows us to produce the denotation of the new system formed by basing a module on a system directly from the denotation of the old system itself; given that d is the denotation of the old system s, and the module m to be based on s, the denotation of the new system (s, m) is simply $M[\![m]\!]d$.

Note that its is possible to derive from a compositional semantics a denotational semantics in the sense of definition 32. For the compositional semantics of the above definition, this would be the model $((\mathrm{DSys}_i)_{i\in\mathbf{N}}, ({M^\#}_i)_{i\in\mathbf{N}})$, where $M^\#$ is constructed by component-wise application of M:

$$\begin{array}{rcl} M_0^\#[\![\Lambda]\!] & = & \beta, \\ \forall s \in \mathrm{Mod}_{i-1}^\#, m \in \mathrm{Mod}_i.\ M_i^\#[\![(s,m)]\!] & = & M_i[\![m]\!](M_{i-1}^\#[\![s]\!]) \end{array}$$

It is not difficult to demonstrate that $((\mathrm{DSys}_i)_{i\in\mathbf{N}}, ({M^\#}_i)_{i\in\mathbf{N}})$ satisfies all the requirements of definition 32. Note that as its name suggests, β is used in this semantics as the denotation of the base system (which contains no modules).

7.4 Proving Stability

Recall that in chapter 3 a modular language was described as *stable* precisely any two systems which were behaviourally equivalent could be interchanged without affecting the observable behaviour of any system produced by basing modules upon them. We formalized this in chapter 6 by saying that a modular language $\mathcal{L} = (\mathrm{Mod}, \mathrm{Obs}, V, \mathrm{eval})$ is stable if and only if for all $i \in \mathbf{N}$ and $s_1, s_2 \in \mathrm{Mod}_i^\#$, $s_1 \overset{\mathrm{lbeh}}{=} s_2 \Rightarrow \forall m \in \mathrm{Mod}_{i+1}.\ (s_1, m) \overset{\mathrm{lbeh}}{=} (s_2, m)$.

Given that we have related correctness, abstractness and compositionality to system specifications in a modular language, and have shown how the denotations of systems may be thought of as specifications, it is probably not surprising to learn that the presence or absence of these properties in denotational models of a language is strongly related to its stability.

Consider a language $\mathcal{L} = (\mathrm{Mod}, \mathrm{Obs}, V, \mathrm{eval})$ for which there is an abstract, compositional, correct denotational semantics $\mathcal{S} = ((\mathrm{DSys}_i)_{i\in\mathbf{N}}, (M_i)_{i\in\mathbf{N}^+}, \beta)$. Now it follows that for all $i \in \mathbf{N}$, $s_1, s_2 \in \mathrm{Mod}_i^{\#}$ and $m \in \mathrm{Mod}_{i+1}$:

$$\begin{array}{rlll} s_1 \overset{\mathrm{lbeh}}{=} s_2 & \Rightarrow & M^{\#}[\![s_1]\!] = M^{\#}[\![s_2]\!] & \text{(Abstractness of } \mathcal{S}\text{)} \\ & \Rightarrow & M^{\#}[\![m]\!](M^{\#}[\![s_1]\!]) = M^{\#}[\![m]\!](M^{\#}[\![s_2]\!]) & \\ & \Rightarrow & M^{\#}[\![(s_1, m)]\!] = M^{\#}[\![(s_2, m)]\!] & \text{(Compositionality of } \mathcal{S}\text{)} \\ & \Rightarrow & (s_1, m) \overset{\mathrm{lbeh}}{=} (s_2, m) & \text{(Correctness)} \end{array}$$

Therefore, $\mathcal{L}$ is stable. Thus:

Theorem 3 *A modular language with an abstract, compositional, correct denotational semantics is stable.*

7.5 Full Abstraction

Our main motivation in defining the properties of correctness and abstractness for denotational models was the investigation of specifications for modular languages (and object-oriented languages in particular). Similar concepts, however, have also been the concern of researchers devising denotational semantics for other sorts of programming languages.

In the main, investigations of such researchers start with the proposition that one's primary objective when using a programming language is the production of a *program*. A program is simply a kind of language construct or *term* which may be directly executed. The results of executing a program determines it *operational behaviour*; in a sequential language, the operational behaviour of a program might simply be the transformation it performs between its inputs and outputs, in concurrent languages, it might include a description of event-histories, and so on. In general, a program is constructed by assembling its constituent terms. Two such terms are *operationally equivalent* if they may be interchanged in any program in which they figure without changing that program's operational behaviour. (As an example, in most programming languages, the terms "x+1" and "1+x" are operationally equivalent.) Conventionally, a denotational semantics for a programming language assigns denotations to the terms of the language. Two terms are *semantically equivalent* with respect to a semantics if it gives them equal denotations. Now a denotational semantics is *fully abstract* when the semantic equivalence it defines on terms coincides precisely with the operational equivalence of those terms.

First ventilated by Milner in [Mil75] and Plotkin in [Plo77], fully abstract denotational semantics have proven an enduring subject of study. In the main, these studies may be divided into two classes:

- Examinations of the characteristics which a language must possess in order that it have fully abstract denotational models within a given mathematical framework (such as complete partial orders and continuous functions).

- Investigations of fully abstract models for particular programming languages.

Examples of the first kind of study are [Mil77], [Ber79] and [Mul87]; an excellent synopsis and development of such research is to be found in [Sto88]. Many of the positive results in this area are produced by means of *term models*, constructed by collecting the syntactic constructs of a language into operationally equivalent sets.

In the main, term models are not sought in the second class of investigation, on the grounds that they are not felt to be useful in reasoning about particular languages. Researchers in this area seek *natural* denotational models, constructed without using terms in this way. Most conventional denotational models of the Scott-Strachey kind are natural. However, many of the models constructed using conventional techniques are not fully abstract. This was first pointed out by Plotkin in [Plo77], where it was shown that the usual complete-partial-order-and-continuous-function model for a programming language based on the typed lambda-calculus is not fully abstract. Though Plotkin was able to show that subtly changing the subject language does make the model fully abstract, the problem of finding fully abstract semantics for the original language is still open—see [BCL85] for a survey of approaches. Indeed, finding natural fully abstract models for a variety of languages has proven an extremely perplexing exercise; problems arise when dealing with functions of higher type [BCL85], concurrency [HP79], and storage-allocation [HMT84].

By equating the notion of the operational equivalence of terms with that of behavioural equivalence of systems set out in the previous chapter,[2] strong similarities emerge between full abstraction and the concepts of abstractness and correctness given in the above. It might be said that the models given in the case studies later on constitute fully abstract semantics for certain object-oriented languages.

However, there are certain distinctions which should be bourne in mind. The first (and relatively minor) difference is that when viewed in this way, full abstraction actually

[2]This implies a notion of a "program" in an OOL as a system and a module based upon it; this view is taken up in greater length in [Yel90].

encompasses both correctness and abstractness: a fully abstract model attributes equal denotations to terms *if and only if* they are operationally equivalent, whereas correct models and abstract models only satisfy one direction of the bi-implication. The second difference (and the main reason we have avoided using the expression "fully abstract" to describe the models given here) is that the definition of full abstraction looks at the denotations given to *every* kind of language construct. By contrast, correctness and abstractness are characterized by the denotations of only one kind of construct, namely systems. We make no pretence, for example, that the models of the case studies give abstract denotations to assignment statements in the languages they describe. Indeed, in the next section, we illustrate the difficulties which would be associated in particular with giving abstract denotations to the object- or class-declarations of an object-oriented language.

7.6 Abstract Denotations for Modules

Recall that in our abstract framework, the object- or class-definitions of OOL's correspond to modules of modular programming languages. In order to determine the abstractness or otherwise of a denotational semantics with respect to modules, we need to define the l-behavioural equivalence of modules. The definition selected here says that two modules are behavourally equivalent precisely when based on behaviourally equivalent systems, they produce systems which are in their turn behaviourally equivalent. In symbols:

Definition 36 *Let* $\mathcal{L} = ((\mathrm{Mod}_i)_{i \in \mathbf{N}^+}, (\mathrm{Obs}_i)_{i \in \mathbf{N}^+}, V, (\mathrm{eval}_i)_{i \in \mathbf{N}^+})$ *be a modular language. For all* $i \in \mathbf{N}^+$, *modules* $m_1, m_2 \in \mathrm{Mod}_i$ *are* l-behaviourally equivalent *(written* $m_1 \overset{\mathrm{lbeh}}{=} m_2$*) iff for all* $s_1, s_2 \in \mathrm{Mod}^{\#}_{i-1}$*:*

$$s_1 \overset{\mathrm{lbeh}}{=} s_2 \;\Rightarrow\; (s_1, m_1) \overset{\mathrm{lbeh}}{=} (s_2, m_2)$$

We are concerned with *definitionally correct and abstract* semantics—compositional semantics for modular languages which assign equal denotations to modules if and only if they are l-behaviourally equivalent:

Definition 37 *A compositional denotational semantics* $\mathcal{S} = ((\mathrm{DSys}_i)_{i \in \mathbf{N}}, (M_i)_{i \in \mathbf{N}^+}, \beta)$ *for a modular language* $\mathcal{L}$ *is* definitionally correct and abstract *iff for all* $i \in \mathbf{N}^+$ *and* $m_1, m_2 \in \mathrm{Mod}_i$*:*

$$m_1 \overset{\mathrm{lbeh}}{=} m_2 \iff M[\![m_1]\!] = M[\![m_2]\!]$$

It turns out that a correct abstract semantics is definitionally abstract if the sets of system denotations, $(\mathrm{DSys}_i)_{i \in \mathbf{N}}$, are so arranged that they contain only the denotations of systems in the subject language. Such a semantics is termed *surjective*:

Definition 38 *A compositional denotational semantics* $\mathcal{S} = ((\mathrm{DSys}_i)_{i \in \mathbf{N}}, (M_i)_{i \in \mathbf{N}^+}, \beta)$ *for a modular language* $\mathcal{L}$ *is* surjective *iff for all* $i \in \mathbf{N}$*:*

$$\forall d \in \mathrm{DSys}_i.\ \exists s \in \mathrm{Mod}_i^{\#}.\ d = M^{\#}[\![s]\!]$$

So we have:

Theorem 4 *A compositional denotational semantics* $\mathcal{S} = ((\mathrm{DSys}_i)_{i \in \mathbf{N}}, (M_i)_{i \in \mathbf{N}^+}, \beta)$ *for a modular language* $\mathcal{L}$ *is definitionally correct and abstract iff it is correct, abstract and surjective.*

Proof:

Let i, s_1, s_2, m_1, m_2 be as in definition 36.

Assume the correctness, abstractness and surjectivity of $\mathcal{S}$, and furthermore that $m_1 \stackrel{\mathrm{lbeh}}{=} m_2$. Trivially, $\forall s \in \mathrm{Mod}_{i-1}^{\#}.\ s \stackrel{\mathrm{lbeh}}{=} s$, and so by abstractness $M[\![m_1]\!]M^{\#}[\![s]\!] = M[\![m_2]\!]M^{\#}[\![s]\!]$. By surjectivity, this implies that $\forall d \in \mathrm{DSys}_i.\ M[\![m_1]\!]d = M[\![m_2]\!]d$, and so by functional extensionality, $M[\![m_1]\!] = M[\![m_2]\!]$.

Next, assume abstractness and correctness, and that $M[\![m_1]\!] = M[\![m_2]\!]$.
Then $\forall d \in \mathrm{DSys}_i.\ M[\![m_1]\!]d = M[\![m_2]\!]d$. For $s_1 \stackrel{\mathrm{lbeh}}{=} s_2$, by abstractness, $M^{\#}[\![s_1]\!] = M^{\#}[\![s_2]\!] \Rightarrow M[\![m_1]\!]M^{\#}[\![s_1]\!] = M[\![m_2]\!]M^{\#}[\![s_2]\!] \Rightarrow M^{\#}[\![(s_1, m_1)]\!] = M^{\#}[\![(s_2, m_2)]\!] \Rightarrow (s_1, m_1) \stackrel{\mathrm{lbeh}}{=} (s_2, m_2)$, by correctness. Therefore $m_1 \stackrel{\mathrm{lbeh}}{=} m_2$.

Thus, if a semantics is correct, abstract and surjective, then $M[\![m_1]\!] = M[\![m_2]\!] \iff m_1 \stackrel{\mathrm{lbeh}}{=} m_2$, as required.
□

Unfortunately, as has been demonstrated by those attempting to formulate fully abstract models for languages with higher-order functions (see[BCL85]), achieving surjectivity in a semantics can be extremely difficult. For this reason, we have chosen to avoid tackling definitional abstractness in this dissertation, concentrating instead on the more tractable problem of abstractness.

7.7 Existing Denotational Models for OOL's

We promised in chapter 1 that the models for object-oriented languages presented in this dissertation differ from existing models of realistic OOL's in that they are (correct and) abstract. In this section, we will discuss the reasons why existing models are not abstract.

There is certainly no dearth of denotational semantics for OOL's. The most notable of these—and the ones which deal with languages most closely related to those studied here—were produced by Wolczko [Wol87] [Wol88], Kamin [Kam88], Cook [Coo89a] (and Cook and Palsberg [CP89]), Reddy [Red88] and North [Nor88]. All of these models are correct (though only [Coo89a] demonstrates this formally). The models of North, Reddy, Cook and Palsberg are also compositional; the non-compositionality of Kamin and Wolcko's models results largely from the treatment of inheritance and similar language features not examined in this dissertation—restricting them to the compass of our investigations renders them compositional as well.

However, the claims of Kamin, North and Reddy notwithstanding, none of these models are abstract in the sense defined above.[3] This is mainly a result of the way in which all of these models use some form of *global object memory*, as described in the following.

7.7.1 Global Object Memories

Global object memories are mathematical contrivances used to deal with references and shared objects in OOL's—two features highlighted in chapter 2. An object is shared when two or more references to it are present in the system of objects under consideration. More particularly, sharing occurs when two or more variables of objects in the system contain references to the same object. Shared objects are a form of *aliasing*—a phenomenon frequently exhibited by more traditional programming languages with pointers or reference parameters [Bro85]. Aliasing arises in general when some part of a computer's memory (the shared object in this case) is referred to by more than one name (the variables containing the references).

Aliasing is conventionally dealt with in denotational semantics by associating variables with their contents using two separate mathematical entities, commonly known as an *environment* and a *store*. The environment is represented by a function which maps variable names to *locations*, which are auxiliary values indicating specific areas of the computer memory. The store maps locations to the contents of memory and also indicates which locations are available for use in the environment. The aliasing of two variable names may be represented by binding them to the same location in the environment.

All of the denotational models for OOL's highlighted in the previous section use similar constructions to deal with shared objects. Here, locations are commonly termed *object-*

[3]A formal demonstration of the non-abstractness of a model based on Reddy's work may be found in [Yel89].

pointers, and the store—the *object-memory*—maps them to objects. By associating each object with a unique location, this location may be used as a representation of that object's identity. Two variables share the same object when they are both bound in the environment to that object's location. When a new object is created, it is assigned an unused location from the store, and that location is then marked as "in use." It is important to note that in order to deal properly with references and shared objects, the object memory in each of these existing models is manipulated directly by the representations of all objects (hence the term *global* object memory).

Although global object-memories are an intuitive means of tackling shared objects (being close to they way in which memory is actually managed in many implementations of OOL's [GR83]), their use normally results in the attribution of non-abstract denotations to objects and classes. To see why this is, recall the definition of class Bit in chapter 2, and consider the following two class-declarations:

class Cell **has variables** bit
    functions value = bit.value
    procedures init: bit ← **new** Bit
        tick: bit.tick

class WasteCell **has variables** bit, waste
    functions value = bit.value
    procedures init: bit ← **new** Bit;
            waste ← **new** Bit;
        tick: bit.tick

Informally, a Cell is merely a container for a Bit; its interface and observable behaviour is the same as the Bit it contains. A WasteCell is a cell with a extra variable ("waste"), which is assigned a new Bit when the object is initialized. It is not difficult to see that this extra variable is irrelevant to the user of a WasteCell (it is inaccessible through the operations in the object's interface), and thus that the classes Cell and WasteCell are behaviourally equivalent.

However, a model which uses a global object memory gives them unequal denotations. Imagine creating and initializing instances of Cell and WasteCell. When a Cell is created, it is assigned a new location from the object memory. Another location is used when the Cell is initialized, as the Bit in variable bit is created. A new WasteCell is also assigned a single new location, but when it is initialized, the creation of two new Bits to occupy variables bit and waste requires *two* new locations. Thus the object-memory which results from the creation of a Cell differs from that resulting from the creation of a WasteCell (in that the former has one more free location than the latter). Therefore, the denotations of Cell and WasteCell must be different in such a model, even though they are observably equivalent.

In the next chapter, we will outline a new approach to the semantics of modular languages. As is shown in the case studies, this approach may be used to develop models of OOL's featuring shared objects which, unlike the models described here, are both correct and abstract.

Chapter 8

Algebraic Models for OOL's

We have established that existing models of object-oriented languages are not abstract. Remember that in chapter 5, we pointed out that algebras may be used to produce abstract specifications of data-abstractions, but also indicated that the application of algebras to OOL's should be undertaken with great caution. This chapter sets out a mathematical framework for just such an exercise. We will also show that provided they are correctly applied, algebraic techniques may be used to produce correct, abstract denotational semantics for OOL's.

8.1 ΣW-Semantics

The most significant elements of modular languages embodied in the mathematical descriptions of chapter 6 were systems, and the observable behaviour of those systems. In this section we will introduce a kind of denotational model for modular languages which expresses both of these constituents using the algebraic ideas of chapter 5. We call such a model a *ΣW-semantics*.

Recall the mathematical definition of a compositional denotational semantics for a modular language $\mathcal{L} = ((\mathrm{Mod}_i)_{i\in\mathbf{N}^+}, (\mathrm{Obs}_i)_{i\in\mathbf{N}^+}, V, (\mathrm{eval}_i)_{i\in\mathbf{N}^+})$ as a triple $((\mathrm{DSys}_i)_{i\in\mathbf{N}}, (M_i)_{i\in\mathbf{N}^+}, \beta)$, where the family $(\mathrm{DSys}_i)_{i\in\mathbf{N}}$ consists of sets of *system denotations* representing the systems $s \in \mathrm{Mod}^{\#}$ of the language, $(M_i)_{i\in\mathbf{N}}$ is a family of *semantic functions* which map the modules of the language to functions on system denotations, $M_i : \mathrm{Mod}_i \to \mathrm{DSys}_{i-1} \to \mathrm{DSys}_i$, and $\beta \in \mathrm{DSys}_0$ is used as the denotation of the empty system $\Lambda \in \mathrm{Mod}_0^{\#}$.

In a *ΣW-semantics*, the system denotations are algebras which are persistent extensions of the algebra V of observation results in the language. We fix separate sets of algebras for each of the sets of systems $\mathrm{Mod}_i^{\#}$, $i \in \mathbf{N}$. These sets of algebras are described by an $\mathbf{N}$-indexed family of signatures, $(\Sigma_i)_{i\in\mathbf{N}}$. By arranging that the first of these signatures, Σ_0, is the signature of V, the definition of compositional semantics fixes the denotation of the empty system as V itself.

To account for the l-behaviour of systems, a ΣW-semantics also associates a definition of a-behaviour with each set of system denotations, by supplying an $\mathbf{N}$-indexed family of sets of terms $(W_i)_{i\in\mathbf{N}}$. Each set W_i contains visible terms on (Σ_i, V). Again, the stipulation that Σ_0 be the signature of V implies that $W_0 = T_{\mathrm{Sig}(V)}$. (In the next section, we will examine the conditions under which the *a*-behaviour of system *denotations* may be taken as a yardstick for the *l*-behaviour of *systems*.)

The formal definition of a ΣW-semantics is as follows:[1]

Definition 39 *A* ΣW-semantics *for a modular language* $\mathcal{L} = ((\mathrm{Mod}_i)_{i\in\mathbf{N}^+}, (\mathrm{Obs}_i)_{i\in\mathbf{N}^+}, V, (\mathrm{eval}_i)_{i\in\mathbf{N}^+})$ *is a compositional denotational semantics* $((\mathrm{DSys}_i)_{i\in\mathbf{N}}, (M_i)_{i\in\mathbf{N}^+}, \beta)$*, such that for all* $i \in \mathbf{N}$*,* $\mathrm{DSys}_i = \mathbf{Alg}(\Sigma_i, V, W_i)$*, where:*

- Σ_i *is a signature such that* $\mathrm{Sig}(V) \subseteq \Sigma_i$ *and* (Σ_i, V) *has no inputs.*
- $W_i \in T^{\mathrm{v}}_{\Sigma_i,V}$ *is a set of visible terms.*
- Σ_0 *is* $\mathrm{Sig}(V)$ *and* $W_0 = T_{\mathrm{Sig}(V)}$*.*

It follows from the definition of compositional semantics that for all $i \in \mathbf{N}^+$*,* $M_i : \mathrm{Mod}_i \to \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1}) \to \mathbf{Alg}(\Sigma_i, V, W_i)$ *and* $\beta \in \mathbf{Alg}(\mathrm{Sig}(V), V, T_{\mathrm{Sig}(V)}) = V$*.*

Note that such a ΣW-semantics is determined uniquely by the triple $((\Sigma_i)_{i\in\mathbf{N}^+}, (W_i)_{i\in\mathbf{N}^+}, (M_i)_{i\in\mathbf{N}^+})$, and so we will often refer to a ΣW-semantics such as that in the definition as "the ΣW-semantics $((\Sigma_i)_{i\in\mathbf{N}^+}, (W_i)_{i\in\mathbf{N}^+}, (M_i)_{i\in\mathbf{N}^+})$".

8.2 Congruence

We said that the motivation for deriving the algebraic behaviour of system denotations in a ΣW-semantics is to produce a reliable analogue of the linguistic behaviour of systems themselves. We indicated in previous chapters that the two notions of behaviour are generally quite different, and there is no guarantee that they can be equated in a ΣW-semantics. ΣW-semantics in which both notions *are* equivalent—that is, where two systems are l-behaviourally equivalent precisely when their denotations are a-behaviourally equivalent—are termed *congruent*. Congruence is defined formally as follows:

[1] For the definition of $\mathbf{Alg}(\Sigma, W, V)$, refer to chapter 5.

Definition 40 *A ΣW-semantics $((\Sigma_i)_{i\in\mathbf{N}^+}, (W_i)_{i\in\mathbf{N}^+}, (M_i)_{i\in\mathbf{N}^+})$ for a modular language $\mathcal{L} = ((\mathrm{Mod}_i)_{i\in\mathbf{N}^+}, (\mathrm{Obs}_i)_{i\in\mathbf{N}^+}, V, (\mathrm{eval}_i)_{i\in\mathbf{N}^+})$ is* congruent *iff for all $i \in \mathbf{N}$:*

$$\forall s_1, s_2 \in \mathrm{Mod}_i^{\#}.\ s_1 \overset{\mathrm{lbeh}}{=}_{\mathcal{L}} s_2 \iff M^{\#}[\![s_1]\!] \overset{\mathrm{abeh}}{=} M^{\#}[\![s_2]\!]$$

Congruence is of critical concern in producing ΣW-models. It implies that the model is correct according to definition 33:

Theorem 5 *A congruent ΣW-semantics $((\Sigma_i)_{i\in\mathbf{N}^+}, (W_i)_{i\in\mathbf{N}^+}, (M_i)_{i\in\mathbf{N}^+})$ for a modular language $\mathcal{L} = ((\mathrm{Mod}_i)_{i\in\mathbf{N}^+}, (\mathrm{Obs}_i)_{i\in\mathbf{N}^+}, V, (\mathrm{eval}_i)_{i\in\mathbf{N}^+})$ is also correct.*

Proof:

For $s_1, s_2 \in \mathrm{Mod}_i^{\#}$, $M^{\#}[\![s_1]\!] = M^{\#}[\![s_2]\!] \Rightarrow M^{\#}[\![s_1]\!] \overset{\mathrm{abeh}}{=} M^{\#}[\![s_2]\!] \Rightarrow s_1 \overset{\mathrm{lbeh}}{=} s_2$ by congruence.
$\square$

In addition, by applying the term realization construction described in chapter 5, a correct, abstract model for a modular language may be derived automatically from a congruent ΣW-semantics for it. This is demonstrated in the following section.

8.3 Abstractness and Stability

Recall from chapter 5 the definition of the term realization $\mathcal{I}(A)$ of a (Σ, V, W)-algebra A. Also recall that two such algebras have the same realization if and only if they have the same a-behaviour. Now consider the following construction. It produces from a given ΣW-semantics another ΣW-semantics in which the denotation of any system is the term realization of its original denotation.[2]

Definition 41 *Let $\mathcal{S} = ((\Sigma_i)_{i\in\mathbf{N}^+}, (W_i)_{i\in\mathbf{N}^+}, (M_i)_{i\in\mathbf{N}^+})$ be a ΣW-semantics for a modular language $\mathcal{L} = ((\mathrm{Mod}_i)_{i\in\mathbf{N}^+}, (\mathrm{Obs}_i)_{i\in\mathbf{N}^+}, V, (\mathrm{eval}_i)_{i\in\mathbf{N}^+})$. The* term realization *of $\mathcal{S}$ is the ΣW-semantics $\mathcal{I}(\mathcal{S}) = ((\Sigma_i)_{i\in\mathbf{N}^+}, (W_i)_{i\in\mathbf{N}^+}, ({M'}_i)_{i\in\mathbf{N}^+})$, where for all $i \in \mathbf{N}^+$, $m \in \mathrm{Mod}_i$, $s \in \mathrm{Mod}_{i-1}^{\#}$:*

$$M_i'[\![m]\!]s = \mathcal{I}(M_i[\![m]\!]s)$$

[2]To show that $\mathcal{I}(\mathcal{S})$ is well-defined, observe that for all $i \in \mathbf{N}^+$ and $A \in \mathbf{Alg}(\Sigma_i, V, W_i)$, $\mathcal{I}(A) \in \mathbf{Alg}(\Sigma_i, V, W_i)$, so $M'[\![m]\!] : \mathbf{Alg}(\Sigma_{i-1}, W_{i-1}, V) \to \mathbf{Alg}(\Sigma_i, V, W_i)$ as required.

The term realization of a ΣW-semantics forms the denotations of systems by repeatedly forming term realizations of the denotations of its components.[3] For example, consider the system $(m_1, m_2, \ldots, m_n)$, formed by concatenating modules m_1, m_2 to m_n. In the original semantics, the denotation of this system is:

$$M[\![m_n]\!](\ldots M[\![m_2]\!](M[\![m_1]\!]V)\ldots)$$

In the term realization of the semantics, its denotation is:

$$\mathcal{I}(M[\![m_n]\!](\ldots(\mathcal{I}(M[\![m_2]\!](\mathcal{I}(M[\![m_1]\!]V)))\ldots))$$

Note the outermost application of $\mathcal{I}$ in the above, and recall that it produces the same result for all a-behaviourally equivalent algebras. If the original semantics is congruent, then systems which are l-behaviourally equivalent will have a-behaviourally equivalent denotations. This means that in the term realization of the semantics, l-behaviourally equivalent systems will have *equal* denotations. Therefore, according to definition 34, the term realization of the semantics is *abstract*. And since the term realization shares the signatures, visible-term sets and algebra V of the original semantics, it too is congruent and hence correct. In summary:

Theorem 6 *If $\mathcal{S}$ is a congruent ΣW-semantics for a modular language, then the term realization, $\mathcal{I}(\mathcal{S})$, of $\mathcal{S}$ is a correct, abstract semantics for that language.*

Next, observe that in addition to being abstract, the term realization of a congruent ΣW-semantics is also compositional, by definition. Thus by theorem 3 of chapter 7, we also have the following:

Theorem 7 *A modular language with a congruent ΣW-semantics is stable.*

ΣW-semantics show how algebraic techniques—in the right circumstances—may be used to produce correct abstract semantics, and in doing so, to establish the stability of a modular language. In the next chapter, we will show how these results may be applied practically to object-oriented languages.

[3]This two-stage process, constructing a non-abstract representation of a system using the original semantics and then abstracting it by forming the term realization is reminiscent of the "constructor-abstractor" approach to algebraic specifications of abstract data-types described in [ST88].

Chapter 9

Prelude to the Case-Studies

In the following chapters, we will apply the theory developed in these last few chapters to four simple programming languages. These languages express features we stated in chapter 2 were found in commonly-used OOL's. This chapter serves as a "road-map" to the study of these languages.

The languages examined vary quite significantly: The first is a "static" language (in the parlance of chapter 2) where objects are declared explicitly. In this language, objects may be shared by means of references, but no cycles of reference may develop between objects. This means that no object may refer to itself, directly or indirectly. The second language develops on the first by adding (an attenuated form of) cycles. The third language is a "dynamic" language, defining objects by means of class-declarations. In this language, objects may not be shared. Finally, the fourth language incorporates both class-declarations and sharing by means of references.

Despite their differences, the languages studied are sufficiently alike, and our framework sufficiently general, that we can treat all of them in the same general fashion. We will sketch this broadly in the following.

9.1 Deriving Modular Languages

Our first task in dealing with each language is to fit it into the mathematical framework described in chapter 6. This entails deriving from each OOL a modular language such as was defined in section 6.2. We begin by constructing a pre-modular language.

9.1.1 Pre-Modular Languages

Making a pre-modular language (which, from section 6.1, you will recall is simply an $\mathbf{N}^+$-indexed family of sets of modules) from an OOL requires that its class- or object-declarations be separated into sets Mod_i, for $i \in \mathbf{N}^+$.[1] This is made much easier by a slight idiosyncrasy in the syntax of the languages studied: Rather than having every object-declaration identified by a programmer-supplied name (such as "Bit", or "Counter" in chapter 2), strictly positive natural numbers (which we call *indices*) are used instead. All modules with index i then comprise the set Mod_i required.

We are thus delivered a pre-modular language. From the definition in 6.1, it follows that systems in our case languages are sequences of object-definitions which occur in order of their indices, beginning with 1. To complete a modular language, we need to augment this pre-modular language with observations.

9.1.2 Observations

To decide how systems are to be observed in our case languages, we take recourse to some basic intuition about OOL's. Recall that in our framework, observations on a system are carried out by modules based upon it. Thus in the case languages, the class- or object-declarations in a system are observed by basing other class- or object-declarations upon them. Clearly, it would be contrary to the spirit of object-oriented programming to decree that the internal details (i.e. the private memory of objects or instances of classes) of such a system are open to observation. From the point of view of a programmer basing a new object-declaration on a system, the objects defined by that system are not amenable to direct examination of this kind.

What that programmer can determine, however, is the way in which the make-up of the system affects the execution of the code used to construct the new declaration. To see why this is, note that—as in the example language of chapter 2—there are two main constituents of object- or class-declarations in the case languages:

- Commands, used to change the state of newly-defined objects (and in static languages, of the objects in the system).
- Expressions, which compute values depending on the state of the new object (and, in the static case, of objects in the system).

[1] As before, we will normally refrain from distinguishing explicitly between class- and object-declarations, referring to both as "object-declarations".

In general, commands and expressions invoke interactions with the system upon which a new declaration is based. More precisely, both may apply operations to objects defined either directly (in the case of static languages) or indirectly (in dynamic languages) by the system. As an example, consider the programming exercise of chapter 2. (As you will recall, in this exercise, the declaration of the class Counter is based upon a system containing the declaration of class Bit.) Here, the effect of the command "bit_1.tick" (used to define the "tick" operation on Counters) depends intimately on the definition of tick attributed to Bits in the declaration of their class. Similarly, the value of the expression "bit_1.value+2*bit_2.value+4*bit_4.value" (the definition of "value" in Counter) depends upon the particular definition of operation value in Bit.

This remark forms the basis of the definition of observations in the case studies. According to our lights, an observation in the case languages begins by supposing the existence of a new object whose definition (either by direct declaration, or using a class-declaration) is based upon the system under examination. This object will normally start in some initial state, as dictated by its declaration (in the example language, in an object's initial state, all of its private variables had undefined contents.)[2] The observation itself is determined by selecting a command and an expression which might potentially occur in the new object's declaration. The command is executed in the object (which has been placed in its initial state), and this is followed by the evaluation of the expression. The result of applying the observation to the system is the result of this evaluation.

More formally therefore, if sets Cmd_i and Exp_i (for $i \in \mathbf{N}^+$) comprise all possible commands and expressions respectively which may occur in object-declarations with index i, then the set of observations Obs_i which may be applied to systems in $\mathrm{Mod}^{\#}_{i-1}$ is represented simply by the set of all pairs of commands and expressions, $\mathrm{Cmd}_i \times \mathrm{Exp}_i$.

It follows from this definition that in each language, the algebra V of observation results must contain all possible results of evaluating expressions in that language. In fact, we shall define expressions in each language so that these results are either boolean values or natural numbers. In the interests of technical convenience the algebra V in all four cases will be a trivial one containing only booleans and natural numbers (in a single carrier of sort "Val"), and possessing no operations. Adding operations to V would present no theoretical difficulties, but it would introduce unnecessary detail into the presentation of the studies.

[2]As the discussion later shows in more detail, when an observation starts in static languages, all the objects declared in the system must be in their initial states. Any instances of system classes referred to by the new object in dynamic languages must be similarly initialized.

9.1.3 Evaluation Functions

We have described how modules, systems, observations and observation results are to be derived for the case languages. To complete a modular language, we need also to supply a family of evaluation functions in each case. Here we must admit to a little artifice. You will recall from the definition of a modular language (definition 29), that each evaluation function eval_i of a modular language must apply an observation drawn from Obs_i to a system from $\mathrm{Mod}^{\#}_{i-1}$ to produce a result in V; in symbols, for all $i \in \mathbf{N}^+$, $\mathrm{eval}_i : \mathrm{Obs}_i \times \mathrm{Mod}^{\#}_{i-1} \to V$.

Now imagine a model of that language which draws the denotations of systems from the sets of the family $(\mathrm{DSys}_i)_{i\in\mathbf{N}}$. Let us further imagine that we have a set of *semantic evaluation functions*, which apply observations not to systems themselves, but to the *denotations* of systems; for all $i \in \mathbf{N}^+$, $\mathrm{eval}^{\mathcal{S}}_i : \mathrm{Obs}_i \times \mathrm{DSys}_{i-1} \to V$.

It is then possible to *define* the evaluation functions of the language using the semantic evaluation functions of the model; to apply an observation to a system, one simply applies that observation to the system's denotation. Thus if S is the semantic function deriving denotations for systems, for all $i \in \mathbf{N}^+$, we have:

$$\forall o \in \mathrm{Obs}_i, s \in \mathrm{Mod}^{\#}_{i-1}.\ \mathrm{eval}_i(o, s) \stackrel{\mathrm{def}}{=} \mathrm{eval}^{\mathcal{S}}_i(o, S[\![s]\!])$$

This is the tactic we will adopt in the case studies; we will use the denotational ΣW-semantics produced for each language to *define* the evaluation functions of that language. It is not difficult, of course, to show that this implies that each model is automatically correct. This begs the question of independently substantiating their correctness. In this dissertation, as in almost all the existing semantic studies of OOL's described in the previous chapter, we must rely on intuitive justification of correctness of the models produced. To do otherwise, we would need to provide an additional formal model of each language examined to define the evaluation functions, and establish the correctness of the ΣW-semantics according to definition 33. Such an exercise is illustrated in [Yel89]; to repeat similar procedures for each of the case studies would unduly lengthen our presentation. Semantic evaluation functions are described in more detail in the next section.

9.2 Producing Congruent ΣW-Semantics

Having derived a modular language from each of the OOL's in the case studies, our second task it to define a congruent ΣW-semantics. As we pointed out in the previous chapter, this entails supplying a family of signatures $(\Sigma_i)_{i\in\mathbf{N}^+}$, a family of sets $(W_i)_{i\in\mathbf{N}^+}$ of visible terms on those signatures, and a family $(M_i)_{i \in \mathbf{N}^+}$ of semantic functions producing algebras as the denotations of systems.

9.2.1 Signatures and Terms

Recall that the signature Σ_i of a ΣW-semantics is the signature of denotations of systems with i modules (with object-declarations indexed 1 to i in the languages of the case studies). According to the definitions, Σ_i must include the signature of the algebra of observation results, so that $\mathrm{Sig}(V) \subseteq \Sigma_i$. Since we have said that in the case studies, the signature simply names the sort Val, this merely means that Σ_i should contain the sort Val.

In addition to this, to produce a congruent semantics, Σ_i must also contain sufficient sorts and operations to describe all possible interactions which may occur between a system with i modules and a module based on such a system. As we will see, the precise requirements differ quite markedly from language to language; in each individual case study, we will try to point them out in some detail.

The sets of visible terms, $(W_i)_{i \in \mathbf{N}^+}$, are needed to fix the behaviour of system denotations in each model. Again, if congruent models are to be produced, the set W_i must contain *precisely* those terms necessary to describe possible interactions between a system with i modules and a new module based upon it. These sets also vary widely from language to language, as the case studies will bear out.

9.2.2 Denotations of Commands and Expressions

In the discussion above, we sketched the construction of observations as pairs of commands and expressions. We also showed how we could define the evaluation functions of a modular language using a ΣW-semantics. To see how this is done in more detail, we should examine how the state of objects and systems is to be represented.

Stores

In all the models of the case studies, we will use mathematical representations of *stores* (whose associated symbolism is discussed in greater detail below) to frame all the state which might be altered by a command executed—or interrogated by an expression evaluated—in a new object whose definition is based on a system. The store combines the state of private variables in the new object together with, in the case of the static languages studied, the state of all objects in the system accessible to the new object, and in the dynamic languages, the state of any instances of classes in the system which may be referred to by the new object. While stores bear some resemblance to the global object memories analyzed in the last chapter, the rôle they play in the models of the case-studies is quite different. Whereas global object memories are apparent in the final denotations of systems, the stores in our models figure explicitly only in intermediate representations; as we will see later, they are effectively concealed when system denotations are expressed as algebras.

In all the languages studied, the names of the private variables used in an object are not explicitly declared; each variable is simply quoted as required. This effectively means that all objects contain variables with the same (infinite) set of names. Immediately after the creation of an object, all of its variables have undefined contents (as in the example language of chapter 2).

The remarks of the previous paragraph imply that the representation of the store of a new object based upon a system depends only upon the system, and not on the particular object involved. Accordingly, in the models of the case studies, the store representation is a function of the system's denotation. More precisely, each model is supplied with a family $(\mathrm{Store}_i)_{i\in\mathbf{N}^+}$ of sets. Each Store_i (for all $i \in \mathbf{N}^+$) should contain representations of the states of the store of an object based on any system with i modules. In addition, we furnish a family of functions $(\mathrm{sto}_i)_{i\in\mathbf{N}^+}$, such that for all $i \in \mathbf{N}^+$, $\mathrm{sto}_i : \mathbf{Alg}(\Sigma_i, V, W_i) \to \mathcal{P}(\mathrm{Store}_i)$; sto_i produces from the denotation of a system with i modules the states of the store of an object based upon that system. Finally, we have a family of functions which establish the initial state of such a store, viz. $(\mathrm{init}_i)_{i\in\mathbf{N}^+}$, such that for all $i \in \mathbf{N}^+$, and $A \in \mathbf{Alg}(\Sigma_i, V, W_i)$, $\mathrm{init}_i(A) \in \mathrm{sto}_i(A)$. (We adhere to our customary practice and omit where possible the subscripts on sto, init and so on.)

In actual fact, it normally turns out more convenient—having defined sto_i independently—to fix Store_i implicitly, so that for all $i \in \mathbf{N}^+$:

$$\mathrm{Store}_i \stackrel{\mathrm{def}}{=} \bigcup_{A\in\mathbf{Alg}(\Sigma_i,V,W_i)} \mathrm{sto}_i(A)$$

Semantic Functions

We said that the application of an observation in the case languages amounted to the execution of a command, and the subsequent evaluation of an expression. In the case studies, these operations are accomplished by families of semantic functions $(C_i)_{i\in\mathbf{N}^+}$ and $(E_i)_{i\in\mathbf{N}^+}$ respectively. For $i \in \mathbf{N}^+$, functions C_i and E_i return the denotations of commands and expressions respectively which may occur in an object-declaration with index i. In general, command executions and expression evaluations are affected by the system upon which is based the object in which they are carried out (this, after all, was the motivation for our definition of observations). Thus the denotations of commands and expressions in the models are both parameterized by system denotations (drawn, for $i \in \mathbf{N}^+$, from $\mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1})$). In addition, the interpretation of a command must also take the state of a store (as described in above) and transform it into a new state, so as to reflect possible changes in the store. An expression's interpretation produces a value dependent upon the state of the store to which it applies.

Formally, for all $i \in \mathbf{N}^+$, we have:

$$\begin{array}{l} C_i : \mathrm{Cmd}_i \to \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1}) \to \mathrm{Store}_i \to \mathrm{Store}_i \\ E_i : \mathrm{Exp}_i \to \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1}) \to \mathrm{Store}_i \to V \end{array}$$

Recall from our discussions in section 9.1.3 that in each case study, the semantic evaluation function $\mathrm{eval}_i^{\mathcal{S}}$ (used to define the evaluation function eval_i of the modular language defined in the study) must apply an observation to a system denotation. This entails executing a command (from the set Cmd_i) in the initial state of the store (represented by $\mathrm{init}(A)$, where A is the system denotation) and subsequently evaluating an expression (from Exp_i). Following from the definitions of C_i and E_i above, we have, for all $i \in \mathbf{N}^+$:

$$\mathrm{eval}_i^{\mathcal{S}} = \lambda(c, e), A.\ E[\![e]\!]A\,(C[\![c]\!]A\,\mathrm{init}(A))$$

Thus by definition, for $s \in \mathrm{Mod}_i^{\#}$:

$$\mathrm{eval}_i = \lambda(c, e), s.\ E[\![e]\!](M^{\#}[\![s]\!])\,(C[\![c]\!](M^{\#}[\![s]\!])\,\mathrm{init}(A))$$

9.2.3 Denotations of Modules and Systems

To complete a ΣW-semantics, we must provide the family of semantic functions $(M_i)_{i \in \mathbf{N}^+}$ which derive the denotations of modules. (You will recall that for all $i \in \mathbf{N}^+$ $M_i : \mathrm{Mod}_i \to \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1}) \to \mathbf{Alg}(\Sigma_i, V, W_i)$.) The precise definitions of these functions in the case studies are quite diverse (in keeping with the variations in Σ_i and W_i we remarked above.) Nonetheless (since all object-declarations are expressed using commands and expressions), they invariably make use of private stores and the semantic functions for statements and expressions described above.

9.3 Proving Congruence

Our final task in each case study is to establish the congruence of the ΣW-semantics presented. Rather than proceeding directly from the definition of congruence given in section 8.2, we will use the following intermediate results:

Theorem 8 *Let S be a ΣW-semantics $((\Sigma_i)_{i\in\mathbf{N}^+}, (W_i)_{i\in\mathbf{N}^+}, (M_i)_{i\in\mathbf{N}^+})$ for a modular language $\mathcal{L} = ((\mathrm{Mod}_i)_{i\in\mathbf{N}^+}, (\mathrm{Obs}_i)_{i\in\mathbf{N}^+}, V, (\mathrm{eval}_i)_{i\in\mathbf{N}^+})$, associated with a family $(\mathrm{eval}^{\mathcal{S}}{}_i)_{i\in\mathbf{N}^+}$ of semantic evaluation functions used to define $(\mathrm{eval}_i)_{i\in\mathbf{N}^+}$ in accordance with section 9.1.3.*

If $\forall i \in \mathbf{N}^+, o \in \mathrm{Obs}_i, A \in \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1})$. $\mathrm{eval}^{\mathcal{S}}_i(o, A) = \mathrm{eval}^{\mathcal{S}}_i(o, \mathcal{I}(A))$, then $\forall i \in N, s_1, s_2 \in \mathrm{Mod}^{\#}_i$. $M^{\#}[\![s_1]\!] \overset{\mathrm{abeh}}{=} M^{\#}[\![s_2]\!] \;\Rightarrow\; s_1 \overset{\mathrm{lbeh}}{=} s_2$.

Proof:

Assuming the premise, we have $\forall i \in N, s_1, s_2 \in \mathrm{Mod}^{\#}_i$:

$$\begin{array}{rcl} M^{\#}[\![s_1]\!] \overset{\mathrm{abeh}}{=} M^{\#}[\![s_2]\!] & \Rightarrow & \mathcal{I}(M^{\#}[\![s_1]\!]) = \mathcal{I}(M^{\#}[\![s_2]\!]) \\ & \Rightarrow & \forall o \in \mathrm{Obs}_{i+1}.\ \mathrm{eval}^{\mathcal{S}}(o, M^{\#}[\![s_1]\!]) = \mathrm{eval}^{\mathcal{S}}(o, M^{\#}[\![s_2]\!]) \\ & \Rightarrow & s_1 \overset{\mathrm{lbeh}}{=} s_2 \end{array}$$

□

Theorem 9 *Let S be a ΣW-semantics $((\Sigma_i)_{i\in\mathbf{N}^+}, (W_i)_{i\in\mathbf{N}^+}, (M_i)_{i\in\mathbf{N}^+})$ for a modular language $\mathcal{L} = ((\mathrm{Mod}_i)_{i\in\mathbf{N}^+}, (\mathrm{Obs}_i)_{i\in\mathbf{N}^+}, V, (\mathrm{eval}_i)_{i\in\mathbf{N}^+})$, associated with a family $(\mathrm{eval}^{\mathcal{S}}{}_i)_{i\in\mathbf{N}^+}$ of semantic evaluation functions used to define $(\mathrm{eval}_i)_{i\in\mathbf{N}^+}$ in accordance with section 9.1.3.*

If for all $i \in \mathbf{N}^+$, there is a total *function $\psi : W_{i-1} \to \mathrm{Obs}_i$ such that $\forall A \in$ $\mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1})$,*
$t \in W_{i-1}$. $\mathrm{eval}^{\mathcal{S}}(\psi(t), A) = t_A$, then $\forall i \in N, s_1, s_2 \in \mathrm{Mod}^{\#}_i$. $s_1 \overset{\mathrm{lbeh}}{=} s_2 \;\Rightarrow\; M^{\#}[\![s_1]\!] \overset{\mathrm{abeh}}{=}$ $M^{\#}[\![s_2]\!]$.

Proof:

Assuming the premise, we have $\forall i \in N,\ t \in W_{i-1}.\ \exists o \in \mathrm{Obs}_i.\ \mathrm{eval}^{\mathcal{S}}(o, A) = t_A$. Then further, for all $s_1, s_2 \in \mathrm{Mod}^{\#}_i$:

$$\begin{array}{rcl} s_1 \overset{\mathrm{lbeh}}{=} s_1 & \Rightarrow & \forall o \in \mathrm{Obs}_{i+1}.\ \mathrm{eval}^{\mathcal{S}}(o, M^{\#}[\![s_1]\!]) = \mathrm{eval}(o, M^{\#}[\![s_2]\!]) \\ & \Rightarrow & \forall t \in W_i.\ t_{M^{\#}[\![s_1]\!]} = t_{M^{\#}[\![s_2]\!]} \\ & \Rightarrow & M^{\#}[\![s_1]\!] \overset{\mathrm{abeh}}{=} M^{\#}[\![s_2]\!] \end{array}$$

□

In light of the comments made in section 9.2.2 concerning the way in which observations are applied in the case studies, theorem 8 may be developed further. Let $\mathcal{S}$ be the ΣW-semantics defined in theorem 8, and $\mathrm{init}(A), C_i$ and E_i be defined for $\mathcal{S}$ in accordance with the discussion above. Recall that $\mathrm{sto}(A)$ represents the (states of) the store of an object based on a system with denotation A. Note that we can apply the function "sto" to the term realization of A, forming $\mathrm{sto}(\mathcal{I}(A))$. Now we have the following:

Theorem 10 *If, for all* $A \in \mathbf{Alg}(\Sigma_i, V, W_i)$*, there exists:*

a) *A set* $\mathrm{Feas} \subseteq \mathrm{sto}(\mathcal{I}(A))$ *such that for all* $c \in \mathrm{Cmd}_i$, $C[\![c]\!]\mathcal{I}(A)\,\mathrm{init}(\mathcal{I}(A)) \in \mathrm{Feas}$, *and*

b) *A function* $\phi : \mathrm{sto}(\mathcal{I}(A)) \to \mathrm{sto}(A)$*, such that:*

  i) $\mathbf{const}(\mathrm{init}(A)) = \mathbf{const}(\mathrm{init}(\mathcal{I}(A))); \phi$,

  ii) $\phi|_{\mathrm{Feas}}; (E[\![e]\!]A) = E[\![e]\!]\mathcal{I}(A)$ for $e \in \mathrm{Exp}_i$,

  iii) $\phi; (C[\![c]\!]A) = (C[\![c]\!]\mathcal{I}(A)); \phi$ for $c \in \mathrm{Cmd}_i$.

Then for all $(c, e) \in \mathrm{Obs}_{i+1}$, $\mathrm{eval}^{\mathcal{S}}((c, e), A) = \mathrm{eval}^{\mathcal{S}}((c, e), \mathcal{I}(A))$.

Proof:

From condition a) of the premise, we know that:

$$\mathbf{const}(\mathrm{init}(\mathcal{I}(A))); C[\![c]\!]\mathcal{I}(A); \phi = \mathbf{const}(\mathrm{init}(\mathcal{I}(A))); C[\![c]\!]\mathcal{I}(A); \phi|_{\mathrm{Feas}}$$

By definition,

$$\begin{aligned}
\mathrm{eval}^{\mathcal{S}}((c, e), A) &= \mathrm{init}(A); C[\![c]\!]A; E[\![e]\!]A \\
&= \mathrm{init}(\mathcal{I}(A)); \phi; C[\![c]\!]A; E[\![e]\!]A \\
&= \mathrm{init}(\mathcal{I}(A)); C[\![c]\!]\mathcal{I}(A); \phi; E[\![e]\!]A \\
&= \mathrm{init}(\mathcal{I}(A)); C[\![c]\!]\mathcal{I}(A); \phi|_{\mathrm{Feas}}; E[\![e]\!]A \\
&= \mathrm{init}(\mathcal{I}(A)); C[\![c]\!]\mathcal{I}(A); E[\![e]\!]\mathcal{I}(A) \\
&= \mathrm{eval}((c, e), \mathcal{I}(A)).
\end{aligned}$$

□

To summarize, therefore, proving congruence in each case-study will amount to establishing the following:

Corollary 1 *Let $\mathcal{S}$ be a ΣW-semantics as defined in 8, and* $\mathrm{init}(A), C_i$ *and* E_i *be defined for* $\mathcal{S}$ *in accordance with the discussion above. Then* $\mathcal{S}$ *is congruent if the following may be shown for all* $i \in \mathbf{N}^+$ *and* $A \in \mathbf{Alg}(\Sigma_i, V, W_i)$*:*

1. *There is a set* $\mathrm{Feas} \subseteq \mathrm{sto}(\mathcal{I}(A))$ *such that for all* $c \in \mathrm{Cmd}_i$, $C[\![c]\!]\mathcal{I}(A)\,\mathrm{init}(\mathcal{I}(A)) \in \mathrm{Feas}$.

2. *There is a function* $\phi : \mathrm{sto}(\mathcal{I}(A)) \to \mathrm{sto}(A)$*, such that:*

   i) $\mathbf{const}(\mathrm{init}(A)) = \mathbf{const}(\mathrm{init}(\mathcal{I}(A))); \phi$,

   ii) $\phi|_{\mathrm{Feas}}; (E[\![e]\!]A) = E[\![e]\!]\mathcal{I}(A)$ for $e \in \mathrm{Exp}_i$,

   iii) $\phi; (C[\![c]\!]A) = (C[\![c]\!]\mathcal{I}(A)); \phi$ for $c \in \mathrm{Cmd}_i$.

3. *There is a* total *function* $\psi : W_{i-1} \to \mathrm{Obs}_i$ *such that* $\forall A \in \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1})$, $t \in W_{i-1}.\ \mathrm{eval}^{\mathcal{S}}(\psi(t), A) = t_A$.

9.4 More Notation

We round off this chapter by defining some additional mathematical notation used in the case-studies.[3]

9.4.1 Elementary Sets and Functions

Start with some notation for fundamental sets and functions:

Definition 42 *Let* $\mathbf{1}$ *be the one-point set* $\{*\}$*. It is assumed to be disjoint from any other set of interest unless otherwise indicated.*

Definition 43 *For any set A, let* $\mathrm{id} : A \to A$ *be the identity function on A:*

$$\mathrm{id} \stackrel{\mathrm{def}}{=} \lambda a{:}A.\ a{:}A$$

Definition 44 *Define the successor function for natural numbers:*

$$\mathrm{succ} \stackrel{\mathrm{def}}{=} \lambda n{:}\mathbf{N}.\ n + 1{:}\mathbf{N}$$

[3]The notation used here is a less formal paraphrasing of that proposed by Wagner in [Wag84]; while some of it is possibly a little unfamiliar, it does enable us to define the models in the case studies very conveniently, and greatly facilitates the required proofs.

9.4.2 Sums and Products

We will often make use (especially in the forth case-study) of constructions based on cartesian products and disjoint sums of sets. The former were defined in chapter 4. Add the following definitions:

Definition 45 *The* disjoint sum *of sets $A_1, A_2, \ldots, A_n$, denoted $A_1 + A_2 + \ldots + A_n$ is the set $(\{1\} \times A_1) \cup (\{2\} \times A_2) \cup \ldots \cup (\{n\} \times A_n)$.*

Definition 46 *For sets $A_1, A_2, \ldots, A_n$, for all $i \in \{1, \ldots, n\}$, the* i-th projection, pr_i, *from $A_1 \times A_2 \times \ldots \times A_n$ is defined:*

$$\mathrm{pr}_i \stackrel{\mathrm{def}}{=} \lambda(a_1, a_2, \ldots, a_n){:}A_1 \times A_2 \times \ldots A_n.\ a_i{:}A_i$$

Definition 47 *For sets $A_1, A_2, \ldots, A_n$, for all $i \in \{1, \ldots, n\}$, the* i-th injection, in_i, *into $A_1 + A_2 + \ldots + A_n$ is defined:*

$$\mathrm{in}_i \stackrel{\mathrm{def}}{=} \lambda a{:}A_i.\ (i, a){:}A_1 + A_2 + \ldots + A_n$$

Functions on Sums and Products

Its is convenient to construct functions on products and sums without explicitly mentioning the elements of such sets. The following definitions are useful in this respect:

Definition 48 *Let there be sets $A_1, A_2, \ldots, A_n, B$ and functions $f_1 : A_1 \to B, f_2 : A_2 \to B, \ldots, f_n : A_n \to B$. Then the function $\mathbf{cases}(f_1, f_2, \ldots, f_n) : A_1 + A_2 + \ldots + A_n \to B$ is defined such that for all $(i, a) \in A_1 + A_2 + \ldots + A_n$:*

$$\mathbf{cases}(f_1, f_2, \ldots, f_n)(i, a) = f_i(a)$$

Definition 49 *Let there be sets $A_1, A_2, \ldots, A_n, B$ and functions $f_1 : B \to A_1, f_2 : B \to A_2, \ldots, f_n : B \to A_n$. Then the function $\mathbf{tuple}(f_1, f_2, \ldots, f_n) : B \to A_1 \times A_2 \times \ldots \times A_n$ is defined such that for all $b \in B$:*

$$\mathbf{tuple}(f_1, f_2, \ldots, f_n)(b) = \begin{cases} \perp & \text{if } \exists i \in \{1, \ldots, n\}.\ f_i(b) = \perp \\ (f_1(b), f_2(b), \ldots, f_n(b)) & \text{otherwise} \end{cases}$$

Definition 50 *Let there be sets $A_1, A_2, \ldots, A_n$, $B_1, B_2, \ldots, B_n$ and functions $f_1 : B_1 \to A_1, f_2 : B_2 \to A_2, \ldots, f_n : B_n \to A_n$. Then the function $\mathbf{mult}(f_1, f_2, \ldots, f_n) : B_1 \times B_2 \times \ldots \times B_n \to A_1 \times A_2 \times \ldots \times A_n$ is defined:*

$$\mathbf{mult}(f_1, f_2, \ldots, f_n) \stackrel{\mathrm{def}}{=} \mathbf{tuple}(\mathrm{pr}_1; f_1, \mathrm{pr}_2; f_2, \ldots, \mathrm{pr}_n; f_n)$$

9.4.3 Conditionals

A useful construction in interpreting conditional expressions in OOL's is defined as follows:

Definition 51 *For set A, let there be partial functions $f : A \to \{\mathbf{true}, \mathbf{false}\}, g : A \to A, h : A \to A$. Then $(f \to g,\ h) : A \to A$ is defined such that for $a \in A$:*

$$(f \to g,\ h)(a) = \begin{cases} \perp & \text{if } f(a) = \perp\ , \\ g(a) & \text{if } f(a) = \mathbf{true}, \\ h(a) & \text{if } f(a) = \mathbf{false} \end{cases}$$

9.4.4 Stores

We finish with a formal notation for defining the states of the stores we discussed earlier. These are represented simply as tuples of functions between disjoint sets of *locations* and corresponding sets of *contents*.

Definition 52 *Assume we are given pairwise disjoint sets $L_1, L_2, \ldots, L_n$ and pairwise disjoint sets $C_1, C_2, \ldots, C_n$. A* store*, which binds locations $L_1, L_2, \ldots, L_n$ to contents $C_1, C_2, \ldots, C_n$ is member $\mu \in (L_1 \to C_1) \times (L_2 \to C_2) \times \ldots \times (L_n \to C_n)$.*

Where convenient in the following definitions, we will abbreviate "$L_1, L_2, \ldots, L_n$", "$C_1, C_2, \ldots, C_n$",
and "$(L_1 \to C_1) \times (L_2 \to C_2) \times \ldots \times (L_n \to C_n)$" to "$L$", "$C$" and "$(L \to C)$" respectively.

The functions defined next are used to "retrieve" the contents of locations in a store. They simply apply the appropriate functions from the store to a given location:

Definition 53 *Let sets L and C be as above. For all $i \in \{1, \ldots, n\}$ and all elements $l \in L_i$, define the function* $\mathbf{valueof}(l) : (L \to C) \to C_i$*:*

$$\mathbf{valueof}(l) \stackrel{\text{def}}{=} \lambda\mu.\, (\mathrm{pr}_i \mu) l$$

We also define notation for re-binding a location to new contents in a store. Rather than assuming that we are given a location and its new contents, we assume two *functions*. The first of these, when applied to a store, will return the location whose contents are to be altered; the second, similarly applied, produces the new contents. The new construct merely selects the appropriate component from a memory and re-binds the location returned by the former function to that produced by the latter.

Definition 54 *Lets set L and C be as above. For all $i \in \{1, \ldots, n\}$ and functions $f : (L \to C) \to L_i$, $g : (L \to C) \to C_i$, define the function $\{g/f\} : (L \to C) \to (L \to C)$ such that for any $\mu \in (L \to C)$:*

$$\{g/f\}\mu = \mathrm{pr}_1\mu \times \mathrm{pr}_2\mu \times \ldots \mathrm{pr}_i\mu[g(\mu)/f(\mu)] \times \ldots \mathrm{pr}_n\mu$$

In the interests of brevity, we will normally abbreviate terms of the form "$\{g/\mathbf{const}(a)\}$" to "$\{g/a\}$" where the context allows.

9.4.5 Algebras

Frequently when constructing ΣW-semantics, we will find that the the empty system denotation, V, differs markedly in structure from the denotations of systems containing modules. This can lead to a proliferation of special cases in—for example—the definitions of stores and module-denotations. The following notation can help ameliorate such problems:

For algebra A, sort S and operator σ let:

$$\begin{aligned} S_{(A)} &\stackrel{\text{def}}{=} \begin{cases} S_A & \text{if } S \in \mathrm{Sorts}(A) \\ \emptyset & \text{otherwise} \end{cases} \\ \sigma_{(A)} &\stackrel{\text{def}}{=} \begin{cases} \sigma_A & \text{if } \sigma \in \mathrm{Ops}(A) \\ \Omega & \text{otherwise} \end{cases} \end{aligned}$$

By abuse of notation, we will sometimes write S_A and σ_A instead of $S_{(A)}$ and $\sigma_{(A)}$ respectively, should use of the latter become unduly onerous.

Chapter 10

Case Study 1: A Static Language with Shared Objects

10.1 Introduction

The first language studied is possibly the simplest imaginable "object-oriented language". Though it shares many syntactic elements with the example language of chapter 2, it differs most profoundly in that it is a *static* language (whereas the example language was *dynamic*). Recall that this means that objects are introduced individually by declarations, rather than *en masse* using classes, as was the case in the example language. In accordance with the guidelines set out in the last chapter, the object-declarations in this language associate newly-introduced objects with a given *index* (a strictly positive natural number). The index given in an object's declaration constitutes a reference to it; two objects both quoting the object-index i, say, are referring to a single, shared object.

This language, in common with all of the languages in the case studies, does not allow the definition of non-terminating computations. Proper treatment of such features would require additional mathematical apparatus whose presentation would unduly lengthen the dissertation. In this case, prohibiting non-termination entails eliminating the possibility of cycles of references between objects; if an object is allowed to refer to itself directly or indirectly, then the definition of an operation on that object may cause infinite recursion if it incurs the unguarded invocation of the same operation on the same object using the self-reference (a concrete illustration of this is given in the next chapter—see [Coo89b] for a thorough discussion of this and related issues in OOL's). We exclude cycles of reference by allowing the definitions of an object's operations to quote only the indices of objects which are strictly less than its own.

The private memory of objects in this language is expressed as a set of variables. As was foreshadowed in the last chapter, an object's variables are not declared explicitly, and any legal variable identifier may be used in its definition to name a variable. In the initial state of an object, all variables have undefined contents. Unlike the example language, the variables of objects in this language may contain only numeric or boolean values (*not* objects or references to objects); if an object is to refer to another object, it must do so by explicitly quoting that object's index.

As before, there are two kinds of operations associated with objects. When a *function* is applied to an object, it returns a number or boolean value, without altering the contents of any variables. *Procedures*, on the other hand, are applied to an object to change its state; they do not return a value.

Functions and procedures are defined using a simple notation broadly similar to that in used the example language. However, since objects are referred to explicitly in this language, statements of the form $i.p$ are used to apply procedure p to an object with index i—likewise, $i.f$ applies function f to object i.

10.2 Syntax

To begin the study of this language, we will formally describe its syntax, and fit it into our framework for modular languages. The first step is to define the sets Mod_i, for $i \in \mathbf{N}^+$, which comprise the components of a pre-modular language. Following the blueprint set out in the last chapter, each Mod_i consists of all object-declarations in the language with index i. The syntax of these object-declarations is defined using a form of Backus-Naur notation. Assume the following (non-empty) mutually disjoint sets:

$$\begin{array}{ll} n \in \mathbf{N} & \text{—natural numbers} \\ b \in \mathbf{B} & \text{—booleans} \\ v \in \mathrm{Id}_v & \text{—variable identifiers} \\ f \in \mathrm{Id}_f & \text{—function identifiers} \\ p \in \mathrm{Id}_p & \text{—procedure identifiers} \end{array}$$

Now, for all $i \in \mathbf{N}^+$, define the set of object-declarations Mod_i such that a typical member $m \in \mathrm{Mod}_i$ is of the form:

$$\begin{array}{ll} \textbf{object}\ \ i\ \ \textbf{has} & \\ & \textbf{functions}\ \ f_1 = e_1, \ldots, f_m = e_m \\ & \textbf{procedures}\ \ p_1 : c_1, \ldots, p_n : c_n \end{array}$$

Here, $e_1, \ldots, e_m \in \mathrm{Exp}_i$, and $c_1, \ldots c_n \in \mathrm{Cmd}_i$ are members of families of sets of *expressions* and *commands*, $(\mathrm{Exp}_i)_{i \in \mathbf{N}}$ and $(\mathrm{Cmd}_i)_{i \in \mathbf{N}}$ respectively. You will recall from the last chapter that these are used to define the observations of a modular language: for all $i \in \mathbf{N}^+$, $\mathrm{Obs}_i = \mathrm{Cmd}_i \times \mathrm{Exp}_i$. The commands and expressions of this language are defined such that for $i \in \mathbf{N}^+$, and j ranging over $1, \ldots, i-1$, typical elements $e \in \mathrm{Exp}_i$ and $c \in \mathrm{Cmd}_i$ are of the form:

$$\begin{array}{rcl} e & ::= & b \mid n \mid v \mid j.f \\ c & ::= & c_1; c_2 \mid \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid v \leftarrow e \mid j.p \end{array}$$

Following the outline given in the last chapter, we now have a modular language whose modules are the object-declarations in the OOL presented, with sets of observations formed from the commands and expressions described above. The algebra V of observation results (in accordance with the outline) is simply the set $\mathbf{B} \cup \mathbf{N}$, of boolean values and natural numbers, regarded as an algebra with a single sort, "Val". As you will recall from section 9.2.2, to complete the modular language, we can derive the evaluation functions of this modular language from the ΣW-semantics described below.

10.3 Preface to the Semantics

Rather than proceed directly with the formal exposition of a congruent ΣW-semantics for the language, we will first try to point out informally how it is affected by the character of the language itself.

The most fundamental influence on the model arises from the possibility of sharing objects in the language. As an example of this, consider a simple system of two objects defined as follows:

object 1 **has**
    functions f = x
    procedures p: x $\leftarrow$ 4
        q: x $\leftarrow$ 5

object 2 **has**
    functions
    procedures r: 1.q

Note the reference to object 1 in the definition of procedure r in object 2. Since any observer of this system would have an index 3, it too would be able to use the same reference. Object 1, then, may be shared by object 2 and any potential observer.

Now imagine that such an observer applies the observation "(1.p, 1.f)" to the system. This involves first executing the command "1.p" (which applies the procedure p to object

1), and then evaluating the expression "1.f" (which applies function f to the same object). The result is 4.

Picture the same observer applying the observation "(1.p; 2.r, 1.f)". In this case, p is applied to object 1 as before, but now the procedure r is subsequently applied to object 2. According to the definition of r, this causes procedure q to be applied to 1, changing its state. Now the result of evaluating expression 1.f is 5. This despite the fact that the sequence of operations which the observer has applied to object 1 (viz. p, then f) is precisely the same as in first observation. The observer has thus witnessed a change in an object's state brought about by an operation (in this case r) not associated with that object in its definition. In fact, generally in this language, applying a procedure to an object in a system may result in a change in the state of any object with a lesser index.

With regard to the model, this effect means that rather than representing the state of the objects in a system as a collection of separate entities (as one might be given to expect), the state of the *entire* system is represented as a single unit. In addition, the representations of procedures and functions in the semantics apply to this single unit, rather than to representations of individual objects themselves.

10.4 Semantics

We will set out the description of the ΣW-semantics according to the scheme suggested in the previous chapter. We begin by describing the families of signatures which establish the classes of algebras from which system denotations are drawn.

10.4.1 Signatures

Recall from section 9.2.1 that we need to produce a family $(\Sigma_i)_{i \in \mathbf{N}^+}$. Each Σ_i (for all $i \in \mathbf{N}^+$) must be both an extension of $\mathrm{Sig}(V)$—the signature of observation results—and also sufficient to represent all of the interactions between a system with i object-declarations and an observer of it. We will examine the sorts and operators of Σ_i in turn.

Sorts

Since the signature of V must be a subsignature of Σ_i, we know that the sorts of Σ_i must include the sort "Val" (this being the sole sort of V).

In addition to this, Σ_i has one other sort, "State". This represents the possible states of a system defining i objects—recall we pointed out that because of the possible occurence of shared objects, the states of all i objects are treated as a single entity.

Operators

It is necessary that operators of Σ_i must contain all the operators of $\mathrm{Sig}(V)$, and also be sufficient to model the way in which observing objects may affect and interrogate a system with i objects. The first requirement is trivially satisfied, of course, since $\mathrm{Sig}(V)$ has no operators.

We will need a constant, "new", which names the initial state of the system (in this state, the values of the variables of all i objects are undefined).

As for the remaining operators, considerable technical convenience accrues if we consider that a function or procedure may be applied to *any* object, regardless of whether that object's declaration actually includes a definition for it. Of course, if such a definition does not actually exist, then the result of the application is undefined. This mirrors the behaviour of dynamically-typed languages, such as Smalltalk, where erroneous message-sends are indicated by run-time (not compile-time) errors.

To deal properly with shared objects, operators defined on any object in the system act on the state of the system as a whole. Therefore, in keeping with the remarks made in the previous paragraph, we need to include in Σ_i operators corresponding to functions f and procedures p of object j, for all function names f, procedure names p, and object-indices j, where $f \in \mathrm{Id}_f$, $p \in \mathrm{Id}_p$ and $1 \leq j \leq i$. In order to disambiguate functions and procedures with the same name in different objects, we will tag these operators. Form tagged copies of the sets of function and procedure names thus:

$$\begin{aligned}\forall i \in \mathbf{N}.\ \mathrm{Id}_f^i &\stackrel{\mathrm{def}}{=} \{i\} \times \mathrm{Id}_f \\ \forall i \in \mathbf{N}.\ \mathrm{Id}_p^i &\stackrel{\mathrm{def}}{=} \{i\} \times \mathrm{Id}_p\end{aligned}$$

Functions $\_^i : \mathrm{Id}_f \to \mathrm{Id}_f^i$ and $\_^i : \mathrm{Id}_p \to \mathrm{Id}_p^i$ prepend these tags, so that $\forall f \in \mathrm{Id}_f, i \in \mathbf{N}.\ f^i = (i, f)$ and $\forall p \in \mathrm{Id}_p, i \in \mathbf{N}.\ p^i = (i, p)$.

So as to represent the functions and procedures of all the objects defined in a system with objects indexed 1 to i, Σ_i has operators f^j and p^j, for $1 \leq j \leq i$, $f \in \mathrm{Id}_f$ $p \in \mathrm{Id}_p$. (Operator f^j represents the function named f in object j, and p^j represents procedure p in object j.) Since all of these operators act upon the state of the system as a whole, they have types $f^i : \mathrm{State} \to \mathrm{Val}$ and $p^i : \mathrm{State} \to \mathrm{State}$.

To summarize, we have, for all $i \in \mathbf{N}^+, p \in \mathrm{Id}_p, f \in \mathrm{Id}_f$:

$$
\Sigma_i \stackrel{\text{def}}{=} \begin{array}{ll} \textbf{sorts} & \text{State}, \text{Val} \\ \textbf{operators} & \text{new} : \text{State} \\ & f^1 : \text{State} \to \text{Val}, , \ldots , f^i : \text{State} \to \text{Val} \\ & p^1 : \text{State} \to \text{State}, \ldots , p^i : \text{State} \to \text{State} \end{array}
$$

It is easy to show that as required, for all $i \in \mathbf{N}^+$, (Σ_i, V) has no inputs.

10.4.2 Visible Terms

Next, we need to formulate the sets of visible terms $(W_i)_{i \in \mathbf{N}^+}$ which fix the (algebraic) behaviour of system denotations. Note that in order to produce a congruent semantics, each set W_i must describe *precisely* those responses which may be elicited by the observer of a system with i objects.

To see how these sets are produced, let us examine informally the interaction between such a system and its observer. All such interactions begin with the system in its initial state, named by the constant operator "new" in the signature of the system denotation. The observer may then cause a finite sequence of procedures to be applied to objects in the system. These procedures are named by p^j, for $p \in \text{Id}_p$, $1 \le j \le i$. After a number of procedure applications, the observer may apply a function $f \in \text{Id}_f$ to any object with index $j \in \{1, \ldots, i\}$ in the system. This would correspond to the application of operator f^j in the signature.

In summary then, a computation elicited by an observer would be described by a term over Σ_i of the form $f^j(p_n^{j_n}(\ldots p_1^{j_1}(\text{new})\ldots))$, for $j, j_1, \ldots, j_n \in \{1, \ldots, i\}$, $f \in \text{Id}_f$, $p_1, \ldots, p_n \in \text{Id}_p$. Terms of this form comprise the entire set, $T^{\text{v}}_{\Sigma_i,V}$, of visible terms on Σ, which you will recall from chapter 5 is the union of the sets $(T_{\Sigma_i})_s$, for $s \in \text{Sorts}(V)$. Thus, for all $i \in \mathbf{N}^+$, we have:

$$
W_i \stackrel{\text{def}}{=} T^{\text{v}}_{\Sigma_i,V}
$$

10.4.3 Stores

Our next task is to show how the commands and expressions of the language are to be interpreted. This allows us to apply observations to systems and to form the denotations of systems themselves. Here, a central rôle is played by a store like those described in informally in section 9.2.2 and formally in 9.4.4.

According to the plan set out in chapter 9, we need to formulate a function sto_i (for all $i \in \mathbf{N}^+$), such that for $A \in \mathbf{Alg}(\Sigma_i, V, W_i)$, $\text{sto}_i(A)$ (abbreviated $\text{sto}(A)$) represents all the state accessible to an observer (which you will recall is simply a new object) based upon a system with denotation A. In the language studied here, this state comprises the contents of the observer's private variables (whose names are drawn from Id_v), and the state of the objects in the system (which is represented as a single value from the carrier State_A in the denotation of the system). Thus $\mathrm{sto}(A)$ binds (elements of) Id_v to $\mathbf{B} \cup \mathbf{N}$, and the distinguished element $* \in \mathbf{1}$ (assumed disjoint from Id_v) to State_A. Thus:[1]

$$\mathrm{sto}(A) \quad \stackrel{\mathrm{def}}{=} \quad (\mathrm{Id}_v \to \mathbf{B} \cup \mathbf{N}) \times (* \to \mathrm{State}_{(A)})$$

Remember that Store_i is the set $\bigcup_{A \in \mathbf{Alg}(\Sigma_i, V, W_i)} \mathrm{sto}(A)$—i.e. $\bigcup_{A \in \mathbf{Alg}(\Sigma_i, V, W_i)} (\mathrm{Id}_v \to \mathbf{B} \cup \mathbf{N}) \times (* \to \mathrm{State}_{(A)})$.

Next, we have to define $\mathrm{init}(A)$, which represents the initial state of system and observer. In this state, all the variables of the observer have undefined contents, and the system is in its initial state—named by the constant new_A.

$$\mathrm{init}(A) \quad \stackrel{\mathrm{def}}{=} \quad (\Omega, \Omega[\mathrm{new}_A/*])$$

10.4.4 Interpreting Commands and Expressions

Next we turn to the interpretation of commands and expressions in the language. We need to provide, for all $i \in \mathbf{N}^+$, two semantic functions—the first for commands, the second for expressions:

$$\begin{array}{l} C_i : \mathrm{Cmd}_i \to \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1}) \to \mathrm{Store}_i \to \mathrm{Store}_i \\ E_i : \mathrm{Exp}_i \to \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1}) \to \mathrm{Store}_i \to V \end{array}$$

These functions are defined in conventional Scott-Strachey style. Using the notation defined in section 9.4 allows us to elide the third argument (the store) to the functions. Dropping decorations, we have, for $j \in \{1, \ldots, i\}$:

$$\begin{array}{lcl} C[\![c_1; c_2]\!]A & = & C[\![c_1]\!]A; C[\![c_2]\!]A, \\ C[\![\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!]A & = & E[\![e]\!]A \to C[\![c_1]\!]A,\ C[\![c_2]\!]A, \\ C[\![v \leftarrow e]\!]A & = & \{E[\![e]\!]A/v\}, \\ C[\![j.p]\!]A & = & \{\mathbf{valueof}(*); p_A^j/*\} \\ & & \\ E[\![v]\!]A & = & \mathbf{valueof}(v), \\ E[\![b]\!]A & = & \mathbf{const}(b), \\ E[\![n]\!]A & = & \mathbf{const}(n), \\ E[\![j.f]\!]A & = & \mathbf{valueof}(*); f_A^j. \end{array}$$

[1] For an explanation of notation used in the term "$\mathrm{State}_{(A)}$", refer to section 9.4.5.

Most of the clauses are straightforward applications of the notation. Sequential composition of two commands is interpreted as the diagrammatic composition of their interpretations. Conditional commands are interpreted directly using the conditional function construct defined earlier. An assignment command which assigns to the variable v the value of expression e is interpreted as a function which appropriately re-binds v in any store to which it is applied. A command $j.p$, which applies procedure p to object with index i, is interpreted as a function which rebinds "$*$" (to which the state of the system is bound) to the result of applying to the original system state the representation of the procedure supplied by the system's denotation. The interpretation of a variable name is simply a function which extracts the value bound to that name from any store to which it is applied. A constant is represented by the corresponding constant function. Finally, an expression $j.f$, applying the function f to object j, becomes a function which when applied to a store, first extracts the state of the system from it, and then applies to that state the representation of the function from the denotation of the system.

Having defined C_i and E_i, from the previous chapter we have, for all $i \in \mathbf{N}^+$:

$$\mathrm{eval}_i^{\mathcal{S}} = \lambda(c, e), A.\ E[\![e]\!]A\,(C[\![c]\!]A\,\mathrm{init}(A))$$

And to complete the modular language, for $s \in \mathrm{Mod}_i^{\#}$:

$$\mathrm{eval}_i = \lambda(c, e), s.\ E[\![e]\!](M^{\#}[\![s]\!])\,(C[\![c]\!](M^{\#}[\![s]\!])\,\mathrm{init}(A))$$

10.4.5 Interpreting Modules

Finally, we must present the semantic functions $(M_i)_{i \in \mathbf{N}^+}$ which interpret object definitions in the language. Select any $i \in \mathbf{N}^+$. From the syntax definition, a typical object definition $m \in \mathrm{Mod}_i$ is of the form:

$$\begin{array}{ll} \textbf{object}\ \ i\ \ \textbf{has} & \\ & \textbf{functions}\ \ f_1 = e_1, \ldots, f_m = e_m \\ & \textbf{procedures}\ \ p_1 : c_1, \ldots, p_n : c_n \end{array}$$

Now, for any $A \in \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1})$, let $M_i[\![m]\!]A$ be the algebra $A' \in \mathbf{Alg}(\Sigma_i, V, W_i)$, such that:

$$
\begin{array}{ll}
\textbf{carriers} & \mathrm{State}_{A'} = \mathrm{sto}(A), \mathrm{Val}_{A'} = \mathbf{B} \cup \mathbf{N} \\
\textbf{operations} & \mathrm{new}_{A'} = \mathrm{init}(A), \\
& f^1_{A'} = \mathbf{valueof}(*); f^1_A, \ldots, f^{i-1} = \mathbf{valueof}(*); f^{i-1}_A \\
& p^1_{A'} = \{\mathbf{valueof}(*); p^1_A/*\}, \ldots, p^{i-1}_{A'} = \{\mathbf{valueof}(*); p^{i-1}_A/*\} \\
& f^i_{A'} = \begin{cases} E[\![e_j]\!]A & \text{if } f = f_j \in \{f_1, \ldots, f_m\} \\ \Omega & \text{otherwise} \end{cases} \\
& p^i_{A'} = \begin{cases} C[\![c_j]\!]A & \text{if } p = p_j \in \{p_1, \ldots, p_n\} \\ \Omega & \text{otherwise} \end{cases}
\end{array}
$$

If A represents a system consisting of i objects, then A' is the system obtained by adding to it the object defined in m. As with A, the carrier $\mathrm{Val}_{A'}$ is determined by definition. Since the set $\mathrm{sto}(A)$ described above represents the state of a system with i objects together with any object based upon it, it is used as the carrier $\mathrm{State}_{A'}$ of the new system's denotation.

Operations $p^1, \ldots, p^{i-1}$ and $f^1, \ldots f^{i-1}$, relate to the objects in the original system. Their interpretations in A' extract the state of the original system from the store and then apply the corresponding operation drawn from A. Operations p^i and f^i are applied to the new object introduced by m. If they are defined by the declarations in m, then their interpretation in A' is the interpretation of the corresponding statement or expression. If they are not defined, then their interpretation in A' is an undefined function.

10.5 Proving Congruence

We have now assembled a modular language, based upon the original programming notation and a notion of l-behaviour such as was described in the previous chapter. We also have a ΣW-semantics for the modular language which provides interpretations of systems as algebras, and a definition of a-behaviour for those algebras. To complete the presentation, we must show that the semantics is *congruent*—i.e. that the l-behaviour of systems and the a-behaviour of their denotations coincide. According to the results of chapter 8, this automatically delivers a correct, compositional, abstract semantics for the modular language (using the term realization construction), and also establishes the stability of the modular language (and hence of the programming notation with respect to the notion of l-behaviour chosen).

To do this, we make use of corollary 1 (from section 9.3). To re-iterate, this requires, for all $i \in \mathbf{N}^+$ and $A \in \mathbf{Alg}(\Sigma_i, V, W_i)$, we show that:

1. There is a set $\mathrm{Feas} \subseteq \mathrm{sto}(\mathcal{I}(A))$ such that for all $c \in \mathrm{Cmd}_i$, $C[\![c]\!]\mathcal{I}(A)\,\mathrm{init}(\mathcal{I}(A)) \in$

Feas.

2. There is a function $\phi : \mathrm{sto}(\mathcal{I}(A)) \to \mathrm{sto}(A)$, such that:

   i) $\mathbf{const}(\mathrm{init}(A)) = \mathbf{const}(\mathrm{init}(\mathcal{I}(A))); \phi$,

   ii) $\phi|_{\mathrm{Feas}}; (E[\![e]\!]A) = E[\![e]\!]\mathcal{I}(A)$ for $e \in \mathrm{Exp}_i$,

   iii) $\phi; (C[\![c]\!]A) = (C[\![c]\!]\mathcal{I}(A)); \phi$ for $c \in \mathrm{Cmd}_i$.

3. There is a *total* function $\psi : W_{i-1} \to \mathrm{Obs}_i$ such that $\forall A \in \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1})$, $t \in W_{i-1}.\ \mathrm{eval}^{\mathcal{S}}(\psi(t), A) = t_A$.

In this proof, we may put Feas $= \mathrm{sto}(\mathcal{I}(A))$, so that we need only provide functions $\phi : \mathrm{sto}(\mathcal{I}(A)) \to \mathrm{sto}(A)$ and $\psi : W_{i-1} \to \mathrm{Obs}_i$ such that:

i) $\mathbf{const}(\mathrm{init}(A)) = \mathbf{const}(\mathrm{init}(\mathcal{I}(A))); \phi$,

ii) $\phi; (E[\![e]\!]A) = E[\![e]\!]\mathcal{I}(A)$ for $e \in \mathrm{Exp}_i$,

iii) $\phi; (C[\![c]\!]A) = (C[\![c]\!]\mathcal{I}(A)); \phi$ for $c \in \mathrm{Cmd}_i$.

iv) ψ is *total*, and $\forall A \in \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1}), t \in W_{i-1}.\ \mathrm{eval}^{\mathcal{S}}(\psi(t), A) = t_A$

The set $\mathrm{sto}(\mathcal{I}(A))$ is derived by precise analogy with $\mathrm{sto}(A)$:

$$\mathrm{sto}(A) \quad \stackrel{\mathrm{def}}{=} \quad (\mathrm{Id}_v \to \mathbf{B} \cup \mathbf{N}) \times (* \to \mathrm{State}_{\mathcal{I}(A)})$$

Likewise, $\mathrm{start}_{\mathcal{I}(A)}$ is defined:

$$\mathrm{start}_{\mathcal{I}(A)} \quad \stackrel{\mathrm{def}}{=} \quad (\Omega, *[\Omega/\mathrm{new}_{\mathcal{I}(A)}])$$

In a member of $\mathrm{sto}(\mathcal{I}(A))$, $*$ is bound to a *term* in the carrier $\mathrm{State}_{\mathcal{I}(A)}$, rather than a state from A itself as it is in a member of $\mathrm{sto}(A)$. The first function required, ϕ, transforms an element of $\mathrm{sto}(\mathcal{I}(A))$ by evaluating the term bound to $*$, so as to produce a state. The values bound to the variables of the new object (bound to Id_v), being booleans and numbers, are left untouched. Formally (using the function $\mathrm{ev}_A = \lambda t.\, t_A$ from chapter 5), we have:

$$\phi = \mathbf{mult}(\mathrm{id}, \mathrm{ev}_A)$$

For the first part of the proof of congruence, we have:

Theorem 11 *As defined, ϕ satisfies conditions* i), ii) *and* iii) *above.*

Proof:

Involves establishing each of the sub-conditions by structural induction on commands and expressions; details are given in appendix B.

□

To fulfil the second condition for proof of congruence, we must produce for each visible term $t \in W_i$ $(i \in \mathbf{N}^+)$, an observation $\psi(t) \in \mathrm{Obs}_{i+1}$ which elicits the same response from the system to which it is applied. For the model given here, this is not too difficult. In general, t calls for the application of a given sequence of procedures to the objects in a system, and concludes with the application of a function; it is typically of the form:

$$f^j(p_n^{j_n}(\ldots p_1^{j_1}(\mathrm{new})\ldots)), \quad \text{for } j, j_1, \ldots, j_n \in \{1, \ldots, i\}, f \in \mathrm{Id}_f, p_1, \ldots, p_n \in \mathrm{Id}_p$$

The observation $\psi(t)$ consists of a sequence of commands which applies the same sequence of procedures as occurs in t, and an expression which applies the same concluding function; symbolically, it is written:

$$(j_1.p_1; \ldots; j_n.p_n,\ j.f)$$

We have:

Theorem 12 *As defined, ψ satisfies condition 2 above.*

Proof:

Directly from the definitions, we have that:

$$\begin{array}{l}
\mathrm{eval}(A, (j_n.p_n; j_{n-1}.p_{n-1}; \ldots; j_1.p_1, f^j)) \\
\quad = \ (\{\mathbf{valueof}(*); (p_1^{j_1})_A/*\}; \ldots; \{\mathbf{valueof}(*); (p_n^{j_n})_A/*\}; \mathbf{valueof}(*); f_A^j)\ (\Omega, \Omega[\mathrm{new}_A/*]) \\
\quad = \ f_A^j((p_n^{j_n})_A \ldots (p_1^{j_1})_A(\mathrm{new}_A)\ldots) \\
\quad = \ \mathrm{ev}_A(t)
\end{array}$$

□

We conclude, therefore, that the ΣW-semantics given in this chapter is congruent, and that the modular language it describes is stable.

Chapter 11

Case Study 2: A Static Language with Shared Objects and Cycles

11.1 Introduction

In this study, we will extend the language of the last chapter by introducing a limited form of cyclic reference between objects. We pointed out that in general, introducing cycles of reference would involve us in a thorough-going treatment of non-termination. This is easily illustrated using a syntax like that of the previous language (without restriction on the use of indices). Consider the simple two-object system:

object 1 **has procedures** p: 2.q
object 2 **has procedures** q: 1.p

Here, a cycle of reference arises from the way in which the definition of procedure p of object 1 refers to procedure q of object 2, and vice-versa. It should be evident that invoking either p on 1, or q on 2 will cause a non-terminating computation. In the last chapter, ensuring termination meant forbidding objects to reference objects with indices greater than or equal to their own. In this chapter, we still want to preclude non-termination, and maintain this restriction on the use of indices. Instead, this language offers a more restricted form of cyclic reference, constituted using a programming technique known as the *call-back*. Used commonly in operating systems and window-management software [Sun88], this technique involves a system component (the *client*) requesting a service from another component (the *server*) providing the server with a means (often a procedure or procedural closure) for invoking operations on the client.

The language in this chapter features a very simple form of call-back; when a procedure is applied to an object, an assignment statement is also supplied, to be executed during the course of the procedure *within the object applying the procedure.* This does allow a limited form of cyclic reference, in that it allows an object to cause the execution of code (to wit, an assignment statement) in another object invoking an operation on it. However, since the assignment statement cannot invoke further procedures, the possibility of non-termination is eliminated. The actual syntax used to apply a procedure takes the form $i.p$ **calling** $(v \leftarrow e)$, where i is the index of the target object, p the name of the procedure, and $v \leftarrow e$ the assignment statement supplied. In order to accommodate call-backs, procedure-definitions now have two components, the first to be executed before the supplied assignment statement, and the second afterwards.

Since this language shares much in common with the previous one, and since the steps undertaken producing a modular language and ΣW-semantics for it are similar, the presentation here is rather less expository than that of the foregoing chapter; we concentrate mainly on the differences between the two languages.

11.2 Syntax

Assume sets $n \in \mathbf{N}, b \in \mathbf{B}, v \in \mathrm{Id}_v$, and so on, as before. For all $i \in \mathbf{N}^+$, a typical object-declaration (i.e. module $m \in \mathrm{Mod}_i$) is of the form:

$$\begin{array}{ll} \textbf{object}\ \ i\ \ \textbf{has} & \\ & \textbf{functions}\ \ f_1 = e_1, \ldots, f_m = e_m \\ & \textbf{procedures}\ \ p_1 : c_{1,1};\ \textbf{callback};\ c_{1,2}, \ldots, p_n : c_{n,1};\ \textbf{callback};\ c_{n,2} \end{array}$$

Commands and expressions are such that for $i \in \mathbf{N}^+$ and j ranging over $1, \ldots, i-1$, typical elements $e \in \mathrm{Exp}_i$ and $c \in \mathrm{Cmd}_i$ are of the form:

$$\begin{array}{lcl} e & ::= & b \mid n \mid v \mid j.f \\ c & ::= & c_1; c_2 \mid \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid v \leftarrow e \mid j.p\ \textbf{calling}\ (v \leftarrow e) \end{array}$$

We have a modular language whose modules are the object-declarations in the OOL presented, with sets of observations pairs of commands and expressions. Again, the algebra V of observation results, is $\mathbf{B} \cup \mathbf{N}$, regarded as an algebra with a single sort, Val.

11.3 Preface to the Semantics

As in the previous chapter, we will make some prefatory comments concerning the intuitive semantics of the language and their impact upon the model supplied.

First of all, being an extension of the language in the previous chapter, the presence of shared objects in this language means that the state of all objects in a system is again modeled as a single indivisible unit.

The introduction of call-backs has a further significant effect upon the model. For an example of how this comes about, define a system with just one object as follows:

object 1 **has functions** f: x
procedures p: x $\leftarrow$ 1; **callback**; x $\leftarrow$ 2

Now consider the result of applying the following observation to this system:

$$(1.\mathsf{p}\ \textbf{calling}\ (y \leftarrow 1.\mathsf{f}),\ \ y)$$

Applying this observation invokes procedure p on object 1. This causes the private variable x of the object to adopt the value 1. At this point, the supplied assignment statement ($y \leftarrow 1.\mathsf{f}$) is executed, copying the value of x into variable y of *the observer*. Control returns to the procedure, where x is assigned the value 2. The result of the observation is the final value of y—to wit, the value 1. Note that this value has been extracted from object 1 in a state intermediate between the initial one, where 1.f is undefined, and that which results from the complete execution of procedure p, where 1.f returns 2. Such intermediate states arise generally in observing the systems of this language, and they must be properly accounted for in the model.

11.4 Semantics

As before, to define a ΣW-semantics for for this language, we must provide a family of signatures $(\Sigma_i)_{i\in\mathbf{N}^+}$, a family of sets of visible terms $(W_i)_{i\in\mathbf{N}^+}$, semantic functions $(C_i)_{i\in\mathbf{N}^+}$ and $(E_i)_{i\in\mathbf{N}^+}$ interpreting commands and expressions, as well as the stores used to define these semantic functions, and finally semantic functions $(M_i)_{i\in\mathbf{N}^+}$ which extract the denotations of object-definitions, or modules.

11.4.1 Signatures

Examine a typical Σ_i, for some $i \in \mathbf{N}^+$.

Sorts

The sorts of Σ_i in this model are precisely the same as in the last model; sort "Val", so as to make the signature of V a subsignature of Σ_i, and "State", the sort containing representations of system states.

Operators

Recall the requirements that operators of Σ_i must contain all the operators of $\mathrm{Sig}(V)$, and also be sufficient to model the way in which observing objects may affect and interrogate a system with i objects. As usual, the first charge confers no operations on Σ_i.

In complying with the second demand, we can use some of the same operators as in the previous model; a constant operator "new", which names the initial state of system, and an operator f^j : State $\rightarrow$ Val for all $1 \leq j \leq i$ representing each function f, $f \in \mathrm{Id}_f$ on objects in the system.

However, the representation of procedures in this model differs from the treatment of the last chapter. Things are complicated by the presence of intermediate states as described above. From the syntax, we see that the definition of procedure p must be of the form $p:\ c_1$; **callback**; c_2, where c_1 is the command executed before the call-back, and c_2 that executed afterwards. It was demonstrated that an intermediate observable state of the system exists after c_1 has been executed. To narrate this properly in the semantics, procedure p of object j is represented not by a single operator, but by two, $\overrightarrow{p}^j$ and $\overleftarrow{p}^j$, where operator $\overrightarrow{p}^j$ represents the execution of c_1 (leading to the intermediate state) and $\overleftarrow{p}^j$ that of c_2 (which produces the final observable state). We are assume tagged copies $\overrightarrow{\mathrm{Id}_p}^j$ and $\overleftarrow{\mathrm{Id}_p}^j$ of Id_p^j, together with injections $\rightharpoondown : Id_p^j \rightarrow \overrightarrow{\mathrm{Id}_p}^j$ and $\leftharpoondown : Id_p^j \rightarrow \overleftarrow{\mathrm{Id}_p}^j$.

To summarize, we have, for all $i \in \mathbf{N}^+, p \in \mathrm{Id}_p, f \in \mathrm{Id}_f$:

$$\begin{array}{lll} \Sigma_i \stackrel{\text{def}}{=} & \textbf{sorts} & \text{State}, \text{Val} \\ & \textbf{operators} & \text{new} : \text{State} \\ & & f^1 : \text{State} \rightarrow \text{Val}, , \ldots, f^i : \text{State} \rightarrow \text{Val} \\ & & \overrightarrow{p}^1 : \text{State} \rightarrow \text{State}, \ldots, \overrightarrow{p}^i : \text{State} \rightarrow \text{State} \\ & & \overleftarrow{p}^1 : \text{State} \rightarrow \text{State}, \ldots, \overleftarrow{p}^i : \text{State} \rightarrow \text{State} \end{array}$$

11.4.2 Visible Terms

Next, we need to formulate the sets of visible terms $(W_i)_{i \in \mathbf{N}}$ which fix the (algebraic) behaviour of system denotations. Again, in order to produce a congruent semantics, each set W_i must describe *precisely* those interactions which may be elicited by an observer of a system with i object-definitions.

Consider a typical interaction between such a system and an observer which produces a visible value. All such interactions begin with the system in its initial state, named by the constant operator "new" in the signature of the system denotation. Thereafter, the observer may apply a procedure $p \in \mathrm{Id}_p$ to an object with index $j \in \{1, \ldots, n\}$ in the system. The execution of such a procedure, as we saw above, is divided into two phases. The first phase, corresponding to operator $\overrightarrow{p}^j$ in Σ_i, causes the system to enter an intermediate state, where it awaits the execution of the call-back. Once this is done, the procedure is completed by the operator $\overleftarrow{p}^j$. From the perspective of the system (which does not itself experience the effects of the call-back executing in the observer), this means that each application of $\overrightarrow{p}^j$ must be followed *immediately* by an application of $\overleftarrow{p}^j$.

After an arbitrary finite number of procedure applications, the observer must apply a function $f \in \mathrm{Id}_f$ to any object with index $j \in \{1, \ldots, i\}$ in the system. This would correspond to the application of operator f^j in the signature.

Thus a term describing such a typical interaction would consist of the operator "new", followed by a finite number of applications of pairs of operators $\overrightarrow{p}^j$ and $\overleftarrow{p}^j$, and concluding with an application of f^j. Formally, for all $i \in \mathbf{N}$, let U_i, U_i', W_i be the smallest sets such that for all $p \in \mathrm{Id}_p$, $f \in \mathrm{Id}_f$, $1 \le j \le i$:

$$\begin{array}{rcl} & & \mathrm{new} \in U_i, \\ t \in U_i & \Rightarrow & \overrightarrow{p}^j(t) \in U_i' \\ t \in U_i' & \Rightarrow & \overleftarrow{p}^j(t) \in U_i \\ t \in U_i & \Rightarrow & f^j(t) \in W_i \end{array}$$

11.4.3 Stores

Though the introduction of call-backs changes the nature of the interactions between a system and an observer (as compared with the previous chapter), it does not alter the nature of the state accessible to the observer; it still comprises the state of the system and the contents of the observer's variables. Thus we may use the same definitions for stores as we did in the last chapter:

$$\begin{array}{rcl} \mathrm{sto}(A) & \stackrel{\mathrm{def}}{=} & (\mathrm{Id}_v \to \mathbf{B} \cup \mathbf{N}) \times (* \to \mathrm{State}_{(A)}) \\ \mathrm{init}(A) & \stackrel{\mathrm{def}}{=} & (\Omega, \Omega[\mathrm{new}_A/*]) \end{array}$$

11.4.4 Interpreting Commands and Expressions

Proceed as before; for all $i \in \mathbf{N}^+$

$$\begin{array}{l} C_i : \mathrm{Cmd}_i \to \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1}) \to \mathrm{Store}_i \to \mathrm{Store}_i \\ E_i : \mathrm{Exp}_i \to \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1}) \to \mathrm{Store}_i \to V \end{array}$$

For $j \in \{1, \ldots, i\}$ (without decorations):

$$\begin{array}{lcl} C[\![c_1; c_2]\!]A & = & C[\![c_1]\!]A; C[\![c_2]\!]A, \\ C[\![\,\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!]A & = & E[\![e]\!]A \to C[\![c_1]\!]A,\ C[\![c_2]\!]A, \\ C[\![v \leftarrow e]\!]A & = & \{E[\![e]\!]A/v\}, \\ C[\![j.p\ \mathbf{calling}\ (v \leftarrow e)]\!]A & = & \{\mathbf{valueof}(*); \overrightarrow{p}^j_A/*\}; \{E[\![e]\!]A/v\}; \{\mathbf{valueof}(*); \overleftarrow{p}^j_A/*\} \end{array}$$

$$\begin{array}{lcl} E[\![v]\!]A & = & \mathbf{valueof}(v), \\ E[\![b]\!]A & = & \mathbf{const}(b), \\ E[\![n]\!]A & = & \mathbf{const}(n), \\ E[\![j.f]\!]A & = & \mathbf{valueof}(*); f^j_A. \end{array}$$

The only significant difference between these definitions and the corresponding ones from the previous chapter concerns (as one might expect) the interpretation of procedure calls. Informally, executing a statement of the form $j.p$ **calling** $(v \leftarrow e)$ involves initially applying the operator $\overrightarrow{p}^j$ to the state of the system bound to "$*$" in the store. This invokes the first stage of the procedure, and, as we have said, places the system in an intermediate state where it awaits execution of the callback. This is achieved by executing the supplied assignment statement in the observer. Finally the procedure is completed by applying $\overleftarrow{p}$ to the intermediate state, bindng the result to $*$.

Again, we should point out that the families of functions $(C_i)_{i \in \mathbf{N}^+}$ and $(E_i)_{i \in \mathbf{N}^+}$ provide the basis for evaluating observations in the semantics, and hence in the modular language.

11.4.5 Interpreting Modules

From the syntax, a typical object definition $m \in \mathrm{Mod}_i$ is of the form:

object i **has**
  functions $f_1 = e_1, \ldots, f_m = e_m$
  procedures $p_1 : c_{1,1}$; **callback**; $c_{1,2}, \ldots, p_n : c_{n,1}$; **callback**; $c_{n,2}$

Now, for any $A \in \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1})$, let $M_i[\![m]\!]A$ be the algebra $A' \in \mathbf{Alg}(\Sigma_i, V, W_i)$, such that:

$$
\begin{array}{ll}
\textbf{carriers} & \mathrm{State}_{A'} = \mathrm{sto}(A), \mathrm{Val}_{A'} = \mathbf{B} \cup \mathbf{N} \\
\textbf{operations} & \mathrm{new}_{A'} = \mathrm{init}(A), \\
& f^1_{A'} = \mathbf{valueof}(*); f^1_A, \ldots, f^{i-1}_{A'} = \mathbf{valueof}(*); f^{i-1}_A \\
& \overrightarrow{p}^1_{A'} = \{\mathbf{valueof}(*); \overrightarrow{p}^1_A/*\}, \ldots, \overrightarrow{p}^{i-1}_{A'} = \{\mathbf{valueof}(*); \overrightarrow{p}^{i-1}_A/*\} \\
& \overleftarrow{p}^1_{A'} = \{\mathbf{valueof}(*); \overleftarrow{p}^1_A/*\}, \ldots, \overleftarrow{p}^{i-1}_{A'} = \{\mathbf{valueof}(*); \overleftarrow{p}^{i-1}_A/*\} \\
& f^i_{A'} = \begin{cases} E[\![e_j]\!]A & \text{if } f = f_j \in \{f_1, \ldots, f_m\} \\ \Omega & \text{otherwise} \end{cases} \\
& \overrightarrow{p}^i_{A'} = \begin{cases} C[\![c_{1,j}]\!]A & \text{if } p = p_j \in \{p_1, \ldots, p_n\} \\ \Omega & \text{otherwise} \end{cases} \\
& \overleftarrow{p}^i_{A'} = \begin{cases} C[\![c_{2,j}]\!]A & \text{if } p = p_j \in \{p_1, \ldots, p_n\} \\ \Omega & \text{otherwise} \end{cases}
\end{array}
$$

As with the interpretation of commands and expressions, the only significant differences between these definitions and those in the previous chapter pertain to procedures. As in the previous model, operations relating to objects in the original system ($f_1, \ldots, f_{i-1}$, $\overrightarrow{p}_1, \ldots, \overrightarrow{p}_{i-1}$, $\overleftarrow{p}_1, \ldots, \overleftarrow{p}_{i-1}$) are interpreted by extracting the state of the system and applying the corresponding operator from A—the original system's denotation. Functions applied to the new object introduced by m are interpreted in the same way as in the last chapter. However in this model, each procedure p of the new object gives rise to two the operations $\overrightarrow{p}^i$ and $\overleftarrow{p}^j$. Their interpretations in the new system denotation are derived from the interpretations of the corresponding components of the procedure definition.

11.4.6 Proving Congruence

We will omit a detailed proof of the congruence of the model in this chapter; it may be adapted straightforwardly from the proof in the previous chapter.

Chapter 12

Case Study 3: A Dynamic Language

12.1 Introduction

The language studied in this chapter is closer (at least in terms of syntax) to the example language of chapter 2 than those examined in the previous two. It differs from the languages of the last two case-studies in three main respects:

- While languages 1 and 2 were *static*, introducing objects by means of individual declarations, the language studied here is *dynamic*. This means that rather than defining objects, declarations in this language describe *classes*, which may be used to create unlimited numbers of objects.[1]
- Since objects may be created dynamically, it becomes impractical to refer to them by means of explicit identifiers (the indices of languages 1 and 2). Instead, objects gain access to other objects by placing them in special private variables, which we call *object-valued* variables. An object-valued variable may be loaded with a newly-formed object, and its contents may be transferred (or more precisely *copied*—see next item) using a form of assignment command. Objects also have variables for storing booleans and numbers—these are known as *primitive-valued* variables.
- Unlike objects in previous case languages, objects in this language may *not* be shared. An object may be referred to only by one variable of one other object. When assignment occurs between object-valued variables, it is a *distinct copy* of the object in the source variable (and not a reference to it) which is placed in the target.

Syntactically, this language deviates slightly from the example language: First, distinct sets of identifiers are used to name primitive- and object-valued variables (these identifiers

[1] Refer to chapter 2 for more details.

are drawn from sets Id_v and Id_q respectively). Second, in order properly to conform with the framework set out in chapter 9, classes in this language are labelled with (strictly positive) natural numbers, rather than names selected by the programmer.

We should also point out that in eschewing references and shared objects, this language is also significantly dissimilar to most commercially-available OOL's—a more representative language is examined in the next chapter. It does, however, closely resemble the language investigated by Hoare in the seminal paper [Hoa72], where he established a verification method whose application to OOL's has been widely advocated. We will examine the implications of this fact at greater length in chapter 14.

12.2 Syntax

A formal description of this language's syntax follows. Assume the following (mutually disjoint) sets:

$$\begin{array}{ll} n \in \mathbf{N} & \text{—natural numbers} \\ b \in \mathbf{B} & \text{—booleans} \\ v \in \mathrm{Id}_v & \text{—primitive-valued variable identifiers} \\ q \in \mathrm{Id}_q & \text{—object-valued variable identifiers} \\ f \in \mathrm{Id}_f & \text{—function identifiers} \\ p \in \mathrm{Id}_p & \text{—procedure identifiers} \end{array}$$

Then a class-declaration with index i, for all $i \in \mathbf{N}^+$, is of the form:

$$\begin{array}{ll} \textbf{class}\ \ i\ \ \textbf{has} & \\ & \textbf{functions}\ \ f_1 = e_1, \ldots, f_m = e_m \\ & \textbf{procedures}\ \ p_1 : c_1, \ldots, p_n : c_n \end{array}$$

For $i \in \mathbf{N}^+$, and $j \in \{1, \ldots, i-1\}$, typical commands and expressions $c \in \mathrm{Cmd}_i$ and $e \in \mathrm{Exp}_i$ are of the form:

$$\begin{array}{rcl} e & ::= & b \mid n \mid v \mid q.f \\ c & ::= & c_1; c_2 \mid \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid v \leftarrow e \mid q_1 \leftarrow q_2 \mid \\ & & q \leftarrow \textbf{new}\, j \mid q.p \end{array}$$

In most respects other than those indicated above, the syntax of this language resembles that used in the programming example of chapter 2. Note command "$q \leftarrow \textbf{new}\, j$" which places a newly created instance of class j into object-valued variable q. As in the example language, an expression of the form "$q.f$" applies a function f to the object in variable q, and "$q.p$" similarly applies procedure p to the object in q.

Adopting the same approach as was used in the last two chapters, classes do not explicitly declare the primitive- and object-valued variables used in their instances; the definition of an operation may make use of any legal variable name (drawn from sets Id_v or Id_q as appropriate).

In keeping with the notion of objects as "black-boxes", not amenable to direct external scrutiny, we have chosen to use the same algebra for observations results as we did in the previous two chapters: V is simply the set of boolean values and natural numbers, $\mathbf{B} \cup \mathbf{N}$, viewed as a degenerate algebra with only one sort, Val, and no operations. Note in particular that this means that objects—even though they may be held in private variables—are not considered to be valid results of observations.

Since expressions as defined above may produce only booleans or natural numbers, we can use the same formulation for observations in this language as we did in the previous two; observations in Obs_i (for all $i \in \mathbf{N}^+$) are simply pairs of commands and expressions, $(c, e) \in \mathrm{Cmd}_i \times \mathrm{Exp}_i$. However, whereas in languages 1 and 2, observations were executed in an object introduced directly by a new declaration, in this language, they are executed in an *instance* of a newly declared class.

Again, we have derived from the programming notation three of the four components of a modular language (to wit, families $(\mathrm{Mod}_i)_{i \in \mathbf{N}^+}$, $(\mathrm{Obs}_i)_{i \in \mathbf{N}^+}$ and algebra V). As before, we rely on the semantics presented below to defined the forth component—the family of evaluation functions $(\mathrm{eval}_i)_{i \in \mathbf{N}^+}$.

12.3 Preface to the Semantics

Before proceeding with the detailed description of the semantics, we will pause—as on previous occasions—to examine the broad considerations which have shaped it.

The language in this chapter presents two issues which were not examined in the preceding chapters:

- Rather than describing the state of a fixed number of objects, the denotation of a system in this language must provide representations for a potentially unlimited number of instances of classes.

- The model must properly account for the more sophisticated form of *dynamic binding* featured in this language. This term was introduced in chapter 2; it refers to the way in which applying the same operation to instances of different classes may have significantly different results. Dynamic binding figured in the programming exercise in that chapter, where, for example, the effect of applying procedure tick to Bits was quite different from applying tick to $\mathsf{Counter}$s. In languages 1 and 2, since constant indices were used to specify the target object in the application of an operation, it was always apparent from the text which particular version of that operation was required. In this language, operations are applied to the contents of object-valued variables, and these contents may vary over time.

To see how these problems are tackled in a ΣW-semantics, consider representing a system of i classes in this language as an algebra A.

Actually, the fact that an unlimited number of objects may be generated does not cause significant problems. In the model, a denotation such as A will contain a single set named "State", as did the algebras of the previous two models. However rather than representing the state of all the objects in a system, an element of this set represents the state of an instance of a class in the system. In the previous chapters, the store of an object contained only one element of State (bound to $*$), representing the state of the single system upon which it was based. In this model, an object's store may contain any number of elements of State, each representing the state of an object contained in an object-valued variable.

Properly accounting for dynamic binding requires that we take a little care in constructing the set "State". The key is the use of disjoint sums, introduced in section 9.4. Imagine, for the moment that we have two separate sets, S_1 and S_2, which contain representations of the state of instances of classes C_1 and C_2, respectively. Further, let function op_1 (with $\mathrm{dom}(\mathsf{op}_1) = S_1$) be the interpretation of operation op which is appropriate to instances of C_1, and let op_2 be the corresponding interpretation for instances of C_2. Combine sets S_1 and S_2 using a disjoint sum to form a single set S, which represents the states of instances of both classes: $S = S_1 + S_2$. By putting $\mathsf{op}_s = \mathbf{cases}(\mathsf{op}_1, \mathsf{op}_2)$, we produce a single function which may be applied to any element of S. However, the result of such an application depends upon whether that element is drawn from set S_1 or S_2—that is, whether it represents an instance of class C_1 or C_2. In the model, the set State and all the relevant operations of a system's denotation are constructed in just this manner.

12.4 Semantics

We will present a ΣW-semantics for this language just as we did for the previous two. That is to say, we require a family of signatures, $(\Sigma_i)_{i\in\mathbf{N}^+}$, of sets of visible terms $(W_i)_{i\in\mathbf{N}^+}$, a description of the store of a new object in the language, two families of semantic functions

$(C_i)_{i\in\mathbf{N}^+}$ and $(E_i)_{i\in\mathbf{N}^+}$ for commands and expressions respectively, and finally, a family $(M_i)_{i\in\mathbf{N}^+}$ of semantic functions interpreting class-declarations.

12.4.1 Signatures

If we are to produce a congruent ΣW-semantics, then each component signature Σ_i (for all $i \in \mathbf{N}^+$) must be both an extension of $\mathrm{Sig}(V)$—the signature of observation results—and also sufficiently rich to represent all of the interactions between the objects of a system with i class-declarations and a new object based upon it. As usual, we will examine the sorts and operators of Σ_i in turn.

Sorts

Since, by definition, Σ_i must contain the signature $\mathrm{Sig}(V)$ of the algebra of observation results (which, as described above, is an algebra with the single sort Val, and no operations), it must include the sort Val.

The only other sort which Σ_i needs contain is the sort State. This, as we outlined above, names the carrier in a system's denotation whose elements represent the states of instances of the classes in that system.

Operators

Since V has no operations, the stipulation that Σ_i be an extension of $\mathrm{Sig}(V)$ confers no operators on it. The remaining operators in Σ_i are to represent the instances of classes in a system to an observer.

The first requirement is a collection of constants $\mathrm{new}^1, \ldots, \mathrm{new}^i$. Each such new^j (for all $1 \leq j \leq i$) names an element of State which represents the initial state of an instance of the class with index j.

As was the case with languages 1 and 2 (and reflecting the use of run-time typing in the case languages), in producing a semantics for this language, we will assume that any operation may be applied to an instance of any class. This means a representation of all procedures $p \in \mathrm{Id}_p$ and functions $f \in \mathrm{Id}_f$, for each class in the system. We illustrated in the previous section how the use of disjoint sums allowed us to combine the different versions of an operation in a number of classes into a single mathematical function. Thus Σ_i needs only one operator $f \in \mathrm{Id}_f$ for each function-name, and similarly, one operator $p \in \mathrm{Id}_p$ for each procedure.

To summarize, we have, for all $i \in \mathbf{N}^+, p \in \mathrm{Id}_p, f \in \mathrm{Id}_f$:

$$\begin{array}{lll} \Sigma_i \stackrel{\text{def}}{=} & \textbf{sorts} & \mathrm{State}, \mathrm{Val} \\ & \textbf{operators} & \mathrm{new}^1 : \mathrm{State}, \ldots, \mathrm{new}^i : \mathrm{State} \\ & & f : \mathrm{State} \to \mathrm{Val} \\ & & p : \mathrm{State} \to \mathrm{State} \end{array}$$

12.4.2 Visible Terms

The next requirement is the family of sets of visible terms, $(W_i)_{i \in \mathbf{N}^+}$, which determine the algebraic behaviour of system denotations. As before, if we are to produce a congruent semantics, these terms must precisely characterize the interactions between a system and an observer.

In this language, the observer of a system is presented not with a collection of existing objects—as was the case with the static languages—but with a set of classes which may be used to produce new objects on demand. In making use of such a system, the observer may initially create an instance of any of its classes. Recall that the initial state of such an object is named by the constant new^j, where j is the index of the class in question.

Once such an instance has been created, the observer may subject it to a finite number of procedure applications—each named by an operator p (for procedure named $p \in \mathrm{Id}_p$). Finally, the observer may apply a function (named $f \in \mathrm{Id}_f$) to the object, producing a natural number or a boolean value.

Such an interaction, therefore, corresponds to the evaluation of a term of the form $f(p_n(\ldots p_1(\mathrm{new}^j)\ldots))$, where $1 \leq j \leq i$, $p_1, \ldots, p_n \in \mathrm{Id}_p$ and $f \in \mathrm{Id}_f$. It is not difficult to show that—as in the first language we studied—the collection of all such terms is simply the set $T^{\mathrm{v}}_{\Sigma_i,V} = (T_{\Sigma_i})_{\mathrm{Val}}$ of all visible terms on Σ_i. Thus, for all $i \in \mathbf{N}^+$, we have:

$$W_i \stackrel{\text{def}}{=} T^{\mathrm{v}}_{\Sigma_i,V}$$

12.4.3 Stores

The next stage of the presentation involves the description of the state which is accessible to the commands and expressions executed in the observer of a system. You will recall that in the previous chapters, this state comprised the (primitive-valued) variables of a new object, and the state of the objects in the system. In this language, of course, a new object is not defined directly; instead it created as a instance of a newly-defined class. In addition, systems do not offer directly-accessible objects, but rather collections of classes, whose instances may be stored in the (object-valued) variables of such a new object. These latter instances we will term the *component* objects (or simply *components*) of the new object. An object is said to be the *parent* of its component objects.

The state accessible to an object in this language, therefore, is composed of the contents of its own private variables and the states of its components. This means that a store in the model defines the contents of a new object's primitive-valued variables, and the states of the component objects stored in its object-valued variables.

Now if the components are instances of classes drawn from a system whose denotation is the algebra A, the states of these component objects are drawn from the carrier State_A. In addition, observe that the names of primitive-valued variables are drawn from the set Id_v, whereas those of object-valued variables come from Id_q. Thus the store may be written in symbols:

$$\mathrm{sto}(A) \quad \stackrel{\mathrm{def}}{=} \quad (\mathrm{Id}_v \to \mathbf{B} \cup \mathbf{N}) \times (\mathrm{Id}_q \to \mathrm{State}_{(A)})$$

In the initial state of a new object, the contents of all of its variables are undefined, so we have:

$$\mathrm{init}(A) \quad \stackrel{\mathrm{def}}{=} \quad (\Omega, \Omega)$$

12.4.4 Interpreting Commands and Expressions

We must define the semantic functions $(C_i)_{i \in \mathbf{N}^+}$ and $(E_i)_{i \in \mathbf{N}^+}$ which interpret commands and expressions relative to stores such as were defined above and algebras delineated by the signatures described earlier:

$$\begin{array}{l} C_i : \mathrm{Cmd}_i \to \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1}) \to \mathrm{Store}_i \to \mathrm{Store}_i \\ E_i : \mathrm{Exp}_i \to \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1}) \to \mathrm{Store}_i \to V \end{array}$$

We proceed in the same style as before:

$$\begin{array}{lcl} C[\![c_1; c_2]\!]A & = & C[\![c_1]\!]A; C[\![c_2]\!]A, \\ C[\![\,\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!]A & = & E[\![e]\!]A \to C[\![c_1]\!]A,\ C[\![c_2]\!]A, \\ C[\![v \leftarrow e]\!]A & = & \{E[\![e]\!]A/v\}, \\ C[\![q_1 \leftarrow q_2]\!]A & = & \{\mathbf{valueof}(q_2)/q_1\}, \\ C[\![q \leftarrow\ \mathbf{new}\ j]\!]A & = & \{\mathbf{const}(\mathrm{new}^j_A)/q\}, \\ C[\![q.p]\!]A & = & \{\mathbf{valueof}(q); p_A/q\} \end{array}$$

$$\begin{array}{lcl} E[\![v]\!]A & = & \mathbf{valueof}(v), \\ E[\![b]\!]A & = & \mathbf{const}(b), \\ E[\![n]\!]A & = & \mathbf{const}(n), \\ E[\![q.f]\!]A & = & \mathbf{valueof}(q); f_A. \end{array}$$

Interpretations of commands of the form "$c_1; c_2$", "**if** e **then** c_1 **else** c_2", and "$v \leftarrow e$" are derived in the same way as they were in languages 1 and 2. In interpreting an assignment between two object-valued variables, we simply copy the representation of the state of the object in the source variable into the target variable. This signifies that the object in the first has been copied into the second; any operation applied to one will not affect the other. To create a new component object of class j in the system, to be stored in variable q, we invoke the operator new^j in the system's denotation, and bind the resulting initial state representation to q. The interpretation of a procedure application of the form $q.p$ first extracts the representation of the state of the object in q, applies the representation of p to it—so as to produce the new state of that object—and stores the result back in q. Expressions v, b and n are interpreted as before. To apply a function to an object, using an expression of the form $q.f$, we extract the state of the object, and then apply the representation of f provided by the system denotation.

12.4.5 Interpreting Modules

To see how module definitions are interpreted, note that a typical $m \in \text{Mod}_i$, for any $i \in \mathbf{N}^+$, is of the form:

$$\begin{array}{ll} \textbf{class}\ i\ \textbf{has} & \\ & \textbf{functions}\ f_1 = e_1, \ldots, f_m = e_m \\ & \textbf{procedures}\ p_1 : c_1, \ldots, p_n : c_n \end{array}$$

Now, for any $A \in \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1})$, let $M_i[\![m]\!]A$ be the algebra $A' \in \mathbf{Alg}(\Sigma_i, V, W_i)$, such that:

$$\begin{array}{ll} \textbf{carriers} & \text{State}_{A'} = \text{sto}(A) + \text{State}_{(A)} \\ \textbf{operations} & \text{new}^1_{A'} = \text{new}^1_A; \text{in}_2, \ldots, \text{new}^{i-1}_{A'} = \text{new}^{i-1}_A; \text{in}_2, \\ & \text{new}^i_{A'} = \text{init}(A); \text{in}_1, \\ & f_{A'} = \begin{cases} \textbf{cases}(E[\![e_j]\!]A,\ f_{(A)}) & \text{if}\ \ f = f_j \in \{f_1, \ldots, f_m\} \\ \textbf{cases}(\Omega,\ f_{(A)}) & \text{otherwise} \end{cases} \\ & p_{A'} = \begin{cases} \textbf{cases}(C[\![c_j]\!]A; \text{in}_1,\ p_{(A)}; \text{in}_2) & \text{if}\ \ p = p_j \in \{p_1, \ldots, p_n\} \\ \textbf{cases}(\Omega,\ p_{(A)}; \text{in}_2) & \text{otherwise} \end{cases} \end{array}$$

Recall that if A represents a system consisting of i classes, then A' is the system obtained by adding to it the class defined in m. To begin with, A' must contain (in carrier $\text{State}_{A'}$) the representations of the states of the new class's instances. In order that we can use the disjoint-sum construction (as described in the introduction to the semantics) properly to discriminate between (the states of) instances of the new class and instances of classes already in the system, this set is formed as the disjoint sum of the stores described above (representing states of the instances of the new class) and the carrier State_A drawn from the original system denotation (which contains representations of states of the instances of classes in the original system). To define the constants $\mathrm{new}^1 \ldots \mathrm{new}^{i-1}$, we can use corresponding operators from the system, but their results must be injected into the second component of the new state carrier. The constant new^i—the initial state of the new class's instances—is simply the initial state of the store, injected into the other component of the sum (so that it may be distinguished from instances of the other classes). Forming the interpretation of each function f and procedure p amounts to adding a new case to their representations in the original system which deals with the new component of the State carrier. If there is a definition for the operation in question in the new class, then this new case is its interpretation—if there is no such definition, then the new case is an undefined function. Note that we must preserve the separation of states by the use of appropriate injections in the interpretation of procedures.

12.4.6 Proving Congruence

Again, we will omit the proof that the model presented in this chapter is congruent—it is straightforward, and represents a simplification of the proof given in the next chapter.

Chapter 13

Case Study 4: A Dynamic Language with Shared Objects

13.1 Introduction

The language studied in this chapter combines the shared objects and references of languages 1 and 2 with the class-based dynamic object creation of language 3. Though it still lacks many of the features of the OOL's in everyday use, the resulting language is significantly more realistic than those we have studied up to now. Unfortunately, as we will see, producing a congruent ΣW-semantics for it is significantly more complicated than was the case for any of its forbears.

Superficially, this OOL is similar to that of the last chapter; modules define classes (each with an index), which may be used to create arbitrary numbers of objects. Objects have two sorts of variables—those containing booleans or numbers, and those used to access other objects. However, rather than accessing an object directly, all interactions in this language are mediated using references, as in languages 1 and 2. Thus the variables used to access objects now contain *references* to them, not the objects themselves. For this reason, we will call such variables *reference-valued.* When an assignment takes place between reference-valued variables, the *reference* in the source is placed in the target, so that after the assignment, both variables now refer to the same object. (Contrast this with assignments between object-valued objects in the previous language, where two distinct copies of an object were generated.)

References may be further propagated by the use of *reference-valued functions.* Such a function, applied to an object, returns the reference be found in a given reference-valued variable of the object. Objects also have functions returning booleans or numbers, like those of previous languages—here they are named *primitive-valued functions.*

Since objects are created dynamically, it is far from convenient (and unrealistic) to use explicit references such as were used in the static languages we studied. Instead, a reference to an object is generated when the object is created—it is this reference, and not the object itself, which is returned to the object's creator. The reference may be passed between the reference-valued variables of the creator, or to the creator's parent[1] by means of reference-valued functions. It is important to note that *all* of the references used in this language must be produced in this way—the language offers no means by which natural numbers (for example) may be used as references, as was possible in languages 1 and 2.

We should also note that references may only be passed between the variables of a single object, or from objects to their parents. There are no means (such as the ability to use references as parameters to functions or procedures) for passing references from an object to its components—nor is there any way in which an object may be provided with a reference to itself. This excludes the possibility of cycles of references, and the attendant problems described in chapter 12.

13.2 Syntax

The syntax of this language—as we indicated above—is similar to that of the language studied in the last chapter. Slight extensions to the syntax of class-definitions and commands enable reference-valued functions to be defined and applied. The names of reference-valued variables and functions are are distinct from those of primitive valued variables and functions. Assume the following disjoint sets:

$n \in \mathbf{N}$	—natural numbers
$b \in \mathbf{B}$	—booleans
$v \in \mathrm{Id}_v$	—primitive-valued variable identifiers
$q \in \mathrm{Id}_q$	—reference-valued variable identifiers
$f \in \mathrm{Id}_f$	—primitive-valued function identifiers
$g \in \mathrm{Id}_g$	—reference-valued function identifiers
$p \in \mathrm{Id}_p$	—procedure identifiers

For all $i \in \mathbf{N}^+$, a typical module $m \in \mathrm{Mod}_i$ is a class definition of the form:

class i **has**
value functions $f_1 = e_1, \ldots, f_m = e_l$
reference functions $g_1 = q_1, \ldots, g_m = q_m$
procedures $p_1 : c_1, \ldots, p_n : c_n$

[1]Terms *parent* and *component* are defined by analogy with their use in the previous chapter. Thus an object is termed the parent of another if it contains a reference to the latter. In such a situation, the latter object is a component of the former.

Note that a reference-valued function is defined by giving the name of the (reference-valued) variable whose contents it is to return.

Commands and expressions are such that for $i \in \mathbf{N}^+$ and j ranging over $1, \ldots, i-1$, typical elements $e \in \mathrm{Exp}_i$ and $c \in \mathrm{Cmd}_i$ are of the form:

$$\begin{aligned} e &::= b \mid n \mid v \mid q.f \\ c &::= c_1; c_2 \mid \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid v \leftarrow e \mid q_1 \leftarrow q_2 \mid \\ & q \leftarrow \textbf{new } j \mid q.p \mid q_1 \leftarrow q_2.g \end{aligned}$$

Most of these constructs are familiar from the last chapter. Commands of the form "$q_1 \leftarrow q_2.g$" are used to apply a reference-valued function g to the object referred to by (the reference in) variable q_2, placing the resulting reference in q_1.

We have a modular language whose modules are the class-declarations in the OOL presented, with sets of observations pairs of commands and expressions. Again, the algebra V of observation results, is $\mathbf{B} \cup \mathbf{N}$, regarded as an algebra with a single sort, Val.

13.3 Preface to the Semantics

To see why the semantics of this language are complicated, consider the following example system, which defines versions of the classes Bit and Cell introduced in chapters 2 and 7 respectively.

class Bit **has variables** state
    value functions value = state
    procedures init: state ← 0
        tick: **if** state = 0 **then** state ← 1 **else** state ← 0

class Cell **has variables** bit
    value functions value = bit.value
    reference functions bitRef = bit
    procedures init: bit ← **new** Bit;
            bit.init
        tick: bit.tick

The class Bit, as before, describes objects which simulate two-state switches. Their internal state is made available through the value function, and they are forced into a fixed state by the procedure init. The state of a Bit may be inverted by application of procedure tick. Cells are merely containers for Bits. When procedure init is applied to a new Cell, a reference to a new Bit is loaded into the Cell's reference-valued variable bit and initialized. This Bit may be tick'ed by applying tick to the Cell itself. The state of the Bit referred to by a Cell may be retrieved by applying function value to the Cell. In addition, a Cell also provides a reference-valued function bitRef, which returns the reference which the Cell holds to its Bit.

Now compare the results of applying the following two observations which, as you will recall, are simply pairs of commands and expressions. (We have placed quotes around commands and expressions in the interests of clarity):

$$(\ulcorner \mathsf{cell} \leftarrow \textbf{new}\ \mathsf{Cell};\ \mathsf{cell.init} \urcorner,\ \ulcorner \mathsf{cell.value} \urcorner)$$

$$(\ulcorner \mathsf{cell} \leftarrow \textbf{new}\ \mathsf{Cell};\ \mathsf{cell.init};\ \mathsf{bitR} \leftarrow \mathsf{cell.bitRef};\ \mathsf{bitR.tick} \urcorner,\ \ulcorner \mathsf{cell.value} \urcorner)$$

The first of these observations prompts the creation of a new Cell, a reference to which is placed in variable cell of the observer. The Cell is initialized, and the value of the observation is the result of subsequently applying function value to it. This value is 0.

The second observation also creates and initializes a Cell. At this point, a reference which alludes to the Bit referred to by the Cell is extracted from the Cell and placed in the observer's variable bitR. Note that the Bit is now shared by the Cell and the observer. Thus when the command bitR.tick is executed in the observer, it affects the state of the same Bit that the Cell uses in computing the result of the function value. This means that the value of the observation is 1.

The difference between the results of these two observations is another manifestation of the effect we first observed in language 1; in the presence of references it is possible change the result of applying a function to an object by applying a procedure to a different object. In this particular example, the fact that the value of the Cell is changed by tick'ing the Bit accounts for the discrepancy observed.

Recall that in the static languages, because an observer had access to references to all of the objects in a system, we had to give a proper account by treating all of those objects as one unit, modified and inspected as a whole by the representations of operations on objects. To distinguish the particular object in the system to which the representation of an operation applied, we decorated it with the index of that object. Thus the representations of procedures were of the form p^i : State $\rightarrow$ State, and those of (primitive-valued) functions of the form f^i : State $\rightarrow$ Val, where i was the index of the object, "State" the set whose elements described the state of the system as a whole, and "Val" the set of booleans and natural numbers.

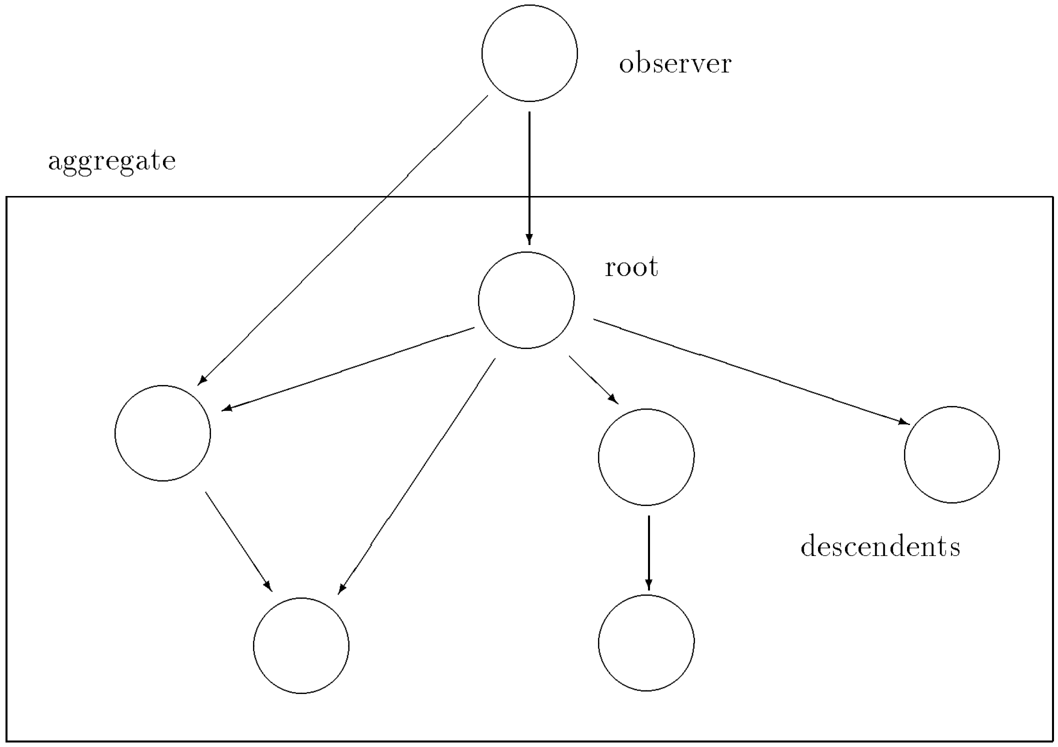

Figure 13.1: Observing an Aggregate

We will adopt a similar approach here. In this language, of course, systems contain classes, not objects, and an observer inspects instances of them. In this case, the references available to the observer of an instance comprise the reference to the object itself (returned when the object is created), the references to the component objects held in its variables (accessible using reference-valued functions), the references in those component objects (drawn out by applying reference-valued functions to them), the components of those components, and so on.[2] The original object, from which all the references are derived in this way, we will term the *root* object, and we will refer to the other objects as *descendents* of the root. We will use the term *aggregate* to describe a root object and all of its descendents, considered as a single unit. In the model, objects are not considered in isolation, but rather as members of aggregates. For example, rather than creating an object and returning a reference to it, we think instead of creating an aggregate, and returning a reference to its root object.

Therefore the state of the aggregate containing a root object and all of its descendents is treated as a whole. In order to specify to which particular object an operation's representation applies, we need to supply a reference to that object. As we will show, it is

[2]In the example, the references available to the observer of a Cell are the reference of the Cell and that of the Bit it holds.

possible to collect into a single set all the references (or more precisely, representations of references) used to specify members of all aggregates created by a particular system. This set is named Ref. Rather than decorating operations with references (as we did in the models of the static languages), it is more convenient in this case to make an operation a binary function, one of whose arguments represents the state of an aggregate, while the other argument is a reference to the object in the aggregate to which the operation is applied. Assuming that the states of aggregates are drawn from the set State, representations of procedures in this language are of the form $p : \mathrm{State} \times \mathrm{Ref} \to \mathrm{State}$, those of primitive valued of the form $f : \mathrm{State} \times \mathrm{Ref} \to \mathrm{Val}$, and of reference-valued functions of the form $g : \mathrm{State} \times \mathrm{Ref} \to \mathrm{Ref}$. Since classes will in general give different definitions to operations with the same name, we will contruct sets State and Ref using disjoint sums as we did in the last chapter.

Finally, we should note another aspect of the way in which references are used in this language. In the static languages, an observer could access any object in a system, simply by quoting the natural number which was its index. In this case, however, when the observer creates a new object, only the reference to the object itself is returned; references to its descendents may only be extracted by the application of reference-valued functions. We will see later on how this fact affects the sets of terms required to produce a congruent ΣW-semantics.

13.4 Semantics

We will use the usual form of presentation for this model; first a family of signatures, $(\Sigma_i)_{i \in \mathbf{N}^+}$, then sets of visible terms $(W_i)_{i \in \mathbf{N}^+}$, a description of the state accessed by an observer, two families of semantic functions $(C_i)_{i \in \mathbf{N}^+}$ and $(E_i)_{i \in \mathbf{N}^+}$ for commands and expressions respectively, and finally, a family $(M_i)_{i \in \mathbf{N}^+}$ of semantic functions interpreting class-declarations.

13.4.1 Signatures

As before, each component signature, Σ_i (for all $i \in \mathbf{N}^+$) must be an extension of the signature $\mathrm{Sig}(V)$, of observation results. It must also contain all the sorts and expressions needed to account for the interactions between observer and system we illustrated above.

Sorts

In the usual way, Σ_i must contain the sort Val if it is to be an extension of $\mathrm{Sig}(V)$.

It must also contain the sorts State and Ref described in the previous section; State names a set whose elements represent the aggregate state of an object and all of its descendents, while members of Ref are references which may be used to identify particular objects in such an aggregate.

Operators

As usual, Sig(V) confers no operations on Σ_i.

As in the previous model, we need constants $\text{new}^1, \ldots, \text{new}^i$. In this model, new^j (for all $1 \leq j \leq i$) represents the initial state of a newly-created object of class j and all of its descendents (i.e. an aggregate whose root is of class j). In addition to this, we also require constants $\text{root}^1, \ldots, \text{root}^i$. Each root^j represents the reference returned when an object is created; it is used to identify the root object in an aggregate.

The remaining operators in Σ_i represent the functions (primitive- and reference-valued) and procedures of the classes in a system. Making our established assumption that any operation may be applied to any object, the discussions in the preface entail the inclusion of functions $f : \text{State} \times \text{Ref} \rightarrow \text{Val}$, for all primitive-valued function names $f \in \text{Id}_f$, $g : \text{State} \times \text{Ref} \rightarrow \text{Ref}$ for all reference-valued function names $g \in \text{Id}_g$, and $p : \text{State} \times \text{Ref} \rightarrow \text{State}$ for all procedure names $p \in \text{Id}_p$.

Thus, for all $i \in \mathbf{N}^+, p \in \text{Id}_p, g \in \text{Id}_g, f \in \text{Id}_f$:

$$
\begin{array}{lll}
\Sigma_i \stackrel{\text{def}}{=} & \textbf{sorts} & \text{State}, \text{Ref}, \text{Val} \\
& \textbf{operators} & \text{new}^1 : \text{State}, \ldots, \text{new}^i : \text{State} \\
& & \text{root}^1 : \text{Ref}, \ldots, \text{root}^i : \text{Ref} \\
& & f : \text{State} \times \text{Ref} \rightarrow \text{Val} \\
& & g : \text{State} \times \text{Ref} \rightarrow \text{Ref} \\
& & p : \text{State} \times \text{Ref} \rightarrow \text{State}
\end{array}
$$

13.4.2 Visible Terms

We have noted on several occasions that in producing congruent ΣW-semantics, great care should be taken to ensure that the families of visible terms which determine the behaviour of system denotations characterize *precisely* the interactions between an observer and the system it is inspecting. In this instance, we must take into account the remark made at the end of the preface, concerning the way in which references are extracted from aggregates.

Recall how we characterized observer/system interactions in the previous case-study; an observer could create a new instance of a class in a system, subject it to a sequence of procedure applications, and then apply a function to produce a visible result. One would suspect that the interactions in this language are analogous; an observer creates an aggregate, subjects the objects in it to a series of procedure applications, supplying references to identify the object involved in each case, and finally applies a primitive-valued function to extract a visible result. Unfortunately, unless the references supplied are very carefully delimited, it is easy to produce sets of visible terms too large to produce a congruent model.

To see why this is, consider how an observer might apply a particular operation (be it a primitive-valued function, a reference-valued function or a procedure) to an object in an aggregate. In the model (assuming that the system observed has i classes), this might be represented by a term of the form $\alpha(s, r)$, where term $s \in (T_{\Sigma_i})_{\text{State}}$ represents the state of the aggregate, $r \in (T_{\Sigma_i})_{\text{Ref}}$ stands for the reference to the target object of the application, and α is the representation of the operation. As we pointed out in the preface, the reference used in this application is either root^i, or it has been obtained by applying a reference-valued function to the aggregate either in its present state or some state through which it has already passed. No other references to objects in the aggregate are available to the observer.

To reflect this stricture in our model, we first require some means of taking a term in $(T_{\Sigma_i})_{\text{State}}$ describing the state of an aggregate and deriving formally those terms (also in $(T_{\Sigma_i})_{\text{State}}$) which represent states through which the aggregate must have passed. Hence the following definitions:

Definition 55 *Let Σ be an $\mathcal{S}$-sorted signature, and t, t' be Σ-terms. Then t' is an* immediate subterm *of t iff t is of the form $\sigma(t_1, \ldots, t_n)$, for $\sigma \in \text{Ops}(\Sigma), n \in \mathbf{N}$, and there is an $i \in \{1, \ldots, n\}$ such that $t_i = t'$.*

A subterm *of a term T is either t itself, or a subterm of an immediate subterm of t.*

An s-subterm *of t is a subterm t' such that $\text{sort}(t) = \text{sort}(t')$.*

$t' \leq t \overset{\text{def}}{\iff} t'$ *is an s-subterm of t.*

$t' < t \overset{\text{def}}{\iff} t' \leq t \wedge t' \neq t.$

As required, if $s' < s$, for $s, s' \in (T_{\Sigma_i})_{\text{State}}$, then s' describes an earlier state of the object whose state is described by s.

Now for each $s \in (T_{\Sigma_i})_{\text{State}}$, we define a set $F(s)$, such that if s describes the state of an aggregate, then $F(s)$ contains—in addition to the constant root^i, representing the reference to the root of the aggregate—all those terms which represent applications of reference-valued functions in states which preceded s. As pointed out above, these terms describe all of the references available to the observer.

Definition 56 *For* $s \in (T_{\Sigma_i})_{\text{State}}$*, define* $F : (T_{\Sigma_i})_{\text{State}} \to \mathcal{P}((T_{\Sigma_i})_{\text{Ref}})$ *such that for all* $s \in (T_{\Sigma_i})_{\text{State}}$*,* $F(s)$ *is the smallest set satisfying:*

- $\text{new}^j \leq s \;\Rightarrow\; \text{root}^j \in F(s)$, for $1 \leq j \leq i$
- $s' \leq s \;\wedge\; r \in F(s') \;\Rightarrow\; g(s', r) \in F(s)$

Given that only these feasible references may be used to modify the state of an object, we can now define a predicate "Reach", which characterizes the set of states which might arise during the course of an observation:

Definition 57 *For* $s \in (T_{\Sigma_i})_{\text{State}}$*, define* $\text{Reach}(s)$ *inductively:*

- $\text{Reach}(\text{new}^j)$, for $1 \leq j \leq i$
- $\text{Reach}(s) \wedge r \in F(s) \;\Rightarrow\; \text{Reach}(p(s, r))$

Now we are able properly to restrict the set of visible terms so as to produce a behavioural semantics. The terms we require describe primitive-valued function applications $f(s, r)$, where s represents a state which may arise during an observation, and r a reference available to the observer in that state:

$$W_i = \{f(s, r) \;\mid\; f \in \text{Id}_f \wedge s \in (T_{\Sigma_i})_{\text{State}} \wedge \text{Reach}(s) \wedge r \in F(s)\}$$

13.4.3 Stores

We come to the definition of the store which describes the state accessible to the commands and expressions of an observing object. In this language, this state includes the values of the observer's primitive-valued variables and the objects referred to by its reference-valued variables (the latter, you will recall, are represented by the aggregates in which they figure).

In the previous model, we were able to bind object-valued variables directly to their contents. In this language, however, two different reference-valued variables may identify the same object, or more generally, two objects in the same aggregate: we saw in the example of section 13.3 (where variables cell and bitR both referred to objects in the same aggregate) that this can lead to interference. In order to account for such effects, we use a form of object-memory like those we described in section 7.7.1. Despite our insistence that the use of global object memories undermines the abstractness of previous

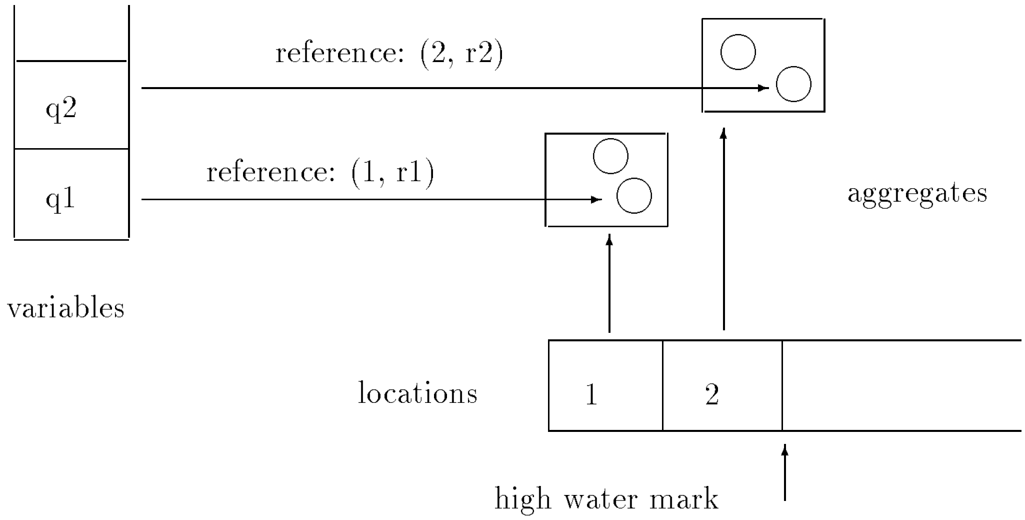

Figure 13.2: Aggregates and References in Stores

models for OOL's, those used here are strictly local to each object, and are carefully "concealed" within the algebras comprising class denotations—so we are still able to produce an abstract semantics using the term realization construction.

In the stores used in this model, the states of aggregates created by an observer are bound to locations represented by natural numbers. The store contains a counter (bound to the constant $*$) known as the *high water-mark*. This indicates the location reserved for the next aggregate to be created, and is incremented when creation takes place.

The reference-valued variables of the observer contain references to objects in the aggregates in the store. If we assume that the system upon which the observer is based has the algebra A as its denotation, then we know that the references which may be used to access objects in the aggregates created by its classes are to be found in carrier Ref_A. Thus to identify a particular object in an aggregate created by the observer, we need to supply a natural number giving the aggregate's location, and a member of Ref_A specifying the actual object within that aggregate. Thus in the store, the contents of reference-valued variables are drawn from the product $\mathbf{N} \times \mathrm{Ref}_A$. (The treatment of aggregates and references is depicted in the figure.)

Finally, the primitive-valued variables of the observer are bound to booleans and natural numbers, as in previous models. Thus we have:

$$
\begin{array}{lcll}
\mathrm{sto}(A) & \stackrel{\mathrm{def}}{=} & (\mathrm{Id}_v \to \mathbf{B} \cup \mathbf{N}) \times & \text{—primitive-valued variables} \\
 & & (\mathrm{Id}_q \to \mathbf{N} \times \mathrm{Ref}_{(A)}) \times & \text{—reference-valued variables} \\
 & & (\mathbf{1} \to \mathbf{N}) \times & \text{—high water mark} \\
 & & (\mathbf{N} \to \mathrm{State}_{(A)}) & \text{—immediate components}
\end{array}
$$

In the store's initial state, the values of all variables and locations are undefined, and the high water mark is set to its initial value:

$$\text{init}(A) \quad \stackrel{\text{def}}{=} \quad (\Omega, \Omega, \Omega[1/*], \Omega)$$

13.4.4 Interpreting Commands and Expressions

We must define the semantic functions $(C_i)_{i \in \mathbf{N}^+}$ and $(E_i)_{i \in \mathbf{N}^+}$ which interpret commands and expressions:

$$\begin{aligned} C_i &: \text{Cmd}_i \to \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1}) \to \text{Store}_i \to \text{Store}_i \\ E_i &: \text{Exp}_i \to \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1}) \to \text{Store}_i \to V \end{aligned}$$

In this model, we have (for $1 \le j < i$):

$$\begin{array}{lcl} C[\![c_1; c_2]\!]A & = & C[\![c_1]\!]A; C[\![c_2]\!]A, \\ C[\![\,\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!]A & = & E[\![e]\!]A \to C[\![c_1]\!]A,\ C[\![c_2]\!]A, \\ C[\![v \leftarrow e]\!]A & = & \{E[\![e]\!]A/v\}, \\ C[\![q_1 \leftarrow q_2]\!]A & = & \{\mathbf{valueof}(q_2)/q_1\}, \\ C[\![q \leftarrow\ \mathbf{new}\ j]\!]A & = & \{\mathbf{valueof}(*); \text{succ}/*\}; \{\mathbf{const}(\text{new}_A^j)/\mathbf{valueof}(*)\}; \\ & & \{\mathbf{tuple}(\mathbf{valueof}(*),\ \mathbf{const}(\text{root}_A^j))/q\}, \\ C[\![q_1 \leftarrow q_2.g]\!]A & = & \{\mathbf{tuple}(\text{id},\ \mathbf{valueof}(q_2)); \text{apply}_A^g/q_1\} \\ C[\![q.p]\!]A & = & \mathbf{tuple}(\text{id},\ \mathbf{valueof}(q)); \text{apply}_A^p \\ & & \\ E[\![v]\!]A & = & \mathbf{valueof}(v), \\ E[\![b]\!]A & = & \mathbf{const}(b), \\ E[\![n]\!]A & = & \mathbf{const}(n), \\ E[\![q.f]\!]A & = & \mathbf{tuple}(\text{id},\ \mathbf{valueof}(q)); \text{apply}_A^f. \end{array}$$

where for all $f \in \text{Id}_f, g \in \text{Id}_g, p \in \text{Id}_p$:

$$\begin{array}{lcl} \text{apply}_A^f & \stackrel{\text{def}}{=} & \lambda\mu, (n, r).\ f_A(\mathbf{valueof}(n)\mu, r) \\ \text{apply}_A^g & \stackrel{\text{def}}{=} & \lambda\mu, (n, r).\ (n, g_A(\mathbf{valueof}(n)\mu, r)) \\ \text{apply}_A^p & \stackrel{\text{def}}{=} & \lambda\mu, (n, r).\ \mu[p_A(\mathbf{valueof}(n)\mu, r)/n] \end{array}$$

Many of these equations—those interpreting sequences of commands, conditional commands, assignments to primitive-valued variables, and expressions consisting of constants or primitive-valued variables—are familiar from previous models.

Note that while in essence, the interpretation of a command $q_1 \leftarrow q_2$ is the same as in the model of the last language studied ("copy the contents of q_2 into q_1"), the effect is quite different. In the previous model, a distinct copy the object contained in q_2 was placed in q_1. In this model, the *reference* in q_2 is loaded into q_1; after the assignment, both variables refer to the same object.

A command of the form $q \leftarrow \mathbf{new}\, j$ creates an instance of class j (that is the root object of a new aggregate) and places a reference to it in the variable q. Its interpretation first increments the high water-mark in the store, and then binds the initial state of the new aggregate to the location produced. The final contents of the variable is a pair consisting of the location of the aggregate and a reference to its root object.

In statement $q_1 \leftarrow q_2.g$, the reference-valued function g is applied to the object referred to by q_2, and the result placed in q_1. In its interpretation, the two components of the pair in q_2 are separated, producing the location of an aggregate and a reference to an object within it. These are passed to the auxiliary apply^g_A, which returns a new pair, consisting of the location of the aggregate itself (since the result must identify an object within the same aggregate as the target of the function application), and the reference given by the representation of g in A when applied to the state of the aggregate and the reference originally in q_2.

Statement $q.p$ applies procedure p to the object referred to by q. The contents of q are separated as above into the location of and aggregate and a reference to an object within it, and passed to auxiliary apply^p_A. The auxiliary updates the state of the aggregate by applying the representation of p to it and to the reference in q.

Expressions of the form $q.f$ are interpreted similarly.

13.4.5 Interpreting Modules

To finish the model, we need to supply the functions $(M_i)_{i \in \mathbf{N}^+}$ which produce the denotations of modules. A typical $m \in \text{Mod}_i$, for any $i \in \mathbf{N}^+$, is of the form:

$$
\begin{array}{ll}
\textbf{class} \;\; i \;\; \textbf{has} & \\
& \textbf{value functions} \;\; f_1 = e_1, \ldots, f_m = e_l \\
& \textbf{reference functions} \;\; g_1 = q_1, \ldots, g_m = q_m \\
& \textbf{procedures} \;\; p_1 : c_1, \ldots, p_n : c_n
\end{array}
$$

For any $A \in \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1})$, we will define the algebra $A' \in \mathbf{Alg}(\Sigma_i, V, W_i)$ such that $A' = M_i[\![m]\!]A$.

To begin with, we see from the definition of the signatures $(\Sigma_i)_{i \in \mathbf{N}^+}$ that A' must contain carriers $\text{Ref}_{A'}$ and $\text{State}_{A'}$. Elements of the first of these carriers, $\text{Ref}_{A'}$, represent references to objects in the aggregates formed when instances of the classes in the new system are created. The second carrier, $\text{State}_{A'}$, contains representations of the states of those aggregates.

These latter elements may be derived, as in the previous model, by forming the disjoint sum of the set of stores described above and the carrier State_A from the denotation of the original system. Here the stores are used as representations of aggregates formed by instances of the new class; the disjoint sum allows the correct implementations of operations to be selected according to the class of the objects to which they apply.

To form $\mathrm{Ref}_{A'}$, we use another disjoint sum. The second component of this sum is the set of references Ref_A from the original system denotation. These are used to refer to objects in aggregates formed using classes in the old system. The first component must contain references identifying objects in the aggregates formed by instances of the new class. The states of these new aggregates are represented by stores, as described above. We can think of such a store as representing the internal memory of the root object of an aggregate (the bindings it gives to members of Id_v and Id_q reflect the contents of the root object's variables), and the descendents of the root (in the shape of the aggregates in the store). Now the reference-valued variables of the root object are used to identify its descendents. We saw that the contents of such a variable could be represented in the store using a pair, the first element of which was the location of the aggregate (in the store) in which the component occurred, while the second element was the reference to the object within that aggregate. In fact, we will use the same representation in the new system to refer to the descendents of the root object. In addition, we require a reference to the root object itself, represented by the distinguished constant "$*$".

In brief, then, we have established the following:

$$\begin{array}{ll}\textbf{carriers} & \mathrm{State}_{A'} = \mathrm{sto}(A) + \mathrm{State}_{(A)} \\ & \mathrm{Ref}_{A'} = \mathrm{ref}(A) + \mathrm{Ref}_{(A)}\end{array}$$

where $\mathrm{ref}(A)$ is defined:

$$\mathrm{ref}(A) \quad \stackrel{\mathrm{def}}{=} \quad \mathbf{1} + (\mathbf{N} \times \mathrm{Ref}_{(A)})$$

Next we turn to the constants $\mathrm{new}^1_{A'}, \ldots, \mathrm{new}^i_{A'}$ and $\mathrm{root}^1_{A'}, \ldots, \mathrm{root}^i_{A'}$ which must be provided according to the signature of A'. Of these, $\mathrm{new}^1_{A'}, \ldots, \mathrm{new}^i_{A'}$ may be defined just as we did in the previous chapter—$\mathrm{new}^1_{A'}, \ldots, \mathrm{new}^{i-1}_{A'}$ are simply the injections into the second component of $\mathrm{State}_{A'}$ of the corresponding values taken from A, while new^i is the initial state of the store, injected into the first component. Operations $\mathrm{root}^1, \ldots, \mathrm{root}^i$ (which, recall, represent references to the root objects of aggregates formed by the system) may be treated similarly; $\mathrm{root}^1, \ldots, \mathrm{root}^{i-1}$ are the references from the original system's denotation, injected appropriately, and root^i is the injection of the new root reference, $*$:

$$\begin{array}{ll}\textbf{operations} & \mathrm{new}^1_{A'} = \mathrm{new}^1_A; \mathrm{in2}, \ldots, \mathrm{new}^{i-1}_{A'} = \mathrm{new}^{i-1}_A; \mathrm{in2} \\ & \mathrm{root}^1_{A'} = \mathrm{root}^1_A; \mathrm{in2}, \ldots, \mathrm{root}^{i-1}_{A'} = \mathrm{root}^{i-1}_A; \mathrm{in2} \\ & \\ & \mathrm{new}^i_{A'} = \mathrm{init}(A); \mathrm{in1} \\ & \mathrm{root}^i_{A'} = *; \mathrm{in1}; \mathrm{in1}\end{array}$$

Next, operations $f_{A'}$, $g_{A'}$ and $p_{A'}$ must be defined on the product $\mathrm{State}_{A'} \times \mathrm{Ref}_{A'}$. From the definitions, we see that:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathrm{State}_{A'} \times \mathrm{Ref}_{A'}} \\
= & (\mathrm{sto}(A) + \mathrm{State}_{(A)}) \times (\mathrm{ref}(A) + \mathrm{Ref}_{(A)}) \\
= & \mathrm{sto}(A) \times \mathrm{ref}(A) + \mathrm{State}_{(A)} \times \mathrm{ref}(A) + \mathrm{sto}(A) \times \mathrm{Ref}_{(A)} + \mathrm{State}_{(A)} \times \mathrm{Ref}_{(A)} \\
= & \mathrm{sto}(A) \times (\mathbf{1} + (\mathbf{N} \times \mathrm{Ref}_{(A)})) + \mathrm{State}_{(A)} \times \mathrm{ref}(A) + \mathrm{sto}(A) \times \mathrm{Ref}_{(A)} + \\
= & \mathrm{State}_{(A)} \times \mathrm{Ref}_{(A)} \\
= & \mathrm{sto}(A) \times \mathbf{1} + \mathrm{sto}(A) \times (\mathbf{N} \times \mathrm{Ref}_{(A)}) + \mathrm{State}_{(A)} \times \mathrm{ref}(A) + \mathrm{sto}(A) \times \mathrm{Ref}_{(A)} + \\
 & \mathrm{State}_{(A)} \times \mathrm{Ref}_{(A)}
\end{array}
$$

For convenience's sake, let α be a member of Id_f, Id_g, or Id_p (representing, respectively, a primitive-valued function, a reference-valued function, or a procedure). We define $\alpha_{A'}$ using five component operations, corresponding to the five components of the disjoint sum expansion given above. Thus:

$$\alpha_{A'} \stackrel{\mathrm{def}}{=} \mathbf{cases}((\alpha^1)_{A'}, (\alpha^2)_{A'}, (\alpha^3)_{A'}, (\alpha^4)_{A'}, (\alpha^5)_{A'})$$

We will define each of these components individually.[3]

Operators of the form α^1 are applied to the state of an aggregate formed by the new class, and the reference to its root object. The effect of these operators is given by the definition of the new class, as in previous chapters:

$$
\begin{array}{lcll}
(f^1)_{A'} & = & \mathrm{pr1}; E[\![e_j]\!]A & \text{if } f = f^j \in \{f_1, \ldots, f_l\} \\
(g^1)_{A'} & = & \mathrm{pr1}; \mathbf{valueof}(q_j); \mathrm{in1}; \mathrm{in2} & \text{if } g = g_j \in \{g_1, \ldots, g_m\} \\
(p^1)_{A'} & = & \mathrm{pr1}; C[\![c_j]\!]A; \mathrm{in1} & \text{if } g = g_j \in \{g_1, \ldots, g_m\} \\
(\alpha^1)_{A'} & = & \Omega & \text{otherwise}
\end{array}
$$

Passing over operators α^2 for the moment, we come to α^3 and α^4. The former (domain $\mathrm{State}_{(A)} \times \mathrm{ref}(A)$) represents an attempt to apply an operation to an aggregate formed from the old system and a reference to an aggregate from the new one, while the latter ($\mathrm{sto}(A) \times \mathrm{Ref}_{(A)}$) tries to do the same for a new object and an old reference. Clearly, both manoeuvres are meaningless, and we may safely put:

$$
\begin{array}{lcl}
(\alpha^3)_{A'} & \stackrel{\mathrm{def}}{=} & \Omega \\
(\alpha^4)_{A'} & \stackrel{\mathrm{def}}{=} & \Omega
\end{array}
$$

Operations α^5 (domain $\mathrm{State}_{(A)} \times \mathrm{Ref}_{(A)}$) apply to states and references from the old system, and may thus be derived directly from algebra A:

[3]The only instance in which either of $\mathrm{State}_{(A)}$ or $\mathrm{Ref}_{(A)}$ are empty occurs when $A \in \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1})$, so that $\mathrm{State}_{(A)} = \mathrm{Ref}_{(A)} = \emptyset$. In such circumstance, of course, only case $(\alpha^1)_{A'}$ need be defined.

$$
\begin{array}{rcl}
(f^5)_{A'} & \stackrel{\text{def}}{=} & f_{(A)} \\
(g^5)_{A'} & \stackrel{\text{def}}{=} & g_{(A)}; \text{in2} \\
(p^5)_{A'} & \stackrel{\text{def}}{=} & p_{(A)}; \text{in2}
\end{array}
$$

Returning to α^2 (domain $\text{sto}(A) \times (\mathbf{N} \times \text{Ref}_{(A)})$); such operators apply to the state of a new aggregate and a reference to a descendent of its root object. Observe that the reference given is of precisely the same structure as contents of reference-valued private variables in the state of the root object itself. Thus we can use the auxiliary functions apply^f_A, apply^g_A and apply^p_A defined above:

$$
\begin{array}{rcl}
(f^2)_{A'} & \stackrel{\text{def}}{=} & \text{apply}^f_A \\
(g^2)_{A'} & \stackrel{\text{def}}{=} & \text{apply}^g_A; \text{in1}; \text{in2} \\
(p^2)_{A'} & \stackrel{\text{def}}{=} & \text{apply}^p_A; \text{in1}
\end{array}
$$

13.5 Proving Congruence

We have now defined a ΣW-semantics for the modular language presented in the opening sections. We will demonstrate its congruence following to the outline in chapter 9.

Recall that corollary 1 (on page 63) requires that, for all $i \in \mathbf{N}^+$ and $A \in \mathbf{Alg}(\Sigma_i, V, W_i)$:

1. There is a set $\text{Feas} \subseteq \text{sto}(\mathcal{I}(A))$ such that for all $c \in \text{Cmd}_i$, $C[\![c]\!]\mathcal{I}(A)\,\text{init}(\mathcal{I}(A)) \in \text{Feas}$.

2. There is a function $\phi : \text{sto}(\mathcal{I}(A)) \to \text{sto}(A)$, such that:

   i) $\mathbf{const}(\text{init}(A)) = \mathbf{const}(\text{init}(\mathcal{I}(A))); \phi$,

   ii) $\phi|_{\text{Feas}}; (E[\![e]\!]A) = E[\![e]\!]\mathcal{I}(A)$ for $e \in \text{Exp}_i$,

   iii) $\phi; (C[\![c]\!]A) = (C[\![c]\!]\mathcal{I}(A)); \phi$ for $c \in \text{Cmd}_i$.

3. There is a *total* function $\psi : W_{i-1} \to \text{Obs}_i$ such that $\forall A \in \mathbf{Alg}(\Sigma_{i-1}, V, W_{i-1})$, $t \in W_{i-1}$. $\text{eval}^{\mathcal{S}}(\psi(t), A) = t_A$.

We define the set Feas such that for all $\mu \in \text{sto}(\mathcal{I}(A))$, $\mu \in \text{Feas}$ iff:

$$\forall q \in \text{Id}_q.\ \mathbf{valueof}(q)\mu = (n, r) \Rightarrow \text{Reach}(\mathbf{valueof}(n)\mu) \wedge r \in F(\mathbf{valueof}(n)\mu)$$

Proof that Feas satisfies condition 1. is given in appendix C.

The function ϕ is defined as follows:

$$\phi \quad \stackrel{\text{def}}{=} \quad \mathbf{mult}(\mathrm{id}, \mathrm{id}, \mathbf{tuple}(\mathrm{id}, \mathrm{ev}_A), \mathrm{ev}_A)$$

Again, proof that ϕ satisfies condition 2. may be found in the appendix.

To define ψ, we need the following:

Definition 58 *For $r \in (T_{\Sigma_i})_{\mathrm{Ref}}, s \in (T_{\Sigma_i})_{\mathrm{State}}$, define:*

$$\begin{aligned} \mathrm{origin}(g(s,r)) &= s \\ \mathrm{radius}(g(s,r)) &= r \end{aligned}$$

Definition 59 *For $s \in (T_{\Sigma_i})_{\mathrm{State}}$ define an ordering, $\sqsubset$, the smallest relation on $F(s)$ such that for all $r, r' \in (T_{\Sigma_i})_{\mathrm{Ref}}$:*

$$r \sqsubset r' \stackrel{\text{def}}{\Longleftrightarrow} \begin{cases} r = \mathrm{root}^j, \ \text{for } 1 \le j \le i \text{ or} \\ \mathrm{origin}(r) < \mathrm{origin}(r') \text{ or} \\ \mathrm{origin}(r) = \mathrm{origin}(r') \text{ and } \mathrm{radius}(r) \sqsubset \mathrm{radius}(r') \end{cases}$$

For $s \in (T_{\Sigma_i})_{\mathrm{State}}$, and $P \subseteq F(s)$, let $\max(P)$ be the $\sqsubset$-maximal element of P.[4]

We assume that there is an injection $\nu : (T_{\Sigma_i})_{\mathrm{Ref}} \hookrightarrow \mathrm{Id}_q$ which ascribes to each term describing a reference a reference-valued variable name. (To ensure this, we require that the set Id_q be denumerably infinite; we stand with other authors on the subject [Gol81] in insisting that this is not too unreasonable a demand.) Now, define $\gamma(s, P)$ inductively:

$\gamma(s, P) =$

If $s = \mathrm{new}^j$, and $P = \{\mathrm{root}^j\}$ then $\ulcorner \nu(\mathrm{root}^j) \leftarrow \mathbf{new}\ j \urcorner$

Otherwise, let $r = \max(P)$ in

If $\mathrm{origin}(r) = s$, then let $r' = \mathrm{radius}(r)$ in

$\ulcorner \gamma(s, P \cup \{r'\} \setminus \{r\}); \nu(r) \leftarrow \nu(r').g \urcorner$

Otherwise, let $s' = \mathrm{origin}(r), r' = \mathrm{radius}(r)$ in

$\ulcorner \gamma(s', P \cup \{r'\}); \nu(r').p \urcorner$

Here quotes are used to delimit elements of the programming language syntax, so that, for example, $\ulcorner \nu(\mathrm{root}^j) \leftarrow \mathbf{new}\ j \urcorner$ denotes the command which places a newly-formed instance of class j into the variable named by the injection of root^j.

Finally (noting that each visible term in the model is of the form $f(s,r)$, for $s \in (T_{\Sigma_i})_{\mathrm{State}}, r \in (T_{\Sigma_i})_{\mathrm{Ref}}$), put $\psi(f(s,r)) = (\gamma(s, \{r\}), \nu(r))$. Proof that this function satisfies the required condition is also relegated to the appendix.
□

[4]That $\max(P)$ is well-defined is demonstrated in the appendix.

Chapter 14

Lessons Learned

We have presented four cases studies, each examining a simple object-oriented language which embodies a particular combination of characteristics exhibited by real-life OOL's. We have produced congruent ΣW-semantics for each of these languages that express the denotations of systems as algebras; the algebraic behaviour of these algebras faithfully reflect the linguistic behaviour of the systems they represent.

According to the results of chapter 8, applying the term realization construction to these ΣW-semantics automatically delivers a correct, compositional, abstract model for their respective languages. We are also assured that these languages are stable.

Some additional conclusions drawn from the case-studies are set out below.

14.1 Properties of OOL's

In producing congruent models for the languages in the case studies, we have had to address certain characteristics of OOL's currently in use which are not widely adduced. We will highlight a few of the more notable of these characteristics.

A fundamental tenet of most authorities on object-oriented programming is that the declarations of classes and objects define data-abstractions. These, as you will recall from chapter 3 are bodies of data accessible only by means of a set of operations. We saw in the same chapter, however, that class- or object-declarations by themselves are not generally sufficient to define data-abstractions. This is because objects usually rely on other objects to achieve certain functions; defining the functionality of these latter objects may require other declarations. We pointed out that only self-contained collections of declarations—which we called *systems*—could actually be guaranteed to define data-abstractions. However, our case-studies have shown that the correspondence between the declarations in a system and the kind of data-abstraction which it defines is not always straightforward.

The main reason for this is the somewhat liberal interpretation of "encapsulation" which underlies the design of many OOL's. The definition of encapsulated objects given in chapter 2 is the one normally employed in this connection; it states that all access to the private variables of an encapsulated object is restricted to the operations of that object. All of the languages examined in the case studies have encapsulated objects. The main idea of encapsulation is—quoting [GR83]—to arrange things so that "the only way to interact with an object is through its operations".

Consider, however, the case studies of languages 1, 2 and 4. All of these languages feature shared objects and references (the first two static references, and the latter dynamic ones). Recall that in each language, we established that applying operations to a shared object may affect the results of operations later applied to objects which share it. (For examples, refer to sections 10.3 and 13.3.) This would appear largely to undermine the intention of encapsulation set out in the previous paragraph.

In studying language 2, we also discovered that cycles of reference may lead to objects being observed while "suspended" in intermediate states attempting to complete internal computation. (These are the states whose representations result from applying operations of the form $\overrightarrow{p}^j$.) The nature of these states can be determined only by examining the internal details of objects.

Clearly, these remarks have bearing on the design of object-oriented languages in general, and we will return them in the final chapter. For the present, our concern is with the implications of such language features for the algebraic representation of objects.

14.2 Algebras and Objects

It might be pointed out that the ΣW-semantics presented here are merely a selection from the gamut of potential algebraic models for OOL's. As far as we can determine however, the case studies set out in this dissertation comprise the first thorough-going attempt 1) to formulate a notion of observation with respect to which (more-or-less) realistic object-oriented languages are stable, and 2) to represent objects in those languages as algebras whose algebraic behaviour agrees precisely with that of the objects they describe. Furthermore, there is every reason to suspect that any other attempt to meet these conditions within the algebraic framework would share at least some of the characteristics of the ΣW-semantics given. Therefore, while one would readily concede that other algebra-based models of realistic OOL's might be devised, we have some confidence in insisting that there may be valuable lessons to be learnt from the models presented here.

14.2.1 Objects, Signatures and Algebras

The models in studies of languages with references do not exhibit the close correspondence between the traditional conception of object "interfaces" and the signatures of their algebraic representations. Such a correspondence seems to be implicit in many influential writings on the subject, including [GM86] [Mey88] [LZ74] and [Sha81].

For an illustration, consider the representation of two systems—one written in language 3, the other in language 2. (Recall that the former is a class-based dynamic language without references and shared objects,[1] while the latter is a static language with shared objects and a limited form of cyclic references.)

The system in the first language consists of two classes, both of which simulate 1-bit counters (similar to the class Bit we saw in the programming exercise of chapter 2). The first class implements such counters in the usual way. Instances of the second class are simply "containers" for instances of the first; they each hold an instance of the first class, to which they delegate all operations.

class 1 **has functions** value = state
        procedures init: state ← 0
                tick: **if** state = 0 **then** state ← 1 **else** state ← 0

class 2 **has functions** value = bit.value
        procedures init: bit ← **new** 1; bit.init
                tick: bit.tick

This system may be described using the sort of informal "interface specification" espoused in writings such as [LAB+81], [Sha81], [GR83] and [Mey88]. For these writers, the interface of a class (and by extension, its instances) consists soley of the operations (the functions and procedures) which it defines. An interface specification, therefore, consists of a description of these operations. Thus in this case, we have:

Interface Specification

class 1 *simulation of modulo 2 counter*

[1] As we remarked earlier, of all the languages studied, this is the one which most closely resembles that examined by Hoare in [Hoa72]; Hoare's results have influenced many later writings concerned with object-oriented programming including [LAB+81], [Sha81] and [Mey88].

functions

value	*the value of the counter*

procedures

init	*initialize the counter so that its value is 0*
tick	*increment the value of the counter modulo 2*

class 2 *<same specification as class 1>*

Compare the general structure of this interface specification with the signature of this system's denotation in the model of the case study. (Omit operators which are not defined in the system; we know these operators correspond to undefined functions in the denotation.) We have:

$$\begin{array}{ll}\textbf{sorts} & \mathrm{State}, \mathrm{Val} \\ \textbf{operators} & \mathrm{new}^1 : \mathrm{State}, \\ & \mathrm{new}^2 : \mathrm{State}, \\ & \mathsf{value} : \mathrm{State} \to \mathrm{Val}, \\ & \mathrm{init} : \mathrm{State} \to \mathrm{State}, \\ & \mathsf{tick} : \mathrm{State} \to \mathrm{State}\end{array}$$

Note the correspondence between this signature and the interface specification; the only operators in the signature which do not figure in the specification are the constants new^1 and new^2 which produce instances of the classes in the system. Furthermore, we know from the model that in the denotation (A, say) of the system, the operation value_A is formed by the disjoint sum construction $\mathbf{cases}(\mathsf{value}^1, \mathsf{value}^2)$, where value^1 represents the function value on instances of class 1 and value^2 represents value on instances of 2. Similar comments apply to operations init_A and tick_A. Thus in this language, there is a cohesion between the traditional apprehension of a system's interface specification and the structure of its algebraic representation.

Contrast this with a similar exercise using language 2. Here, the system consists of two 1-bit counter objects, the first of which uses an integer to represent its state, and the second of which simply delegates its operations to the first. (The command "**skip**" abbreviates some nilpotent assignment, such as "$\mathsf{x} \leftarrow \mathsf{x}$", and a command of the form "i.op" abbreviates "i.op **calling**(**skip**)"):

object 1 **has functions** $\mathsf{value} = \mathsf{state}$
  procedures init: $\mathsf{state} \leftarrow 5$; **callback**; $\mathsf{state} \leftarrow 0$
    tick: **if** $\mathsf{state} = 0$ **then** $\mathsf{state} \leftarrow 3$ **else** $\mathsf{state} \leftarrow 2$;
      callback;

state ← state-2;

object 2 **has functions** value = 1.value
procedures init: **skip**; **callback**; 1.init
tick: 1.tick; **callback**; **skip**;

We could formulate an interface specification by analogy with above:

Interface Specification

object 1	*simulation of modulo 2 counter*
functions	
value	*the value of the counter*
procedures	
init	*initialize the counter so that its value is 0*
tick	*increment the value of the counter modulo 2*
object 2	*<same specification as object 1>*

On the face of it, this is a reasonable description of the system involved. Applying the procedure init to object 1, for example, has exactly the same effect as applying the same procedure to an instance of class 1 in the previous system. However, it ignores pivotal characteristics of this system which are consequences of effects noted in the case study:

- The procedures of object 2 may alter the state of object 1. For example, applying procedure tick to object 2 and then applying it to object 1 leaves both objects unchanged. No hint of this is given in the interface specification.
- Intermediate states occur during the execution of procedures which thanks to the occurrence of call-backs, may be observed by users of the system. For example, when the call back from procedure init of object 1 is executed, the value of this "modulo 2 counter" is 5!

This leads to a significant mismatch between the interface of the system given above and the structure of its denotation. If we again leave out operators which we know name undefined functions, the denotation of this system has the signature:

$$
\begin{array}{ll}
\textbf{sorts} & \mathrm{State}, \mathrm{Val} \\
\textbf{operators} & \mathrm{new} : \mathrm{State}, \\
 & \mathsf{value}^1 : \mathrm{State} \to \mathrm{Val}, \\
 & \mathsf{value}^2 : \mathrm{State} \to \mathrm{Val}, \\
 & \overrightarrow{\mathrm{init}}^1 : \mathrm{State} \to \mathrm{State}, \\
 & \overleftarrow{\mathrm{init}}^1 : \mathrm{State} \to \mathrm{State}, \\
 & \overrightarrow{\mathrm{init}}^2 : \mathrm{State} \to \mathrm{State}, \\
 & \overleftarrow{\mathrm{init}}^2 : \mathrm{State} \to \mathrm{State}, \\
 & \overrightarrow{\mathsf{tick}}^1 : \mathrm{State} \to \mathrm{State}, \\
 & \overleftarrow{\mathsf{tick}}^1 : \mathrm{State} \to \mathrm{State}, \\
 & \overrightarrow{\mathsf{tick}}^2 : \mathrm{State} \to \mathrm{State}, \\
 & \overleftarrow{\mathsf{tick}}^2 : \mathrm{State} \to \mathrm{State}
\end{array}
$$

Note the proliferation of operators; for example, the procedure tick of object 1 in the interface specification is represented here by two operators, $\overrightarrow{\mathsf{tick}}^1$ and $\overleftarrow{\mathsf{tick}}^1$. Also note how both objects have been combined, in that we cannot distinguish components of the carrier State corresponding to each, as we could in the previous example.

14.2.2 Algebraic Notions of Behaviour

Our experiences in devising congruent models for the languages in the case studies also casts doubt on another assumption concerning algebraic models of data-abstractions in programming languages. According to this conviction, the behaviour of a (Σ, V, W)-algebra is *always* characterized by the entire set of visible terms on its signature, $T^{\mathrm{v}}_{\Sigma,V}$. This belief appears to inform certain writings in the area, such as [GM85], [GM86].

This conviction is borne out in models of languages 1 and 3 (chapters 10 and 12 resp.); you will recall in each of these cases, the set of visible terms, W_i, used to fix the a-behaviour of a denotation of a system in $\mathbf{Alg}(\Sigma_i, V)$ is the entire set $T^{\mathrm{v}}_{\Sigma_i,V}$. However, in order to produce congruent models of languages 2 and 4 (chapters 11 and 13), we had to use a proper subset of $T^{\mathrm{v}}_{\Sigma_i,V}$. This would tend to suggest that the assumption of the previous paragraph is inadequate to model realistic OOL's, and gives support to more wide-ranging conceptions of a-behaviour, such as those set out in [ST87].[2]

That certain of the models in the case studies contradict accepted conceptions of algebraic representation of abstract data-types is reflected in the way in which systems in these languages are *verified*, as we will see next.

[2] In fact, the definition of behaviour given in chapter 5 is based on ideas from [ST87].

14.3 Verifying the Correctness of Systems

Let the algebra D be the denotation of a system in a congruent ΣW-semantics. *Verifying* the system in general means establishing the a-behavioural equivalence of D with another given algebra—E, say. It may be that E models an algebraic specification along the lines of those given in chapter 5, in which case verifying the system demonstrates that it properly satisfies that specification. Alternatively, E might be the denotation of another system; verification then amounts to showing that both systems may be used interchangeably.

In his seminal paper [Hoa72], Hoare proposed a method for verifying systems along these lines. (Published by Milner at about the same time, [Mil71] explores a related technique.) Today, the same technique—refined and adapted by Jones [Jon86]—provides the basis for proof methods used in the formal specification language VDM (which, of course, does not exhibit the reference-related characteristics of OOL's). Its use in the context of object-oriented languages has been advocated by [LAB+81] [Sha81] [Mey88] [Ame90a]. Its mathematical underpinnings have been extended by [Nip87] and [Sch87]. Hoare and his colleagues have also updated his original treatment in accordance with these extensions [HHS85].

As we have stated, the language used in [Hoa72] closely resembles the third of our case-languages, and in the next section, we will show how a fairly direct derivation of Hoare's method may be applied to it.

14.3.1 Verifying Language 3

First of all, we will assume that for both algebras $D, E \in \mathbf{Alg}(\Sigma_i, V, W_i)$, $1 \in \mathbf{N}^+$:

$$\begin{array}{l}\mathrm{State}_D = \mathrm{State}_D^1 \cup \ldots \cup \mathrm{State}_D^i \ \text{ and}\\ \mathrm{State}_E = \mathrm{State}_E^1 \cup \ldots \cup \mathrm{State}_E^i\end{array}$$

for families of disjoint sets $\mathrm{State}_D^1, \ldots, \mathrm{State}_D^i$ and $\mathrm{State}_E^1, \ldots, \mathrm{State}_E^i$.

Furthermore, we will assume that for all $p \in \mathrm{Id}_p$, $1 \leq j \leq i$ and all $d \in \mathrm{State}_D^j$, $e \in \mathrm{State}_E^j$,

$$\begin{array}{l}p_D(d) = \perp \ \text{ or } p_D(d) \in \mathrm{State}_D^j \ \text{ and}\\ p_E(e) = \perp \ \text{ or } p_E(e) \in \mathrm{State}_E\end{array}$$

Intuitively, this means that we can determine the class of the object whose state is represented by an element of State_D or State_E, and that applying a procedure to an object does not change its class. It is not difficult to show that the denotation of a system in the model satisfies these conditions. It does not seem too stringent a condition to place on E even if it is in fact a model of an algebraic specification (and not a system denotation).

The key to verification is a relation, θ, which associates representations object states taken from D with state-representations from E. If it can be shown that θ is a (strong) *correspondence* [Sch87], then $D \stackrel{\text{abeh}}{=} E$. To demonstrate that θ is a correspondence, we show that for all $1 \leq j \leq i$:

i) $\text{new}_D^j \; \theta \; \text{new}_E^j$

ii) For all $p \in \text{Id}_p, d \in \text{State}_D^j, e \in \text{State}_E^j$:

$$d \, \theta \, e \quad \Rightarrow \quad \begin{cases} p_D(d) = p_E(e) = \bot \text{ , or} \\ p_D(d) \; \theta \; p_E(e) \end{cases}$$

iii) For all $f \in \text{Id}_f, d \in \text{State}_D^j, e \in \text{State}_E^j$:

$$d \, \theta \, e \quad \Rightarrow \quad f_D(d) =_{\bot} f_E(e)$$

An important point to note is that in general, θ need only be a proper subset of the product $\text{State}_D \times \text{State}_E$, so that it leaves elements of State_D and State_E unrelated. In fact it can be shown that to satisfy the above conditions, θ need only relate the *reachable* elements of State_D and State_E. These reachable elements comprise the images of $(T_{\Sigma_i})_{\mathsf{State}}$ under ev_D and ev_E respectively. Intuitively, they are the representations of object states which may actually arise during observations.[3] Reachable elements are usually isolated with the aid of a *data-type invariant*, which is a predicate on the elements of an algebra preserved by all its operations—see [Hoa72], [Jon86] and [Sch87] for more details. Actually, when formulating θ, it is often helpful to eliminate from consideration as many unreachable elements as possible, since it may be that some unreachable elements of D or E cannot possibly be related to elements of the other algebra.

Proof that the existence of a correspondence does indeed establish the behavioural equivalence D and E is as follows:

Theorem 13 *If $D, E \in \mathbf{Alg}(\Sigma_i, V, W_i)$, for Σ_i, W_i, taken from the ΣW-semantics for language 3, and there is a correspondence $\theta \subseteq \text{State}_D \times \text{State}_E$, then $D \stackrel{\text{abeh}}{=}_{W_i} E$.*

Proof:

[3]In the modulo-2 counter example, the reachable states of instances of class 1 are those in which the private variable state has values 0 and 1.

Since in the model, $W_i = T^{\mathrm{v}}_{\Sigma_i,V}$, we need to show that the premise implies that $\forall t \in T^{\mathrm{v}}_{\Sigma_i,V} .\, t_D =_\perp t_E$.

Assume the premise. Define a family of sets of terms $(U_j)_{j \in \{1,\ldots,i\}}$, such that for all $1 \leq j \leq i$, U^j is the smallest set satisfying:

- $\mathrm{new}^j \in U^j$
- $t \in U^j \;\Rightarrow\; p(t) \in U^j$ for all $p \in \mathrm{Id}_p$

It is easy to establish that every $t \in T^{\mathrm{v}}_{\Sigma_i,V}$ must be of the form $f(t')$, for some $t' \in U^j$, $j \in \{1,\ldots,i\}$. We are done, therefore, if we can show that for all $1 \leq j \leq i$ and $t \in U^j$, either $t_D = t_E = \perp$ or $t_D \;\theta\; t_E$. For each j, proceed by induction on the number of procedure-names, n, which occur in t.

The base case ($n = 0$) follows directly from the condition that $\mathrm{new}^j_D \;\theta\; \mathrm{new}^j_E$. For the induction case, we know that t is of the form $p(t')$, where either $t'_D = t'_E = \perp$ or $t'_D \,\theta\, t'_E$. In the first case, the result follows by strictness of p_D and p_E. In the second case $p_D(t'_D) \,\theta\, p_E(t'_E) \;\Rightarrow\; t_D = t_E$.
$\square$

Notice how in the process of establishing that θ was a correspondence, we were able to examine the classes in the system one at a time. Furthermore, the procedures and functions which need to be examined may be read directly from the system's interface specification.

Now compare a similar verification method for language 2.

14.3.2 Verifying Language 2

We will devise a notion of "correspondence" meet for this language. In doing so, we need to take into account the points discussed in the previous sections. To re-iterate:

- Operators on different objects in a system may interfere; the states of all objects are combined and treated as a single unit.
- Each procedure $p \in \mathrm{Id}_p$ on object j in a system is represented by two operators, $\overrightarrow{p}^j$ and $\overleftarrow{p}^j$, in its denotation.
- The set of visible terms, W_i, which characterize the behaviour of denotations is a proper subset of the entire set of visible terms $T^{\mathrm{v}}_{\Sigma_i,V}$. Recall that in a term in W_i, each application of an operator of the form $\overrightarrow{p}^j$ (which represents the system entering

an intermediate state in a procedure, awaiting the execution of a call-back) must be followed immediately by an application of $\overleftarrow{p}^j$ (which completes the procedure after the call-back). This means that the set of reachable object states—those which may actually arise during an observation—is of a more restricted nature than those of the previous example. We must ensure that the method does not demand that a correspondence contain any unreachable states, for as we noted above, this could lead to behaviourally equal algebras being rejected.

The first two points are addressed by suitably adjusting the quantifications used to test for a correspondence. The latter entails breaking a correspondence into a pair of relations, θ and θ'. The first of these relates the non-intermediate states of a system, and the second relates intermediate ones.

In summary, for $D, E \in \mathbf{Alg}(\Sigma_i, V, W_i)$, $D \overset{\text{abeh}}{=} E$ if there exist relations $\theta, \theta' \in \mathrm{State}_D \times \mathrm{State}_E$ such that:

i) $\mathrm{new}_D \; \theta \; \mathrm{new}_E$

ii) For $1 \leq j \leq i$, and all $p \in \mathrm{Id}_p, d \in \mathrm{State}_D, e \in \mathrm{State}_E$:

$$d \, \theta \, e \quad \Rightarrow \quad \begin{cases} \overrightarrow{p}^j_D(d) = \overrightarrow{p}_E(e) = \bot \text{ , or} \\ \overrightarrow{p}^j_D(d) \, \theta' \, \overrightarrow{p}^j_E(e) \end{cases}$$

iii) For $1 \leq j \leq i$, and all $p \in \mathrm{Id}_p, d \in \mathrm{State}_D, e \in \mathrm{State}_E$:

$$d \, \theta' \, e \quad \Rightarrow \quad \begin{cases} \overleftarrow{p}^j_D(d) = \overleftarrow{p}^j_E(e) = \bot \text{ , or} \\ \overleftarrow{p}^j_D(d) \, \theta \, \overleftarrow{p}^j_E(e) \end{cases}$$

iv) For $1 \leq j \leq i$, and all $f \in \mathrm{Id}_f, d \in \mathrm{State}_D, e \in \mathrm{State}_E$:

$$d \, \theta \, e \quad \Rightarrow \quad f_D(d) =_\bot f_E(e)$$

Proof of the correctness of this method follows the same lines as that of the previous one.

Observe that in this case, all objects in the system must be considered at once—not separately as the classes in the method for language 3. Furthermore, operations such as $\overrightarrow{p}^j$ and $\overleftarrow{p}^j$ have been introduced whose effects could not be determined from an informal interface description such as that given in section 14.2.1.

14.4 Summary

We have seen how the case studies force us to acknowledge certain discrepancies between the way in which object-oriented languages actually function and received wisdom—pragmatic and theoretical—on the subject.

It turns out that the third of the case languages (which most closely resembled that studied by Hoare) does actually accord fairly well with the received wisdom. However, we should insist that is so far as it lacks the shared objects and cyclic structures of the other languages, it does not present a complete and accurate picture of OOL's in use today.

However, before going on to draw firm conclusions from these results, one further question should be addressed. Although the features embodied in languages 1, 2 and 4 (to wit, shared objects and cycles) do actually occur in real-life OOL's, they might be employed only rarely in practice. If this is the case, and OOL's are actually used in a manner more in keeping with the functionality of language 3, then traditional ideas are actually applicable in the majority of situations.

We will try to determine to what extent this supposition is true in the next chapter.

Chapter 15

Experimental Results

15.1 Motivation

The purpose of the experiments set out in this chapter is to determine just how frequently shared objects and cycles of reference do occur in the practical use of object-oriented systems. The first pre-requisite, of course, is a suitably representative "real-life" object-oriented system. We have chosen the Smalltalk-80 system [GR83], [Gol84], for the following reasons:

- *Smalltalk-80 is well established.* Though no longer the only mainstream Smalltalk-related commercial product, Smalltalk-80 is the direct precursor of most of the systems sold today, including the most prominent: "ObjectWorks\Smalltalk", from ParcPlace Systems, Inc. [Par90] and the "Smalltalk/V" family from Digitalk, Inc. [Dig86]. (Smalltalk-80 is itself available from ParcPlace Systems and Smalltalk Express Ltd. Despite the increasing popularity of alternatives such as C++ [SE90], it is probably still true to say that Smalltalk-related products are amongst the most widely used of all object-oriented systems. For our purposes, the differences between these systems and Smalltalk-80 itself (most of them concerned with programming tools) are not significant. (In the remainder of this chapter, therefore, we will use the terms "Smalltalk-80" and "Smalltalk" interchangeably.)

- *The Smalltalk-80 system is a Smalltalk program.* Smalltalk-80 is not simply a programming language; the system comes complete with a program development environment [Gol84]. This development environment is constructed almost entirely using the Smalltalk language, and must represent one of the most widespread object-oriented systems in use. Given the fact that practically all the source-code of the Smalltalk system is available to the user, it is also probable that the programming style and idioms used in the system have been widely copied [Deu89]. Therefore, we have some grounds for claiming that the Smalltalk-80 system is representative of a sizable sample of object-oriented systems.

- *Smalltalk-80 is large.* In total, the Smalltalk-80 system (including the program development environment), comprises over 50,000 lines of Smalltalk-80 language code. Given the diversity of functions which it performs, this probably means that it contains a fair cross-section of the kinds of program constructed in (at least one particular) object-oriented language.

- *Smalltalk's run-time system is easily accessible.* Our investigations involve detailed examination of the execution mechanisms of whichever object-oriented language is selected. Choosing Smalltalk makes such an activity particularly straightforward; the implementation of a Smalltalk system is defined by a bytecode interpreter, with reasonably simple and accessible run-time data-structures. In fact, Smalltalk makes data-structures such as stack-frames and process-bases into fully-fledged Smalltalk objects, and this eases our task still further.

- *Smalltalk is a "pure" object-oriented language.* Unlike hybrid object-oriented languages such as C++, all computation in Smalltalk is effected by the application of operations to objects. This allows us to monitor much of the activity in the language simply by observing the application of operations, greatly reducing the amount of data which has to be collected. (In the following, we will use Smalltalk's own terminology, referring to operations as "messages" and the application of an operation as the "sending" of such a message.)

There were also pragmatic reasons for choosing Smalltalk-80. An extensive collection of simulators and instrumentation harnesses for Smalltalk-80 systems has been designed and built by researchers at Manchester University. These were generously made available to the author, and greatly facilitated the conduct of the experiments. Kind permission was also granted for the use of the "Goldskin/Brouhaha" Smalltalk-80 interpreter [Mir87], providing another highly suitable base for instrumentation.

15.2 Overview

Two experiments were conducted; each involved observing the run-time behaviour of a Smalltalk system during several sessions of interactive use. The first experiment attempted to detect objects shared using references, while the second monitored possible cycles of reference between objects.

In principle, both experiments require details of all the storage activity which occurred during a session. Unfortunately this would be prohibitively expensive, since upwards of 100,000,000 store actions may take place in a typical Smalltalk session. However, with a slight change of emphasis, and by making a few simplifying assumptions, we can take advantage of the fact that Smalltalk is a pure OOL (so that all computation takes place using messages) to significantly reduce the amount of information which needs to be recorded.

Fundamentally, we take the view that a reference held by one object to another may be ignored if it is not used to send a message to its referee. Though two objects share a third object by holding references to it, for example, this need have no effect on the behaviour of the system unless both objects apply operations using those references. Similarly, a cycle of references is only of note if it results in an object which has sent a message being in receipt of another one before processing of the first message has been completed. This means that much of the information required for both experiments may be gathered by monitoring the way in which messages are sent during a session, thus significantly simplifying the instrumentation required.

The experiments, therefore, attempt to detect *shared* and *cyclic message-sends*, which are defined as follows:

- A *shared message-send* occurs in a session if, during the course of that session, an object receives messages from more than one other object.
- A *cyclic message-send* occurs in a session if, during the course of that session, an object sends a message and receives a message from another object before processing of the first is complete.

15.3 The Smalltalk-80 System

To delineate the remaining assumptions that have been made in designing the experiments, we need to describe some aspects of the Smalltalk-80 system.

15.3.1 Objects and Oops

Just as in language 4 of the case-studies, all objects in Smalltalk are accessed by reference. When it is created, each object is assigned a reference—normally known as its *oop* (short for "*o*bject-*o*riented *p*ointer")—which (at least in the Smalltalk implementations which hosted the experiments) is unique to it throughout its lifetime.[1] These oops are usually represented by integers.[2]

15.3.2 Messages

Despite connotations of concurrency and distribution, "message-sending" in Smalltalk is actually implemented as a form of (synchronous, local) procedure call; a message is effectively a procedure applied by one object (the message's *sender*) to another (the *receiver*). The actual body of code associated with a particular message name is determined at run-time, according to the message-name and the class of the receiver. Smalltalk makes use of stacks—composed of *stack-frames*—to record run-time information. The majority of stack-frames are created in response to message-sends; such stack-frames are known as *method contexts*.[3] For our purposes, it is sufficient to note that it is possible to determine from a method context the (oops of the) sender and receiver of the message whose dispatch caused it to be created.

15.3.3 Processes

Smalltalk offers cooperative pseudo-concurrency in the form of multiple independent processes. Each process is associated with a separate stack, which corresponds to a particular thread of control. A process is actually represented by a Smalltalk object, with its own oop.

15.3.4 Garbage Collection

Storage in Smalltalk is managed automatically, so that the user need make no special provision for the deletion of objects no longer in use. Instead, objects are *deallocated* by automatic garbage collection routines once all references to it have been discarded. In the Smalltalk implementations studied here, deallocating involves the return for further use of both the memory occupied by an object's data and the oop of the object. A consequence of this is that during a session, several objects may be associated with the same oop,

[1]Some Smalltalk implementations [PC86] use the address of the area of store used by an object as its oop—this means that an object's oop may change during its life.

[2]Oops and the *bone fide* integers used by objects are kept distinct by tagging.

[3]There is a second form of stack-frame which occurs occasionally in Smalltalk stacks: *block contexts* are used to facilitate the use of procedural closures. We have chosen to ignore block-contexts in the experiments; there is evidence to suggest that they are created relatively infrequently [Fal83], and furthermore, (since they provide a means by which code in one object may be executed by another) their inclusion would only support the hypothesis that encapsulation is broken significantly often in Smalltalk.

though while an object is alive, no other object is associated with its oop.

In the first experiment—detecting shared message-sends—we must take precautions against the possible effects of garbage-collection. In both experiments, objects are identified by the oop assigned them on creation. As we have remarked, garbage collection may cause two objects to be assigned the same oop during the course of a session. This means that it is possible for an objects to send a message to an object with a given oop, for the receiving object to be deallocated, and for a new object to be assigned its oop. Later on in the session, another object may send a message to the new object (quoting the same oop as did the earlier sender). Unless special provision is made, it is possible that this will lead to the spurious identification of a shared message-send.[4]

15.3.5 Classes with Immutable Instances

Certain "primitive" classes in Smalltalk, such as those describing characters or integers, have *immutable* instances whose state cannot be changed. It is easy to see that such objects may be shared with impunity. The first experiment, therefore, ignores shared message-sends if the receiver of the message involved is an instance of such a primitive class. Certain OOL's, such as CLU [LAB+81], permit user-defined classes to be designated immutable, with the compiler verifying that the state of their instances are not subject to change. Smalltalk, however, offers no such mechanism, and so we must assume (as should the user) that any shared object is mutable, unless it is an instance of these designated primitive classes.

15.3.6 Stack Behaviour

Smalltalk offers a variety of control structures, such as exceptions and non-local returns [Par90], as well as more exotic facilities such as first-class continuations [Deu81], which may disrupt the conventional last-in-first-out behaviour of execution stacks. Fortunately, there is fair evidence that Smalltalk stacks behave abnormally only very infrequently—in fact, many of today's highly optimized Smalltalk implementations depend heavily on such observations [DS84]. This means that particularly in the second experiment (when attempting to detect that the processing of a message has been completed), we may rely on stacks acting conventionally.

---

[4]No such provision need be made when looking for cyclic message sends, since while an object is awaiting the reply to a message, it cannot be deallocated.

15.4 The Experiments

Both experiments followed the same broad outline: The Smalltalk run-time system was modified so as to collect data onto a log file during an interactive session. Thus resulting log was supplied to an analyzer, which attempted to identify the frequency with which the given pattern of message sending (shared or cyclic) occurred during the session.

15.4.1 Experiment 1: Shared Message Sends

The information collected for this experiment recorded the oops of the sender and receiver of each message dispatched during a session. The oops of objects deallocated in the course of the session were also logged.

The analyzer detects sharing by effectively associating with each receiving object's oop a set containing the oops of the objects sending messages to it. This set we term the receiver's *senders-set*. Each time a message is sent, the analyzer adds the oop of the sender of that message to the senders-set of the receiver; if after the addition, the senders-set contains more than one object, then a shared message-send is deemed to have been detected.

In fact, the analyzer does not record the complete set of senders, since it is only of note if it contains more than one element. An empty senders-set is represented by a distinguished constant, **empty**, a singleton set by its sole element, and a set of greater cardinality by another constant, **full**.

When the analyzer encounters a record indicating that an object has been deallocated, the senders-sets are adjusted: First, the senders-set associated with this object's oop is emptied, to prevent spurious detection of shared sends should a new object be created with the same oop. Second, any senders-set containing only this object's oop is set to the distinguished constant **deallocated**; should a new object with the same oop be added to the set, a shared send has to be recorded.

Pseudo-code outlining the pertinent parts of the analyzer is given below.

function AddElement(set, element) =

—Add an element to a representation of a senders-set and return the result.

```
if set = empty or set = element then
        element
else
        full
```

end function

procedure SendMessage(senderOop, receiverOop)

> —*Analyze a message sent from object with oop* senderOop *to that with* receiverOop.
> —*Array "sendersSet" maps the oop of an object to its senders-set.*
>
> sendersSets[receiverOop] ← AddElement(sendersSets[receiverOop], senderOop);
> **if** *the receiver is not of an immutable class* **then**
>   **if** senderSets[receiverOop] = **full then**
>   tallyOfSharedMessages ← tallyOfSharedMessages+1;

end procedure

procedure Deallocate(oop)

> —*Analyze the deallocation of the object with the oop given.*
>
> sendersSets[oop] ← **empty**;
> **for** i = 1 **to** size of sendersSets **do**
>   **if** sendersSets[i] = oop **then**
>   sendersSets[i] ← **deallocated**;

end procedure

15.4.2 Experiment 2: Cyclic Message Sends

Information collected for the second experiment related the oops of the sender and receiver of each message dispatched during a session, as before. A record was emitted when the processing of a message was completed (so that control returned to its sender) which identified that message's sender and receiver. Records were also produced each time a new process became active (the record contained the oop of the Smalltalk object representing that process).

The analyzer for this experiment maintains a mapping which associates an "activity count" with (the oop of) each object, recording the number of messages sent by that object whose processing is yet to be completed. This activity count is incremented each time the analyzer notes that an object has sent a message, and decremented when it is found that the processing of a message dispatched by that object has been completed. A cyclic send is detected when the activity count of an object rises above 1.

Two issues slightly complicate the analyzer: A message sent from an object to itself (forbidden in all of the case-studies, but permitted in Smalltalk) is ignored, since it is to all intents and purposes invisible to other objects. Furthermore, if an object participates in more than one process, we choose not to record a cyclic send if it sends a message in one process and receives one in another process. This entails the analyzer keeping separate sets of activity counts for each process executed during a session. It also identifies the process currently active at any point in the session.

As before, pseudo-code for the more important parts of the analyzer is given below:

procedure SendMessage(senderOop, receiverOop)

> —*Sending a message from an object with oop* senderOop *to one with* receiverOop.
> —currentProcess *is a number unique to the currently active process.*
> —activityCounts *holds the activity count for each object in each process.*
>
> **if** senderOop $\neq$ receiverOop **then**
>   activityCount $\leftarrow$ activityCounts[currentProcess, senderOop]+1;
>   activityCounts[currentProcess, senderOop] $\leftarrow$ activityCount;
> **if** activityCount $>$ 1 **then**
>   tallyOfCyclicMessages $\leftarrow$ tallyOfCyclicMessages+1;

end procedure

procedure CompleteMessage(senderOop, receiverOop)

> —*The completion of a message sent from object with* senderOop *to* receiverOop.
>
> **if** senderOop $\neq$ receiverOop **then**
>   activityCount $\leftarrow$ activityCounts[currentProcess, receiverOop]-1;
>   activityCounts[currentProcess, receiverOop] $\leftarrow$ activityCount;

end procedure

15.4.3 Results

Each experiment was conducted for three interactive sessions. Each session characterized a particular type of activity supported by the Smalltalk system: manipulation of the windows in the programming environment, text editing, and compilation of Smalltalk language code. The results of these analyses are summarized in the six graphs overleaf. (Note that we have displayed the cumulative density of shared and cyclic sends—that is, the ratio of the number of shared or cyclic sends up to a point in the session as a percentage of the total number of messages sent.)

15.5 Findings

The results of our preliminary investigations appear to be quite significant. An average density of about 50% for shared message sends, and approximately 30% for cyclic sends, appears to be the norm for these types of activity in Smalltalk. Of course, our results are subject to the caveats made in previous sections, and we have investigated only one of many systems written in object-oriented languages. Even so, it would appear that the effects of sharing objects and creating cyclic patterns of reference do play a major role in the behaviour of object-oriented systems.

Chapter 16

Conclusions

This chapter summarizes the investigation recounted in the dissertation, outlines how its scope might be extended, and attempts to draw some conclusions from its findings.

16.1 Summary

We set out to explore the basic mechanisms at work in object-oriented programming languages by investigating their mathematical semantics. We elected to employ denotational semantics, and aimed to produce so-called *abstract* denotational models. These models exhibit the minimum of "implementation detail", and we felt (in common with other researchers in the area) that they would supply informative accounts of the languages studied. In producing such abstract models, we resolved not to eschew discussion of problematic features exhibited by object-oriented languages in everyday use, especially those involving references and shared objects. Broadly speaking, previous theoretical accounts chose either to ignore such features so as to facilitate the production of abstract models, or to forgo abstractness altogether.

We described a broad framework within which to study object-oriented languages. We assumed in constructing it that the main point of these languages was the production of *systems*—self contained collections of object- or class-declarations. We considered that *observations* were carried out on a system by programmers basing new declarations upon it. The *l-behaviour* of a system was established by the results of carrying out a given set of observations on it. We summarized our ideas with the definition of the *modular language*—a generalized description of object-oriented languages. *Specifications* for systems were investigated in the context of modular languages; *correct* and *abstract* specifications were characterized. We also formalized a concept of *stability* for modular languages; a modular language was stable if there was a reasonable fit between its observations and class- or object-declarations.

We explored denotational semantics for modular languages, framing definitions for *correct*, *compositional* and *abstract* semantics. We pointed out the desirability of correct, compositional, abstract models—especially with regard to the specification of systems. We also demonstrated how existing models for realistic OOL's—though they might be correct and compositional—failed to be abstract.

We introduced a particular kind of denotational semantics for modular languages—ΣW-semantics—which used *algebras* to represent systems. The observations in a modular language were mimicked by sets of *visible terms* in such a semantics. The latter were evaluated with respect to algebras to determine their *a-behaviour*. When the a-behaviour of algebras in a ΣW-semantics corresponded precisely with the l-behaviour of the systems they represented, the semantics was termed *congruent*. We showed how given a congruent ΣW-semantics, one could produce a correct, compositional, abstract semantics simply by taking the *term realizations* of system denotations. The existence of a congruent ΣW-semantics also guaranteed the stability of that language.

We proceeded to study four languages, each one of which reflected a particular combination of features found in "real-life" object-oriented languages. The first two of these languages were *static*, so that objects were defined explicitly, not generated from classes or exemplars. Objects in these languages were accessed exclusively by *reference*, permitting them to be shared. The second language was largely similar to the first, but introduced a limited form of *cycles* of reference. The third and the fourth languages studied were *dynamic*, in that they allowed objects to be created on demand from classes. The third language did not provide references; objects were *copied* on assignment. (Of all the languages studied, this language was closest to that discussed by Hoare in his influential paper [Hoa72].) The fourth language did include references, and permitted such references to be passed between objects. For each of the languages examined, we formulated a collection of sets of observations. By adjoining such observations to each language, we were able to produce modular language. For each of the modular languages, we presented a ΣW-semantics, and showed it to be congruent. According to previous results, this delivered a correct, compositional, abstract semantics for the language, and also established its stability.

The case-studies determined that facilities for producing shared objects and cycles of reference significantly complicated the semantics of the languages which provided them. In the congruent ΣW-semantics produced for languages with such features, the algebras used as denotations for systems figured operations which were not present in the declared interfaces for those systems given in the program. This contrasts with the received wisdom concerning object-oriented languages, which says that "objects may only be manipulated through the operations declared in their interfaces." In addition, the sets of visible terms required to mimic observations in such languages were at variance with traditional theoretical ideas about abstract data-types. Both these phenomena had knock-on effects for verification methods, and we showed how it was only for the third of our languages (dynamic, no references) that commonly-suggested verification methods were sound.

Our first suspicion was that shared objects and cycles of reference, though they feature in real-life object-oriented languages, are relatively infrequently used. If this were the case, traditional assumptions about semantics and verification might be broadly correct in most circumstances. We constructed two simple experiments, designed to test this hypothesis. These experiments were applied to the Smalltalk-80 system—an object-oriented system widely used in research and commercial environments. Our results appear to show that on the contrary, references and cycles *are* frequently employed in realistic object-oriented programs.

16.2 Extending the Treatment

Before we go on to draw any firm conclusions from our investigations, let us explore the ways in which the treatment might be extended:

- We could permit classes (or objects) to be identified by *names* chosen by the user, rather than merely by fixed indices. To ensure the compatibility of systems and class- (or object-) declarations, one would need to associate with systems the names of sets of classes which they defined, and declarations with the classes or objects they required. Induction methods such as we have used in the proofs in chapters 10 and 11 would be based upon the number of classes defined by a system.

- Iterative and recursive constructs could be incorporated. "While-loops" and similar iterative constructs do not present any major problems. One would simply need to make the carriers of algebras used in ΣW-semantics into flat complete partial orders.[1] As to recursive class or object definitions—we have already seen (in chapter 11) how cyclic references can have unexpected consequences. Though we have constructed a congruent ΣW-semantics for a static object-oriented language permitting fully general cycles of references, the result was far too complicated to be included in this dissertation. Cycles which result from the unconstrained use of references as function- or procedure-parameters in dynamic languages (a much closer reflection of

[1] Thanks to our use of partial algebras, this is a straightforward operation.

the nature of "real-life" OOL's, as discussed below) are likely to lead to still more complicated semantics.

- Unlike "real-life" OOL's, the languages studied in the case-studied do not allow parameters to be supplied when applying procedures or functions to objects. Adding parameters with primitive values (such as integers or booleans) would not be too difficult; an operation $p(x)$, whose parameter x ranges over the primitive values $\{a_1, a_2, \ldots\}$ could be simply "expanded" into the collection of (parameterless) operations $p_{a_1}, p_{a_2}, \ldots$ (An allied technique is used in models of CCS [Mil80].) Unfortunately, commonly-used object-oriented languages also permit object-references to be used as parameters, and these are much less tractable. We witnessed in chapter 12 how the use of dynamic references—even in a very restricted context—significantly complicated the semantics of OOL's. The prospect of formulating abstract models for languages with unrestricted reference parameters is even less pleasant.

- Even when the above issues have been addressed, there still remain a flock of more peripheral language-features to be considered. These include meta-classes [Coi87], reflective constructs [Mae87], concurrency [YT87], sophisticated control facilities [Deu81], amongst others.

As well as extending our treatment to cover the features described above, it is also possible that the models themselves could also be made *surjective* by ensuring they contained only those denotations which represented systems. This was discussed in chapter 7, where we pointed out that the experience of other researchers desirous of surjectivity in models [BCL85] hints at possible difficulties which might be encountered.

At the outset, we decided to avoid a detailed discussion of inheritance in OOL's. While it is possible that the treatment presented in this dissertation could be extended to take account of such features, we feel it might be more appropriate to try to use the insights gained here to inform the discussion of inheritance itself. In chapter 2, we enumerated two main aspects of the inheritance facilities of most OOL's—viz. *sub-typing* and *code-sharing*:

Sub-typing relates the way in which objects interact with each other—a type A is a *sub-type* of type B if objects of type A may replace those of type B in all contexts in which they are used. In the vocabulary of chapters 3 and 6, this amounts to the observable behaviour of type B objects "including" in some fashion the observable behaviour of type A objects. This view is formalized in [Ame90a], where a definition of sub-typing is produced for a language like that of chapter 12 (dynamic, no shared objects or references) based on the traditional verification method related in chapter 14. We showed in the same chapter how corresponding verification methods might be developed for more realistic object-oriented languages. An evolution of these methods similar to that in [Ame90a] would lead to formal definitions of subtyping for such languages.

The other facet of inheritance—*code-sharing*—permits parts of an object-declaration to be re-used to define other kinds of object. As was pointed out in chapter 2, researchers have drawn increasingly close parallels between code-sharing and object composition (wherein one object makes use of another to achieve its function). In particular, [Sak89] shows how this form of inheritance may be elegantly expressed as a form of "direct aggregation"—rather like a static form of the object-composition found in language 3. As chapter 14 shows, the ΣW-semantics presented in the case studies may be used to guide the specification of objects used in this way. The second case-study (of a static language with cyclic references) also throws light on another inheritance mechanism. Sometimes called the "*self* facility", this permits an object-declaration to refer to the compound objects whose definitions might share it. A detailed exploration is given in [Coo89a]. The self facility is analogous to the provision of call-backs in language 2, where objects are allowed to cause the execution of code in objects which make use of them.

Clearly, much remains to be done. Even so, there are conclusions which may be drawn even from the preliminary investigations described here.

16.3 A Little Learning

To draw any lessons from the exercises described in this dissertation, we should first try to divine the "true purpose" of an object-oriented programming language. The fulcrum of our development has been a postulation to the effect that:

> An object-oriented language is a programming notation designed to promote modular programming by supporting the disciplined production of executable descriptions of data-abstractions.

We have seen the basic means by which most OOL's try to achieve this: they allow the user to list a collection of operations acting upon a object, and (by encapsulation) to ensure that the variables the object are only accessible by means of those operations. This having been done, it is commonly believed that the operations comprise the object's *interface*—the only means by which it may interact with other objects. This is a conviction which has underlain many mathematical investigations of the subject.

Unfortunately, the findings of the case studies show how some of the most fundamental characteristics of OOL's give lie to this commonly-held view. In particular, we have seen how the presence of shared objects and the unconstrained use of object-references can lead to a bourgeoning of interactions between objects whose nature cannot be gleaned from such "interfaces."

In the main, there are two possible responses to this difficulty:

- Accept object-oriented languages as they stand, and attempt to remedy possible defects either by careful programming or design conventions.
- Adjust the semantics of present-day OOL's, so as to produce new languages in which these defects are ameliorated.

The first of these options, of course, is merely an extrapolation of present practice. In fact of the two, it is probably the option which would be most popular with the majority of programmers. Restrictions on the ways in which objects may be shared or referred to in programs are likely to result—to a greater or lesser extent—in a decrease in execution efficiency and programming convenience. The popularity of languages such as "C" [KR88], which place great emphasis on efficiency and allow the programmer to sacrifice safety in favour of flexibility, attests to the desirability of such traits. There is also anecdotal evidence that for developers of small systems, the presence of shared objects causes only minor annoyance. Unfortunately, often painful experience [Bro75] shows that much greater discipline is required in constructing large software systems than is needed to produce small ones, and it is quite possible that the effects of such language features will become more evident as more ambitious projects are undertaken in OOL's.

In fact, some authorities on the design of object-oriented systems have acknowledged the potential hazards of unrestricted use of object-references. Meyer [Mey88], for example, draws attention to "dynamic aliasing" (the sharing of objects by means of references), and points out that it might undermine attempts at verification; Booch [Boo91] makes similar remarks about "structural sharing".[2] The Hierarchical Object-Oriented Design (HOOD) method [CM87] forbids cyclic references between classes (though is it unclear whether the motivation for this is the existence of the intermediate states uncovered in chapter 11). None of these admonitions, however, are reflected in OOL's themselves, and all must be enforced by convention (which, given the dynamic nature of patterns of reference in most OOL's is normally very difficult to enforce). This largely undermines our proposition that OOL's support the "disciplined description of objects", for as Stroustrup points out [Str88], "a language does not support a technique if it takes exceptional skill to use that style; in that case, the language merely enables programmers to use the technique".

[2] Booch incorporates elements into his design notation with especial regard to shared objects.

We are left, therefore, with the second option; to alter the character of object-oriented programming languages so as to control the use of object-references.[3] This would reduce the disparity between the actual behaviour of objects and the picture which might be gained from their declared interfaces (descriptions of the latter might need to be augmented somewhat for greater flexibility). The evidence presented here suggests that this would increase the effectiveness of OOL's as regards disciplined software engineering. Unfortunately, such developments might also increase the restrictions on the users of such languages, or degrade the efficiency of their implementations. However, the very fact that OOL's are themselves increasingly popular suggests that programmers are willing to trade off a certain loss of licence and performance against an increase in effectiveness [Boo91].

Whichever option is finally adopted, it is evident that object-oriented programming as it is currently prosecuted represents only the beginning, and not the end of a line of development. Certainly, that there is such a widespread misapprehension of the consequences of encapsulation as it is implemented in present-day OOL's suggests that there are significant questions still to be addressed.

16.4 The Great White Hope

In light of this, it might be wise to introduce a modicum of restraint into the approbation of object-oriented programming. Such is the degree of enthusiasm which presently surrounds the subject, that it is possible to see object-orientation as the "artificial intelligence of the 1990's". This might imply a "boom-and-bust" cycle of popularity and support, as hopes are in turn overly-inflated and then punctured. Certainly, the increasing use of "object-orientation" as a weapon with which to drub competitors—rather than any intrinsic merit of the final product—cannot be a good thing. It would be a pity if these fears were realized, and object-oriented programming became victim of fashion and fad, because within its limits, it is without doubt a promising technique.

On a rather narrower scale, object-oriented programming languages themselves should not be seen as the "great white hope." Some authors see OOL's presaging a significant shift in the way software is produced, with much greater emphasis on the use of reusable components [Cox86]. Unfortunately, current evidence points to the contrary—US defence contractors, for example, remunerated on the basis of the number of lines of code written, are most reluctant to "buy-in" prefabricated software components, despite using an object-based language (viz. Ada) designed to promote the use of such components.

[3] For example, one might copy objects as in language 3, rather than copying references to them as most OOL's do. Alternatively, one might only allow patterns of sharing to be specified statically, as in languages 1 and 2.

It is probable that no single "technological fix", will cure the ills which presently beset the computer software industry [Bro87]. As General Motors found when faced with keen competition in the 1980's, many of the root causes of low productivity and poor quality are social, and not technological [Ano90]. With software, as with automobiles, no amount of apparent technical innovation will substitute for good training, good management and good sense.

Appendix A

Universes

The problems of expressing formally in naive set-theory ideas such the "set" of all algebras with a given signature are well-known.[1] Unfortunately, such expressions form the basis for much of the semantic framework developed in the dissertation.

One established device by which these difficulties may be avoided involves the addition to the usual formation rules for sets (cartesian products, unions, powersets and limited comprehension) the assumed existence of a *universe* [Mac72].

A universe is defined as a set U satisfying the following conditions:

1. $x \in u \in U$ implies $x \in U$,
2. $u \in U$ and $v \in U$ implies $\{u, v\}, (u, v)$ and $u \times v \ \in U$,
3. $x \in U$ implies $\mathcal{P}(x) \in U$ and $\bigcup x \ \in U$,
4. $\mathbf{N} \in U$,
5. If $f : a \to b$ is a surjective function with $a \in U$ and $b \subset U$ then $b \in U$.

If we assume the existence of such a universe, then, for example, the set of algebras above may be expressed by limited comprehension as the "set of all algebras with signature Σ which are members of U", and we are assured of its well-formedness. Throughout the dissertation, we make the implicit assumption that such limited comprehension over U is applied where appropriate.

[1] The upward Löwenheim-Skolem theorem shows that such algebras exist with arbitrary cardinality, and so such a "set" is not well-formed in naive set theory.

Appendix B

Proof of Congruence for Language 1

To recall, we have:

$$\phi = \mathbf{mult}(\mathrm{id}, \mathrm{ev}_A)$$

We must show that:

i) $\mathbf{const}(\mathrm{init}(A)) = \mathbf{const}(\mathrm{init}(\mathcal{I}(A))); \phi$,

ii) $\phi; (E[\![e]\!]A) = E[\![e]\!]\mathcal{I}(A)$ for $e \in \mathrm{Exp}_i$,

iii) $\phi; (C[\![c]\!]A) = (C[\![c]\!]\mathcal{I}(A)); \phi$ for $c \in \mathrm{Cmd}_i$.

Proceed by each condition in turn:

i) By the definitions:

$$\begin{array}{rcl} \mathbf{const}(\mathrm{init}(\mathcal{I}(A))); \phi; \mathbf{valueof}(b) & = & \perp \\ & = & \mathbf{const}(\mathrm{init}(A)); \mathbf{valueof}(b) \end{array}$$

In addition:

$$\begin{array}{rcl} \mathbf{const}(\mathrm{init}(\mathcal{I}(A))); \phi; \mathbf{valueof}(*) & = & \mathrm{new}_{\mathcal{I}(A)}; \mathrm{ev}_A \\ & = & \mathrm{new}_A \\ & = & \mathbf{const}(\mathrm{init}(A)); \mathbf{valueof}(*) \end{array}$$

The result follows by extensionality.

ii) Most cases are simple re-writes. The only case of difficulty is that of expressions of the form $j.f$. For this, we have:

$$\begin{array}{rcll} \phi; E[\![j.f]\!]A & = & \phi; \mathbf{valueof}(*); f^j_A & (\text{definition of} E[\![j.f]\!]A) \\ & = & \mathbf{valueof}(*); \mathrm{ev}_A; f^j_A & (\text{definition of } \phi) \\ & = & \mathbf{valueof}(*); f^j_{\mathcal{I}(A)} & (\text{definition of } \mathcal{I}(A)) \\ & = & E[\![j.f]\!]\mathcal{I}(A) & (\text{definition of} E[\![j.f]\!]\mathcal{I}(A)) \end{array}$$

iii) Proceed by structural induction on c:

Case: $c_1; c_2$

$$\begin{aligned}
\phi; C[\![c_1; c_2]\!]A &= \phi; C[\![c_1]\!]A; C[\![c_2]\!]A \\
&= C[\![c_1]\!]\mathcal{I}(A); \phi; C[\![c_2]\!]A \\
&= C[\![c_1]\!]\mathcal{I}(A); C[\![c_2]\!]\mathcal{I}(A); \phi \\
&= C[\![c_1; c_2]\!]\mathcal{I}(A); \phi
\end{aligned}$$

Case: **if** e **then** c_1 **else** c_2

$$\begin{aligned}
\phi; C[\![\,\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2]\!]A &= \phi; (E[\![e]\!]A \to C[\![c_1]\!]A,\ C[\![c_2]\!]A) \\
&= (\phi; E[\![e]\!]A) \to (\phi; C[\![c_1]\!]A),\ (\phi; C[\![c_2]\!]A) \\
&= E[\![e]\!]\mathcal{I}(A) \to C[\![c_1]\!]\mathcal{I}(A),\ C[\![c_2]\!]\mathcal{I}(A) \\
&= C[\![\,\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2]\!]\mathcal{I}(A)
\end{aligned}$$

Case: $v \leftarrow e$

Result follows by extensionality from the following:

$$\begin{aligned}
\phi; C[\![v \leftarrow e]\!]A; \textbf{valueof}(v') &= \phi; \{E[\![e]\!]A/b\}; \textbf{valueof}(v') \\
&= \begin{cases} \phi; \textbf{valueof}(v') & \text{if } v' \neq v \\ \phi; E[\![e]\!]A & \text{if } v' = v \end{cases} \\
&= \begin{cases} \textbf{valueof}(v') & \text{if } v' \neq v \\ E[\![e]\!]\mathcal{I}(A) & \text{if } v' = v \end{cases} \\
&= C[\![v \leftarrow e]\!]\mathcal{I}(A); \textbf{valueof}(v') \\
&= C[\![v \leftarrow e]\!]\mathcal{I}(A); \phi; \textbf{valueof}(v')
\end{aligned}$$

$$\begin{aligned}
\phi; C[\![v \leftarrow e]\!]A; \textbf{valueof}(*) &= \phi; \textbf{valueof}(*) \\
&= \textbf{valueof}(*); \mathrm{ev}_A \\
&= C[\![v \leftarrow e]\!]\mathcal{I}(A); \textbf{valueof}(*); \mathrm{ev}_A \\
&= C[\![v \leftarrow e]\!]\mathcal{I}(A); \phi; \textbf{valueof}(*)
\end{aligned}$$

Case: $j.p$

$$\begin{aligned}
\phi; C[\![j.p]\!]A; \textbf{valueof}(*) &= \phi; \{\textbf{valueof}(*); p^j_A/*\}; \textbf{valueof}(*) \\
&= \phi; \textbf{valueof}(*); p^j_A \\
&= \textbf{valueof}(*); \mathrm{ev}_A; p^j_A \\
&= \textbf{valueof}(*); p^j_{\mathcal{I}(A)}; \mathrm{ev}_A \\
&= C[\![j.p]\!]\mathcal{I}(A); \textbf{valueof}(*); \mathrm{ev}_A \\
&= C[\![j.p]\!]\mathcal{I}(A); \phi; \textbf{valueof}(*)
\end{aligned}$$

$\square$

Appendix C

Proof of Congruence for Language 4

To recall, we have:

- A set Feas $\subseteq$ sto($\mathcal{I}(A)$), such that for all $\mu \in$ sto($\mathcal{I}(A)$), $\mu \in$ Feas iff:

  $$\forall q \in \mathrm{Id}_q.\ \mathbf{valueof}(q)\mu = (n, r) \ \Rightarrow\ \mathrm{Reach}(\mathbf{valueof}(n)\mu) \wedge r \in F(\mathbf{valueof}(n)\mu)$$

  where for all $s \in (T_{\Sigma_i})_{\mathrm{State}}$, $F(s)$ is the smallest set satisfying:

  - $\mathrm{new}^j \leq s \ \Rightarrow\ \mathrm{root}^j \in F(s),\ \text{ for } 1 \leq j \leq i$
  - $s' \leq s \ \wedge\ r \in F(s') \ \Rightarrow\ g(s', r) \in F(s)$

  and Reach(s) is defined inductively:

  - $\mathrm{Reach}(\mathrm{new}^j),\ \text{ for } 1 \leq j \leq i$
  - $\mathrm{Reach}(s) \wedge r \in F(s) \ \Rightarrow\ \mathrm{Reach}(p(s, r))$

- A function $\phi : \mathrm{sto}(\mathcal{I}(A)) \to \mathrm{sto}(A)$, such that:

  $$\phi \ \stackrel{\mathrm{def}}{=}\ \mathbf{mult}(\mathrm{id}, \mathrm{id}, \mathbf{tuple}(\mathrm{id}, \mathrm{ev}_A), \mathrm{ev}_A)$$

- For $s \in (T_{\Sigma_i})_{\mathrm{State}}$, an ordering $\sqsubset$, the smallest relation on $F(s)$ such that for all $r, r' \in (T_{\Sigma_i})_{\mathrm{Ref}}$:

  $$r \sqsubset r' \ \stackrel{\mathrm{def}}{\Longleftrightarrow}\ \begin{cases} r = \mathrm{root}^j,\ \text{ for } 1 \leq j \leq i \text{ or} \\ \mathrm{origin}(r) < \mathrm{origin}(r') \text{ or} \\ \mathrm{origin}(r) = \mathrm{origin}(r') \text{ and } \mathrm{radius}(r) \sqsubset \mathrm{radius}(r') \end{cases}$$

- A function $\psi : W_{i-1} \to \mathrm{Obs}_i$, such that $\psi(f(s, r)) = (\gamma(s, \{r\}), \nu(r))$, where $\gamma(s, P)$ defined inductively:

$\gamma(s, P) =$

$$\begin{array}{l}
\text{If } s = \text{new}^j \text{, and } P = \{\text{root}^j\} \text{ then } \ulcorner \nu(\text{root}^j) \leftarrow \textbf{ new } j \urcorner \\
\text{Otherwise, let } r = \max(P) \text{ in} \\
\qquad \text{If origin}(r) = s\text{, then let } r' = \text{radius}(r) \text{ in} \\
\qquad\qquad \ulcorner \gamma(s, P \cup \{r'\} \setminus \{r\}); \nu(r) \leftarrow \nu(r').g \urcorner \\
\qquad \text{Otherwise, let } s' = \text{origin}(r), r' = \text{radius}(r) \text{ in} \\
\qquad\qquad \ulcorner \gamma(s', P \cup \{r'\}); \nu(r').p \urcorner
\end{array}$$

where, for $s \in (T_{\Sigma_i})_{\text{State}}$, and $P \subseteq F(s)$, $\max(P)$ is the $\sqsubset$-maximal element of P, $\nu : (T_{\Sigma_i})_{\text{Ref}} \hookrightarrow \text{Id}_q$ is an injection, and:

$$\begin{array}{rcl}
\text{origin}(g(s, r)) & = & s \\
\text{radius}(g(s, r)) & = & r
\end{array}$$

We will show that:

1. For all $c \in \text{Cmd}_i$, $C[\![c]\!]\mathcal{I}(A)\,\text{init}(\mathcal{I}(A)) \in \text{Feas}$.
2. i) $\textbf{const}(\text{init}(A)) = \textbf{const}(\text{init}(\mathcal{I}(A))); \phi$,

   ii) $\phi|_{\text{Feas}}; (E[\![e]\!]A) = E[\![e]\!]\mathcal{I}(A)$ for $e \in \text{Exp}_i$,

   iii) $\phi; (C[\![c]\!]A) = (C[\![c]\!]\mathcal{I}(A)); \phi$ for $c \in \text{Cmd}_i$.
3. The relation $\sqsubset$ is a total ordering on $F(s)$, for $s \in (T_{\Sigma_i})_{\text{State}}$
4. Function ψ is total and $\forall A \in \textbf{Alg}(\Sigma_{i-1}, V, W_{i-1}), t \in W_{i-1}.\ \text{eval}^{\mathcal{S}}(\psi(t), A) = t_A$.

Condition 1. This follows from the following:

i) $\text{init}(\mathcal{I}(A)) \in \text{Feas}$

ii) $\forall c \in \text{Cmd}_i.\ \mu \in \text{Feas} \Rightarrow C[\![c]\!]\mathcal{I}(A)\,\mu \in \text{Feas}$

The first case is trivially demonstrated. The second may be established by structural induction on c. The only results which are not routine are derived as follows:

Case: $q_1 \leftarrow q_2.g$

$$\begin{array}{l}
C[\![q_1 \leftarrow q_2.g]\!]\mathcal{I}(A)\,\mu \\
\quad = \{\textbf{tuple}(\text{id}, \textbf{valueof}(q_2)); \text{apply}^g_{\mathcal{I}(A)}/q_1\} \\
\quad = \{\textbf{tuple}(\text{id}, \textbf{valueof}(q_2)); \lambda\mu, (n, r).\ (n, g_A(\textbf{valueof}(n)m, r))/q_1\}
\end{array}$$

So we are done if we can show that:

$$\mu \in \text{Feas} \,\wedge\, \mathbf{valueof}(q_2)\mu = (n, r) \;\Rightarrow\; g_{\mathcal{I}(A)}(\mathbf{valueof}(n)\mu, r) \in F(\mathbf{valueof}(n)m)$$

This follows directly from the definition of F, since we know that $r \in F(\mathbf{valueof}(n)m)$, because $\mu \in \text{Feas}$.

Case: $q.p$

$$\begin{array}{rcl} C[\![q.p]\!]\mathcal{I}(A)\,m & = & \mathbf{tuple}(\text{id},\ \mathbf{valueof}(q)); \text{apply}^p_A \\ & = & \mathbf{tuple}(\text{id},\ \mathbf{valueof}(q)); \lambda\mu, (n, r).\ \mu[p_A(\mathbf{valueof}(n)\mu, r)/n] \end{array}$$

If we assume $\mu \in \text{Feas}$, and that $\mathbf{valueof}(q)\mu = (n, r)$, then we know that $\mathbf{valueof}(n)\mu \in \text{Reach}$ and $r \in F(\mathbf{valueof}(n)\mu)$. By the definition of $\mathcal{I}(A)$:

$$\mathbf{valueof}(n)\mu \leq p_{\mathcal{I}(A)}(\mathbf{valueof}(n)\mu, r)$$

Therefore, by the definitions of F and Reach:

$$\begin{array}{c} \text{Reach}(p_{\mathcal{I}(A)}(\mathbf{valueof}(n)\mu, r)) \,\wedge \\ \forall r' \in F(\mathbf{valueof}(n)\mu).\ r' \in F(p_{\mathcal{I}(A)}(\mathbf{valueof}(n)\mu, r)) \end{array}$$

The result then follows from the definition of $\{\_/\_\}$.
$\square$

Condition 2. This is established by structural induction much as in the proof of appendix B. The only non-routine cases are as follows:

Case: $q.f$

By the definition of $E[\![q.f]\!]$, Feas and W_i:

$$\begin{array}{rcl} E[\![q.f]\!]\mathcal{I}(A) & = & \mathbf{tuple}(\text{id},\ \mathbf{valueof}(q)); \text{apply}^f_{\mathcal{I}(A)} \\ & = & \mathbf{tuple}(\text{id},\ \mathbf{valueof}(q)); \lambda\mu, (n, r).\ f_{\mathcal{I}(A)}(\mathbf{valueof}(n)\mu, r) \\ & = & \mathbf{tuple}(\text{id},\ \mathbf{valueof}(q)); \lambda\mu, (n, r).\ \text{ev}_A\,|_{W_i}(f(\mathbf{valueof}(n)\mu, r)) \\ & = & \mathbf{tuple}(\text{id},\ \mathbf{valueof}(q)); \lambda\mu, (n, r).\ \text{ev}_A(f(\mathbf{valueof}(n)\mu, r))\,|_{\text{Feas}} \\ & = & \mathbf{tuple}(\text{id},\ \mathbf{valueof}(q)); \lambda\mu, (n, r).\ f_A(\text{ev}_A(\mathbf{valueof}(n)\mu), \text{ev}_A(r))\,|_{\text{Feas}} \\ & = & \phi; \mathbf{tuple}(\text{id},\ \mathbf{valueof}(q)); \lambda\mu, (n, r).\ f_A(\mathbf{valueof}(n)\mu, r)\,|_{\text{Feas}} \\ & = & \phi\,|_{\text{Feas}}; \mathbf{tuple}(\text{id},\ \mathbf{valueof}(q)); \lambda\mu, (n, r).\ f_A(\mathbf{valueof}(n)\mu, r) \\ & = & \phi\,|_{\text{Feas}}; E[\![q.f]\!]A \end{array}$$

Case: $q_1 \leftarrow q_2.g$

Follows by extensionality from the following:

$$
\begin{aligned}
C[\![q_1 \leftarrow q_2.g]\!]\mathcal{I}(A); \mathbf{valueof}(q_1); \phi & \\
&= \mathbf{tuple}(\mathrm{id},\ \mathbf{valueof}(q_2)); \mathrm{apply}^g_A; \phi \\
&= \mathbf{tuple}(\mathrm{id},\ \mathbf{valueof}(q_2)); \lambda\mu, (n, r).\ (n, g_{\mathcal{I}(A)}(\mathbf{valueof}(n)\mu, r)); \phi \\
&= \mathbf{tuple}(\mathrm{id},\ \mathbf{valueof}(q_2)); \lambda\mu, (n, r).\ (n, \mathrm{ev}_A(g_{\mathcal{I}(A)}(\mathbf{valueof}(n)\mu, r))) \\
&= \mathbf{tuple}(\mathrm{id},\ \mathbf{valueof}(q_2)); \lambda\mu, (n, r).\ (n, g_A(\mathrm{ev}_A(\mathbf{valueof}(n)\mu), \mathrm{ev}_A(r))) \\
&= \phi; \mathbf{tuple}(\mathrm{id},\ \mathbf{valueof}(q_2)); \lambda\mu, (n, r).\ (n, g_A(\mathbf{valueof}(n)\mu, r)) \\
&= \phi; C[\![q_1 \leftarrow q_2.g]\!]A; \mathbf{valueof}(q_1)
\end{aligned}
$$

Case: $q.p$

$$
\begin{aligned}
C[\![q.p]\!]\mathcal{I}(A); \phi &= \mathbf{tuple}(\mathrm{id},\ \mathbf{valueof}(q)); \mathrm{apply}^p_{\mathcal{I}(A)}; \phi \\
&= \mathbf{tuple}(\mathrm{id},\ \mathbf{valueof}(q)); \lambda\mu, (n, r).\ \mu[p_{\mathcal{I}(A)}(\mathbf{valueof}(n)\mu, r)/n]; \phi \\
&= \mathbf{tuple}(\mathrm{id},\ \mathbf{valueof}(q)); \lambda\mu, (n, r).\ \phi(\mu)[\mathrm{ev}_A(p_{\mathcal{I}(A)}(\mathbf{valueof}(n)\mu, r))/n] \\
&= \mathbf{tuple}(\mathrm{id},\ \mathbf{valueof}(q)); \\
&= \quad \lambda\mu, (n, r).\ \phi(\mu)[p_A(\mathrm{ev}_A(\mathbf{valueof}(n)\mu), \mathrm{ev}_A(r))/n] \\
&= \phi; \mathbf{tuple}(\mathrm{id},\ \mathbf{valueof}(q)); \lambda\mu, (n, r).\ \mu[p_A(\mathbf{valueof}(n)\mu, r)/n] \\
&= \phi; C[\![q.p]\!]A
\end{aligned}
$$

□

For the remaining proofs, we employ the principle of *noetherian induction*. This may be introduced for reference purposes as follows:

Definition 60 *An element a of a set A partially ordered by $<$ is a* minimal element *of A iff $\nexists x \in A.\ x < a$.*

A set A satisfies the minimum condition *iff every non-empty subset of A has a minimal element.*

Theorem 14 (Principle Of Noetherian Induction) *Let A be a partially ordered set which satisfies the minimum condition. Let $B \subseteq A$ contain an element $a \in A$, whenever it contains all the elements $x \in A$ such that $x < a$. Then $B = A$.*[1]

That $(T_{\Sigma_i})_{\mathrm{State}}$, ordered by $<$, satisfies the minimum condition follows from the fact that for all $s \in (T_{\Sigma_i})_{\mathrm{State}}$, the set $\{s' \mid s' \le s\}$ is totally ordered by $<$.

Condition 3. This may be demonstrated by a noetherian induction on $s \in (T_{\Sigma_i})_{\mathrm{State}}$. The proposition is trivially true for start^j. Assume it is true for all $s' < s$. Let $r, r' \in F(s)$,

[1] A familiar special case of this principle, known as *transfinite*, or *well-ordered* induction results when A is *well-ordered*—i.e. when every subset of A has a *unique* minimal element. We require the fully general case, since as defined, the relation $<$ on a carrier of a term algebra is not always a well-ordering.

for $r \neq r'$. If either $r = \text{root}^j$ or $r' = \text{root}^j$, the hypothesis holds, for from the definition of $F(s)$, if $r = \text{root}^{j_1}$ and $r = \text{root}^{j_2}$, then $j_1 = j_2$. Otherwise we see that either $\text{origin}(r) < \text{origin}(r')$ or $\text{origin}(r') < \text{origin}(r)$ or $\text{origin}(r) = \text{origin}(r')$. In the first two cases, we are done. In the final case, by the construction of $F(s)$, r and r' must be of the form:

$$\begin{aligned} r &= \overbrace{g(s, \ldots g(s,}^{j \text{ times}} r_1) \ldots) \\ r' &= \underbrace{g(s, \ldots g(s,}_{k \text{ times}} r_2) \ldots) \end{aligned}$$

for $j, k \geq 1$, $r_1, r_2 \in (T_{\Sigma_i})_{\text{Ref}}$, and $\text{origin}(r_1), \text{origin}(r_2) < s$. If $j \neq k$, then we are done. Otherwise, the proposition follows from the induction assumption.
□

Note that the above establishes the well-definedness of $\max(P)$.

A similar argument proves the following result:

Lemma 4 *For $s \in (T_{\Sigma_i})_{\text{State}}$, $r \in F(s)$, $g \in \text{Id}_g$, $r \sqsubset g(s, r)$.*

To establish noetherian induction on sets (s, P), $P \subseteq F(s)$, define:

Definition 61 *For $s \in (T_{\Sigma_i})_{\text{State}}$, $s_1, s_2 \leq s$, $P_1 \subseteq F(s_1), P_2 \subseteq F(s_2)$, let:*

$$(s_1, P_1) < (s_2, P_2) \stackrel{\text{def}}{\Longleftrightarrow} \begin{cases} s_1 < s_2 & \text{or} \\ s_1 = s_2 \wedge \max(P_1) < \max(P_2) \end{cases}$$

Lemma 5 *As defined above, $<$ on the set $\{(s, P) | s \leq s', P \subseteq F(s)\}$, for some fixed $s \in (T_{\Sigma_i})_{\text{State}}$, satisfies the minimum condition.*

Proof:

From the fact that $<$ totally orders $\{s' \mid s' \leq s\}$ and $\sqsubset$ totally orders $F(s)$.
□

Now define induction hypothesis, $H(s, P)$:

$$H(s, P) \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \forall r \in P.\ \mathbf{valueof}(\nu(r))\mu = (1, r_A) \ \wedge\ \mathbf{valueof}(1)\mu = s_A,$$

where $\mu = C[\![\gamma(s, P)]\!]A\,\text{init}(A)$.

By noetherian induction, the truth of the proposition follows if we can show that for all $s \in (T_{\Sigma_i})_{\text{State}}$ such that $\text{Reach}(s)$, and all $P \subseteq F(s)$:

$$\forall((s', P') < (s, P).\ H(s, P)) \ \Rightarrow\ H(s, P)$$

The proof requires examination of three cases:

Case: $(s, P) = (\mathrm{new}^j, \{\mathrm{root}^j\})$

Directly from the definitions.

Case: $\mathrm{origin}(\max(P)) = s$

Then $\max(P) = g(s, r')$, for some $r' \in F(s)$. By lemma 4, $r' \sqsubset g(s, r')$, so $(s, P \cup \{r'\} \setminus \{r\}) < (s, P)$, and thus $H(s, P \cup \{r'\} \setminus \{r\})$. Now let:

$$\mu' \stackrel{\mathrm{def}}{=} C[\![\gamma(s, \{r'\} \setminus \{r\})]\!] A \, \mathrm{init}(A)$$

By the induction hypothesis:

$$\begin{array}{lcl} \mathbf{valueof}(\nu(r'))\mu' & = & (1, r'_A) \\ \mathbf{valueof}(1)\mu' & = & s_A \end{array}$$

So:

$$\begin{array}{lcl} \mu & = & C[\![\gamma(s, P)]\!] A \, \mathrm{init}(A) \\ & = & \{\mathbf{tuple}(\mathrm{id}, \mathbf{valueof}(\nu(r)); \mathrm{apply}^g_A)/\nu(r)\}\mu' \\ & = & \{(1, g_A(\mathrm{rval}(1)\mu', r'_A))/\nu(r)\}\mu' \\ & = & \{(1, g_A(s_A, r'_A))/\nu(r)\}\mu' \\ & = & \{(1, r_A)/\nu(r)\}\mu' \end{array}$$

Case: $\mathrm{origin}(\max(P)) < s$

Then $s = p(s', r')$, for some $s' < s$ and $r' \in F(s')$ (by $\mathrm{Reach}(s)$). Since $\mathrm{origin}(\max(P)) \leq s'$ and $P \cup r' \subseteq F(s')$, we have that $(s', P \cup r') < (s, P)$, and thus $H(s', P \cup r')$. Now let:

$$\mu' \stackrel{\mathrm{def}}{=} C[\![\gamma(s', P \cup r')]\!] A \, \mathrm{init}(A)$$

By the induction hypothesis:

$$\begin{array}{lcl} \mathbf{valueof}(\nu(r'))\mu' & = & (1, r'_A) \\ \mathbf{valueof}(1)\mu' & = & s_A \end{array}$$

So:

$$\begin{array}{lcl} \mu & = & C[\![\gamma(s, P)]\!] A \, \mathrm{init}(A) \\ & = & \{\mathbf{tuple}(\mathrm{id}, \mathbf{valueof}(\nu(r'))); \mathrm{apply}^p_A/1\}\mu' \\ & = & \{p_A(s'_A, r'_A)/1\}\mu' \\ & = & \{s_A/1\}\mu' \end{array}$$

□

Bibliography

[AdBKR86] P. America, J. de Bakker, J. N. Kok, and J. Rutten. A denotational semantics of a parallel object-oriented language. Technical Report CS-R8626, Center for Mathematics and Computer Science, Amsterdam, 1986.

[Agh86] G. Agha. *ACTORS: A Models of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[AGOO86] A. Albano, G. Ghelli, M. E. Occhiuto, and R. Orsini. A strongly-typed interactive object-oriented database programming language. In *International Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, 1986.

[AK90] S. Abiteboul and C. Kanellakis. Object identity as a query language primitive. In *School/Workshop on Foundations of Object-Oriented Languages*, May 1990.

[AM74] M. A. Arbib and E.G. Manes. Machines in a category: An expository introduction. *SIAM Review*, 16(2):163–192, April 1974.

[Ame90a] P. H. M. America. A behavioural approach to subtyping. In *School/Workshop on Foundations of Object-Oriented Languages*, May 1990.

[Ame90b] P. H. M. America. A parallel object-oriented language with inheritance and subtyping. In *Joint Conference on Object-Oriented Programming: Systems, Languages and Applications and European Conference on Object-Oriented Programming*, 1990.

[Ano90] Anonymous. Culturing change. *The Economist*, July 1990.

[B$^+$84] G. M. Birtwistle et al. *Simula Begin*. Chartwell-Bratt, 1984. 2nd edition.

[BCL85] G. Berry, P.-L. Curien, and J.-J. Lévy. Full abstraction for sequential languages: The state of the art. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*. Cambridge University Press, 1985.

[BCS86] J. Bonar, R. Cunningham, and J. Schultz. An object-oriented architecture for intelligent tutoring systems. In *Conference on Object-Oriented Programming, Systems and Languages*, 1986.

[BDG+87] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S. Keene, G. Kiczales, and D.A. Moon. The Common Lisp Object System, 1987. X3J13 Document 87-002.

[Ber79] G. Berry. *Modèles Complètement Adéquats et Stables des Lambda-Calculs*. PhD thesis, UniversitParis VII, 1979.

[BG80] R. M. Burstall and J. A. Goguen. The semantics of Clear, a specification language. In *Advanced Course on Abstract Software Specifications*, Lecture Notes in Computer Science. Springer-Verlag, 1980.

[BM84] R. J. R. Back and H. Mannila. A semantic approach to program modularity. *Information and Control*, 60:138–167, 1984.

[Boo86] G. Booch. Object-oriented development. *IEEE Transactions on Software Engineering*, February 1986.

[Boo91] G. Booch. *Object Oriented Design with Applications*. Benjamin/Cummings, 1991.

[Bro75] F. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1975.

[Bro85] S. Brookes. A fully abstract semantics and a proof system for an ALGOL-like language with sharing. In *Conference on the Mathematical Foundations of Programming Semantics*, Kansas State University, 1985.

[Bro87] F. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4), 1987.

[BW82] M. Broy and M. Wirsing. Partial abstract types. *Acta Informatica*, 18(1), 1982.

[CM87] CISI Ingenierie CRI and Matra. HOOD: Hierarchical object-oriented design. Paris, 1987.

[Coi87] P. Cointe. Metaclasses are first class: The ObjVLisp model. In *Conference on Object-Oriented Programming, Systems and Languages*, Orlando, Florida, 1987.

[Coo89a] W. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, March 1989.

[Coo89b] W. Cook. A proposal for making Eiffel type-safe. In *European Conference on Object-Oriented Programming*, Nottingham, 1989.

[Cox86] B. Cox. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1986.

[CP89] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *Conference on Object-Oriented Programming, Systems, Languages and Applications*, 1989.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4), December 1985.

[Deu81] L. P. Deutsch. Building control structures in the Smalltalk-80 system. *Byte*, 6(8), August 1981.

[Deu89] L. P. Deutsch. The past, present and future of Smalltalk. In *European Conference on Object-Oriented Programming*, Nottingham, 1989.

[Dig86] Digitalk, Inc., Los Angeles, CA. *Smalltalk/V—Tutorial and Programming Handbook*, 1986.

[DS84] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *11th Anual Symposium on Principles of Programming Languages*, 1984.

[DT88] S. Danforth and C. Thomlinson. Type theories and object-oriented programming. *Computing Surveys*, 20(1), March 1988.

[Duf86] C. B. Duff. Designing an efficient language. *Byte*, August 1986.

[EFH83] H. Ehrig, W. Fey, and H. Hansen. ACT ONE—An algebraic specification language with two levels of semantics. FB 20 83-03, T.U. Berlin, 1983.

[EW67] S. Eilenberg and J. B. Wright. Automata in general algebras. *Information and Control*, 11:452–470, 1967.

[Fal83] J. R. Falcone. The analysis of the Smalltalk-80 system at Hewlett-Packard. In G. Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, 1983.

[Fis87] M. Fisher. Abstractness and compositionality. Technical report, Manchester University, 1987.

[FS83] J. R. Falcone and J. R. Stinger. The Smalltalk-80 implementation at Hewlett-Packard. In G. Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, 1983.

[GA68a] Y. Give'on and M. A. Arbib. Algebra automata I: Prologomena. *Information and Control*, 12:331–345, 1968.

[GA68b] Y. Give'on and M. A. Arbib. Algebra automata II: The categorical framework for dynamic analysis. *Information and Control*, 13:346–370, 1968.

[Gan83] H. Ganzinger. Parameterized specifications: Parameter passing and implementation with respect to observability. *ACM Transactions on Programming Languages and Systems*, 5(3), 1983.

[GM82] J. Goguen and J. Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. In *9th Colloquium on Automata, Languages and Programming (LNCS 140)*. Springer-Verlag, 1982.

[GM85] J. Goguen and J. Meseguer. Initiality, induction and computability. In M. Nivat and J.C. Reynolds, editors, *Algebraic Methods in Semantics*. Cambridge University Press, 1985.

[GM86] J. A. Goguen and J. Meseguer. Extensions and foundations of object-oriented programming. *SIGPLAN Notices*, October 1986.

[Gol81] R. Goldblatt. *Axiomatizing the Logic of Computer Programming*. Lecture Notes in Computer Science (130). Springer Verlag, 1981.

[Gol84] A. Goldberg. *Smalltalk 80: The Interactive Programming Environment*. Addison-Wesley, 1984.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Gra69] G. Graetzer. *Universal Algebra*. Van Nostrand, 1969.

[GTW78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebras approach to the specification, correctness and implementation of abstract datatypes. In R.T. Yeh, editor, *Current Trends in Programming Methodology (Volume IV: Data Structuring)*. Prentice-Hall, 1978.

[Gut75] J. V. Guttag. The specification and application to programming of abstract data types. Technical Report CSRG-59, University of Toronto, 1975.

[Hen90] A. V. Hense. Denotational semantics of an object-oriented programming language with explicit wrappers. Technischer Bericht A 11/90, Universitaet des Saarlandes, June 1990.

[HHS85] J. He, C.A.R. Hoare, and J.W. Sanders. Data refinement refined (resumé). Technical report, Oxford University Computer Laboratory Programming Research Group, 1985.

[HMT84] J. Halpern, A. Meyer, and B. Trakhtenbrot. The semantics of local storage, or: What makes the free-list free? In *11th ACM Symposium on Principles of Programming Languages*, 1984.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1, 1972.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[Hol82] W. M. L. Holcombe. *Algebraic Automata Theory*. Cambridge Unversity Press, 1982.

[HP79] M. Hennessy and G. Plotkin. Full abstraction for a simple parallel programming language. In *8th MFCS (LNCS 74)*. Springer-Verlag, 1979.

[Irv72] A. H. Irvine, editor. *Collins English Dictionary*. Collins, 1972.

[Joh87] P. T. Johnstone. *Notes on Logic and Set Theory*. Cambridge University Press, 1987.

[Jon86] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.

[Kam88] S. Kamin. Inheritance in Smalltalk-80: A denotational definition. In *ACM Symposium on the Principles of Programming Languages*, 1988.

[Kap58] J. D. Kaplan, editor. *The Pocket Aristotle*. Washington Square Press, 1958.

[KKW84] J. C. Kuntz, T. P. Kehler, and M. D. Williams. Applications development using a hybrid AI development system. *AI Magazine*, September 1984.

[KMMPN85] B. O. Kristensen, O. Madsen, B. Moller-Pederson, and K. Nygaard. Multisequential execution for the BETA programming language. *SIGPLAN Notices*, April 1985.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language (2nd ed.)*. Prentice-Hall, 1988.

[LAB+81] B. Liskov, R. Atkinson, T. Bloom, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*. Lecture Notes in Computer Science 114. Springer-Verlag, 1981.

[Lew90] T. G. Lewis. Object-oriented programming a hit. *IEEE Spectrum*, January 1990.

[LZ74] B. Liskov and S. Zilles. Programming with abstract data types. *SIGPLAN Notices*, 9(4), 1974.

[Mac72] S. MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1972.

[Mae87] P. Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, January 1987.

[Man89] T. Manuel. What's behind all the software delays? *Electronics*, June 1989.

[Mar85] J. L. Martin. MACH-1: An operating environment for large-scale multiprocessor applications. *IEEE Software*, July 1985.

[Men90] B. Meng. No tricky loops with the oops troops. *MacWorld*, January 1990.

[Mey88] B. Meyer. *Object-Oriented Software Software Construction*. Prentice-Hall, 1988.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Second International Joint Conference on Artificial Intelligence*, London, 1971.

[Mil75] R. Milner. Processes: a mathematical model of computing agents. In *Logic Colloquium 75*. North Holland, 1975.

[Mil77] R. Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 1977.

[Mil80] R. Milner. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science 92. Springer Verlag, 1980.

[Mir87] E. Miranda. BrouHaHa—A portable Smalltalk interpreter. In *Conference on Object-Oriented Programming, Systems and Languages*, 1987.

[Mul87] K. Mulmuley. *Full Abstraction and Semantic Equivalence.* MIT Press, 1987.

[Nip87] T. Nipkow. *Behavioural Implementation Concepts for Nondeterministic Data-Types.* PhD thesis, Dept. of Computer Science University of Manchester, 1987. Rep. UMCS-87-5-3.

[Nor88] N. D. North. Progress report on a framework for the semantics of object-oriented languages. Technical Report DITC 114/88, National Physical Laboratory, April 1988.

[OH84] E.-R. Olderog and C. A. R. Hoare. Specification-oriented semantics for communicating processes. Technical Monograph PRG-37, Oxford University Computing Laboratory Programming Research Group, February 1984.

[Par72] D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5), May 1972.

[Par90] ParcPlace Systems, Inc., San Fransisco, CA. *ObjectsWorks\Smalltalk (Version 4)—User's Guide*, 1990.

[PC86] A. Wirfs-Brock P. Caudill. A third-generation Smalltalk-80 interpreter. In *Conference on Object-Oriented Programming, Systems and Languages*, 1986.

[Plo77] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5, 1977.

[Pou86] D. Pountain. Object-oriented FORTH. *Byte*, August 1986.

[Red88] U. S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *ACM Symposium on the Principles of Programming Languages*, 1988.

[Ren82] T. Rensch. Object-oriented programming. *ACM SIGPLAN Notices*, 17(9), September 1982.

[Sak89] M. Sakkinen. Disciplined inheritance. In *European Conference on Object-Oriented Programming*, Nottingham, 1989.

[Sau89] J. H. Saunders. A survey of object-oriented programming languages. *Journal of Object-Oriented Programming*, 1(6), March/April 1989.

[SC89] S. L. Stepoway and M. G. Christiansen. Object-oriented fractal modeling on a shared-memory mimd machine. *Journal of Object-Oriented Programming*, 2(1), May/June 1989.

[SCBK86] C. Schaffert, T. Cooper, B. Bullis, and M. Kilian. An introduction to Trellis/Owl. In *Conference on Object-Oriented Programming, Systems and Languages*, Portland, Oregon, 1986.

[Sch87] O. Schoett. *Data Abstraction and the Correctness of Modular Programming*. PhD thesis, University of Edinburgh, 1987. CST-42-87.

[SE90] B. Stroustrup and M. A. Ellis. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[Sha81] M. Shaw, editor. *Alphard: Form and Content*. Springer-Verlag, 1981.

[Sny87] A. Snyder. Inheritance and the development of encapsulated software components. In P. Wegner and B. Shriver, editors, *Research Directions on Object-Oriented Programming*. MIT Press, 1987.

[Spi88] J. M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge Tracts in Theoretical Computer Science 3. Cambridge University Press, 1988.

[SS75] G. Sussmand and G. Steele. SCHEME: An interpreter for extended lambda calculus. Technical Report Memo 349, MIT, 1975.

[ST87] D. Sanella and A. Tarlecki. On observational equivalence and algebraic specification. *Journal of Computer and Systems Science*, 43:150–178, 1987.

[ST88] D. Sanella and A. Tarlecki. Toward formal development of programs from algebraic specifications: Implementations revisited. *Acta Informatica*, 25:233–281, 1988.

[Sto88] A. Stoughton. *Fully Abstract Models of Programming Languages*. Research Notes in Theoretical Computer Science. Pitman, 1988.

[Str88] B. Stroustrup. What is object-oriented programming? *IEEE Software*, 5(3), 1988.

[Sun88] Sun Microsystems, Inc., San Fransisco, CA. *SunView I Programmer's Guide*, 1988.

[Wag84] E. G. Wagner. Semantics of block-structured languages with pointers. In *Mathematical Foundations of Programming Language Semantics*, LNCS (298). Springer-Verlag, 1984.

[Weg87] P. Wegner. Dimensions of object-based language design. In *Conference on Object-Oriented Programming, Systems and Languages*, 1987.

[Wir82] N. Wirth. *Programming in Modula-2.* Springer-Verlag, 1982.

[Wol87] M. Wolcko. Semantics of Smalltalk-80. In *European Conference on Object-Oriented Programming.* Springer-Verlag, June 1987.

[Wol88] M. Wolczko. *Semantics of Object-Oriented Languages.* PhD thesis, University of Manchester Department of Computer Science, 1988. Technical Report UMCS-88-6-1.

[YBS86] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference on Object-Oriented Programming, Systems and Languages*, Portland, Oregon, 1986.

[YC79] E. Yourdon and L. Constantine. *Structured Design.* Prentice-Hall, 1979.

[Yel89] P. M. Yelland. First steps towards fully abstract semantics for object-oriented languages. In *European Conference on Object-Oriented Programming*, Nottingham, 1989.

[Yel90] P. M. Yelland. Producing abstract models for object-oriented languages. In *School/Workshop on Foundations of Object-Oriented Languages*, May 1990.

[YT86] Y. Yokote and M. Tokoro. The design and implementation of Concurrent Smalltalk. In *Conference on Object-Oriented Programming, Systems and Languages*, Portland, Oregon, 1986.

[YT87] A. Yonezawa and M. Tokoro, editors. *Object-Oriented Concurrent Programming.* MIT Press, 1987.

[Zil73] Stephen N. Zilles. Procedural encapsulation: A linguistic protection mechanism. *SIGPLAN Notices*, 8(9), 1973.