Department of Computer Science
University of Manchester

Manchester M13 9PL, England

Technical Report Series

UMCS–88–6–1

Mario Wolczko

Semantics of Object-Oriented Languages

SEMANTICS OF OBJECT-ORIENTED LANGUAGES

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE

By
Mario I. Wolczko
Department of Computer Science
March 1988

DECLARATION

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Abstract

Object-oriented programming is becoming an important technique in the construction of large software systems. Compelling arguments, like reduced maintenance costs, are advanced to encourage its use. To maximise the advantages of such methods, object-oriented programming languages need to be well-designed. When selecting the main features of a programming language, or choosing between alternative designs, formal methods of semantic analysis are invaluable. To date little attention has been given to the formal description of object-oriented languages. This thesis introduces a framework for describing the semantics of object-oriented languages.

To characterise the important features of object-oriented languages, an idealised object-oriented language is described and its semantics specified formally, using the denotational style of VDM. Design alternatives are explored in the same way. Several general principles of object-oriented language design are introduced, and the alternatives reviewed in light of these principles. By choosing apposite semantic domains, the fundamental concepts of object-oriented systems are exposed: two message-passing schemes, based on dynamically-bound procedure call and delegation, are presented; class- and prototype-based systems are described; and special emphasis is given to the different approaches to class-based multiple inheritance. The encapsulation of behaviour within classes is discussed, and suggestions are made as to how this might be best achieved. A variety of object-oriented control mechanisms are also surveyed.

Для батька, матері й брата Левка

Дякую Вам за все.

Acknowledgements

Many people have beneficially influenced this thesis. Foremost among them is Cliff Jones, my supervisor. His deep insight into programming language design and description, and his ability to ask simple yet penetrating questions, have vastly improved the quality of this work. I must also thank Ifor Williams and Trevor Hopkins for numerous stimulating conversations about object-oriented programming, and my colleagues in 2.90 for a convivial atmosphere in which to do research. Many coffee-break discussions with Michael Fisher were also helpful in crystallising my ideas. Andrew Barnard and Michael Fisher receive special thanks for their careful reading of a draft of the thesis.

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

Colophon: This thesis was typeset in Times Roman using LaTeX, and printed on an Apple LaserWriter Plus.

Contents

List of Figures

Chapter 1

Introduction

> Computer science is the first engineering discipline ever in which the complexity of the objects created is limited by the skill of the creator and not limited by the strength of the raw materials. If steel beams were infinitely strong and couldn't ever bend no matter what you did, then skyscrapers could be as complicated as computers.
>
> *Brian K. Reid*

The aim of the object-oriented approach to programming is to control complexity. Object-oriented design consists of decomposing a problem into subproblems, each of which can be mapped onto an "object." The major advance made by object-oriented programming languages over conventional languages is that modularity is enforced at the semantic level as well at the syntactic level. In a pure object-oriented language, side effects cannot occur.

The structure imposed by the object-oriented approach enables larger, more complex systems to be built and maintained than was previously possible. In

the opinion of the author, this key advance is likely to lead to the widespread adoption of object-oriented programming techniques.

Although object-oriented programming is not a recent invention, there is no widely-agreed definition of the term. In 1982, Rentsch [Ren82] wrote:

> What is object oriented programming? 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 favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is.

Unfortunately, his predictions seem to be coming true. "Object-oriented programming", together with "structured programming" and "user-friendly" are rapidly becoming "noise" words.

This thesis attempts to define "object-oriented programming", by defining what an object-oriented programming language is. Essential features and important characteristics of object-oriented languages are described using formal techniques. A series of small, hypothetical object-oriented languages is developed, and their semantics are defined denotationally. Major features of object-oriented languages are surveyed formally.

1.1 Background and Aims

Until recently there were few object-oriented programming languages. Their history begins with Simula, which introduced the notions of object, class and inheritance [BDMN73]. Dahl, one of the inventors of Simula, has described the motivation behind the introduction of classes and objects into Simula from a desire to have multiple, co-existing instances of ALGOL-like blocks [Dah87]. The identifiers within each block were to be in private scopes; this is the hallmark of an object-oriented language. These blocks are now known as objects. Objects with identical behaviour were grouped into classes, and parts of the code associated with each class could be inherited from other classes. Every ALGOL-like procedure was associated with a class, and only the internal variables of objects belonging to that class were in scope within a procedure. These restricted forms of procedure have come to be known as methods.

Building on the ideas from Simula, the series of Smalltalk languages refined the notion of object-oriented programming, and were the first "pure" object-oriented languages [Ing83, Sho79, Ing78, GR83]. (By "pure", we mean that all values are objects.) Simula allowed one object to access the internal state of another directly; Smalltalk insists that all inter-object communication is accomplished by objects sending messages to each other, where a message is a dynamically-bound procedure call invoking a method.

Since Smalltalk-80,[1] new object-oriented languages have been invented. Most are hybrid languages, mixing object-oriented features with older, conventional languages. For example, C++ and Objective-C add object-oriented features to C [Str86, Cox86]. A family of Lisp-based object-oriented languages has arisen: LOOPS and Flavors, along with their Common Lisp successors, CommonLoops, New Flavors, CommonObjects and the Common Lisp Object System [BS83, WM80, BKK+86, Moo86, Sny85, DG87]. New, pure object-oriented languages have also been invented (e.g., Trellis/Owl [SCB+86], Eiffel [Mey87] and POOL [Ame85]). These languages have contributed several new ideas to object-oriented programming. The development of the actor-based languages [Agh86] has progressed in parallel; their emphasis is on concurrent models of computation.

Whilst this list is not exhaustive, it gives some impression of the diversity of object-oriented languages. However, this thesis shows that a few simple principles underlie all the object-oriented features of these languages. The essence of object-oriented languages is explained by identifying their intrinsic features and describing each in isolation. Using a minimal core language, the principles of objects, messages, dynamic binding and inheritance are explained formally, using denotational semantics. This has two benefits: a denotational definition abstracts away irrelevant detail, concentrating on the important aspects; it also serves as a comparison with the denotational definitions of other programming styles [BJ82, RC86, Sch78, Ten73]. In this way, this thesis answers the questions, "What is object-oriented programming?" and, "How is it related to other programming styles?"

Having provided a formal taxonomy for object-oriented languages, another

[1] Henceforth, "Smalltalk" will be used to denote the Smalltalk-80 programming language, as defined in [GR83].

aim of this thesis is to outline principles and guidelines for the designers of future object-oriented languages. This task is inseparable from the formal definition; the definition provides many insights into the design of these languages, and suggests ways of improving them. In the author's opinion, the best foundation for language design is the study of semantic models; others have also concluded this [Ten77, AW82]. It is hoped that the models and principles presented here will aid future designers.

The semantics presented in this thesis are denotational, in the style of Bjørner and Jones [BJ82], although the notation is based on more recent VDM [Jon86]. A model-oriented approach has been used in preference to an axiomatic approach because it is felt that language designs can be analysed more clearly in this way; axiomatic semantics are better suited to reasoning about programs.

1.2 Limitations of this Work

Language features not germane to object-oriented programming have been ignored. This means that some features, important to the art of programming and useful in object-oriented programs, have been left out. In particular, the interaction between concurrency and object-oriented programming has not been treated thoroughly. Agha provides an operational definition of concurrency in the actor model of computation in [Agh86], and America *et al.* define concurrency in the POOL languages in [AdBKR86]. Chapter 2 discusses these models and explains why they are not suited to the wider variety of languages described here.

The treatment of the notion of "program" is also somewhat unusual. Conventional denotational definitions (e.g., the definition of Pascal in [BJ82]) model a program as a function from inputs to outputs; program executions have a finite duration, and their only observable effects (semantically speaking) are their outputs. This model is inappropriate when applied to persistent systems [ABC+83], in which the events of semantic interest are mainly changes to the persistent store. In the persistent model of programming, the data manipulated by a program can outlive that program without being explicitly "output." When reasoning about the behaviour of program code in a persistent system, one is interested in the effects of the code on the persistent data, rather than in

any "output."[2] Another property of this approach is that errors do not cause the program to terminate with no output; changes made to persistent data remain. Because a number of object-oriented languages are based on the persistent model, and the semantics of input/output and errors are not particularly connected with object-oriented programming, these aspects have been omitted from the semantics.

Inheritance is described in Chapter 4. Although inheritance is widely associated with object-oriented programming, there is no reason to think that object-oriented programming necessarily includes inheritance, or that inheritance could not be used in other programming styles. However, this thesis makes no attempt to describe inheritance in more general terms. For a discussion of inheritance in relation to logical reasoning, the reader is referred to [Tou86].

Another area not addressed by this thesis is that of type systems for object-oriented languages. The languages described are strongly typed in the sense of Smalltalk (i.e., one cannot apply operations to values on which those operations are undefined), but are not statically typed. More is said about this in Chapter 6.

1.3 Overview of the Thesis

Chapter 2 outlines the scope of the formal models introduced in the thesis. Several previous models are described, and their usefulness with respect to the variety of language features described in this thesis is assessed. In Chapter 3, a simple model of object-oriented languages is introduced, by defining a minimal language and its semantics. Fundamental issues of object-oriented programming are discussed. Chapter 4 adds inheritance to the language of Chapter 3. Single and multiple inheritance schemes are defined, and their relative merits discussed. Control aspects of object-oriented languages are described in Chapter 5. Conclusions are drawn in Chapter 6. The semantics described in Chapters 3–5 are gathered together into complete definitions in Appendices A and B. An index to the types and functions defined in Chapters 3, 4 and 5 can be found in Appendix D.

[2] The Smalltalk system has no fundamental notion of output at all; it is a side effect of manipulating bits in a bitmap, which just happens to be projected onto the screen of a CRT.

Throughout the text the reader is assumed to be familiar with the techniques of denotational semantics. Schmidt [Sch86] is an excellent introduction to the field. To avoid using a diversity of notations, descriptions of previous work in this area have been "translated" into VDM notation where possible. A summary of the notation used can be found in Appendix C. Acquaintance with the basic concepts of an object-oriented language is also helpful. The reader with knowledge of Smalltalk-80 will find the languages described in this thesis easy to assimilate.

The feminine pronoun "she" has been used in preference to the clumsier "he/she"; similarly "hers" has been used in place of "his/hers", etc. A masculine version of this thesis is available on request from the author.

Chapter 2

Formal Semantics of Object-Oriented Languages

> No human investigation can be called real science if it cannot be demonstrated mathematically.
>
> *Leonardo da Vinci*

> It is easier to square the circle than to get round a mathematician.
>
> *Augustus De Morgan*

The central aim of this thesis is to provide the framework for formal models of a variety of object-oriented languages. To achieve this, the semantics of a minimal language are defined in Chapter 3, and extended in Chapters 4 and 5. The purpose of this language is to serve as the glue holding together a number of features that are central to object-oriented programming. Clearly, the choice

of features influences the structure of the formal model. The purpose of this chapter is to outline the set of important features, and in what ways previous models do not support them.

2.1 Objects, their Organisation and Inter-Object Communication

The most important aspect of a formal model of object-oriented languages is its characterisation of objects. All object-oriented systems have similar notions of what an object is, and the formal model should describe those notions succinctly. It should capture the essence of "objectness": the idea that an object has an identity that distinguishes it from all other objects (even those with identical contents), and that an object has internal state that is inaccessible to other objects. Chapter 3 is devoted to this issue, and introduces two principles by which the "object-orientedness" of a system can be judged.

Two other questions need to be addressed: "How are objects organised into systems?" and "How do objects communicate?" Organisation of objects introduces the concepts of class and inheritance. A model of object-oriented languages has to capture the ideas of class and inheritance, and do it in a way that is not specific to any particular scheme of inheritance. Several different schemes for multiple inheritance exist, and a formal model must be able to describe all of them. Some object-oriented languages reject the idea that classes are fundamental concepts, and prefer to base their organisation on prototypes. It is desirable that a formal model accommodate these languages. These ideas are described in Chapters 3 and 4.

In most object-oriented languages, objects communicate by passing messages, where message-passing takes the form of dynamically-bound procedure calls. When a message is sent from one object to another, a token identifying the message, called a *selector*, is used to select the procedure to be invoked. The selector, in effect, names the procedure, but different objects may have different procedures, or *methods*, bound to the same selector. Hence, a selector is an overloaded procedure identifier. The semantics of this form of message-passing are described in Chapter 3.

Other approaches to message-passing, based on communication between

concurrent processes, have also been suggested. In these systems, every object is associated with a process, and a message is either a rendezvous between two objects (their processes synchronise, and exchange data [AdBKR86]), or is a "mail item", queued at the destination object until it is ready to be processed [Agh86]. Although concurrent message-passing may become widespread in the future, this thesis concentrates on the procedure-call model of message-passing, because the techniques for formal description of concurrent computation are not as well-developed or well-understood as those for sequential languages. The use of these techniques can complicate a definition to such an extent that other, important aspects of the definition are obscured. As it is unlikely that a single model could comfortably embrace both kinds of message-passing, the most widespread (and to date, important) kind is described.

A successful model will address these three issues, namely the nature of objects, their organisation and communication, in a way that brings out the crucial aspects of a language's design. Although the naturalness of a semantic definition is subjective, this should not be considered unimportant.

The remainder of this chapter describes previous formal models of object-oriented languages, and outlines why they are unsuitable for the description of the language features in consideration here. The Generalized Object Model of Hailpern and Nguyen [HN87] attempts to cover similar areas to this work, and is reviewed first. Summaries of the concurrent models of POOL [AdBKR86] and Actors [Agh86] close the chapter.

2.2 The Generalized Object Model of Hailpern and Nguyen

In [HN87], Hailpern and Nguyen give an informal description of a formal model for objects and inheritance. Their model maps every object onto a CSP process [Hoa85]; objects communicate by passing messages between CSP ports. Every object has two ports: one is used to communicate source-level messages, and the other inherited attributes. The semantics of an object is the set of possible histories of communications on these ports.

Internally, an object consists of a set of local variables, a set of local procedures, and a set of methods. (See Figure 2.1 for a schematic representation of

Figure 2.1: An object in Nguyen and Hailpern's Object Model

an object.) The local variables and procedures cannot be accessed by external objects. A method is invoked by sending a *message request* to the "message request" port of the object. If the object has a locally-defined method response to the message, it is invoked, and a reply sent back to the message port of the original sender. This is termed *direct* execution of a method. Whilst being executed, the method may access the local variables and procedures of the receiver. An object that has sent a message must wait for the reply before proceeding.

If no local method exists for the message, an *inheritance request* containing the original message may be sent to the "inherit" port of a third object. This is known as execution of an *inherited* method. If an object receiving an inheritance request has a method associated with the message, it returns the method, which can then be executed by the original receiver. Otherwise, the inheritance request may be forwarded to another object, which may in turn forward it, and so on, until either a method is returned or the message rejected. Because supplying an inherited method does not cause a change in the local state of an object supplying the method, an object can process many inheritance requests concurrently, possibly in parallel with a single direct execution. (See Figure 2.2 for an illustration of direct and inherited method execution.)

2.2.1 Guards

Associated with each method is a *guard*, which must match an incoming message for the method to be executed. The guard is a triple of the form:

$$[\{\text{REQUEST}, \text{INHERIT}\}] \times [\mathit{Object\_name}] \times [\mathit{Message\_selector}].$$

Each field of the triple is optional, and can be omitted. If a guard has REQUEST as its first field, it only matches messages received on the message port; if it has INHERIT as its first field, it only matches messages received on the inherit port. If neither is present, then a guard may match messages on either port.

The optional *Object_name* in a guard selects messages from specific objects. If it is not present, then all objects match; otherwise only messages from the single, named object may match the guard.

Every message contains a *Message_selector*; it is analogous to a procedure name. A message only matches guards with the same message selector, or

The upper diagram represents an object executing a direct method. In the lower diagram, an object executes an inherited method by forwarding the request to another object, which may in turn forward the request, until a method body is returned.

Figure 2.2: Method execution in the Generalized Object Model

guards that do not contain a message selector.

In summary, messages to the message-port of an object *dest*, from an object *src*, of the form (*selector, params*) can only match guards of the form (REQUEST, *src, selector*) or similar guards with one or more of the fields omitted; similarly for messages to the inherit-port, with REQUEST replaced by INHERIT in the guard. If more than one guard matches a message, the most specific (i.e., the guard with fewest fields omitted) is chosen if it exists; if there is more than one most specific guard, then a choice is made between them non-deterministically.

With each guard is associated one of two types of method:

- A *direct method* is a piece of executable code; if it is associated with a REQUEST guard then it is executed by the receiver of the REQUEST; if it is associated with an INHERIT guard then it is returned as the result. In order to be independent of any particular programming language, Nguyen and Hailpern do not describe the detailed form that the code might take.
- An *inherited method* is a list of *Object_names*; the message is forwarded to each object in the list in turn, until one returns a method.

So that a guard can match against a selected set of source objects, the *Object_name* in the guard can be the name of a *view*. A view is a collection of object names; it can be updated at run-time. A message from any object within the view may match the guard, provided the other two fields also match.

2.2.2 Example

In [HN87], Nguyen and Hailpern give several examples of how their model can be applied to various object-oriented languages; the following is one of their examples.

To model a Smalltalk object, every object has a *class* variable, and the methods for all REQUESTS are inherited from the class. The associated guard and method is:

(REQUEST, nil, nil) → inherit from *class*.

The inherit from clause indicates that the method is inherited from the object referenced by the *class* variable.

Classes, being special kinds of Smalltalk object, have additional guards. Every class has the following variables: a *superclass*; a *class_view*, containing all the instances of the class; and a *subclass_view*, containing all the subclasses of the class. The associated guards and methods are:

(INHERIT, *class_view, m*) → definition of method *m*
(INHERIT, *subclass_view, n*) → definition of method *n*
rest of methods …
(INHERIT, *class_view*, nil) → inherit from *superclass*
(INHERIT, *subclass_view*, nil) → inherit from *superclass*
(REQUEST, nil, nil) → inherit from *class*

Thus, all messages to an object are forwarded to its class. If the class does not define a method for the message, it forwards it to its superclass.

2.2.3 Problems

The Generalized Object Model has several shortcomings that cause it to be rejected as a suitable model for the variety of languages covered in this work.

First, it seems strange that a model based on concurrency is used to model languages that are inherently sequential in nature. This in itself would be acceptable if the model could also be applied to concurrent languages, but its message-passing semantics, based on procedure call, preclude its application to the concurrent languages of [AdBKR86] and [Agh86], for example.

A more serious problem is that Hoare's CSP [Hoa85] does not admit the notion of dynamic creation of an indeterminate number of processes.[1] Consequently, Nguyen and Hailpern's semantics make no mention of the semantics of object creation. Object creation, and the associated issues of scope, lifetime and garbage collection are important features of languages like Smalltalk, and any model that does not describe these concepts has serious shortcomings.

Also, the distinction between direct and inherited methods means that modelling explicit access to inherited methods from within direct methods (such as is provided by Smalltalk's `super` mechanism) is difficult. However, Hailpern and Nguyen state that they intend to investigate the effect of making inherit from

[1] Although there are versions which do this.

a conventional statement, removing the distinction between direct and inherited methods; this would seem to rectify the problem.

Finally, the procedure call semantics associated with message-sending are not defined when the call graph is cyclic. For example, a form of indirect recursion occurs in Smalltalk when an object sends a message to a second object, which then sends a message to the first object. In the CSP semantics, this causes a deadlock. Nguyen and Hailpern avoid this in the simplest case where an object sends a message to itself by having local procedures; presumably sends to "self" should be mapped into calls of local procedures. However, this does not solve the problem of indirect sends to "self."

2.3 Message Sending—Concurrent or Serial?

Communication in object-oriented languages is one of the most contentious issues in the object-oriented programming community. This is reflected by the variety of binding and communication mechanisms in use. In most object-oriented languages, message-sending is serial: the sender is inactive while the receiver is processing the message, and every object is inactive until it is sent a message. This implies that a single thread of control is passed from one object to another when a message is sent. Some object-oriented languages introduce concurrency by adding conventional multi-processing facilities. In Smalltalk, for example, instances of a special class, Process, can be created and associated with an executable body of code. When a process object is sent the message resume, control returns to the sender immediately, but the code associated with the process can start executing in parallel. To achieve synchronisation between processes, conventional mechanisms (semaphores) are provided. Otherwise, there are no special concurrency features.

Multiple processes in a Smalltalk system can interfere when they simultaneously use the same object; they may even be executing the same method on that object at the same time. The instance variables of the object act as shared variables, and communication (or interference) may take place.

An alternative approach is that objects themselves should be the units of concurrency. In this view, potentially any object can be an independent, active process. To preserve concurrency, an object sending a message does not wait for the receiver of the message to complete processing the message before

continuing with its own task.

In the Parallel Object-Oriented Language (POOL) described by America [Ame86a], this model of concurrent, active objects is used. Message-sending is a form of rendezvous between objects; the sender must wait for the receiver to accept the message, some exchange of data may take place, and then both sender and receiver may continue execution.

The Actor model of computation as described by Agha [Agh86] also uses a concurrent approach, but message processing is asynchronous. An object can send a message to another object at any time, and the message will be placed on a *message queue* pending acceptance by the receiver. When the receiver has processed the message, it may send a reply at any time. No synchronisation need take place at all.

It should be clear from these brief descriptions that these three models of computation differ drastically in their notion of what a message is. Moreover, these differences are reflected in the formal semantics of each language: both the POOL and Actor semantics given in [AdBKR86] and [Agh86] use operational definitions based on state transition rules.

Whilst it is desirable to integrate all the different forms of inter-object communication smoothly into the same formal model, it is beyond the power of current formal description techniques to do this without obscuring much of the semantics. Certainly it would appear that the simple denotational semantics given in the rest of this thesis cannot be extended straightforwardly to embrace concurrent models.

The rest of this chapter is devoted to descriptions of the semantics given in [AdBKR86] and [Agh86]. These should serve as comparisons against the semantics in the rest of the thesis, illustrating the gulf between the two approaches.

2.4 The Semantics of POOL

The Parallel Object-Oriented Language developed as part of ESPRIT project 415 is described informally in [Ame86a]. A formal definition is given in [AdBKR86].

In POOL, every object is an independent process. It has a local, private state, and communicates with other objects by sending messages. So that an

object can function independently of other objects, it has a *body*, which is an executable statement.

Two basic mechanisms are used when a message is sent. The sending object executes a message send expression of the form $r!m(e_1, \ldots, e_n)$, which sends to r the message m with parameters $e_1, \ldots, e_n$. The receiving object executes an answer statement of the form $\mathsf{answer}(m_1, \ldots, m_k)$, where $m \in \{m_1, \ldots, m_k\}$, which indicates that the receiving object is ready to receive any message with one of the listed selectors.

When the sending object executes the message send, it is suspended until the receiving object executes a matching answer statement; similarly if the receiving object executes an answer statement, it is suspended until a matching send occurs. Thus a message creates a rendezvous between two objects; their processes are synchronised. After the message has been received and accepted, the sender is suspended while the receiver processes the message. Later, the receiver returns a result, and the sender proceeds. Meanwhile, the receiver can execute an optional post-processing section, and then resume what it was doing before it answered the message. (See Figure 2.3.)

The operational semantics of POOL are given as a transition system over a set of configurations, *Conf*. The relation describing valid transitions, $\rightarrow \subset \mathit{Conf} \times \mathit{Conf}$, is defined inductively by a set of axioms and rules.

A configuration is a quadruple:

$$\mathit{Conf} = (\mathit{LStat}\text{-}\mathsf{set}) \times \Sigma \times \mathit{Type} \times \mathit{Unit}$$

The first component is a set of labelled statements of the form

$$\{(\alpha_1, s_1), \ldots, (\alpha_n, s_n)\}$$

where each $\alpha_i \in \mathit{AObj}$ is the name of an object, and each s_i is the statement about to be executed by that object. All the α_i must be different. The elements of *AObj* are tagged natural numbers: $\mathit{AObj} :: \mathbb{N}$.

The second component of a configuration is a store:

$$\Sigma = (\mathit{AObj} \xrightarrow{m} \mathit{IVar} \xrightarrow{m} \mathit{AObj}) \times (\mathit{AObj} \xrightarrow{m} \mathit{LVar} \xrightarrow{m} \mathit{AObj}^*)$$

where *IVar* and *LVar* are the sets of instance variable names and local variable names. The first component of the store is the repository for all the instance variables: for every object in the configuration there is an element in

Figure 2.3: Synchronisation between objects in POOL

the map that has the values of the object's instance variables. The second component of the store contains activation stacks for the processes associated with every object.

The third component of a configuration records the class of every object,

$$Type = AObj \xrightarrow{m} Class\_name$$

while the fourth contains the program being executed. (A POOL program is a set of class definitions, each class definition defining the structure of all the instances of the class, and the methods for that class. The detailed definition of the abstract syntax of POOL is omitted here.)

When a program begins execution, a single instance of a distinguished class is created. This object may then create more objects, which may execute in parallel. Thus the initial configuration for a program U is:

$$\langle (\alpha_0, s), (\{\alpha_0 \mapsto \{v \mapsto \textsc{Nil} \mid v \in ivars(C)\}\}, \{\alpha_0 \mapsto [\,]\}), \{\alpha_0 \mapsto C\}, U \rangle,$$

where C is the distinguished class, s is the body associated with C, and the $ivars(C)$ are the instance variable names associated with instances of C.

An example of an axiom defining the transition relation is the axiom for the statement that creates a new object (the new statement):

$$\langle X \cup \{(\alpha, \mathsf{new}(C))\}, (\sigma, l), \tau, U \rangle \rightarrow \langle X \cup \{(\alpha, \beta), (\beta, s)\}, (\sigma', l), \tau', U \rangle$$

where $\beta \notin X$ is a new name, s is the body associated with the class named C, $\sigma' = \sigma \dagger \{\beta \mapsto \lambda x \cdot \mathsf{nil}\}$ is σ updated to include the new object β, and $\tau' = \tau \dagger \{\beta \mapsto C\}$ is τ updated with the class of β.

Having defined the above transition relation, [AdBKR86] then goes on to define a similar relation, $\overset{\|}{\rightarrow} \subset (GConf \times GConf)$, in which each transition represents the execution of many statements concurrently. The global configuration defined earlier is replaced by a set of local configurations, each recording the entire state of an object:

$$GConf = LConf\text{-}\mathsf{set}$$

$$LConf = AObj' \times Stat \times \Sigma' \times Class\_name \times Unit$$

where *Stat* is the set of (unlabelled) statements and Σ' is a local store. Most axioms then take the form

$$\{\langle \alpha, s, \sigma, \tau, U\rangle\} \xrightarrow{\|} \{\langle \alpha, s', \sigma', \tau', U\rangle\}$$

and the following rules for parallel composition are used to describe concurrent transitions:

$$\frac{X \xrightarrow{\|} X', Y \xrightarrow{\|} Y'}{X \cup Y \xrightarrow{\|} X' \cup Y'} \qquad \frac{X \xrightarrow{\|} X'}{X \cup Y \xrightarrow{\|} X' \cup Y}$$

In this latter definition, the set of active object identifiers, *AObj'*, cannot be the simple set used in the earlier definition, as concurrent new statements must create objects with differing names. Each *AObj* is a sequence of natural numbers; when an object α creates a new object for the *n*th time, the name of the new object is $\alpha \frown [n]$.

Points to note:

- The structure of objects, classes and methods is similar to that presented in Chapter 3, save for the extra state within each object for its activation stack.
- POOL does not have inheritance in any form, although it seems that it would be straightforward to add the simple form of inheritance described in Chapter 4.
- As in Nguyen and Hailpern's model, if an object sends a message indirectly to itself, deadlock occurs.

The deadlock problem, together with the lack of inheritance, does not make this model amenable to application to languages such as Smalltalk.

2.5 The Actor Model of Computation

In [Agh86], Agha describes the Actor model of computation developed by Hewitt [Hew79], and defines the semantics of a kernel actor language. The Actor model has evolved over the last decade in parallel with conventional object-oriented languages [Hew79], but each seems to have influenced the other only

slightly. Like the models of Hailpern & Nguyen and POOL, the actor model is based on concurrency. However, the semantics of message-passing, and the internal computation performed by an actor, are quite different in the Actor model.

In the Actor model, every object (known as an actor) is passive until it receives a message. On receipt of a message, an actor processes the message according to its *script*. Scripts are also known as *behaviours*, because a script defines the response of an actor to any message; it is analogous to a class definition in a conventional object-oriented language. In response to a message, an actor can:

1. Create new actors, by evaluating *new-expressions*. Each new-expression is given the initial behaviour of the new actor to be created.

2. Send messages to other actors (or itself), and

3. Specify its behaviour in response to the next message. If a new behaviour is not specified, then the same behaviour is used.

The acceptance of a message by an actor is termed an *event*.

When an actor sends a message, the message is placed in a queue at the destination actor; no synchronisation takes place. Although the Actor model has as one of its premises the guarantee of delivery of messages, no arrival ordering is guaranteed, even for successive messages from one actor to the same destination.

The script executed by an actor in response to a message cannot have any internal iteration or recursion, so that no form of deadlock is possible at the event level. All the statements within a script are executed concurrently. Once a new behaviour has been specified, an actor can start to process the next message in its queue, even though the processing of the current message may not have finished. This is possible because there is no local state that can be modified within an actor. The state of an actor is entirely described by its behaviour. Behaviours are parameterised by two lists: one is a list of *acquaintances*, the other a *communication list*. The acquaintance list is defined when a new behaviour is specified, and the communication list is defined when a message is selected to be processed.

Figure 2.4: A transition in the state of an actor

A behaviour that has been supplied with an acquaintance list is sometimes termed an *actor machine*. During the processing of a message, an actor machine specifies its replacement; once this has happened, the replacement can begin processing the next message. This is illustrated in Figure 2.4, where actor X has processed the nth message in its queue, and has specified a behaviour for the processing of message $n + 1$.

The principal differences between the Actor model and more conventional object-oriented models are:

- Message-sending is asynchronous.
- The behaviour of an object can change from event to event. There is no notion of actors belonging to classes, and having fixed behaviours.[2] Consequently, inheritance is not a primitive concept in actor systems.

[2]The semantics defined by Agha, however, bind behaviours to behaviour identifiers statically,

- The state of an actor is fixed whilst it is processing a message. A state change occurs when a new behaviour is defined for the actor. The new behaviour is only related to the old behaviour inasmuch as the acquaintance list of the new behaviour is specified by the old.

2.5.1 The Semantics of Actor Systems

The semantics of actor systems given by Agha in [Agh86] is in two parts: first, an initial configuration is defined for a particular actor program; second, a transition relation between configurations is defined.

Actor programs may be composed by identifying the *receptionists* of one program (i.e., the actors that may receive communications from external sources) with the *external* actors of another (i.e., actors that are not part of the program but may be sent messages). An important property of actor semantics is that the meaning of the composition of two actor systems is the same as the composition of the meanings of the parts.

The configuration of an actor system is a pair (l, T), where l defines the behaviour of every actor in the configuration, and T is the set of messages that have not been processed yet (known as a *tasks*). The first element of the pair is known as the *local states function*, and is a map from actor names (known as *mail addresses*) to behaviours: $l \in M \xrightarrow{m} B$. As in the second semantic definition of POOL in [AdBKR86] (see §2.4), a mail address is a sequence of natural numbers to allow concurrent allocation of new mail addresses.

A task is a triple of the form (t, m, k), where $t \in Tag$ is a unique tag that distinguishes all tasks, $m \in M$ is the destination mail address of the message, and $k \in M^*$ is the list of parameters in the message:

$$Task = Tag \times M \times M^*.$$

A behaviour is a function of the form

$$B = Task \rightarrow Task\text{-set} \times Actor\text{-set} \times B.$$

When an actor with mail address m and behaviour β processes a task (t, m, k), then $\beta(t, m, k) = (\tau, A, \beta')$ is a triple of new tasks τ, new actors A, and a new

---

so that the complete behaviour of an actor can be said to be completely determined when it is created.

behaviour β', respectively.

A transition relation between configurations c_1 and c_2 can then be defined as follows: if $\tau \in \mathit{tasks}(c_1)$, $\tau = (t, m, k)$, and $\mathit{states}(c_1)(m) = \beta$ where $\beta(t, m, k) = (T, A, \beta')$, then

$$\mathit{tasks}(c_2) = \mathit{tasks}(c_1) - \{\tau\} \cup T, \text{ and}$$
$$\mathit{states}(c_2) = \mathit{states}(c_1) \cup A \dagger \{m \mapsto \beta'\}.$$

Whilst the Actor model is an admittedly elegant model of concurrent object-oriented computation, it is not suited for our purposes. The aim of this thesis is to provide a model for existing features of object-oriented languages, and there is a mismatch between the features of a language like Smalltalk and the Actor model. More specifically, the differences in message-sending semantics and the lack of mutable local state within an object whilst processing a message make the Actor model unsuitable as a general model of object-oriented languages.

In the next chapter a new model of object-oriented computation will be presented that is more suited to the description of the desired language features.

Chapter 3

A Model of Object-Oriented Systems

> The creative mind plays with the objects it loves.
>
> *Carl Gustav Jung*

This chapter describes a simple model of object-oriented languages. As mentioned in the introduction, this model is based on a sequential object-oriented language, and the semantics of the language are defined denotationally.

The main concepts that are introduced formally are:

- objects, and object identity,
- internal state of objects (instance variables), and primitive objects (namely, those without mutable internal state),
- classes,

- messages and methods, with dynamic binding of messages to methods, and
- prototypes and delegation.

The chapter opens by investigating the semantic domains underlying object-oriented languages; a comparison of the domain of object stores and conventional stores illustrates the essential difference between object-oriented and conventional languages. After that the abstract syntax of the minimal object-oriented language is given, and its semantics defined. Then follows a discussion of primitive objects and classes, and the chapter closes by modifying the language and its semantics to be prototype-based, rather than class-based.

An index to the functions and types defined in this and succeeding chapters can be found in Appendix D.

3.1 Objects

Central to the semantics of object-oriented languages is the question, "What is an object?" Clearly, objects are used as the values of expressions in object-oriented languages, but what distinguishes an object from a value in a conventional language?

A fundamental concept in object-oriented programming is the idea of *object identity* [KC86]. Simply put, an object's identity is the property that distinguishes it from all other objects. Many objects with identical internal states may have different identities; however two objects with the same identity are *the same object*.

In general, the existence of an object is independent of all other objects. An object may interact with others, and its creation is usually brought about by another object, but it is a first-class citizen and its existence cannot be terminated by another object.

These two ideas, of identity and first-class citizenship, are reflected in the principal semantic domain in use in this thesis, namely *Object_memory*. An *Object_memory* is a map from object identifiers to objects:

$$Object\_memory = Oop \xrightarrow{m} Object$$

Object identifiers are named *Oops* for historical reasons.[1]

The domain *Oop* can be any set of distinct values; no assumption about ordering within the set is made. The only operator that need be defined on *Oops* is equality. This is in contrast with the object identifiers used in the POOL and actor semantics, which are sequences of natural numbers. In those models, the concurrent creation of objects requires some scheme for allocating unique identifiers. To this end, each actor or POOL object maintains an internal count of how many objects it has created. When an object with identifier α creates its nth object, the new object is assigned the identifier $\alpha \frown [n]$. The POOL and actor models assume that an object cannot change its identity after it has been created.

3.2 Internal States

Oops identify objects; additionally, objects may have internal states. In the model presented here, every object is an instance of a class (see §3.12 for an alternative view):

$$\begin{aligned} \mathit{Object} \;::\; & \mathit{Class} \;:\; \mathit{Class\_name} \\ & \mathit{Body} \;:\; \mathit{Object\_body} \end{aligned}$$

Objects can either be instances of user-defined classes, or primitive objects.

$$\mathit{Object\_body} = \mathit{Plain\_object} \cup \mathit{Primitive\_object}$$

A *Plain_object* contains all the mutable state of an object, stored as values of instance variables. Every instance variable can potentially refer to any object in the *Object_memory*:

$$\mathit{Plain\_object} = \mathit{Id} \xrightarrow{m} \mathit{Oop}$$

Expanding the various definitions, an *Object_memory* looks something like this:

---

[1] OOP is the acronym used in Smalltalk circles for 'object pointer'. More accurately, it stands for Ordinary Object Pointer, namely pointers to objects that have storage allocated to them. This means that OOPs cannot refer to integers, for example, but in this thesis the domain of *Oops* refers to all objects. See §3.10 for more on this subject.

$$Object\_memory = Oop \xrightarrow{m} (\ldots \times Id \xrightarrow{m} Oop)$$

This is the central structure found in almost all object-oriented systems: a flat space of objects, each with their own unique identity and internal, private address space. The identifiers that comprise that address space, i.e., the instance variable names, are only in scope within methods associated with the object (usually by being defined within the object's class).

Contrast this with the semantic domains associated with a conventional programming language: the environment is a mapping from identifiers to denotable values, and the domain of denotable values usually includes *locations*; the store is a mapping from locations to values. Obviously, these two domains differ in significant ways. Explaining the structure of these domains can highlight the key difference between conventional programming languages and object-oriented languages.

3.3 The Nature of Object-Oriented Languages

Let us first consider the domain underlying the simplest kind of imperative language, with only global identifiers, and no procedure or function facilities.[2] The meaning of a statement in such a language is a function from stores to stores, where a store is a mapping from identifiers to values:

$$MStat = Stat \rightarrow Store \rightarrow Store$$

$$Store = Id \xrightarrow{m} Val$$

No environment is required. Clearly, a language based on the *Store* domain is inadequate for the construction of large software systems: it has no *abstraction mechanism*. The abstraction mechanism should package up entities, encapsulating them so that the internal details are hidden. Ideally, the same mechanism should support both data abstraction (i.e., support for abstract data types) and control abstraction.

ALGOL-like languages approach this problem in three steps: First, separate stores are encapsulated within blocks. Identifiers declared within a block

[2] The early versions of BASIC are examples of such a language.

are invisible outside the block. The only way to create multiple instances of a block is to name it, i.e., create a procedure, and use recursion. However, because block activations are in stack-order, and variables created inside a block disappear when the block exits, blocks cannot provide a data abstraction facility. The model of the store is the same as before, except that a "new" store is created when a block is entered, and discarded when it exits. Alternatively, if recursion is not provided by the language (e.g., FORTRAN), then a global store (with identifiers renamed to avoid name clashes between blocks) is an adequate model.

The second step taken by the ALGOL-like languages is to provide a communication mechanism between blocks (in addition to parameter passing), namely shared variables. Any block can access the variables of its enclosing blocks. A more restricted form of this sharing is parameter passing "by reference." Locations are introduced into the domain of stores, and environments facilitate sharing. At any point in the text of a program, the environment binds identifiers to locations, and the store maps locations to values:

$$Env = Id \xrightarrow{m} Loc$$

$$Store = Loc \xrightarrow{m} Val$$

$$MStat = Stat \to Env \to Store \to Store$$

Finally, as the use of global variables can lead to bad programming habits and obscure bugs, and so that long-term storage can be allocated at run-time, the idea of a *heap* is introduced. This enables a block to create a new location in the store, which outlives the block. Access to the storage is enabled by making locations into values called *pointers*.

$$Val = Loc \cup \ldots$$

Most conventional programming languages include one or more of these features. As has been noted by programmers in recent years, the use of shared variables, call-by-reference, and pointers can lead to programs that are not modular [Don77]. Several features have been proposed to ameliorate this problem, such as modules and packages [Wir83, Uni80]. These provide modularity, but not full support for abstract data types, as multiple instances of a module cannot be created at run-time.

The design of object-oriented languages deviates from that approach at the outset. The use of shared variables and aliasing is outlawed. Rather than insist that allocation of name-spaces takes place in stack-order, each name-space, termed an object, has an independent existence. Structurally, objects are identical to the original store:

$$Object = Id \xrightarrow{m} Val$$

New objects, which may be instances of abstract data types, can be created at any time, and exist indefinitely. Object identifiers, *Oops*, are used as internal names to refer to objects.

$$Object\_memory = Oop \xrightarrow{m} Object$$

Because objects must communicate with each other, and therefore must reference each other, *Oops* can also be values:

$$Val = Oop \cup \ldots$$

If primitive values (such as integers) are also modelled by objects, then the only kind of value required is an *Oop*, but then objects must be of two kinds, primitive or non-primitive:

$$Val = Oop$$

$$Object = Primitive\_object \cup Id \xrightarrow{m} Val$$

As with ALGOL-like languages without sharing, the code associated with a name-space has access to the identifiers of that name-space, and no others. In the ALGOL model, the variables of a block only exist when a block is active. In an object-oriented language, the variables exist indefinitely, so different operations can be invoked on them. Each of these operations is termed a *method*.

The object-based store just defined strongly resembles the *Object_memory* defined earlier (§3.1). This structure can also be found in the object store of the POOL definition ($\Sigma = (AObj \xrightarrow{m} IVar \xrightarrow{m} AObj) \times \ldots$, [AdBKR86], p. 199), and is present in a slightly different form in the actor semantics. Actors are not instances of classes, so their "instance variables" (the acquaintance list) can change in structure from event to event, but not during the processing of a single message. Accordingly, actors systems are mappings from mail addresses

(the object identifiers of an actor system) to behaviours, where a behaviour is a function parameterised by an acquaintance list (see §2.5.1).

So important is this structure in an object-oriented system, that one is tempted to say that its presence is a necessary and sufficient condition for a language to be called "object-oriented." Moreover, because this structure can be simulated using conventional data structures, object-oriented programming can be "simulated" in any conventional language by suitable self-discipline by the programmer [Cox86].

3.4 Object-Oriented Principles

The previous section provided a rationale for the development of object-oriented languages.[3] One thing is clear: that the principal aim of the object-oriented style is to provide a single mechanism that facilitates abstraction and encapsulation of both data and control. To reinforce this idea, this section introduces two principles that all languages that seek to call themselves object-oriented should uphold. These principles are modelled after Tennent's Principles of Abstraction, Correspondence and Qualification [Ten77, Ten81].

The concepts of identity and private internal states are enshrined in the Principle of Object Identity and the Principle of Object Encapsulation:

The Principle of Object Identity Every object has a unique identity, which cannot change without the object's cooperation.

The Principle of Object Encapsulation The internal state of an object can only be accessed or modified by the execution of a method associated with the object, in response to a message sent to that object.

Note that the structure of *Object_memory* reflects these principles by using *Oops* as indicators of identity, and independent name-spaces within objects. Object-oriented languages can support these principles by ensuring that the internal identifiers of only one object are in scope at any place in a program.

[3] It is, however, doubtful that all the various aspects were considered together *before* their development. As usual, the evolution of the style developed in part by design, and in part by chance.

3.5 Abstract Syntax

This section defines the abstract syntax of a language invented to explain the essentials of object-oriented programming. Later sections define its semantics, and following chapters extend the language to incorporate inheritance and control features.

The language presented in this section is a simplification of that described in [Wol87]. As an initial draft of this material was being prepared, the definition of the SPOOL language (sequential version of POOL [Ame86b]) was encountered by the author. The SPOOL language and semantics are remarkably similar to the language presented in this chapter, the principal differences being:

- SPOOL is statically typed (where types are classes), and
- sends to *self* are disallowed (as in POOL)

The reader unfamiliar with the notation used here may find the SPOOL definition useful.

3.5.1 Classes

Every object is an instance of a class. Classes organise objects with common behaviour into groups: the class defines that which is common to all objects in the class; the private states of objects reflect the differences.

To define behaviour, a class defines the messages that its instances will respond to, and how they will respond. It does this by associating a *method* with every *message* that its instances respond to. Messages have names (termed *selectors*), and possibly some parameters. Additionally, a class defines the structure of its instances by enumerating the names of their instance variables. Thus one component of a class is a *Id*-set. An invariant of the *Object_memory* is that the structure of every object is consistent with its class. The abstract syntax of a class is:

$$\begin{aligned} \mathit{Class\_body} :: \ & \mathit{Instvars} : \mathit{Id}\text{-set} \\ & \mathit{Methods} : \mathit{Selector} \xrightarrow{m} \\ & \qquad \mathit{Method\_body} \cup \mathit{Primitive\_method} \end{aligned}$$

Note that associated with every selector is either a user-defined method, or a primitive method that cannot be expressed within the language. Primitive methods are used to provide facilities such as basic arithmetic, and will be described more fully in §3.9.

Every class is named, and a set of named classes comprises a program:

$$Program = Class\_map$$

$$Class\_map = Class\_name \xrightarrow{m} Class\_body$$

Note that one of these "programs" does not, by itself, compute anything. It merely provides a context for the evaluation of an expression. Given a program p, and an expression e (with no free variables) to be evaluated in the context of p, the meaning of e is given by the meaning function *MExpression* (defined later), with the meaning of p (as defined by *MProgram*) as part of its "environment." The resulting meaning is a function that transforms an *Object_memory* to a new *Object_memory*.

3.5.2 Methods, Messages and Expressions

The syntax of methods given here is loosely based on that of Smalltalk. Each method can take some arguments; declare variables that are local to the method, and whose lifetime is that of a method invocation (these are known as *temporary variables*, or sometimes just *temporaries*); execute an expression.[4] (Parameters and temporaries are known collectively as *local identifiers*, or just *locals*.) The value returned by the method is the value of the expression.

$$\begin{aligned} Method\_body :: Params &: Ulist(Id) \\ Temps &: Id\text{-set} \\ Expr &: Expression \end{aligned}$$

The constructor $Ulist(X)$ is used to model a sequence of values from the domain X, that has no duplicates. It is defined thus:

$$Ulist(X) = X^*$$

[4] This is not quite like Smalltalk in that the body of a method is an expression, and an expression can be the sequential composition of a list of expressions. In Smalltalk, sequential composition is only allowed at the outermost level.

where

$$\mathit{inv\text{-}Ulist}(X)(l) \triangleq \mathsf{card}\,\mathsf{rng}\,l = \mathsf{card}\,\mathsf{dom}\,l$$

The invariant asserts that all elements of *Ulist* must have the same number of different elements in their range as in their domain, i.e., that the number of different elements is the same as the length of the sequence.

Expressions can be composed sequentially into *Expression_lists*, and come in five other basic flavours:

$$\mathit{Expression\_list} :: \mathit{Expression}^*$$

$$\mathit{Expression} = \mathit{Expression\_list} \cup \mathit{Assignment} \cup \mathit{Object\_name} \cup \mathit{Message} \cup \mathit{New\_object} \cup \mathit{Literal\_object}$$

In an *Assignment*, the value of an expression can be assigned to a temporary or an instance variable of the object that has received the message being processed by the current method[5] (*AVar_id* is the domain of assignable identifiers).

$$\mathit{Assignment} :: \mathit{LHS} : \mathit{AVar\_id}\ \ \mathit{RHS} : \mathit{Expression}$$

$$\mathit{AVar\_id} = \mathit{Temp\_id} \cup \mathit{Inst\_var\_id}$$

$$\mathit{Arg\_id} :: \mathit{Id}$$

$$\mathit{Temp\_id} :: \mathit{Id}$$

$$\mathit{Inst\_var\_id} :: \mathit{Id}$$

(In a concrete syntax, identifiers of instance variables, temporaries and arguments are likely to come from the same domain. However, it is assumed that any applied occurrence of an identifier can always be resolved statically to determine which kind of variable is being used.)

An *Object_name* is either a local identifier, an instance variable, or a "pseudo-variable" standing for the *Oop* of the receiver, usually known as *self* (also

[5] This object is usually known as the "receiver."

known as *this* in Simula and C++, and *current* in Eiffel [BDMN73, Str86, Mey87]).

$$Object\_name = Var\_id \cup \{\text{SELF}\}$$

$$Var\_id = Arg\_id \cup Temp\_id \cup Inst\_var\_id$$

A *Message* contains an expression that evaluates to the intended receiver of the message, the selector of the message to be sent, and a list of expressions (possibly empty) that are the arguments of the message.

$$\begin{aligned} Message :: \; & Rcvr : Expression \\ & Sel : Selector \\ & Args : Expression^* \end{aligned}$$

New objects are created by evaluating a *New_object* expression; it is parameterised by the name of the class of the new object.

$$New\_object :: Class : Class\_name$$

(Smalltalk *aficionados* will know that there is no special syntactic form in Smalltalk for new-expressions; this is elaborated upon in §3.11.)

For the moment, the only kind of literal objects are integers. Other possible choices are discussed in §3.10 and Chapter 5.

$$Literal\_object = Int\_literal$$

$$Int\_literal :: \mathbb{Z}$$

The abstract syntax of the simple language is complete. Now the associated semantic domains and meaning functions will be defined. Context conditions are defined in Chapter 4.

3.6 Semantic Domains

The principal semantic domain, *Object_memory*, has already been described. To complete the definition of *Object_memory*, all that is required is to choose the domain of primitive objects. As integers are the only type of literal objects in the language, they are also the only type of primitive object:

$$Primitive\_object = \mathbb{Z}$$

Several auxiliary functions will be found useful:

$$class : Oop \times Object\_memory \to Class\_name$$
$$class(oop, \sigma) \triangleq Class(\sigma(oop))$$

pre $oop \in \mathsf{dom}\, \sigma$

$$body : Oop \times Object\_memory \to Object\_body$$
$$body(oop, \sigma) \triangleq Body(\sigma(oop))$$

pre $oop \in \mathsf{dom}\, \sigma$

$$inst\_var : Id \times Oop \times Object\_memory \to Oop$$
$$inst\_var(iv, oop, \sigma) \triangleq body(oop, \sigma)(iv)$$

pre $pre\text{-}body(oop, \sigma)$
$\quad \wedge\ body(oop, \sigma) \in Plain\_object$
$\quad \wedge\ iv \in \mathsf{dom}\, body(oop, \sigma)$

$$update\_inst\_var : Id \times Oop \times Oop \times Object\_memory \to Object\_memory$$
$$update\_inst\_var(inst\_var, oop, value, \sigma) \triangleq \sigma \dagger \{oop \mapsto \mu(\sigma(oop), Body \mapsto body(oop, \sigma) \dagger \{inst\_var \mapsto value\})\}$$

pre $pre\text{-}body(oop, \sigma) \wedge body(oop, \sigma) \in Plain\_object$

The denotation of a class is a function, which, given an instance of that class, a message, and an object memory, executes the method associated with the message, returning a (possibly updated) object memory, and a result *Oop*.

$$Class\_den = Selector \xrightarrow{m} Method\_den$$

$$Method\_den = Oop \times (Oop^*) \times Object\_memory \to Oop \times Object\_memory$$

The first argument of a *Method_den* is the *Oop* of the receiver; the second is a list of the *Oops* of the arguments of the method.

The denotation of a *Program* is composed of the denotations of its constituent classes.

$$Program\_den = Class\_name \xrightarrow{m} Class\_den$$

$$MProgram : Program \to Program\_den$$
$$MProgram[\![p]\!] \triangleq$$
$$\quad \mathsf{let}\ pd =$$
$$\quad\quad \{c \mapsto MClass\_body[\![p(c)]\!]\, mk\text{-}SEnv(all\_instvars(p), pd) \mid c \in \mathsf{dom}\, p\}\ \mathsf{in}$$
$$\quad pd$$

$$all\_instvars : Program \to Class\_name \xrightarrow{m} Id\text{-}\mathsf{set}$$
$$all\_instvars(p) \triangleq \{c \mapsto Instvars(p(c)) \mid c \in \mathsf{dom}\, p\}$$

To define the meaning of methods, a fixed point definition is required. This could also be written as

$$fix\big(\lambda pd \cdot \{c \mapsto MClass\_body[\![p(c)]\!]\, mk\text{-}SEnv(all\_instvars(p), pd) \mid c \in \mathsf{dom}\, p\}\big).$$

For such a fixed point to be defined, the functional that is its argument must be continuous, and its domain and co-domain the same pointed complete partial ordering (cpo) [Sto77]. That the domain *Program_den* is a pointed cpo follows from its construction, using primitive domains that are pointed cpos and constructors that build pointed cpos given arguments that are pointed cpos. Continuity follows by examining the definition of *MClass_body*, and its auxiliary functions (see rest of this chapter). The fixed-point construction proceeds by taking an approximation to the meaning of the program, *pd*, and constructing a better approximation by evaluating the meaning function for every class in the program.

The *MProgram* function passes a *static environment* to *MClass_body*. This records the names of the instance variables associated with each class, and the denotation of the rest of the program.

$$SEnv :: Instvars : Class\_name \xrightarrow{m} Id\text{-}\mathsf{set}$$
$$\qquad\quad PD : Program\_den$$

The denotation of a class is composed of the denotations of its constituent methods:[6]

$$MClass\_body : Class\_body \to SEnv \to Class\_den$$
$$MClass\_body\,[\![mk\text{-}Class\_body(\_, meths)]\!]\,\rho \triangleq \{sel \mapsto MMethod\,[\![meths(sel)]\!]\,\rho \mid sel \in \mathsf{dom}\, meths\}$$

$$MMethod : (Method\_body \cup Primitive\_method) \to SEnv \to Method\_den$$
$$MMethod\,[\![m]\!]\,\rho \triangleq \mathsf{if}\ m \in Primitive\_method\ \mathsf{then}\ m\ \mathsf{else}\ MMethod\_body\,[\![m]\!]\,\rho$$

As is apparent from the definition of *MMethod*, a *Primitive_method* is simply a method denotation:

$$Primitive\_method = Method\_den$$

Some primitive methods are defined in §3.9.

During the execution of a method, a *dynamic environment*[7] records the bindings of parameters to arguments and the values of temporaries, and the current receiver. The auxiliary function *update_temp* will prove useful later:

$$\begin{aligned} DEnv :: \quad Rcvr &: Oop \\ Params &: Id \xrightarrow{m} Oop \\ Temps &: Id \xrightarrow{m} Oop \end{aligned}$$

[6]The use of an underscore in the *Class_body* constructor indicates that the value of that particular field is not used in this function. Similarly, arguments to functions which are not used are indicated by an underscore.

[7]The use of the terms "static environment" and "dynamic environment" differs from that in, say, [BJ82], where a static environment is a structure used during the evaluation of context conditions, and a dynamic environment records, e.g., the binding of identifiers to locations. The dynamic environment used here could be part of the store, but the separation of the state local to a method from the global state is a more natural model of object-oriented computation.

$$update\_temp : Id \times Oop \times DEnv \to DEnv$$
$$update\_temp(id, value, \delta) \triangleq \mu(\delta, Temps \mapsto Temps(\delta) \dagger \{id \mapsto value\})$$

The execution of a method consists of creating a dynamic environment, binding the formal parameters of the method to the actual parameters, initialising temporaries (Smalltalk always initialises temporaries to refer to the special object, nil, with *Oop* NILOOP), and executing the body of the method in the new environment.[8]

$$MMethod\_body : Method\_body \to SEnv \to Method\_den$$
$$MMethod\_body[\![mk\text{-}Method\_body(params, temps, expr)]\!]\,\rho \triangleq$$
$$\lambda rcvr, args, \sigma \cdot$$
$$\quad \textsf{let } bindings = bind\_args(params, args) \textsf{ in}$$
$$\quad \textsf{let } \delta = mk\text{-}DEnv(rcvr, bindings, initialise(temps)) \textsf{ in}$$
$$\quad \textsf{let } (result, \delta', \sigma') = MExpression[\![expr]\!]\,\rho\delta\sigma \textsf{ in}$$
$$\quad (result, \sigma')$$

$$bind\_args : Ulist(Id) \times Oop^* \to Id \xrightarrow{m} Oop$$
$$bind\_args(formals, actuals) \triangleq \{formals(i) \mapsto actuals(i) \mid i \in \textsf{dom}\, formals\}$$
$$\textsf{pre len}\, formals = \textsf{len}\, actuals$$

$$initialise : Id\text{-set} \to Id \xrightarrow{m} Oop$$
$$initialise(vars) \triangleq \{id \mapsto \text{NILOOP} \mid id \in vars\}$$

[8] The preconditions given for some of the functions are only satisfied when the meaning functions are applied to programs that satisfy certain well-formedness conditions; these are defined in Chapter 4.

3.7 The Meaning Function for Expressions

Within the execution of a method, assignments to temporary variables can take place, so the dynamic environment is also a result of an expression evaluation:

$$\begin{aligned}
&MExpression : Expression \to SEnv \to \\
&\qquad DEnv \to Object_memory \to \\
&\qquad\qquad Oop \times DEnv \times Object_memory \\
&MExpression[\![mk\text{-}Expression_list(exprs)]\!]\rho\delta\sigma \triangleq \\
&\quad \textsf{let } (oop, \delta', \sigma') = MExpression[\![\textsf{hd } exprs]\!]\rho\delta\sigma \textsf{ in} \\
&\quad \textsf{if len } exprs = 1 \textsf{ then } (oop, \delta', \sigma') \\
&\quad \textsf{else } MExpression[\![mk\text{-}Expression_list(\textsf{tl } exprs)]\!]\rho\delta'\sigma'
\end{aligned}$$

The remaining definitions show how the other various kinds of expressions are evaluated.

The semantics of assignment are straightforward: either a temporary or an instance variable of the current receiver is updated.

$$\begin{aligned}
&MExpression[\![mk\text{-}Assignment(id, rhs)]\!]\rho\delta\sigma \triangleq \\
&\quad \textsf{let } (result, \delta', \sigma') = MExpression[\![rhs]\!]\rho\delta\sigma \textsf{ in} \\
&\quad \textsf{cases } id \textsf{ of} \\
&\quad mk\text{-}Temp_id(t) \to (result, update_temp(t, result, \delta'), \sigma') \\
&\quad mk\text{-}Inst_var_id(iv) \to \big(result, \delta', \\
&\qquad\qquad update_inst_var(iv, Rcvr(\delta), result, \sigma')\big) \\
&\quad \textsf{end}
\end{aligned}$$

References to objects are equally straightforward:

$$MExpression[\![mk\text{-}Arg\_id(id)]\!]\rho\delta\sigma \triangleq (Params(\delta)(id), \delta, \sigma)$$

$$MExpression[\![mk\text{-}Temp\_id(id)]\!]\rho\delta\sigma \triangleq (Temps(\delta)(id), \delta, \sigma)$$

$$\begin{aligned}
&MExpression[\![mk\text{-}Inst_var_id(id)]\!]\rho\delta\sigma \triangleq \\
&\quad (inst_var(id, Rcvr(\delta), \sigma), \delta, \sigma)
\end{aligned}$$

$$MExpression[\![\text{SELF}]\!]\rho\delta\sigma \triangleq (Rcvr(\delta), \delta, \sigma)$$

Before sending a message, the receiver and argument expressions of the message must be evaluated, to determine the receiver and argument objects. In this simple language, as in Smalltalk, arguments are evaluated left-to-right.

$$\begin{aligned}
&MExpression[\![mk\text{-}Message(rcvr, sel, arglist)]\!]\rho\delta\sigma \triangleq \\
&\quad \text{let } (rcvr_oop, \delta', \sigma') = MExpression[\![rcvr]\!]\rho\delta\sigma \text{ in} \\
&\quad \text{let } (actuals, \delta'', \sigma'') = MExpression_list[\![arglist]\!]\rho\delta'\sigma' \text{ in} \\
&\quad \text{let } (result, \sigma''') = perform(sel, PD(\rho), rcvr_oop, actuals, \sigma'') \text{ in} \\
&\quad (result, \delta'', \sigma''')
\end{aligned}$$

The following function evaluates a list of expressions, accumulating the results of each expression into a list of *Oops*.

$$\begin{aligned}
&MExpression_list : Expression^* \to SEnv \to \\
&\qquad DEnv \to Object_memory \to \\
&\qquad\quad Oop^* \times DEnv \times Object_memory \\
&MExpression_list[\![el]\!]\rho\delta\sigma \triangleq \\
&\quad \text{if } el = [\,] \\
&\quad \text{then } ([\,], \delta, \sigma) \\
&\quad \text{else let } (val, \delta', \sigma') = MExpression[\![\text{hd } el]\!]\rho\delta\sigma \text{ in} \\
&\qquad \text{let } (val_list, \delta'', \sigma'') = MExpression_list[\![\text{tl } el]\!]\rho\delta'\sigma' \text{ in} \\
&\qquad ([val]^\frown val_list, \delta'', \sigma'')
\end{aligned}$$

The *perform* function lies at the heart of the message-passing semantics. Note that the method chosen in response to a message is determined by the class of the receiver, which in general cannot be determined statically.

$$perform : Selector \times Program\_den \times Oop \times Oop^* \times Object\_memory \rightarrow Oop \times Object\_memory$$

$$perform(sel, pd, rcvr, args, \sigma) \triangleq$$
$$\quad \text{let } class = class(rcvr, \sigma) \text{ in}$$
$$\quad \text{if } sel \in selectors(class, pd)$$
$$\quad \text{then } method(sel, class, pd)(rcvr, args, \sigma)$$
$$\quad \text{else } message\_not\_understood(sel, pd, class, rcvr, args, \sigma)$$

$$selectors : Class\_name \times Program\_den \rightarrow Selector\text{-set}$$
$$selectors(class, pd) \triangleq \text{dom } pd(class)$$

$$method : Selector \times Class\_name \times Program\_den \rightarrow Method\_den$$
$$method(sel, class, pd) \triangleq pd(class)(sel)$$

When an object does not understand a message (i.e., when its class does not define a response), an error must be signalled, perhaps by stopping the program. This behaviour is encapsulated within the *message_not_understood* function. In Smalltalk, for example, messages that are not understood are not classed as "errors" in the normal sense, but cause a special message, doesNotUnderstand:, to be sent to the receiver. If this message is in turn not understood, the outcome is undefined. Formalising this behaviour requires a definition of the representation of messages as objects, and a capability within the semantics for creating such objects. Although straightforward, this is not of concern here.

3.8 Creating Objects

Evaluating a new-expression creates a new object, which, by definition, is given a new identity. Because only one restriction has been placed on object identities, namely that equality be defined, any identity not in use can be

assigned to the new object. This is manifest in the definition of *create*: a specification for a function is given, rather than an explicit function definition. [9]

$$
\begin{array}{l}
MExpression[\![mk\text{-}New\_object(class)]\!]\rho\delta\sigma \quad \triangleq \\
\quad \textsf{let } new\_obj = mk\text{-}Object(class, \\
\qquad\qquad\qquad\qquad initialise(Instvars(\rho)(class))) \textsf{ in} \\
\quad \textsf{let } (new\_oop, \sigma') = create(new\_obj, \sigma) \textsf{ in} \\
\quad (new\_oop, \delta, \sigma')
\end{array}
$$

$$
\begin{array}{l}
create\ (obj{:}\ Object)\ new\_oop{:}\ Oop \\
\textsf{ext wr } \sigma\ :\ Object\_memory \\
\textsf{post } new\_oop \notin \textsf{dom } \overleftarrow{\sigma} \wedge \sigma = \overleftarrow{\sigma} \cup \{new\_oop \mapsto obj\}
\end{array}
$$

The signature of the *create* function is $Object \times Object\_memory \to Oop \times Object\_memory$.

A literal value denotes an immutable object. Since an equivalent object may not exist in the memory, one may have to be created. However, if one does exist, there is no sense in insisting that it be found, as two equal immutable objects are indistinguishable. Thus the meaning function for integer literals can return an existing equivalent object if there is one present in the store, or make a new one.

$$
\begin{array}{l}
MExpression[\![mk\text{-}Int\_literal(int)]\!]\rho\delta\sigma \quad \triangleq \\
\quad \textsf{let } (oop, \sigma') = find\_or\_make\_immutable(int, \textsf{Integer}, \sigma) \textsf{ in} \\
\quad (oop, \delta, \sigma')
\end{array}
$$

$$
\begin{array}{l}
find\_or\_make\_immutable: \\
\qquad Primitive\_object \times Class\_name \times Object\_memory \to \\
\qquad\qquad\qquad\qquad\qquad\qquad Oop \times Object\_memory
\end{array}
$$

[9] Strictly, this non-determinacy poses some problems in the claim that denotations are functions; see [BJ82]. The proof that these problems have been resolved has been omitted.

$find\_or\_make\_immutable$ (*value*: *Primitive_object*,
class: *Class_name*) *obj*: *Oop*

ext wr σ : *Object_memory*

post $\sigma(obj) = mk\text{-}Object(class, value) \wedge (\sigma = \overleftarrow{\sigma} \vee \{obj\} \lhd \sigma = \overleftarrow{\sigma})$

Most Smalltalk implementations take advantage of this fact by encoding the values of a small range of integers (usually related to the machine word size) into the range of object pointers, by using tags.

The ability of an object-oriented language to create objects at run-time has important consequences for the semantics and implementation of the language. Consider, for example, the effect of a new-expression that does not assign its result to a variable, nor return it as the result of a method. A new object will have been created in the object memory, and assigned an *Oop*, but this *Oop* will not appear anywhere else in the memory, and so be inaccessible. From an operational point of view, such an expression has no effect on the meaning of a program: the result is the same as if the expression had not been evaluated, and the store is not *detectably* different. However, the meanings of two programs as defined by the semantics, one with and one without such an expression, are different. This is because the resulting object memories are different: one contains an object that is not present in the other. From the implementation viewpoint, the redundant object is *garbage*, and the resources that are allocated to it (an identity, and possibly some storage) must be reclaimed by *garbage collection*.

The semantics given in this section do not eliminate garbage from the state. Although this could be done, it would complicate the definition unnecessarily. A better approach is to define a retrieve function that filters out garbage from the state [Wol86], or define an isomorphic equality operator between object memories that ignores garbage [Mas86].

3.9 Primitive Methods

Primitive methods provide functionality that cannot be expressed within the language. For example, primitives are required to perform arithmetic on integers.

The conventional approach to including primitives is to provide special syntactic forms for the primitives (e.g., the arithmetic operators in Pascal), and meaning functions for the special forms. The approach taken in this semantics, as in Smalltalk, is different: primitive classes (such as the class of integers) are defined, and primitive methods associated with messages to instances of those classes. For example, the + message is associated with an addition primitive in the class of integers; it could also be associated with a user-defined method for concatenation in the class of strings. Defining primitives this way has the advantage that a new capability can be added to the language without changing any existing syntactic or semantic definitions. In this section, several primitives are defined by way of example. More on this subject occurs in the section on primitive classes, §3.10, §3.11 and Chapter 5.

The definition of arithmetic and relational primitives on integers is straightforward. The only noteworthy point is what happens when the argument passed to the addition primitive is not an integer (we can be sure that the receiver is an integer by insisting that the primitive is only associated with a selector in the class of integers—this useful property of receivers is used in other primitives, to follow). Plainly, one option is to define this to be an error, and halt the program's execution. However, as was mentioned in the introduction, this is not a satisfactory solution in a persistent system. The Smalltalk approach is to associate a method, written in Smalltalk, with every primitive method. If the pre-conditions of the primitive are not met, the primitive is deemed to have *failed*, and the non-primitive method is executed. A failing primitive must not cause any change to the object memory or environment. The non-primitive method will typically display an error on the screen, but may instead perform some recovery action. For example, the integer primitives are only defined on a subrange of the integers compatible with the machine's word size; the task of the non-primitive method is to deal with overflow.

Rather than complicate the model by describing a primitive failure, it will be assumed that some suitable action is taken; other suitable static exception handling techniques could be used [Bla83, Cot84, Cri84]. In the integer addition primitive, this is the task of the *plus_error* function.[10]

[10]The use of a sequence constructor in the argument list indicates that there must be exactly one argument to the method—this can be guaranteed by the context conditions, described in Chapter 4.

$$
\begin{array}{l}
plus\_primitive : Primitive\_method \\
plus\_primitive \quad \triangleq \\
\quad \lambda rcvr\_oop, [arg\_oop], \sigma \cdot \\
\quad\quad \textsf{let } addend = body(rcvr\_oop, \sigma) \textsf{ in} \\
\quad\quad \textsf{let } augend = body(arg\_oop, \sigma) \textsf{ in} \\
\quad\quad \textsf{if } augend \in \mathbb{Z} \\
\quad\quad \textsf{then } find\_or\_make\_immutable(addend+augend, \textsf{Integer}, \sigma) \\
\quad\quad \textsf{else } plus\_error(rcvr\_oop, arg\_oop, \sigma)
\end{array}
$$

One thing that *plus_error* could do is return a special *Oop* that could not be a valid answer, e.g., NILOOP. Then the primitive could be encapsulated within a general addition method, and suitable action taken if the result was not an integer.

3.9.1 Primitive Equivalence

The *equivalence* primitive compares the *Oops* of its receiver and argument. No extension to the model is required to support this primitive, as equality between *Oops* must be defined anyhow. However, care must be taken in its definition, because arbitrary *Oops* can be chosen when creating primitive objects.

When creating an integer object, *find_or_make_immutable* has the choice of making a new object or returning the *Oop* of an existing, equivalent object. Thus it would seem natural to make a special case for the comparison of integer objects: rather than comparing *Oops*, the values are compared. The same argument can be extended to all immutable objects.[11]

A definition of the equivalence primitive is:

[11] But see §5.4 for a twist in the story.

$$
\begin{array}{l}
equivalence\_primitive : Primitive\_method \\
equivalence\_primitive \triangleq \\
\quad \lambda rcvr\_oop, [arg\_oop], \sigma \cdot \\
\qquad \text{let } r\_body = body(rcvr\_oop, \sigma) \text{ in} \\
\qquad \text{let } a\_body = body(arg\_oop, \sigma) \text{ in} \\
\qquad \text{if (if } \{r\_body, a\_body\} \subset \mathbb{Z} \\
\qquad\quad \text{then } r\_body = a\_body \\
\qquad\quad \text{else } rcvr\_oop = arg\_oop) \\
\qquad \text{then (TRUEOOP, } \sigma) \\
\qquad \text{else (FALSEOOP, } \sigma)
\end{array}
$$

where TRUEOOP and FALSEOOP are the *Oops* of distinguished objects representing truth and falsehood, respectively.

Introduction of the equivalence primitive creates a subtle problem: the argument of the primitive, if not the same as the receiver, gets no opportunity to actively participate in the test. Why this might cause problems is illustrated by the following example. A useful technique in object-oriented programming is the idea of *surrogate* objects: a surrogate object shadows another object, forwarding messages to it automatically, but performing some additional operation, such as monitoring the message stream. Except for the extra behaviour, one would like the surrogate to behave exactly like the shadowed object, by forwarding all messages. However, the equivalence primitive can be used to see through the deceit: if the receiver of the equivalence test is the shadowed object, and the argument is the surrogate, the primitive will return false, because the surrogate was not actively involved in the operation. If the receiver is the surrogate, and the argument is the shadow, then the surrogate will forward the message to the shadow, and true will be returned.

A possible way out of this is to define equivalence in terms of a lower-level concept: a primitive that maps the *Oop* of its receiver into a unique integer. Then the equivalence method can send a message to both objects asking for integers corresponding to their *Oops*, and compare the integers. A surrogate can forward the message to its shadow, and can never be detected.

This primitive can be formally described as follows:

$$
\begin{aligned}
&oopOf\_primitive : Primitive\_method \\
&oopOf\_primitive \triangleq \\
&\quad \lambda rcvr\_oop, \_, \sigma \cdot \\
&\qquad find\_or\_make\_immutable(oop\_of(rcvr\_oop), \mathsf{Integer}, \sigma)
\end{aligned}
$$

where *oop_of* is any bijection from *Oops* to integers.

This example highlights a more general problem: that of non-unary primitives that take arguments of any class. These primitives will always violate the Principle of Object Encapsulation by inspecting the internal state of the argument.

3.9.2 Enumerating Objects by Class

Smalltalk provides two primitives that facilitate enumeration of all the objects in a class. One finds the "first" instance of that class, and the other, given an instance, finds the "next" instance of the class. Together, these can be used to define an iterator that enumerates all instances of a class.

To accommodate these primitives, the model has to be extended so that an ordering is defined on the *Oops* in any *Object_memory*. This in itself is not a problem, but a more serious problem is that enumeration may reincarnate "dead objects" that would otherwise have been garbage collected. Some Smalltalk systems actually do this [CWB86], but it is obviously undesirable. One could circumvent the problem by insisting that garbage collection took place when either of these primitives was invoked, but the cost of using them would be prohibitive, as the time taken to garbage collect is proportional to the number of objects in the memory.

Primitive objects also complicate the definition of enumeration. If one cannot distinguish between two integer objects with equal values (because they behave identically, even to the point of being considered equivalent [§3.9.1]), what should be returned when enumerating instances of primitive classes? Should all the integers objects in the memory be returned, even though there may be some with the same value? Clearly, there is no simple way out.

Furthermore, because enumeration plays havoc with virtual memory systems, it is likely in the future to be seen as too expensive to be implemented as a primitive. In the author's opinion, as object-oriented systems become distributed, there will no place for such a feature, as it will be computationally

intractable. Furthermore, enumeration is somehow "un-object-oriented": enumeration violates the principle that an object can only increase its knowledge about its world by sending or receiving messages.

3.9.3 Changing the Identity of Objects

In [KC86], Khoshafian and Copeland argue for strong support for object identity within object-oriented languages, and describe several operators on identity that they deem to be desirable in a language. Together with the equivalence primitive described in §3.9.1, they mention the *merge* primitive, which merges the identities of two objects into one. Merging an object with *Oop* p_1 into an object with *Oop* p_2 requires changing all references to p_1 to refer to p_2.[12] Clearly, this is in violation of the Principle of Object Encapsulation. Moreover, it is difficult to see how this could be implemented without one of the objects concerned being merged without being sent a message, thereby violating the Principle of Object Identity. This is another illustration of the problem caused by binary primitives.

The become primitive of Smalltalk has similar problems. The idea of become is that the identities of two objects are exchanged. In exchanging the identities of p_1 and p_2, the object memory σ becomes

$$\sigma \dagger \{p_1 \mapsto \sigma(p_2), p_2 \mapsto \sigma(p_1)\}$$

Again, this violates the Principle of Object Identity, as one of the objects has no say in the matter. Some Smalltalk implementations have great trouble supporting this primitive, and their implementors have chosen to abandon it in the long term [SUH86, LGFT86].[13]

The merge and become primitives are examples of a more general problem concerning the change of identity of objects in an object-oriented system. Simple reasoning shows that changing an object's identity will always violate

[12] Or *vice versa*, or alternatively a new *Oop* could be used to refer to the merged object, and all occurrences of both p_1 and p_2 changed to the new *Oop*.

[13] However, it is fair to say that their reasons are more due to implementation problems than concerns for semantic issues. It can be argued, however, that the implementation problems arise from the semantic problems.

either the Principle of Object Encapsulation, of the Principle of Object Identity, or both. If an object's identity is to change, it must assume either the identity of an existing object, violating the Principle of Object Identity, or a new identity. If it assumes a new identity, existing objects that reference that object must either now refer to some other object (dangling references are not allowed), which violates the Principle of Object Encapsulation, or refer to the same object, in which case the *Oop* may have changed but it does not have a detectably different identity (i.e., the new store is isomorphic to the old one). This leads to:

The (Strong) Principle of Object Identity Every object has a unique identity, which cannot change.

Clearly, an object's identity is connected with its *Oop*, but it is not the same thing. If a change in *Oop* leaves a new store isomorphic to the old one, the identity of the object has not changed.

3.10 Primitive Classes

Just as primitive methods are required to supply functionality that cannot be expressed otherwise in the language, primitive objects are required to supply state that cannot be modelled by user-defined objects (or can only be modelled inefficiently). Primitive methods are frequently related to primitive objects, but as the last section showed, need not be.

To keep the model uniform, primitive objects are instances of primitive classes, which differ from conventional classes in that they must be present in all programs (e.g., as part of a *standard environment* or *prelude*), and cannot be subclassed or used in new-expressions.

Instances of primitive classes are usually created by literal expressions in the program, or by primitive methods invoked on other primitive objects. An example of the former is the literal expression "3", which causes an object with *Body* = 3 to appear in the store if one is not present. Alternatively, the object representing 3 can be created by sending the message "+" to an object representing 1, with argument 2 (assuming the primitive method for integer addition is bound to +).

Primitive objects such as those representing integers have no mutable state. Why then represent them as objects, rather than extending the domain of values? For example, *Oops* could be restricted so that they only referred to non-primitive objects:

$$Value = Oop \cup Primitive\_object$$

$$Object\_memory = Oop \xrightarrow{m} Object$$

$$\begin{aligned} Object &:: Class &&: Class\_name \\ &\quad Body &&: Id \xrightarrow{m} Value \end{aligned}$$

This approach is taken in the definition of POOL [AdBKR86]. The advantage of having primitive objects and classes is that the semantics of message-sending are the same for all objects, allowing the user to define additional methods of her own for integers, and making primitive methods appear to the user the same as all other methods. Other languages that take this approach are Scheme [RC86] and PostScript [Ado85]. Another reason for having primitive classes is that there can exist primitive objects with mutable state; see §3.10.4 for an example. All mutable objects require their own identity.

3.10.1 Booleans

Although the definition of the equivalence primitive (§3.9.1) suggested that every store should contain objects representing truth and falsehood, with predetermined *Oops*, it is not necessary for these objects to be primitive. As in the Smalltalk system, the true and false objects can be instances of classes that have conventional definitions. The only requirement is that these classes be defined in every program that uses boolean objects. Also, a choice should be made whether there should only be one object for each of true and false, or whether multiple instances of each are allowed. As boolean objects do not have mutable state, there is little to distinguish between the cases, but it is likely that an implementation can gain from having only one instance of each.

The syntax of boolean objects is:

$$Literal\_object = \ldots \cup Bool\_literal$$

$$Bool\_literal :: \mathbb{B}$$

and the semantics for single instances is:

$$MExpression[\![mk\text{-}Bool\_literal(bool)]\!]\rho\delta\sigma \triangleq$$
$$\text{(if } bool \text{ then TRUEOOP else FALSEOOP, } \delta, \sigma)$$

Owing to the dynamic binding inherent in message-sending, boolean operations can be defined without resort to primitives. For example, conjunction can be defined by a method in the class of true that returns its argument, and a method in the class of false that returns its receiver. If *blocks* are present in the language,[14] even control structures such as if and while can be defined in terms of messages and block evaluations. The section on Boolean objects in [GR83] shows how this can be done. The semantics of blocks are presented in Chapter 5.

3.10.2 Integers

An alternative to having the entire set of integers available as primitive objects might be to restrict the set, say to those easily represented by a machine's hardware. Smalltalk does this, but by using the notion of primitive failure (§3.9), user-definable classes can implement arbitrary precision integers without the user of those classes being aware that multiple classes are involved. The semantics can easily be changed to have $Primitive\_object = \{-maxint, \ldots, maxint\}$, and the primitives suitably redefined.

3.10.3 Symbols

A symbol is a primitive object that represents an identifier of some sort. In Lisp, symbols are created by quoting, thus: `'foo`. Smalltalk also has symbols as objects.

Symbols can be used in "computed sends"; the selector of the message to be sent is determined at run-time by evaluating an expression. This is precisely what the perform primitive does in Smalltalk.

Adding symbols to the syntax of literals gives:

[14] Smalltalk blocks, not to be confused with blocks in block-structured languages.

$$Literal\_object = \ldots \cup Symbol\_literal$$

$$Symbol\_literal :: Id$$

To add symbols to the semantic domain of primitive objects is straightforward:

$$Symbol = Id$$

$$Primitive\_object = \ldots \cup Symbol$$

$$\begin{array}{l} MExpression[\![mk\text{-}Symbol\_literal(s)]\!]\rho\delta\sigma \triangleq \\ \quad \textsf{let } (oop, \sigma') = find\_or\_make\_immutable(s, \textsf{Symbol}, \sigma) \textsf{ in} \\ \quad (oop, \delta, \sigma') \end{array}$$

The perform primitive takes a destination object, a symbol representing a message selector, and a number of arguments, and sends a message containing the selector and arguments to the destination:

$$\begin{array}{l} perform\_primitive : Primitive\_method \\ perform\_primitive \triangleq \\ \quad \lambda rcvr, \textsf{cons}(sel\_oop, args), \sigma \cdot \\ \quad\quad \textsf{let } sel = body(sel\_oop, \sigma) \textsf{ in} \\ \quad\quad \textsf{if } sel \in Symbol \wedge nargs(sel) = \textsf{len } args \\ \quad\quad \textsf{then } perform(sel, pd, rcvr, args, \sigma) \\ \quad\quad \textsf{else } perform\_error(sel\_oop, sel, pd, rcvr, args, \sigma) \end{array}$$

Points to note about *perform_primitive*:

- The program denotation, *pd*, is a free variable of the above definition. It should be bound to the denotation of the surrounding program.
- The function *nargs* determines how many arguments are associated with a selector. In Smalltalk, this can be determined by examining the name of the selector (see §4.1.1 for a definition of *nargs*). For example, the selector foo:bar: always takes two arguments. In languages where selectors do not have this property, a run-time test must ensure that the number of formal parameters matches the number of actual parameters.

- The *perform_error* function encapsulates the behaviour of the system when either the first argument to the primitive is not a symbol, or the wrong number of arguments are passed. It is not specified in detail here.

Another potential use of symbols is to eliminate two types of expression from the language, namely, assignments and references to instance variables. By defining primitive methods for these, subclasses can override the primitives, providing extra behaviour (such as active values and probes [BS83]). To do this, expressions of the form var $\leftarrow \langle value \rangle$ are replaced with messages of the form self assign: $\langle value \rangle$ to: var, and references to an instance variable var are replaced with self instVar: var.[15]

The primitives are defined like this:

$$
\begin{array}{l}
assign\_primitive : Primitive\_method \\
assign\_primitive \triangleq \\
\quad \lambda rcvr, [value, iv\_oop], \sigma \cdot \\
\quad\quad \mathsf{let}\ iv = body(iv\_oop, \sigma)\ \mathsf{in} \\
\quad\quad \mathsf{if}\ iv \in \mathsf{dom}\, body(rcvr, \sigma) \\
\quad\quad \mathsf{then}\ (value, update\_inst\_var(iv, rcvr, value, \sigma)) \\
\quad\quad \mathsf{else}\ assign\_error(rcvr, iv\_oop, value, \sigma)
\end{array}
$$

$$
\begin{array}{l}
reference\_primitive : Primitive\_method \\
reference\_primitive \triangleq \\
\quad \lambda rcvr, [iv\_oop], \sigma \cdot \\
\quad\quad \mathsf{let}\ iv = body(iv\_oop, \sigma)\ \mathsf{in} \\
\quad\quad \mathsf{if}\ iv \in \mathsf{dom}\, body(rcvr, \sigma) \\
\quad\quad \mathsf{then}\ (inst\_var(iv, rcvr, \sigma), \sigma) \\
\quad\quad \mathsf{else}\ reference\_error(rcvr, iv\_oop, value, \sigma)
\end{array}
$$

[15] Obviously, a pre-processing stage would be used to make these transformations; the syntax seen by the user would be unchanged.

3.10.4 Indexable Objects

Most object-oriented languages provide a facility for creating objects with numbered rather than named instance variables. *Indexed instance variables* are analogous to arrays in conventional languages.

There are two approaches to providing indexed instance variables: either objects can have both named and indexed instance variables, or different types of objects have either named or indexed instance variables. In Smalltalk, for example, a class can be declared to have both named and indexed instance variables. Incorporating this into the present syntax, a *Class_body* becomes:

$$\begin{array}{rrl} \mathit{Class\_body} :: & \mathit{Instvars} & : \mathit{Id}\text{-set} \\ & \mathit{Methods} & : \mathit{Selector} \xrightarrow{m} \\ & & \quad \mathit{Method\_body} \cup \mathit{Primitive\_method} \\ & \mathit{Has\_indexed} & : \mathbb{B} \end{array}$$

and the definition of *Plain_object* becomes:

$$\mathit{Plain\_object} = \mathit{Id} \cup \mathbb{N}_1 \xrightarrow{m} \mathit{Oop}$$

(Alternatively, and equivalently, there could be two types of *Class_body*, say *Normal_class_body* and *Indexable_class_body*, and two basic types of object, with and without indexable fields.)

The static environment has an extra component recording which classes have indexable objects:

$$\begin{array}{rrl} \mathit{SEnv} :: & \mathit{Instvars} & : \mathit{Class\_name} \xrightarrow{m} \mathit{Id}\text{-set} \\ & \mathit{PD} & : \mathit{Program\_den} \\ & \mathit{Indexable} & : \mathit{Class\_name}\text{-set} \end{array}$$

A different kind of new-expression would be used to create objects with indexable fields:

$$\mathit{Expression} = \ldots \cup \mathit{New\_indexable\_object}$$

$$\begin{array}{rrl} \mathit{New\_indexable\_object} :: & \mathit{Class} & : \mathit{Class\_name} \\ & \mathit{Size} & : \mathit{Expression} \end{array}$$

$$
\begin{array}{l}
MExpression[\![mk\text{-}New\_indexable\_object(class, size\_expr)]\!]\,\rho\delta\sigma \;\triangleq \\
\quad \textsf{let } (sizeoop, \delta', \sigma') = MExpression[\![size\_expr]\!]\,\rho\delta\sigma \textsf{ in} \\
\quad \textsf{if } body(sizeoop, \sigma') \in \mathbb{N} \wedge class \in Indexable(\rho) \\
\quad \textsf{then let } size = body(sizeoop, \sigma') \textsf{ in} \\
\qquad \textsf{let } new\_obj = mk\text{-}Object(class, \\
\qquad\qquad initialise(Instvars(\rho)(class) \cup \{1 \ldots size\})) \textsf{ in} \\
\qquad \textsf{let } (new\_oop, \sigma'') = create(new\_obj, \sigma') \textsf{ in} \\
\qquad (new\_oop, \delta', \sigma'') \\
\quad \textsf{else } new\_error(sizeoop, class, \delta', \sigma')
\end{array}
$$

(The domain of *initialise* is extended to include natural numbers.)

New primitives could be used to access the indexable fields, or to change the number of indexable fields:[16]

$$
\begin{array}{l}
at\_primitive : Primitive\_method \\
at\_primitive \;\triangleq \\
\quad \lambda rcvr, [index\_oop], \sigma \cdot \\
\qquad \textsf{let } index = body(index\_oop, \sigma) \textsf{ in} \\
\qquad \textsf{if } index \in \mathbb{N}_1 \wedge in\_bounds(rcvr, index, \sigma) \\
\qquad \textsf{then } (body(rcvr, \sigma)(index), \sigma) \\
\qquad \textsf{else } bound\_error(rcvr, index\_oop, \sigma)
\end{array}
$$

$$
\begin{array}{l}
atput\_primitive : Primitive\_method \\
atput\_primitive \;\triangleq \\
\quad \lambda rcvr, [index\_oop, value], \sigma \cdot \\
\qquad \textsf{let } index = body(index\_oop, \sigma) \textsf{ in} \\
\qquad \textsf{if } index \in \mathbb{N}_1 \wedge in\_bounds(rcvr, index, \sigma) \\
\qquad \textsf{then } (rcvr, update\_inst\_var(index, rcvr, value, \sigma)) \\
\qquad \textsf{else } bound\_error(rcvr, index\_oop, \sigma)
\end{array}
$$

[16] The provision of a grow primitive obviates the need to use the troublesome become primitive to change the number of indexable fields.

$$in\_bounds : Oop \times \mathbb{N}_1 \times Object\_memory \rightarrow \mathbb{B}$$
$$in\_bounds(rcvr, index, \sigma) \triangleq index \in \mathsf{dom}\, body(rcvr, \sigma)$$

$$grow\_primitive : Primitive\_Method$$
$$grow\_primitive \triangleq$$
$$\lambda rcvr, [size\_oop], \sigma \cdot$$
$$\quad \mathsf{let}\ size = body(size\_oop, \sigma)\ \mathsf{in}$$
$$\quad \mathsf{if}\ size \in \mathbb{N} \wedge body(rcvr, \sigma) \in Plain\_object$$
$$\quad \mathsf{then}\ (rcvr, grow(rcvr, size, \sigma))$$
$$\quad \mathsf{else}\ grow\_error(rcvr, size\_oop, \sigma)$$

$$grow : Oop \times \mathbb{N} \times Object\_memory \rightarrow Object\_memory$$
$$grow(oop, size, \sigma) \triangleq$$
$$\quad \mathsf{let}\ new\_body = \mathsf{if}\ size \in \mathsf{dom}\, body(oop, \sigma)$$
$$\qquad \mathsf{then}\ \{1, \ldots, size\} \lhd body(oop, \sigma)$$
$$\qquad \mathsf{else}\ \{i \mapsto \textsc{NilOop} \mid i \in \{1, \ldots, size\}\}$$
$$\qquad\quad \dagger\ body(oop, \sigma)$$
$$\quad \mathsf{in}$$
$$(oop, \sigma \dagger \{oop \mapsto \mu(\sigma(oop), Body \mapsto new\_body)\})$$

(The domains of *inst_var* and *update_inst_var* are extended to include natural numbers).

The other approach requires a special class, whose instances are objects with indexed variables:

$$Indexable\_object = \mathbb{N}_1 \xrightarrow{m} Oop$$

$$Primitive\_object = \ldots \cup Indexable\_object$$

Special syntax is required to create instances of the Array class:

$$New\_indexable\_object :: Size : Expression$$

$$
\begin{array}{l}
\mathit{MExpression}[\![\mathit{mk\text{-}New\_indexable\_object}(\mathit{size\_expr})]\!]\,\rho\delta\sigma \quad \triangleq \\
\quad \mathsf{let}\ (\mathit{sizeoop}, \delta', \sigma') = \mathit{MExpression}[\![\mathit{size\_expr}]\!]\,\rho\delta\sigma\ \ \mathsf{in} \\
\quad \mathsf{if}\ \mathit{body}(\mathit{sizeoop}, \sigma') \in \mathbb{N} \\
\quad \mathsf{then}\ \mathsf{let}\ \mathit{size} = \mathit{body}(\mathit{sizeoop}, \sigma')\ \ \mathsf{in} \\
\qquad \mathsf{let}\ \mathit{new\_obj} = \\
\qquad\quad \mathit{mk\text{-}Object}(\mathsf{Array}, \{i \mapsto \textsc{NilOop} \mid i \in \{1 \ldots \mathit{size}\}\})\ \ \mathsf{in} \\
\qquad \mathsf{let}\ (\mathit{new\_oop}, \sigma'') = \mathit{create}(\mathit{new\_obj}, \sigma')\ \ \mathsf{in} \\
\qquad (\mathit{new\_oop}, \delta', \sigma'') \\
\quad \mathsf{else}\ \mathit{new\_error}(\mathit{sizeoop}, \mathit{class}, \delta', \sigma')
\end{array}
$$

The accessing and updating primitives are similar to those for the former case.

Although these alternatives seem similar, the next chapter will show that the latter is preferable.

3.10.5 Other Types of Primitive Objects

In addition to integers, one could consider a subset of the real numbers, namely the floating point numbers, as primitive objects.

A more unusual candidate for the category of primitive objects is that of bitmaps: two-dimensional arrays of pixel values. Although bitmaps can be defined using indexed instance variables, the operations defined on bitmaps are suitably unusual to warrant not doing so [GS82].

3.11 Classes as Objects

In common with some other languages, Smalltalk has classes as objects that can be sent messages. There are two reasons for this:

1. Objects are created (and initialised) by sending messages to the appropriate class; this obviates the need for a special syntactic category of new-expressions.

2. Classes have mutable state, which can be altered by programs written in Smalltalk, enabling both programs and programming environment to be written in the same language, and to share code.

This section discusses how the formal model is changed to encompass these ideas.

Tackling point (1) first, classes can be added into the domain of primitive objects, and a primitive defined to create instances of each class. Because instances of different classes have different structure, the names of the instance variables are part of each class object, and no longer need be in the static environment:

$$Primitive\_object = \ldots \cup Class\_obj$$

$$\begin{aligned} Class\_obj :: \quad & Name : Class\_name \\ & Instvars : Id\text{-set} \\ & Method\_response : Class\_den \end{aligned}$$

The primitive method for object creation is this:

$$\begin{aligned} & new\_primitive : Primitive\_method \\ & new\_primitive \triangleq \\ & \quad \lambda class\_oop, \_, \sigma \cdot \\ & \quad\quad \textsf{let } instvars = Instvars(body(class\_oop, \sigma)) \textsf{ in} \\ & \quad\quad \textsf{let } new\_obj = mk\text{-}Object(Name(body(class\_oop, \sigma)), \\ & \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad initialise(instvars)) \textsf{ in} \\ & \quad\quad create(new\_obj, \sigma) \end{aligned}$$

where the definition of *create* is as before. Non-primitive classes would be instances of a primitive class that provided the new primitive. Primitive classes would be instances of a different primitive class (because one cannot create primitive objects using new); this would be an instance of itself. This is reminiscent of the situation in Smalltalk-76 [Ing78], where all classes were instances of a single class, which was an instance of itself.

A further step would be to redefine *Object* so that the *Class* field referred to an object in the *Object_memory*, rather than to a class definition (cf. p. 27):

$$\begin{aligned} Object :: \quad & Class : Oop \\ & Body : Object\_body \end{aligned}$$

Method lookup proceeds by following the *Class* link in an object to the denotation of the object's class (cf. p. 42):

$$perform : Selector \times Oop \times Oop^* \times Object\_memory \rightarrow Oop \times Object\_memory$$

$$perform(sel, rcvr, args, \sigma) \triangleq$$
  let $class\_oop = class(rcvr, \sigma)$ in
  if $sel \in selectors(class\_oop, \sigma)$
  then $method(sel, class\_oop, \sigma)(rcvr, args, \sigma)$
  else $message\_not\_understood(sel, class\_oop, rcvr, args, \sigma)$

$$selectors : Oop \times Object\_memory \rightarrow Selector\text{-set}$$
$$selectors(class, \sigma) \triangleq \text{dom } Method\_response(body(class, \sigma))$$

$$method : Selector \times Oop \times Object\_memory \rightarrow Method\_den$$
$$method(sel, class, \sigma) \triangleq Method\_response(body(class, \sigma))(sel)$$

This makes the static environment redundant in the meaning functions; instead, it is embedded within the class objects in the object memory.

An alternative approach, exemplified by Smalltalk-80, is to have every class the sole instance of a *metaclass* (the class of a class is termed a metaclass), and every metaclass an instance of a "class of metaclasses", called Metaclass [GR83]. This has the advantage that every class can have extra methods added that not only create a new instance (by invoking the primitive), but also perform class-specific initialisation.

3.11.1 Class and Global Variables

Most languages provide variables that are accessible to all instances of a class. If classes are objects then their denotation can be augmented to include the variables (cf. p. 59):

$$\begin{aligned} Class\_obj :: \quad & Name : Class\_name \\ & Instvars : Id\text{-set} \\ & Method\_response : Class\_den \\ & Classvars : Id \xrightarrow{m} Oop \end{aligned}$$

The domain of identifiers is also extended to include these new variables, and the meaning functions extended to access and update them (cf. pp. 35, 40):

$$Var\_id = Arg\_id \cup Temp\_id \cup Inst\_var\_id \cup Class\_var\_id$$

$$AVar\_id = Temp\_id \cup Inst\_var\_id \cup Class\_var\_id$$

$$Class\_var\_id :: Id$$

$$MExpression[\![mk\text{-}Class\_var\_id(id)]\!]\rho\delta\sigma \triangleq (class\_var(id, class(Rcvr(\delta), \sigma), \sigma), \delta, \sigma)$$

$$\begin{array}{l} MExpression[\![mk\text{-}Assignment(id, rhs)]\!]\rho\delta\sigma \triangleq \\ \quad \text{let } (result, \delta', \sigma') = MExpression[\![rhs]\!]\rho\delta\sigma \text{ in} \\ \quad \text{cases } id \text{ of} \\ \quad \ldots \\ \quad mk\text{-}Class\_var\_id(cv) \rightarrow (result, \delta', \\ \qquad\qquad update\_class\_var(cv, \\ \qquad\qquad\quad class(Rcvr(\delta'), \sigma'), result, \sigma')) \\ \quad \text{end} \end{array}$$

$$\begin{array}{l} update\_class\_var : Id \times Oop \times Oop \times Object\_memory \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow Object\_memory \\ update\_class\_var(id, class, value, \sigma) \triangleq \\ \quad \text{let } old = body(class, \sigma) \text{ in} \\ \quad \text{let } class' = \mu(obj, Classvars \mapsto \\ \qquad\qquad\qquad\qquad Classvars(old) \dagger \{id \mapsto value\}) \text{ in} \\ \quad \sigma \dagger \{class \mapsto \mu(\sigma(class), Body \mapsto class')\} \end{array}$$

If classes are not objects, then an extra component has to be added to *Object_memory* for class variables:

$$\begin{array}{lll} Object\_memory :: & Objects & : Oop \xrightarrow{m} Object \\ & Classvars & : Class\_name \xrightarrow{m} Id \xrightarrow{m} Oop \end{array}$$

However, an alternative to both of these schemes in to abolish class variables entirely. In §3.3 it was argued that one object should not be able to change the internal state of another without sending it a message (the Principle of Object Encapsulation). Modifying class variables directly is in breach of this principle, because they are components of class objects. Rather than having extra syntax and extended meaning functions for class variables, primitive methods can be provided to access them, so long as symbols are present as objects (§3.10.3). Access to class variables then takes place by sending messages.

$$
\begin{array}{l}
cv\_assign\_primitive : Primitive\_method \\
cv\_assign\_primitive \quad \triangleq \\
\quad \lambda class\_oop, [cv\_oop, value], \sigma \cdot \\
\quad\quad \mathsf{let}\ cv = body(cv\_oop, \sigma)\ \mathsf{in} \\
\quad\quad \mathsf{if}\ cv \in \mathsf{dom}\, Classvars(body(class\_oop, \sigma)) \\
\quad\quad \mathsf{then}\ (value, update\_class\_var(cv, class\_oop, value, \sigma)) \\
\quad\quad \mathsf{else}\ cv\_assign\_error(class\_oop, cv\_oop, value, \sigma)
\end{array}
$$

$$
\begin{array}{l}
cv\_reference\_primitive : Primitive\_method \\
cv\_reference\_primitive \quad \triangleq \\
\quad \lambda class\_oop, [cv\_oop], \sigma \cdot \\
\quad\quad \mathsf{let}\ cv = body(cv\_oop, \sigma)\ \mathsf{in} \\
\quad\quad \mathsf{if}\ cv \in \mathsf{dom}\, Classvars(body(class\_oop, \sigma)) \\
\quad\quad \mathsf{then}\ (class\_var(cv, class\_oop, \sigma), \sigma) \\
\quad\quad \mathsf{else}\ cv\_reference\_error(class\_oop, cv\_oop, value, \sigma)
\end{array}
$$

$$
\begin{array}{l}
class\_var : Id \times Oop \times Object\_memory \to Oop \\
class\_var(id, class, \sigma) \quad \triangleq \quad Classvars(body(class, \sigma))(id)
\end{array}
$$

(These primitives are similar to the *assign* and *reference* primitives of §3.10.3.)

Finally, let us note that the argument in favour of abolition of class variables (that they violate the Principle of Object Encapsulation) can also be applied to global variables: they are best eliminated from object-oriented languages.

3.11.2 Mutable Classes

The second use of classes as objects, namely to support the programming environment, is more complex. First, the situation in Smalltalk-80 is described, and then other approaches are outlined.

In the Smalltalk system, classes and methods are objects. However, rather than placing the denotations of methods and classes in the object memory (as has been done above), representations of those denotations, with internal, accessible structure are used. Methods, for example, are represented by sequences of instructions for a *virtual machine*. This enables the Smalltalk compiler, which transforms Smalltalk source code into instruction sequences for the virtual machine, to be written in Smalltalk itself. Based on this, the Smalltalk programming environment allows class and method definitions to be edited interactively within Smalltalk, compiled with the Smalltalk compiler, and installed in the system whilst it is running.

Although it is possible to define such a system formally by defining the virtual machine's semantics, the definition would not be useful when examining the design of the language, or the semantics of any program (other than the compiler, perhaps). The semantics of a programming language are best defined in abstract terms, without recourse to operational notions. Furthermore, one can only sensibly reason about a program when the semantics of the language in which that program is written are fixed; in Smalltalk that need not be the case, as the compiler can be modified and recompiled within the system.

Even if the language semantics are fixed, there is still a problem with denotational definitions if the language supports self-modification of programs. Clearly, if a program can modify itself, its meaning cannot be determined at "compile-time." For example, the meaning function for an eval expression that computes an expression and then evaluates it might be:

$$
\begin{aligned}
&MExpr : Expr \to Env \to Store \to Store \\
&MExpr\,[\![\textsf{eval}\ e]\!]\,\rho\sigma \quad \triangleq \\
&\qquad \textsf{let}\ (expr\_val, \sigma') = MExpr\,[\![e]\!]\,\rho\sigma\ \ \textsf{in} \\
&\qquad MExpr\,[\![expr\_val]\!]\,\rho\sigma'
\end{aligned}
$$

In semantic terms, the denotational definition is not well-founded: the denotation of an expression is not composed from its constituent parts, but from

run-time values.

A technique known as *reflection* has been used to address this problem for Lisp [Smi82], and shows some promise as a technique with wider application, provided that the foundational issues can be resolved [Yel87]. A reflective language has primitives that enable a program to inspect and modify its text, environment and continuation. To do this, the semantic domains involved in the definition of the language must also be part of the value space. One primitive, termed a *reifier*, returns the current text, environment and continuation (or some subset of the three) as a value. Another, termed a *reflective* primitive, takes arguments representing text, environment and continuation, and "installs" them as the current text, environment and continuation. The semantics of a program written in a reflective language are given by a fixed point of the meaning functions; in this sense the meaning functions define an interpreter.

Given that the state of the art regarding the semantic definition of self-modifying programs is not advanced, and that this issue is not an especially important feature of object-oriented languages (it is simply that it is more useful in a dynamically-bound language), no attempt will be made to fit these features into the formal model.

3.12 An Alternative to Classes: Prototypes

Rather than grouping objects with similar behaviour into classes, a more dynamic system of organisation based on *prototypes* and *delegation* can be used. Although the idea of prototype objects has been around for at least a decade (it appears in ThingLab [Bor77, Bor81]), it gained widespread recognition as an important technique when given prominence by Lieberman [Lie86b].

In a prototype-based system, an object is created by *cloning* an existing object; the existing object is said to be the clone's *prototype*. Clones can in turn be cloned; the prototype relation between objects defines a forest of objects.

Immediately after a clone has been created, it has no instance variables or methods of its own; they are all inherited from its prototype. When an assignment to an instance variable takes place in an inherited method, the instance variable is created in the clone if it does not exist; the instance variable in the prototype is unaltered.

An object can choose to *delegate* a message to another object. The object

that has been delegated to has access to the instance variables of the delegating object while processing the message.

Relating these ideas to the semantics presented earlier, the definition of a *Plain_object* changes to use a prototype rather than a class:

$$\begin{aligned} \mathit{Plain\_object} :: \mathit{Prototype} &: [\mathit{Oop}] \\ \mathit{Attributes} &: \mathit{Id} \xrightarrow{m} \mathit{Oop} \end{aligned}$$

Note that the field *Instvars* has been replaced by one named *Attributes*. This is because the private internal state can also include methods. The domain of primitive objects is extended to include method denotations:

$$\mathit{Primitive\_object} = \ldots \cup \mathit{Method\_den}$$

Methods come in two varieties, conventional methods and methods that delegate to another object (for simplicity we shall only allow delegation to objects that are referred to by an instance variable, rather than an arbitrary expression):

$$\mathit{Method\_body} = \mathit{Conventional\_method} \cup \mathit{Delegated\_method}$$

$$\mathit{Delegated\_method} :: \mathit{Id}$$

$$\begin{aligned} \mathit{Conventional\_method} :: \mathit{Params} &: \mathit{Ulist}(\mathit{Id}) \\ \mathit{Temps} &: \mathit{Id}\text{-set} \\ \mathit{Expr} &: \mathit{Expression} \end{aligned}$$

To support delegation, the denotation of a method has to be altered to take two extra parameters (cf. p. 36): the first is the selector of the message, and is used when delegating:

$$\begin{aligned} \mathit{Method\_den} = \mathit{Selector} \rightarrow \\ \mathit{Oop} \times \mathit{Oop} \times \mathit{Oop}^* \times \mathit{Object\_memory} \rightarrow \\ \mathit{Oop} \times \mathit{Object\_memory} \end{aligned}$$

The second parameter is the *Oop* of the message receiver (known as the *client*), and the first is the *Oop* of the object that the message was delegated to (known as the *receiver*); the other parameters are as before.

The dynamic environment also requires an extra component (cf. p. 38):

$$
\begin{array}{rrcl}
DEnv & :: & Rcvr & : & Oop \\
 & & Client & : & Oop \\
 & & Params & : & Id \xrightarrow{m} Oop \\
 & & Temps & : & Id \xrightarrow{m} Oop
\end{array}
$$

3.12.1 Identifiers and Assignments

As methods can be assigned dynamically from object to object, and instance variables created at run-time, a dynamic search must take place for the value of an instance variable, possibly resulting in an error (the error behaviour is left unspecified). Contrast this with §3.6, where the structure and behaviour of every object is completely determined by its class, and cannot change.

Also, when a message has been delegated, self refers back to the object that delegated the message, rather than the object that is processing the message. This is explained fully in [Lie86b]; in short, an object that is delegated to provides the message response, but gets state information from the delegating object.

Therefore, the search begins at the current message receiver. If an attribute is not found, the client is searched; if still not found, the prototype chain is followed.

$$
\begin{array}{l}
find\_attribute : Id \times Oop \times Object\_memory \rightarrow Oop \\
find\_attribute(id, obj, \sigma) \quad \triangleq \\
\quad \text{if } id \in \text{dom } Attributes(\sigma(obj)) \\
\quad \text{then } Attributes(\sigma(obj))(id) \\
\quad \text{else if } Prototype(\sigma(obj)) = \text{nil} \\
\quad\quad \text{then } unfound\_attribute\_error(id, obj, \sigma) \\
\quad\quad \text{else } find\_attribute(id, Prototype(\sigma(obj)), \sigma)
\end{array}
$$

If after searching the entire prototype chain the attribute is not found, an error is signalled (the error behaviour is left unspecified).

$$MExpression : Expression \to DEnv \to Object\_memory \to Oop \times DEnv \times Object\_memory$$

$$MExpression[\![mk\text{-}Inst\_var\_id(id)]\!]\,\delta\,\sigma \triangleq$$
if $id \in \mathrm{dom}\, Attributes(\sigma(Rcvr(\delta)))$
then $(Attributes(\sigma(Rcvr(\delta)))(id), \delta, \sigma)$
else $find\_attribute(id, Client(\delta), \sigma)$

As mentioned earlier, instance variables are created as required:

$$MExpression[\![mk\text{-}Assignment(id, rhs)]\!]\,\delta\,\sigma \triangleq$$
let $(result, \delta', \sigma') = MExpression[\![rhs]\!]\,\delta\,\sigma$ in
cases id of
$mk\text{-}Temp\_id(t) \to (result, update\_temp(t, result, \delta'), \sigma')$
$mk\text{-}Inst\_var\_id(iv) \to (result, \delta', update\_inst\_var(iv, Rcvr(\delta'), result, \sigma'))$
end

where *update_inst_var* and *update_temp* are similar to their previous definitions.

3.12.2 Sending and Delegating Messages

The meaning of a conventional method is similar to the class-based definition:

$$MMethod\_body : Method\_body \to Method\_den$$

$$MMethod\_body[\![mk\text{-}Conventional\_method(formals, temps, expr)]\!] \triangleq$$
$\lambda selector \cdot \lambda rcvr, client, actuals, \sigma \cdot$
let $bindings = bind\_args(formals, actuals)$ in
let $\delta = mk\text{-}DEnv(rcvr, client, bindings, initialise(temps))$ in
let $(result, \delta', \sigma') = MExpression[\![expr]\!]\,\delta\,\sigma$ in
$(result, \sigma')$

$$
\begin{aligned}
&MExpression[\![mk\text{-}Message(rcvr, sel, arglist)]\!]\,\delta\,\sigma \;\triangleq\\
&\quad \text{let } (rcvr\_oop, \delta', \sigma') = MExpression[\![rcvr]\!]\,\delta\,\sigma \text{ in}\\
&\quad \text{let } (actuals, \delta'', \sigma'') = MExpression\_list[\![arglist]\!]\,\delta'\sigma' \text{ in}\\
&\quad \text{let } (result, \sigma''') = send(rcvr\_oop, sel, actuals, \sigma'') \text{ in}\\
&\quad (result, \delta'', \sigma''')
\end{aligned}
$$

$$
\begin{aligned}
&send : Oop \times Selector \times Oop^* \times Object\_memory \to\\
&\qquad\qquad Oop \times Object\_memory\\
&send(rcvr, sel, args, \sigma) \;\triangleq\\
&\quad find\_method(sel, rcvr, \sigma)(sel)(rcvr, rcvr, args, \sigma)
\end{aligned}
$$

$$
\begin{aligned}
&find\_method : Selector \times Oop \times Object\_memory \to Method\_den\\
&find\_method(sel, oop, \sigma) \;\triangleq\\
&\quad \text{let } res = find\_attribute(sel, oop, \sigma) \text{ in}\\
&\quad \text{if } body(res, \sigma) \in Method\_den\\
&\quad \text{then } body(res, \sigma)\\
&\quad \text{else } unfound\_method\_error(sel, oop, \sigma)
\end{aligned}
$$

Note that sending a message in the conventional way identifies the receiver and the client as the same object.

A delegated method, on the other hand, changes only the receiver; the client stays the same:

$$
\begin{aligned}
&MMethod\_body[\![mk\text{-}Delegated\_method(id)]\!] \;\triangleq\\
&\quad \lambda selector \cdot \lambda rcvr, client, actuals, \sigma \cdot\\
&\qquad \text{let } \delta = mk\text{-}DEnv(rcvr, client, \{\,\}, \{\,\}) \text{ in}\\
&\qquad \text{let } (dgt, \delta', \sigma') = MExpression[\![mk\text{-}Inst\_var\_id(id)]\!]\,\delta\,\sigma \text{ in}\\
&\qquad delegate(dgt, selector, actuals, client, \sigma')
\end{aligned}
$$

$$
\begin{aligned}
&delegate : Oop \times Selector \times Oop^* \times Oop \times Object\_memory \to\\
&\qquad\qquad Oop \times Object\_memory\\
&delegate(rcvr, sel, args, client, \sigma) \;\triangleq\\
&\quad find\_method(sel, rcvr, \sigma)(sel)(rcvr, client, args, \sigma)
\end{aligned}
$$

This is similar in action to the inherit from statement in the Generalized Object Model (§2.2).

An object created by cloning has no initial internal state; everything is inherited from its prototype:

$$
\begin{aligned}
&\mathit{clone\_primitive}\ (\mathit{Prototype}: \mathit{Oop})\ \mathit{new\_oop}: \mathit{Oop} \\
&\mathsf{ext\ wr}\ \sigma\ :\ \mathit{Object\_memory} \\
&\mathsf{post}\ \mathit{new\_oop} \notin \mathsf{dom}\ \overleftarrow{\sigma} \\
&\qquad \wedge\ \sigma = \overleftarrow{\sigma} \cup \{\mathit{new\_oop} \mapsto \mathit{mk\text{-}Object}(\mathit{prototype}, \{\})\}
\end{aligned}
$$

3.12.3 Summary

Prototypes organise objects in ways that can change much more dynamically than class-based systems. This has led to their use in languages designed for exploratory AI work, such as the series of Act languages developed at MIT [Lie86a, The83]. As Lieberman has shown [Lie86b], a prototype-based system can simulate a class-based system, but not *vice versa*. Also, the use of delegation can lead to a cleaner organisation in some types of system. However, the extra flexibility gained by prototypes results in new kinds of programming errors (references to instance variables that do not exist, for example), and lacks the organisational framework provided by class-based systems that use inheritance. Both types of system have their merits, and it is unlikely that one will become dominant in the near future.

This chapter has described the fundamentals behind class-based and prototype-based systems. The next chapter introduces inheritance to class-based systems; one form of multiple inheritance is shown to be similar to the prototype approach.

Chapter 4

Inheritance

> Everything in the universe goes by indirection.
>
> *Ralph Waldo Emerson*

The previous chapter described the fundamental concepts of object-oriented programming: objects, methods and classes. Classes describe the similarities between objects, and in many object-oriented languages can themselves be related by inheritance. The purpose of inheritance is to encourage reuse of code; it supports a style of programming known as *differential programming*, in which the programmer develops a new class by stating the differences between the new class and an existing class.

A widely-held opinion is that object-oriented programming is inextricably tied to inheritance [Weg86, Str87], although the previous chapter showed that this view is mistaken. Chapter 3 argued that the central concept of object-oriented programming was strong information-hiding, at the object level. Nevertheless, inheritance is widespread amongst object-oriented languages, and is an important technique in the design and construction of object-oriented programs. This chapter adds inheritance to the model presented in the previous

chapter, and discusses its effects on the semantics of object-oriented languages.

The first addition is the simplest form of inheritance, single inheritance. Later, various forms of multiple inheritance are compared to each other and to single inheritance. Other issues, such as static binding, visibility of methods, and use of inherited methods, are also discussed.

4.1 Single Inheritance

In a single inheritance system, a *superclass* relation is defined on the classes such that each class can have at most one superclass.[1] The superclass relation must be well-founded; hence, the classes are organised into a hierarchy, or tree, or, if there is more than one root, a forest.

The change in structure of a class definition is indicated by an extra field in the abstract syntax (cf. p. 32):

$$\begin{aligned} \mathit{Class\_body} :: \ & \mathit{Instvars} : \mathit{Id}\text{-set} \\ & \mathit{Methods} : \mathit{Method\_map} \\ & \mathit{Parent} : \mathit{Parent\_class} \end{aligned}$$

$$\mathit{Method\_map} = \mathit{Selector} \xrightarrow{m} \mathit{Method\_desc}$$

$$\mathit{Method\_desc} = \mathit{Method\_body} \cup \mathit{Primitive\_method}$$

$$\mathit{Parent\_class} = [\mathit{Class\_name}]$$

$$\begin{aligned} & \mathit{parent} : \mathit{Class\_name} \times \mathit{Class\_map} \to \mathit{Parent\_class} \\ & \mathit{parent}(\mathit{class}, \mathit{class\_map}) \ \triangleq \ \mathit{Parent}(\mathit{class\_map}(\mathit{class})) \end{aligned}$$

[1] To avoid confusion when the terms subclass and superclass may be ambiguous, the terms *parent* and *child* are used to mean direct superclass and direct subclass. The meaning of the terms *ancestor* and *descendant* should be obvious.

(All the auxiliary functions defined on elements of the modified abstract syntax assume that certain context conditions are satisfied; these are set out in §4.1.1.)

A class inherits definitions of instance variables and methods from its parent class, which in turn inherits from its parent, and so on. One distinction is usually made between instance variables and methods: a class may *override* any inherited method, and provide its own definition of the method; it may not alter the inherited definition of instance variables, but only extend it. This means all the methods that a class inherits can be applied to its instances, as they contain all the instance variables defined by its ancestors.

The following functions return all the instance variables and methods of a class, including those inherited from its ancestors:

$$inst\_vars : Class\_name \times Class\_map \rightarrow Id\text{-set}$$
$$inst\_vars(class\_id, class\_map) \triangleq$$
$$\quad Instvars(class\_map(class\_id))$$
$$\quad\quad \cup\ inherited\_inst\_vars(class\_id, class\_map)$$

$$inherited\_inst\_vars : Class\_name \times Class\_map \rightarrow Id\text{-set}$$
$$inherited\_inst\_vars(class\_id, class\_map) \triangleq$$
$$\quad \text{if } parent(class\_id, class\_map) = \text{nil } \text{ then } \{\}$$
$$\quad \text{else } inst\_vars(parent(class\_id, class\_map), class\_map)$$

$$all\_methods\_of : Class\_name \times Class\_map \rightarrow Method\_map$$
$$all\_methods\_of(class\_id, class\_map) \triangleq$$
$$\quad inherited\_methods(class\_id, class\_map)$$
$$\quad\quad \dagger\ Methods(class\_map(class\_id))$$

$$inherited\_methods : Class\_name \times Class\_map \rightarrow Method\_map$$
$$inherited\_methods(class\_id, class\_map) \triangleq$$
$$\quad \text{if } parent(class\_id, class\_map) = \text{nil } \text{ then } \{\}$$
$$\quad \text{else } all\_methods\_of(parent(class\_id, class\_map), class\_map)$$

Inheritance alters the meaning of classes: the meaning of a class must include the meanings of its inherited methods, and its instances include inherited instance variables, as well as any defined locally. Despite this, there is no need to alter any of the meaning functions defined in the previous chapter. Instead, the meaning of a system of classes related by inheritance can be given by "processing out" the inheritance, so that the definition of each class includes all its inherited attributes. Applying the meaning functions of the last chapter to the processed classes yields the desired meaning.

This approach is taken because:

- it shows that inheritance does not require any changes to the semantic domains or meaning functions, and
- it will allow us to compare single inheritance with the schemes for multiple inheritance described in later sections.

Factoring out inheritance is straightforward. If an element of the abstract syntax defined in the previous chapter is denoted by e, and the corresponding element in the inheritance syntax is denoted by e', then $e' = e$, except for *Programs* and *Class_bodies*. Therefore, the functions defined thus far in this chapter should all have signatures with primed types; primes will be used this way for the remainder of the chapter.

The processing of programs and classes uses the following functions:

$$PProgram : Program' \to Program$$
$$PProgram[\![p]\!] \triangleq \{class\_id \mapsto PClass[\![class\_id]\!]p \mid class\_id \in \mathsf{dom}\, p\}$$

$$PClass : Class\_name \to Class\_map' \to Class\_body$$
$$PClass[\![class\_id]\!]\, class\_map \triangleq mk\text{-}Class\_body(inst\_vars(class\_id, class\_map), all\_methods\_of(class\_id, class\_map))$$

The *PClass* function "copies down" the instance variables and methods inherited from its argument's parent using the *inst_vars* and *all_methods_of* functions.

4.1.1 Context Conditions

What restrictions must be imposed on a program so that the processing functions and, hence, meaning functions are always defined? This section states the context conditions that must hold for a program to be considered well-formed. The meaning and processing functions are undefined unless the context conditions hold, i.e.,

$$pre\text{-}PProgram[\![p]\!] \triangleq WFProgram[\![p]\!]$$

The meaning of a program is then given by applying *MProgram* to a processed program, i.e., $MProgram[\![PProgram[\![p]\!]]\!]$.

A context condition environment,[2] *CCEnv*, records the relevant parts of the surrounding context of the syntactic clause being considered. These are:

- the instance variables of the enclosing class (local and inherited),
- the local variables of the surrounding method (distinguishing parameters from temporaries), and
- the names of the classes in the program.

$$\begin{aligned} CCEnv :: \; & Instvars : Id\text{-set} \\ & Params : Id\text{-set} \\ & Temps : Id\text{-set} \\ & Classes : Class\_name\text{-set} \end{aligned}$$

For a program to be well-formed, the superclass relation must be well-founded, the superclass of every class must be defined, and each class must be well-formed:

[2] A context condition environment is referred to as a "static environment" in [BJ82].

$$
\begin{array}{l}
WFProgram : Program' \to \mathbb{B} \\
WFProgram[\![class\_map]\!] \triangleq \\
\quad non\_circular(class\_map) \\
\quad \wedge\ \forall class\_id \in \mathsf{dom}\, class\_map \cdot \\
\qquad \mathsf{let}\ inh\_iv = inherited\_inst\_vars(class\_id, class\_map)\ \mathsf{in} \\
\qquad \mathsf{let}\ \theta = mk\text{-}CCEnv(inh\_iv, \_, \_, \mathsf{dom}\, class\_map)\ \mathsf{in} \\
\qquad parent(class\_id, class\_map) \in \mathsf{dom}\, class\_map \cup \{\mathsf{nil}\} \\
\qquad \wedge\ WFClass[\![class\_map(class\_id)]\!]\,\theta
\end{array}
$$

$$
\begin{array}{l}
non\_circular : Class\_map' \to \mathbb{B} \\
non\_circular(class\_map) \triangleq \\
\quad \forall s \subseteq \mathsf{dom}\, class\_map \cdot \\
\qquad s \neq \{\} \ \Rightarrow\ \exists class\_id \in s \cdot parent(class\_id, class\_map) \notin s
\end{array}
$$

For a class to be well-formed, it must not redeclare any instance variables, and all non-primitive methods must be well-formed:

$$
\begin{array}{l}
WFClass : Class\_body' \to CCEnv \to \mathbb{B} \\
WFClass[\![mk\text{-}Class\_body'(iv, meths, \_)]\!]\,\theta \triangleq \\
\quad \mathsf{let}\ \theta' = \mu(\theta, Instvars \mapsto Instvars(\theta) \cup iv)\ \mathsf{in} \\
\quad is\_disjoint(iv, Instvars(\theta)) \\
\quad \wedge\ \forall sel \in \mathsf{dom}\, meths \cdot \\
\qquad meths(sel) \notin Primitive\_method \\
\qquad\quad \Rightarrow\ nargs(sel) = \mathsf{len}\, Params(meths(sel)) \\
\qquad\qquad \wedge\ WFMethod[\![meths(sel)]\!]\,\theta'
\end{array}
$$

$$
\begin{array}{l}
is\_disjoint : X\text{-}\mathsf{set} \times X\text{-}\mathsf{set} \to \mathbb{B} \\
is\_disjoint(s_1, s_2) \triangleq s_1 \cap s_2 = \{\}
\end{array}
$$

The *nargs* function returns the number of arguments associated with a selector. A Smalltalk-like selector has the following structure:

$$Selector = Unary \cup Binary \cup Keyword$$

$$Unary :: Id$$

$$Binary :: \{+, -, *, /, <, \ldots\}$$

$$Keyword :: Id^*$$

$$nargs : Selector \to \mathbb{N}$$
$$nargs(sel) \triangleq \textsf{cases } sel \textsf{ of}$$
$$\quad mk\text{-}Unary(\_) \to 0$$
$$\quad mk\text{-}Binary(\_) \to 1$$
$$\quad mk\text{-}Keyword(ids) \to \textsf{len } ids$$
$$\quad \textsf{end}$$

Note that a unary selector has no arguments, only a receiver, and a binary selector has one argument.

A method is well-formed if its body is well-formed, in an environment that includes the parameters and temporaries:

$$WFMethod : Method\_body' \to CCEnv \to \mathbb{B}$$
$$WFMethod[\![mk\text{-}Method\_body'(params, temps, expr)]\!]\,\theta \triangleq$$
$$\quad \textsf{let } \theta' = \mu(\mu(\theta, Params \mapsto \textsf{rng } params), Temps \mapsto temps) \textsf{ in}$$
$$\quad WFExpression[\![expr]\!]\,\theta'$$

Now the context conditions for expressions are enumerated.

$$WFExpression : Expression' \to CCEnv \to \mathbb{B}$$
$$WFExpression[\![mk\text{-}Expression\_list'(exprs)]\!]\,\theta \triangleq$$
$$\quad \textsf{len } exprs \geq 1 \wedge \forall e \in \textsf{rng } exprs \cdot WFExpression[\![e]\!]\,\theta$$

Parameters are assumed to be constant within a method, and cannot be assigned to. Hence an assignment expression is well-formed if it assigns to a declared temporary or instance variable, and the assigned expression is well-formed:

$$
\begin{array}{l}
WFExpression[\![mk\text{-}Assignment'(id, rhs)]\!]\,\theta \triangleq \\
\quad WFExpression[\![rhs]\!]\,\theta \wedge \\
\quad \textsf{cases } id \textsf{ of} \\
\quad mk\text{-}Inst\_var\_id(iv) \rightarrow iv \in Instvars(\theta) \\
\quad mk\text{-}Temp\_id(t) \quad \rightarrow t \in Temps(\theta) \\
\quad \textsf{end}
\end{array}
$$

All applied occurrences of identifiers must have matching defining occurrences:

$$WFExpression[\![mk\text{-}Inst\_var\_id(id)]\!]\,\theta \triangleq id \in Instvars(\theta)$$

$$WFExpression[\![mk\text{-}Temp\_id(id)]\!]\,\theta \triangleq id \in Temps(\theta)$$

$$WFExpression[\![mk\text{-}Arg\_id(id)]\!]\,\theta \triangleq id \in Params(\theta)$$

$$WFExpression[\![\textsc{Self}]\!]\,\theta \triangleq \textsf{true}$$

The number of arguments to a message must match the number in the selector (this becomes a dynamic check if the selector does not contain argument information [see §3.10.3]), and the receiver and argument expressions must be well-formed:

$$
\begin{array}{l}
WFExpression[\![mk\text{-}Message'(rcvr, sel, args)]\!]\,\theta \triangleq \\
\quad WFExpression[\![rcvr]\!]\,\theta \\
\quad \wedge\ \textsf{len}\, args = nargs(sel) \\
\quad \wedge\ \forall arg \in \textsf{rng}\, args \cdot WFExpression[\![arg]\!]\,\theta
\end{array}
$$

A new-expression must refer to a valid non-primitive class:

$$
\begin{array}{l}
WFExpression[\![mk\text{-}New\_object(class)]\!]\,\theta \triangleq \\
\quad class \in Classes(\theta) - Primitive\_classes
\end{array}
$$

$$Primitive\_classes = \{\mathsf{Integer}, \mathsf{Symbol}, \ldots\}$$

All literals are well-formed:

$$WFExpression[\![l]\!]\,\theta \triangleq \mathsf{true} \qquad \text{for } l \in Int\_literal \cup \ldots$$

4.2 Static Binding of Messages

A drawback of using the inheritance scheme defined above is that any method that has been overridden in a subclass is inaccessible to instances of that subclass. This is unfortunate because it can be useful to override a method purely to augment its behaviour (e.g., a subclass may want to initialise inherited variables using a method defined in its parent, and then perform extra initialisation on its local variables), but the current scheme bars access to overridden methods.

To circumvent this problem, many object-oriented languages have a mechanism for accessing inherited, overridden methods. In a Smalltalk method, for example, the distinguished identifier super refers to the same object as self, but messages sent to super are bound to methods in the superclass of the method from which the message is sent. An important property of the super mechanism is that overridden methods are still inaccessible to objects other than self.

To model superclass sends formally, we first extend the syntax:

$$\begin{aligned} Expression' = {} & Expression\_list' \cup Assignment' \cup Object\_name \\ & \cup Message' \cup New\_object \cup Literal\_object \cup Super\_send \end{aligned}$$

$$\begin{aligned} Super\_send :: {} & Sel : Selector \\ & Args : Expression'^{*} \end{aligned}$$

Note that the receiver is implicitly the object denoted by self.

Because the graph of classes is static, the method invoked in response to a superclass send can be determined statically. This is done at the pre-processing stage, when inheritance is factored out of the abstract syntax. A new kind of message-send expression, statically binding a message to a method, is used in the processed abstract syntax:

$$\begin{array}{lrcl} Static\_send & :: & Sel & : & Selector \\ & & Args & : & Expression^* \\ & & Class & : & Class\_name \end{array}$$

$$Expression = Expression\_list \cup Assignment \cup Object\_name \cup Message \cup New\_object \cup Literal\_object \cup Static\_send$$

A *Static_send* is equivalent to a conventional procedure call.

Now that the domains of expressions before and after processing are different, i.e., $Expression' \neq Expression$, extra processing functions are required. First, the *all_methods_of* function changes so that the other processing functions are invoked (cf. p. 72):

$$all\_methods\_of : Class\_name \times Class\_map' \rightarrow Method\_map$$
$$all\_methods\_of(c, class\_map) \triangleq inherited\_methods(c, class\_map) \dagger PMethods[\![Methods(class\_map(c))]\!]parent(c, class\_map)$$

The *PMethods* function processes a method body, which in turn involves processing the constituent expressions. The functions that process methods and expressions take the superclass of the current method as an argument so that they can determine which method is invoked in response to a *Static_send*.

$$Method\_map' = Selector \xrightarrow{m} Method\_desc'$$

$$Method\_desc' = Method\_body' \cup Primitive\_method$$

$$Method\_map = Selector \xrightarrow{m} Method\_desc$$

$$Method\_desc = Method\_body \cup Primitive\_method$$

$$PMethods : Method\_map' \rightarrow Class\_name \rightarrow Method\_map$$
$$PMethods[\![meths]\!]parent \triangleq \{sel \mapsto PMethod[\![meths(sel)]\!]parent \mid sel \in \mathsf{dom}\, meths\}$$

$PMethod : Method\_desc' \rightarrow Class\_name \rightarrow Method\_desc$

$PMethod\,[\![m]\!]\,parent \triangleq$

if $m \in Primitive\_method$

then m

else let $mk\text{-}Method\_body'(params, temps, expr) = m$ in

$mk\text{-}Method\_body(params, temps,$

$PExpression\,[\![expr]\!]\,parent)$

$PExpression : Expression' \rightarrow Class\_name \rightarrow Expression$

$PExpression\,[\![expr]\!]\,parent \triangleq$

cases $expr$ of

$mk\text{-}Super\_send(sel, args) \rightarrow mk\text{-}Static\_send(sel,$

$PExpression\_list\,[\![args]\!]\,parent,$

$parent)$

$mk\text{-}Assignment'(l, r) \rightarrow mk\text{-}Assignment(l,$

$PExpression\,[\![r]\!]\,parent)$

$mk\text{-}Message'(r, s, a) \rightarrow mk\text{-}Message($

$PExpression\,[\![r]\!]\,parent,$

$s, PExpression\_list\,[\![a]\!]\,parent)$

…

end

(The arms of the cases expression that simply re-tag expressions have been elided.)

A new meaning function is required for statically-bound messages:

$MExpression : Expression \rightarrow SEnv \rightarrow$

$DEnv \rightarrow Object\_memory \rightarrow$

$Oop \times DEnv \times Object\_memory$

$MExpression\,[\![mk\text{-}Static\_send(sel, args, class)]\!]\,\rho\delta\sigma \triangleq$

let $(actuals, \delta', \sigma') = MExpression\_list\,[\![args]\!]\,\rho\delta\sigma$ in

let $(result, \sigma'')$

$= method(sel, class, PD(\rho))(Rcvr(\delta'), actuals, \sigma')$ in

$(result, \delta', \sigma'')$

Note that no check is required to see whether the message is understood; this is guaranteed by the context conditions (given below).

A field is added to the context condition environment, recording the selectors of methods inherited by a class, and this is used to check that *Super_sends* are well-formed (cf. p. 74).

$$
\begin{array}{rrcl}
\mathit{CCEnv} :: & \mathit{Instvars} & : & \mathit{Id}\text{-set} \\
& \mathit{Params} & : & \mathit{Id}\text{-set} \\
& \mathit{Temps} & : & \mathit{Id}\text{-set} \\
& \mathit{Classes} & : & \mathit{Class\_name}\text{-set} \\
& \mathit{Inherited\_selectors} & : & \mathit{Selector}\text{-set}
\end{array}
$$

$$\mathit{WFProgram} : \mathit{Program}' \to \mathbb{B}$$

$$
\begin{array}{l}
\mathit{WFProgram}\,[\![\mathit{class\_map}]\!] \quad \triangleq \\
\quad \mathit{non\_circular}(\mathit{class\_map}) \wedge \\
\quad \forall \mathit{class\_id} \in \mathsf{dom}\,\mathit{class\_map} \cdot \\
\qquad \mathsf{let}\ p = \mathit{parent}(\mathit{class\_id}, \mathit{class\_map})\ \mathsf{in} \\
\qquad \mathsf{let}\ \mathit{inh\_iv} = \mathit{inherited\_inst\_vars}(\mathit{class\_id}, \mathit{class\_map})\ \mathsf{in} \\
\qquad \mathsf{let}\ \mathit{inh\_sel} = \mathsf{if}\ p = \mathsf{nil}\ \mathsf{then}\ \{\}\ \mathsf{else}\ \mathit{selectors}(p, \mathit{class\_map})\ \mathsf{in} \\
\qquad \mathsf{let}\ \theta = \mathit{mk\text{-}CCEnv}(\mathit{inh\_iv}, \_, \_, \mathsf{dom}\,\mathit{class\_map}, \mathit{inh\_sel})\ \mathsf{in} \\
\qquad p \in \mathsf{dom}\,\mathit{class\_map} \cup \{\mathsf{nil}\} \\
\qquad \wedge\ \mathit{WFClass}\,[\![\mathit{class\_map}(\mathit{class\_id})]\!]\,\theta
\end{array}
$$

$$\mathit{selectors} : \mathit{Class\_name} \times \mathit{Class\_map}' \to \mathit{Selector}\text{-set}$$

$$
\begin{array}{l}
\mathit{selectors}(\mathit{class\_id}, \mathit{class\_map}) \quad \triangleq \\
\quad \mathsf{dom}\,\mathit{all\_methods\_of}(\mathit{class\_id}, \mathit{class\_map})
\end{array}
$$

$$
\begin{array}{l}
\mathit{WFExpression}\,[\![\mathit{mk\text{-}Super\_send}(\mathit{sel}, \mathit{args})]\!]\,\theta \quad \triangleq \\
\quad \mathsf{len}\,\mathit{args} = \mathit{nargs}(\mathit{sel}) \\
\quad \wedge\ \mathit{sel} \in \mathit{Inherited\_selectors}(\theta) \\
\quad \wedge\ \forall \mathit{arg} \in \mathsf{rng}\,\mathit{args} \cdot \mathit{WFExpression}\,[\![\mathit{arg}]\!]\,\theta
\end{array}
$$

4.3 Multiple Inheritance

In a language with multiple inheritance, a class may have any number of parent classes. The child class inherits all the methods and instance variables of all of its parents. The main distinctions between different multiple inheritance schemes are:

- how name conflicts in inherited instance variables are handled, and
- how conflicting inherited methods are handled.

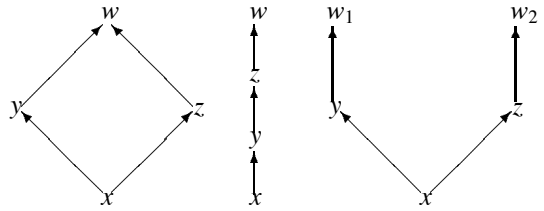
These cases are treated differently because instance variables are private to objects, whereas methods are public. Another distinction arises when an attribute is inherited via different routes from the same class. Both instance variables and methods are deemed to be conflicting if their names (selectors, for methods) conflict; in general one cannot determine if two methods are the same (have the same meaning).

Three basic types of multiple inheritance scheme are described here. In the first, the graph structure formed by the superclass relation is preserved: attributes inherited from the same node via different paths are not considered to be different, and so do not conflict (see Fig. 4.1(a)). When attempting to resolve a conflict in the second scheme, the graph of ancestors of a class is linearised. The resulting total order is used to select amongst the conflicting attributes; no user intervention is required (see Fig. 4.1(b)). The third inheritance scheme "unfolds" the graph from a particular class into a tree, so that there is no merging of inherited attributes. It differs from the first form of inheritance only when an attribute is inherited via different paths (see Fig. 4.1(c)). Snyder has named these schemes graph, linear and tree inheritance [Sny86b].

All three schemes require the following change to the abstract syntax (cf. p. 71):

$$\begin{array}{lrcl} \mathit{Class\_body'} & :: & \mathit{Instvars} & : & \mathit{Id}\text{-set} \\ & & \mathit{Methods} & : & \mathit{Method\_map'} \\ & & \mathit{Parents} & : & \mathit{Parent\_classes} \end{array}$$

$$\mathit{parents} : \mathit{Class\_name} \times \mathit{Class\_map'} \rightarrow \mathit{Parent\_classes}$$
$$\mathit{parents}(\mathit{class}, \mathit{class\_map}) \triangleq \mathit{Parents}(\mathit{class\_map}(\mathit{class}))$$



The child→parent relation for:
(a) Graph inheritance (b) Linear inheritance (c) Tree inheritance

Figure 4.1: Three different inheritance schemes

For the graph and tree inheritance schemes, *Parent_classes* = *Class_name*-set, whereas for the linear inheritance scheme *Parent_classes* = *Ulist*(*Class_name*).[3]

Additionally, the abstract syntax for a *Super_send* changes so that the programmer chooses the class referred to by super (cf. p. 78):

Super_send :: *Sel* : *Selector*
Args : *Expression'**
Parent : *Class_name*

This simplifies pre-processing since *Super_sends* and *Static_sends* now have the same structure.

4.3.1 Graph Inheritance

In graph inheritance, the user must resolve conflicts between inherited methods by writing a new method. For example, in Fig. 4.1(a), if both classes *y* and *z* define a method *m*, then class *x* must override those definitions with a definition of its own. One thing the new method might do is invoke one or more of the conflicting methods using a *Super_send*.

As regards conflicting inherited instance variables, these are not allowed. If both classes *y* and *z* define an instance variable *i*, then the inheritance graph

[3] See p. 33 for a definition of the *Ulist* constructor.

of Fig. 4.1(a) is illegal. Also, if they inherit the same instance variable from different classes (e.g., from $w1$ and $w2$ in Fig. 4.1(c)), then the graph is also illegal. But an instance variable inherited from the same class via multiple routes is not considered to be in conflict (e.g., an instance variable defined in class w in Fig. 4.1(a)). This is a major failing of graph inheritance; that private features of classes become visible when the classes are composed using inheritance (see §4.4). Graph inheritance is used in "extended" Smalltalk-80 [BI82].

Changes to Context Conditions

The context condition environment is changed to record the classes that provide inherited selectors (cf. pp. 74, 81):

$$
\begin{array}{rrcl}
CCEnv :: & Instvars & : & Id\text{-set} \\
& Params & : & Id\text{-set} \\
& Temps & : & Id\text{-set} \\
& Classes & : & Class\_name\text{-set} \\
& Inherited\_selectors & : & Class\_name \xrightarrow{m} Selector\text{-set}
\end{array}
$$

The changes to the context conditions to reflect the new restrictions are as follows (cf. pp. 72, 75):

$$
\begin{array}{l}
WFProgram : Program' \to \mathbb{B} \\
WFProgram\,[\![class\_map]\!] \triangleq \\
\quad non\_circular(class\_map) \\
\quad \wedge\ \forall class\_id \in \mathsf{dom}\, class\_map \cdot \\
\qquad \mathsf{let}\ par = parents(class\_id, class\_map)\ \mathsf{in} \\
\qquad \mathsf{let}\ anc = ancestors(class\_id, class\_map)\ \mathsf{in} \\
\qquad \mathsf{let}\ inh\_iv = inherited\_inst\_vars(class\_id, class\_map)\ \mathsf{in} \\
\qquad \mathsf{let}\ \theta = mk\text{-}CCEnv(inh\_iv, \_, \_, \mathsf{dom}\, class\_map, \\
\qquad\qquad \{p \mapsto selectors(p, class\_map) \mid p \in par\})\ \mathsf{in} \\
\qquad par \subset \mathsf{dom}\, class\_map \\
\qquad \wedge\ non\_conflicting\_instvars(anc, class\_map) \\
\qquad \wedge\ non\_conflicting\_methods(class\_id, par, class\_map) \\
\qquad \wedge\ WFClass\,[\![class\_map(class\_id)]\!]\,\theta
\end{array}
$$

$$ancestors : Class\_name \times Class\_map' \to Class\_name\text{-set}$$
$$ancestors(class, class\_map) \quad \triangleq$$
$$\quad \text{let } p = parents(class, class\_map) \text{ in}$$
$$\quad p \cup \left(\bigcup\{ancestors(c, class\_map) \mid c \in p\}\right)$$

$$inherited\_inst\_vars : Class\_name \times Class\_map' \to Id\text{-set}$$
$$inherited\_inst\_vars(class, class\_map) \quad \triangleq$$
$$\quad \bigcup\{inst\_vars(c, class\_map) \mid c \in parents(class, class\_map)\}$$

The *non_circular* function changes because the previous, simpler version could not cope with relations (cf. p. 75):

$$non\_circular : Class\_map' \to \mathbb{B}$$
$$non\_circular(class\_map) \quad \triangleq$$
$$\quad \forall classes \subseteq \text{dom}\, class\_map \cdot$$
$$\quad\quad classes \neq \{\,\} \Rightarrow$$
$$\quad\quad\quad \exists c_1 \in classes \cdot$$
$$\quad\quad\quad\quad \forall c_2 \in classes \cdot c_2 \notin parents(c_1, class\_map)$$

The instance variables of a set of classes do not conflict iff they are all different:

$$non\_conflicting\_instvars : Class\_name\text{-set} \times Class\_map' \to \mathbb{B}$$
$$non\_conflicting\_instvars(classes, class\_map) \quad \triangleq$$
$$\quad \forall c_1, c_2 \in classes \cdot$$
$$\quad\quad c_1 \neq c_2 \Rightarrow is\_disjoint(Instvars(class\_map(c_1)),$$
$$\quad\quad\quad Instvars(class\_map(c_2)))$$

Two methods conflict when their names are the same and they are not inherited from the same place; the conflict can be resolved by providing a new, overriding method:

$$non\_conflicting\_methods : Class\_name \times Class\_name\text{-set} \times Class\_map' \to \mathbb{B}$$

$$non\_conflicting\_methods(c, parents, class\_map) \triangleq \forall sel \in conflicting\_selectors(parents, class\_map) \cdot sel \in \mathsf{dom}\, Methods(class\_map(c))$$

The test for conflict between a selector inherited from two different classes involves comparing the defining class of the selector: if the selector has been inherited from different classes, then there is a conflict. To do this, the *sel_to_class* function builds a map from all the selectors in a class to the *Class_name* in which each was defined.

$$conflicting\_selectors : Class\_name\text{-set} \times Class\_map' \to Selector\text{-set}$$

$$conflicting\_selectors(classes, class\_map) \triangleq \mathsf{let}\ sel\_set = \bigcup\{all\_selectors(c, class\_map) \mid c \in classes\}\ \mathsf{in}\ \{sel \in sel\_set \mid \exists c_1, c_2 \in classes \cdot sel\_to\_class(c_1, class\_map)(sel) \neq sel\_to\_class(c_2, class\_map)(sel)\}$$

$$all\_selectors : Class\_name \times Class\_map' \to Selector\text{-set}$$

$$all\_selectors(class, class\_map) \triangleq \bigcup\{all\_selectors(p, class\_map) \mid p \in parents(class, class\_map)\} \cup \mathsf{dom}\, Methods(class\_map(class))$$

$$sel\_to\_class : Class\_name \times Class\_map' \to Selector \xrightarrow{m} Class\_name$$

$$sel\_to\_class(class, class\_map) \triangleq \biguplus\{sel\_to\_class(p, class\_map) \mid p \in parents(class, class\_map)\} \dagger \{sel \mapsto class \mid sel \in \mathsf{dom}\, Methods(class\_map(class))\}$$

The special union symbol, $\uplus$, and its distributed form, $\biguplus$, signifies map union with overlapping elements:

$$\_ \uplus \_ : X \xrightarrow{m} Y \times X \xrightarrow{m} Y \to X \xrightarrow{m} Y$$
$$a \uplus b \triangleq (\mathsf{dom}\, b \lhd a) \cup b \qquad \text{iff} \quad \forall x \in (\mathsf{dom}\, a \cap \mathsf{dom}\, b) \cdot a(x) = b(x)$$

This form of "loose" map union is used because classes may inherit the same method via different routes.

One additional change is required, to check that a genuine parent is being used (cf. p. 81):

$$\begin{aligned}
&WFExpression\,[\![mk\text{-}Super_send(sel, args, parent)]\!]\,\theta \quad \triangleq \\
&\quad \mathsf{len}\, args = nargs(sel) \wedge \\
&\quad parent \in \mathsf{dom}\, Inherited_selectors(\theta) \wedge \\
&\quad sel \in Inherited_selectors(\theta)(parent) \wedge \\
&\quad \forall arg \in \mathsf{rng}\, args \cdot WFExpression\,[\![arg]\!]\,\theta
\end{aligned}$$

Changes to Pre-processing

The changes to the pre-processing functions required for *Super_sends* (p. 80) are no longer required, as *Super_sends* must now explicitly name a class. Therefore the pre-processing functions of §4.1 are used. The only necessary change is to *inherited_methods*, and is straightforward (cf. p. 72):

$$\begin{aligned}
&inherited_methods : Class_name \times Class_map' \to Method_map \\
&inherited_methods(class_id, class_map) \quad \triangleq \\
&\quad \biguplus \{all_methods_of(p, class_map) \\
&\qquad\qquad \mid p \in parents(class_id, class_map)\}
\end{aligned}$$

Having ensured that class definitions are restricted in such a way that conflicts do not exist, and that all inheritance has been "processed out", no changes to the meaning functions of Chapter 3 and §4.2 are required.

4.3.2 Linear Inheritance

In contrast to graph inheritance, where the user must resolve conflicts between inherited methods, and ensure there are no conflicts between inherited instance

variables, linear inheritance resolves all conflicts automatically. It does this by creating a total order of a class and its ancestors (known as a *precedence list*), and using the order to select between conflicting methods. Moreover, most systems place a restriction on possible orderings so that some inheritance graphs are disallowed. In the Lisp-based object-oriented systems (e.g., Flavors [Moo86] and CLOS [DG87]), the parents of a class are ordered by the user, and the resulting total order of ancestors must obey the following rules:

- a class must always precede its parents, and
- the user's ordering of a class's parents must be preserved within the precedence list.

If, for example, class A has parents $[X, Y]$, and class B has parents $[Y, X]$, then A and B can have no common offspring.

To model this formally, *Parent_classes* is changed thus:

$$Parent\_classes = Ulist(Class\_name)$$

$$parents : Class\_name \times Class\_map' \rightarrow Class\_name\text{-set}$$
$$parents(class, class\_map) \triangleq \mathsf{rng}\, Parents(class\_map(class))$$

Many different algorithms can be used to create an ordering that satisfies the above rules. Instead of listing different algorithms, a specification is given:

$$\begin{aligned} &ordering\ (c: Class\_name, \\ &\qquad class\_map: Class\_map)\ prec\_list: Ulist(Class\_name) \\ &\mathsf{post}\ \{c\} \cup ancestors(c, class\_map) = \mathsf{rng}\, prec\_list \wedge \\ &\quad \forall class \in \mathsf{rng}\, prec\_list \cdot \\ &\qquad is\_subseq([class]^\frown Parents(class\_map(class)), prec\_list) \end{aligned}$$

The *Ulist* type defined earlier models a total order. The *is_subseq* function describes the rules about ordering of parent classes:

$$
\begin{array}{l}
is\_subseq : X^* \times X^* \to \mathbb{B} \\
is\_subseq(a, b) \triangleq \\
\quad \exists m \in \mathbb{N}_1 \xrightarrow{m} \mathbb{N}_1 \cdot \\
\quad\quad \mathsf{dom}\, m = \mathsf{dom}\, a \wedge \mathsf{rng}\, m \subseteq \mathsf{dom}\, b \wedge \\
\quad\quad \forall i, j \in \mathsf{dom}\, m \cdot \\
\quad\quad\quad a(i) = b(m(i)) \wedge a(j) = b(m(j)) \\
\quad\quad\quad \wedge\ (i < j \iff m(i) < m(j))
\end{array}
$$

Given an ordering function that satisfies the above specification, the context conditions for a program can be defined. The only change from the graph-inheritance context conditions is in *WFProgram*, where the restrictions on methods and instance variables can be relaxed (cf. p. 84):

$$
\begin{array}{l}
WFProgram : Program' \to \mathbb{B} \\
WFProgram[\![class\_map]\!] \triangleq \\
\quad non\_circular(class\_map) \\
\quad \wedge\ \forall class\_id \in \mathsf{dom}\, class\_map \cdot \\
\quad\quad \mathsf{let}\ parents = parents(class\_id, class\_map)\ \mathsf{in} \\
\quad\quad \mathsf{let}\ inh\_iv = inherited\_inst\_vars(class\_id, class\_map)\ \mathsf{in} \\
\quad\quad \mathsf{let}\ \theta = mk\text{-}CCEnv(inh\_iv, \_, \_, \mathsf{dom}\, class\_map, \\
\quad\quad\quad\quad \{p \mapsto selectors(s, class\_map) \mid p \in parents\})\ \mathsf{in} \\
\quad\quad \exists prec\_list \in Ulist(Class\_name) \cdot \\
\quad\quad\quad post\text{-}ordering(class\_id, class\_map, prec\_list) \\
\quad\quad \wedge\ parents \subset \mathsf{dom}\, class\_map \\
\quad\quad \wedge\ WFClass[\![class\_map(class\_id)]\!]\, \theta
\end{array}
$$

Note that in the linear inheritance systems used in Flavors (and its cousins) there can be no conflict in inherited instance variables: variables with the same name are considered the same, even if they are inherited from different classes.

To process a linear multiple-inheritance program into a program without inheritance, only a few of the auxiliary functions used for processing graph-inheritance programs need be changed. The *all_methods_of* function returns the set of all methods defined on a class, using the *ordering* function to resolve conflicts (cf. p. 79):

$$all\_methods\_of : Class\_name \times Class\_map' \rightarrow Method\_map$$
$$all\_methods\_of(class\_id, class\_map) \triangleq$$
$$\quad methods\_from(ordering(class\_id, class\_map), class\_map)$$

$$methods\_from : Ulist(Class\_name) \times Class\_map' \rightarrow Method\_map$$
$$methods\_from(class\_list, class\_map) \triangleq$$
$$\quad \text{if } class\_list = [\,] \text{ then } \{\,\}$$
$$\quad \text{else } methods\_from(\text{tl } class\_list, class\_map)$$
$$\qquad\qquad \dagger\ Methods(class\_map(\text{hd } class\_list))$$

Static Binding in Linear Inheritance Systems

Although the extended form of *Super_send* used in graph inheritance could be adopted in linear inheritance systems, several Lisp-based object-oriented languages (e.g., CommonLoops and CLOS [BKK+86, DG87]) choose to use the simpler version described in §4.2. In these systems super refers to the succeeding class in the precedence list. This need not be a parent class at all: e.g., in Fig. 4.1(b), the precedence list of class x is $[x, y, z, w]$, so that a method defined in class y and inherited by class x has class z as its successor, although z is not an ancestor of y. The use of super in this method will invoke a method in z (if there is one), rather than one in w. Therefore, the meaning of the inherited method is dependent on the part of the precedence list that follows, which may contain arbitrary unrelated classes. Clearly, this can lead to confusion and error, but is in keeping with the almost arbitrary way in which attributes can be merged and ordered in these systems.

Some of these languages go so far as to make this into a virtue by allowing the user to define her own combination techniques for inherited methods based on the precedence list (e.g., invoke methods in reverse order of precedence) [Moo86, DG87]. As Snyder has argued, this leads to classes being heavily dependent on the implementation details of their parents [Sny86b, Sny87]. The next section describes an inheritance scheme that discourages such dependencies.

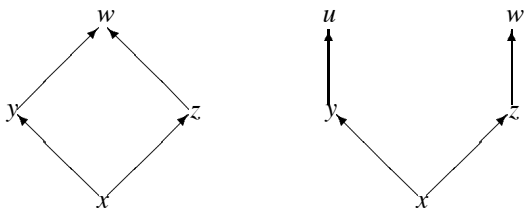
4.3.3 Tree Inheritance

Snyder has argued that both graph and linear inheritance schemes are deficient with respect to the aims of good software engineering practice [Sny86b, Sny86a, Sny87]. His view (and it is one that the author agrees with) is that because classes are the units of modularity in an object-oriented system, the internal details of a class should not be visible to its subclasses. In effect, the external interface of a class is all that is guaranteed constant by the class's designer, and a language should not promote styles of subclassing that depend on the internal details of an ancestor's implementation.

The most obvious way that linear and graph inheritance fail in hiding the internals of a class from its children is that children have direct access to the inherited instance variables. A consequence of this is that a child may suffer from name conflicts when inheriting from two unrelated classes, because they both use the same instance variable name. The linear inheritance solution to this, of making all the names refer to the same variable, suffers from the usual problems of aliasing encountered in conventional languages. In general, it is unlikely that the designers of the classes intentionally chose to use the same name; unintended conflicts should not result in unusual behaviour (as in linear inheritance) or prohibit subclassing (as in graph inheritance).

Another problem with graph inheritance occurs when a class inherits an instance variable via two or more routes: the behaviour of the child class can be affected if the implementor of a parent class decides to change the implementation of that class by inheriting from a different class. For example, in Fig. 4.2(a), if the implementor of class *y* changes its implementation so that it inherits instead from a class *u* (Fig. 4.2(b)) that has an instance variable with the same name as its original parent *w*, the child *x* will either (a) become illegal, because of an instance variable name clash, or (b) have different behaviour, because classes *u* and *w*, though created independently, share the instance variable.

These problems have led Snyder to propose the tree inheritance scheme. In tree inheritance, a class cannot see the implementation details of its parents. In particular, it cannot know which classes they inherit from. To achieve this, a class does not have direct access to inherited instance variables; an instance must send messages to itself to access them. Secondly, instance variable names that are inherited from different classes name different instance variables, even



(a) Before (b) After

A change in the inheritance graph of y causes an unexpected change in the behaviour of x in the graph and linear inheritance schemes.

Figure 4.2: Problems with graph and linear inheritance

if they were originally inherited from a common class.

No abstract syntax changes are required from the graph inheritance model, but the context conditions now restrict access to inherited instance variables (cf. pp. 75, 84):

$$
\begin{array}{l}
WFProgram : Program' \to \mathbb{B} \\
WFProgram\,[\![class\_map]\!] \quad \triangleq \\
\quad non\_circular(class\_map) \\
\quad \wedge\ \forall class\_id \in \mathsf{dom}\, class\_map \cdot \\
\qquad \mathsf{let}\ par = parents(class\_id, class\_map)\ \ \mathsf{in} \\
\qquad \mathsf{let}\ \theta = mk\text{-}CCEnv(\_, \_, \_, \mathsf{dom}\, class\_map, \\
\qquad\qquad \{p \mapsto selectors(p, class\_map) \mid p \in par\})\ \ \mathsf{in} \\
\qquad par \subset \mathsf{dom}\, class\_map \\
\qquad \wedge\ non\_conflicting\_methods(class\_id, par, class\_map) \\
\qquad \wedge\ WFClass\,[\![class\_map(class\_id)]\!]\,\theta
\end{array}
$$

$$
\begin{array}{l}
WFClass : Class\_body' \to CCEnv \to \mathbb{B} \\
WFClass[\![mk\text{-}Class\_body'(iv, meths, \_)]\!]\,\theta \quad \triangleq \\
\quad \forall sel \in \mathsf{dom}\, meths \cdot \\
\quad\quad meths(sel) \notin Primitive\_method \\
\quad\quad\quad \Rightarrow\ nargs(sel) = \mathsf{len}\, Params(meths(sel)) \\
\quad\quad\quad\quad \wedge\ WFMethod[\![meths(sel)]\!]\,\mu(\theta, Instvars \mapsto iv)
\end{array}
$$

Tree inheritance uses a more restrictive notion of conflict between inherited selectors than graph inheritance. A selector inherited from different classes causes a conflict, regardless of where its associated method was defined (cf. p. 86):

$$
\begin{array}{l}
conflicting\_selectors : Class\_name\text{-}\mathsf{set} \times Class\_map' \to \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Selector\text{-}\mathsf{set} \\
conflicting\_selectors(classes, class\_map) \quad \triangleq \\
\quad \mathsf{let}\ sel\_set = \bigcup\{all\_selectors(c, class\_map) \mid c \in classes\}\ \ \mathsf{in} \\
\quad \{sel \in sel\_set \mid \\
\quad\quad \exists c_1, c_2 \in classes \cdot \\
\quad\quad\quad c_1 \neq c_2 \wedge sel \in all\_selectors(c_1, class\_map) \\
\quad\quad\quad\quad \wedge\ sel \in all\_selectors(c_2, class\_map)\}
\end{array}
$$

Whereas no changes in meaning functions are required for single inheritance and both graph and linear inheritance, some modification is required with tree inheritance. This is because the structure of an object now resembles the structure of the classes from which it is derived.

Changes to Semantic Domains

The structure of an object is that of a tree, isomorphic to the tree of its constituent classes. At each node of the tree are the instance variables defined by the corresponding class (cf. p. 27):

$$
\begin{array}{lll}
Plain\_object & :: & Local\_instvars\ :\ Id \xrightarrow{m} Oop \\
 & & Inherited\ :\ Class\_name \xrightarrow{m} Plain\_object
\end{array}
$$

During the invocation of an inherited method, only a part of the receiver is in scope; the path taken to find the inherited method is the same as the path taken to find the part of the object, and is a sequence of class names:

$Access\_path = Ulist(Class\_name)$

New auxiliary functions are required to access and update instance variables; they use the current path (part of the static environment) to determine the sub-object of the current receiver that contains the instance variables (cf. pp. 36, 37):

$$\begin{aligned} SEnv \;::\; Instvars &: Class\_name \xrightarrow{m} Instvar\_desc \\ PD &: Program\_den \\ Class &: Access\_path \end{aligned}$$

$$\begin{aligned} Instvar\_desc \;::\; Local\_instvars &: Id\text{-set} \\ Inherited &: Class\_name \xrightarrow{m} Instvar\_desc \end{aligned}$$

$$inst\_var : Id \times Oop \times Access\_path \times Object\_memory \to Oop$$
$$inst\_var(id, oop, path, \sigma) \triangleq$$
$$\quad Local\_instvars(sub\_body(oop, path, \sigma))(id)$$

$$sub\_body : Oop \times Access\_path \times Object\_memory \to Plain\_object$$
$$sub\_body(oop, path, \sigma) \triangleq sub\_obj(body(oop, \sigma), path)$$

$$sub\_obj : Plain\_object \times Access\_path \to Plain\_object$$
$$sub\_obj(pobj, path) \triangleq$$
$$\quad \text{if } path = [\,] \text{ then } pobj$$
$$\quad \text{else } sub\_obj(Inherited(pobj)(\text{hd}\, path), \text{tl}\, path)$$

$$update\_inst\_var : Id \times Oop \times Oop \times Access\_path \times Object\_memory \to Object\_memory$$
$$update\_inst\_var(id, oop, value, path, \sigma) \triangleq$$
$$\quad \sigma \dagger \{oop \mapsto \mu(\sigma(oop), Body \mapsto update\_body(body(oop, \sigma), id, value, path))\}$$

$$update\_body : Plain\_object \times Id \times Oop \times Access\_path \rightarrow Plain\_object$$

$$update\_body(pobj, id, value, path) \triangleq$$
if $path = [\,]$
then $\mu(pobj, Local\_instvars \mapsto Local\_instvars(pobj) \dagger \{id \mapsto value\})$
else let $inh = Inherited(pobj)$ in
let $new\_part =$
$update\_body(inh(\mathsf{hd}\, path), id, value, \mathsf{tl}\, path)$ in
$\mu(pobj, Inherited \mapsto inh \dagger \{\mathsf{hd}\, path \mapsto new\_part\})$

Changes to Pre-processing

As mentioned, pre-processing cannot now eliminate inheritance, but is used to tag each inherited method with the path by which it was inherited. So, the abstract syntax after processing is like this:

$$Class\_body :: Instvars : Instvar\_desc$$
$$Methods : Method\_map$$

$$Method\_map = Selector \xrightarrow{m} Method\_desc$$

$$Method\_desc :: Original\_class : Access\_path$$
$$Method : Method\_body \cup Primitive\_method$$

To tag the methods, the following auxiliary functions are changed (cf. pp. 72, 79, 87):

$$inst\_vars : Class\_name \times Class\_map' \rightarrow Instvar\_desc$$
$$inst\_vars(class\_id, class\_map) \triangleq$$
$$mk\text{-}Instvar\_desc(Instvars(class\_map(class\_id)),$$
$$\{p \mapsto inst\_vars(p, class\_map) \mid$$
$$p \in parents(class\_id, class\_map)\})$$

$$
\begin{array}{l}
\mathit{all\_methods\_of} : \mathit{Class\_name} \times \mathit{Class\_map}' \to \mathit{Method\_map} \\
\mathit{all\_methods\_of}(\mathit{class\_id}, \mathit{class\_map}) \quad \triangleq \\
\qquad \mathsf{let}\ \mathit{meths} = \mathit{Methods}(\mathit{class\_map}(\mathit{class\_id}))\ \mathsf{in} \\
\qquad \mathit{inherited\_methods}(\mathit{class\_id}, \mathit{class\_map}) \\
\qquad\ \dagger \{\mathit{sel} \mapsto \mathit{mk\text{-}Method\_desc}([\,], \mathit{meths}(\mathit{sel})) \mid \mathit{sel} \in \mathsf{dom}\, \mathit{meths}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{inherited\_methods} : \mathit{Class\_name} \times \mathit{Class\_map}' \to \mathit{Method\_map} \\
\mathit{inherited\_methods}(\mathit{class\_id}, \mathit{class\_map}) \quad \triangleq \\
\qquad \circledast\!\!\!\biguplus\{\mathit{methods\_inherited\_from}(p, \mathit{class\_map}) \\
\qquad\qquad\qquad\qquad \mid p \in \mathit{parents}(\mathit{class\_id}, \mathit{class\_map})\}
\end{array}
$$

Whereas the definition of *inherited_methods* for graph inheritance was based on the “loose” map union $\uplus$, the definition for tree inheritance explicitly excludes any intersection in the maps:

$$
\begin{array}{l}
\_\circledast\!\!\!\uplus\_ : X \xrightarrow{m} Y \times X \xrightarrow{m} Y \to X \xrightarrow{m} Y \\
a \circledast\!\!\!\uplus b \triangleq (\mathsf{dom}\, b \mathbin{⩤} a) \cup (\mathsf{dom}\, a \mathbin{⩤} b)
\end{array}
$$

This is because any common inherited selectors are in conflict, and must be overridden in the child class.

$$
\begin{array}{l}
\mathit{methods\_inherited\_from} : \mathit{Class\_name} \times \mathit{Class\_map}' \to \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Method\_map} \\
\mathit{methods\_inherited\_from}(\mathit{class\_id}, \mathit{class\_map}) \quad \triangleq \\
\qquad \mathsf{let}\ \mathit{meth\_map} = \mathit{all\_methods\_of}(\mathit{class\_id}, \mathit{class\_map})\ \mathsf{in} \\
\qquad \{\mathit{sel} \mapsto \mathit{prepend\_class}(\mathit{class\_id}, \mathit{meth\_map}(\mathit{sel})) \\
\qquad\qquad\qquad\qquad \mid \mathit{sel} \in \mathsf{dom}\, \mathit{meth\_map}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{prepend\_class} : \mathit{Class\_name} \times \mathit{Method\_desc} \to \mathit{Method\_desc} \\
\mathit{prepend\_class}(c, \mathit{mdesc}) \quad \triangleq \\
\qquad \mu(\mathit{mdesc}, \mathit{Original\_class} \mapsto [c] \frown \mathit{Original\_class}(\mathit{mdesc}))
\end{array}
$$

Changes to Meaning Functions

The change to *MClass_body* sets up the path in the static environment (cf. p. 38):

$$
\begin{aligned}
&MClass\_body : Class\_body \to SEnv \to Class\_den \\
&MClass\_body\,[\![mk\text{-}Class\_body(\_, meths)]\!]\,\rho \quad \triangleq \\
&\quad \{sel \mapsto MMethod\,[\![meths(sel)]\!] \\
&\qquad\qquad \mu(\rho, Class \mapsto Original\_class(meths(sel))) \\
&\qquad\qquad\qquad\qquad \mid sel \in \mathsf{dom}\, meths\}
\end{aligned}
$$

Creating an object involves creating all its parts (cf. p. 43):

$$
\begin{aligned}
&MExpression : Expression \to SEnv \to DEnv \to \\
&\qquad\qquad Object\_memory \to \\
&\qquad\qquad\qquad Oop \times DEnv \times Object\_memory \\
&MExpression\,[\![mk\text{-}New\_object(class)]\!]\,\rho\delta\sigma \quad \triangleq \\
&\quad \mathsf{let}\ new\_obj = make\_obj(Instvars(\rho)(class))\ \mathsf{in} \\
&\quad \mathsf{let}\ (new\_oop, \sigma') = create(mk\text{-}Object(class, new\_obj), \sigma)\ \mathsf{in} \\
&\quad (new\_oop, \delta, \sigma')
\end{aligned}
$$

$$
\begin{aligned}
&make\_obj : Instvar\_desc \to Plain\_object \\
&make\_obj(iv\_desc) \quad \triangleq \\
&\quad \mathsf{let}\ local = \{id \mapsto \textsc{NilOop} \mid id \in Local\_instvars(iv\_desc)\}\ \mathsf{in} \\
&\quad \mathsf{let}\ inh = \{c \mapsto make\_obj(Inherited(iv\_desc)(c)) \\
&\qquad\qquad\qquad\qquad \mid c \in \mathsf{dom}\, Inherited(iv\_desc)\}\ \mathsf{in} \\
&\quad mk\text{-}Plain\_object(local, inh)
\end{aligned}
$$

The only other changes are in the interpretation of instance variable identifiers; the path component of the static environment must be passed to the appropriate auxiliary function (cf. p. 40):

$$
\begin{aligned}
&\mathit{MExpression}[\![\mathit{mk\text{-}Assignment}(id, rhs)]\!]\,\rho\delta\sigma \;\triangleq\\
&\quad \text{let } (\mathit{result}, \delta', \sigma') = \mathit{MExpression}[\![\mathit{rhs}]\!]\,\rho\delta\sigma \text{ in}\\
&\quad \text{cases } \mathit{id} \text{ of}\\
&\quad \mathit{mk\text{-}Temp\_id}(t) \quad \rightarrow (\mathit{result}, \mathit{update\_temp}(t, \mathit{result}, \delta'), \sigma')\\
&\quad \mathit{mk\text{-}Inst\_var\_id}(iv) \rightarrow (\mathit{result}, \delta',\\
&\qquad\qquad \mathit{update\_inst\_var}(iv, \mathit{Rcvr}(\delta'),\\
&\qquad\qquad\quad \mathit{result}, \mathit{Class}(\rho), \sigma'))\\
&\quad \text{end}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{MExpression}[\![\mathit{mk\text{-}Inst\_var\_id}(id)]\!]\,\rho\delta\sigma \;\triangleq\\
&\quad (\mathit{inst\_var}(id, \mathit{Rcvr}(\delta), \mathit{Class}(\rho), \sigma), \delta, \sigma)
\end{aligned}
$$

The changes made to semantic domains and meaning functions for tree inheritance solve the problems of modularity, but at the cost of extra complexity. Each object is now a tree-structured entity, with each sub-object corresponding to a node in the (unfolded) inheritance graph.

An alternative route to modelling the semantics of tree inheritance might be to make each sub-object a full object in its own right, i.e., with its own *Oop*. Such a system has already been described: namely, the prototype-based system discussed in §3.12. A prototype-based system can imitate a tree inheritance structure by using delegation. To do this, every object's prototype is an object representing its class. In addition to its own instance variables, an object has one instance variable for each of its class's parents, and this instance variable refers to the corresponding sub-object in the tree inheritance model. Any inherited message is delegated to the corresponding sub-object; because no change in the structure or behaviour of an object can take place at run-time, the appropriate delegation methods can be inherited from the class object, and no errors owing to missing attributes can occur.

4.4 The Interaction between Inheritance and Encapsulation

Inheritance introduces a new dimension to the problem of encapsulating the internals of an object; a class's children are now also clients of the class, and,

according to sound engineering practice, should be insulated from changes in the implementation of their ancestors. Unfortunately, this causes something of a dilemma: the purpose of inheritance is that one class may inherit another's implementation, and modify it in some way; how can this be achieved if the implementation is hidden?

The solution is to note that a class presents two, different interfaces to its users: one to other classes, sending messages to its instances, and one to its children. The purpose of the former interface is to define what an object does without revealing the details of its implementation. The purpose of the latter is to provide a package of re-usable parts, with carefully selected features changeable in subclasses.

The task of the class designer is to choose the best compromise between these two aspects. On the one hand, instances of her class must behave in a way useful to other objects, without revealing details of their internal behaviour via their message protocol. On the other hand, the message protocol must be chosen in such a way that by selectively overriding part of it, new and useful classes can be derived.

These separate aspects of class design should be supported by the language. Thus far, support for the first aspect, namely protection and encapsulation from other objects, has been described, and tree inheritance, which begins to support encapsulation, introduced. The rest of this section discusses other language features for the support of modular inheritance.

4.4.1 Private and Subclass-Visible Methods

Tree inheritance took the first steps towards encapsulation by hiding inherited instance variables. To gain access to its inherited parts, an object has to send messages to itself. Nevertheless, there is no way of preventing other objects invoking these methods. This is unsatisfactory: it leads to unwanted breaches of encapsulation, by necessitating the definition of publicly-accessible methods that do not form part of the external interface of an object. A language that forces the designer along this route is flawed; it does not allow a class to satisfy the following principle:

The Principle of Class Encapsulation The internal construction of an object or class should not be visible outside that object or class unless the object

or class chooses to make it so.

To distinguish between the external messages, and messages that can be invoked by subclasses, the idea of *subclass-visible* methods is introduced (see [SCB+86], for example). A subclass-visible method can be invoked using self or super, but in no other way. This enables a class to present a wider interface to its children than to other classes.

Additionally, the class designer should also be able to categorise some methods as *private*, i.e., invisible to all other classes, including subclasses. Private methods enable a programmer to decompose operations into smaller operations without compromising encapsulation. For example, if two methods *A* and *B* have a common part *C*, then *C* should be a separate method. Invocation of *C* by other objects or subclasses at inopportune moments may compromise integrity, so *C* should be invisible to them.

So important is the idea of decomposition to modern programming techniques, that it is enshrined in the following principle:

The Principle of Decomposition The programmer should always be able to decompose her problem into methods and/or objects without compromising the principles of encapsulation.

To model these three categories of methods, namely public, subclass-visible and private methods, the abstract syntax is extended so that every method is tagged with its type (cf. pp. 82, 79):

$$\begin{aligned} \mathit{Class\_body'} :: \ & \mathit{Instvars} : \mathit{Id}\text{-set} \\ & \mathit{Methods} : \mathit{Method\_map'} \\ & \mathit{Parents} : \mathit{Parent\_classes} \end{aligned}$$

$$\begin{aligned} \mathit{Method\_desc'} :: \ & \mathit{Method} : \mathit{Method\_body'} \cup \mathit{Primitive\_method} \\ & \mathit{Type} : \mathit{Method\_type} \end{aligned}$$

$$\mathit{Method\_type} = \{\textsc{Private}, \textsc{SubV}, \textsc{Public}\}$$

If the locally-defined private, subclass-visible, and public selectors of a class are denoted by $priv_L(c)$, $sv_L(c)$ and $pub_L(c)$ (where these three sets are disjoint), then the complete interface of c, including all inherited methods, is:

$$priv(c) = priv_L(c)$$
$$sv(c) = sv_L(c) \cup \big(\{sv(c') \mid c' \in parents(c)\} - priv_L(c) - pub_L(c)\big)$$
$$pub(c) = pub_L(c) \cup \big(\{pub(c') \mid c' \in parents(c)\} - priv_L(c) - sv_L(c)\big)$$

Changes to Pre-processing

Each different type of method is processed separately; inherited methods are combined with local methods according to the rules above.

Privacy is enforced in the context conditions by not passing selectors of private methods to subclasses. Nevertheless, this does not prevent external access to private and subclass-visible methods from other objects. Because messages are dynamically bound, this restriction must be enforced in the meaning functions, and hence the processed abstract syntax also includes the method type (cf. p. 95):

$$\begin{aligned} Method\_desc :: \; & Original\_class : Access\_path \\ & Method : Method\_body \cup Primitive\_method \\ & Type : Method\_type \end{aligned}$$

$$all\_methods\_of : Class\_name \times Class\_map' \to Method\_map$$
$$\begin{aligned} & all\_methods\_of(class\_id, class\_map) \triangleq \\ & \quad local\_meths\_type(class\_id, class\_map, \textsc{Private}) \\ & \quad \cup\, all\_meths\_type(class\_id, class\_map, \textsc{Public}) \\ & \quad \cup\, all\_meths\_type(class\_id, class\_map, \textsc{SubV}) \end{aligned}$$

$$local\_meths\_type : Class\_name \times Class\_map' \times Method\_type \to Method\_map$$
$$\begin{aligned} & local\_meths\_type(class\_id, class\_map, type) \triangleq \\ & \quad \{sel \mapsto mk\text{-}Method\_desc([\,], \\ & \qquad Method(Methods(class\_map(class\_id))(sel)), type) \\ & \qquad \mid Type(Methods(class\_map(class\_id))(sel)) = type\} \end{aligned}$$

$$all\_meths\_type : Class\_name \times Class\_map' \times Method\_type \to Method\_map$$

$$\begin{array}{l} all\_meths\_type(class\_id, class\_map, type) \triangleq \\ \quad \textsf{let } other\_types = Method\_type - \{type\} \textsf{ in} \\ \quad \textsf{let } others = \textsf{dom} \bigcup \{local\_meths\_type(class\_id, class\_map, t) \\ \qquad\qquad \mid t \in other\_types\} \textsf{ in} \\ \quad (others \lhd inh\_meths\_type(class\_id, class\_map, type)) \\ \qquad\qquad \dagger\ local\_meths\_type(class\_id, class\_map, type) \end{array}$$

$$inh\_meths\_type : Class\_name \times Class\_map' \times Method\_type \to Method\_map$$

$$\begin{array}{l} inh\_meths\_type(class\_id, class\_map, type) \triangleq \\ \quad \overset{*}{\bigcup} \{methods\_inherited\_from(p, class\_map, type) \\ \qquad\qquad \mid p \in parents(class\_id, class\_map)\} \end{array}$$

$$methods\_inherited\_from : Class\_name \times Class\_map' \times Method\_type \to Method\_map$$

$$\begin{array}{l} methods\_inherited\_from(class\_id, class\_map, type) \triangleq \\ \quad \textsf{let } meth\_map = all\_meths\_type(class\_id, class\_map, type) \textsf{ in} \\ \quad \{sel \mapsto prepend\_class(class\_id, meth\_map(sel)) \\ \qquad\qquad \mid sel \in \textsf{dom}\, meth\_map\} \end{array}$$

Changes to Context Conditions

WFProgram is modified so that the methods in a class only have access to inherited subclass-visible and public methods (cf. p. 92):

$$
\begin{array}{l}
WFProgram : Program' \to \mathbb{B} \\
WFProgram\,[\![class\_map]\!] \triangleq \\
\quad non\_circular(class\_map) \\
\quad \wedge\ \forall class\_id \in \mathsf{dom}\, class\_map \cdot \\
\qquad \mathsf{let}\ par = parents(class\_id, class\_map)\ \mathsf{in} \\
\qquad \mathsf{let}\ inh\_selectors = \\
\qquad\quad \{p \mapsto \mathsf{dom}\, all\_meths\_type(p, class\_map, \textsc{Public}) \\
\qquad\qquad \cup\ \mathsf{dom}\, all\_meths\_type(p, class\_map, \textsc{SubV}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid p \in par\}\ \mathsf{in} \\
\qquad \mathsf{let}\ \theta = \\
\qquad\quad mk\text{-}CCEnv(\_, \_, \_, \mathsf{dom}\, class\_map, inh\_selectors)\ \mathsf{in} \\
\qquad par \subseteq \mathsf{dom}\, class\_map \\
\qquad \wedge\ non\_conflicting\_methods(class\_id, par, class\_map) \\
\qquad \wedge\ WFClass\,[\![class\_map(class\_id)]\!]\,\theta
\end{array}
$$

Changes to Semantic Domains and Meaning Functions

The denotation of a class has been a function that interprets messages (pp. 36, 38); now the domain of class denotations must be extended to distinguish between messages sent from the object denoted by self, and messages from other objects. Note that the context conditions and pre-processing section ensure that a class cannot gain access to its parents' private methods.

$$Class\_den = Selector \xrightarrow{m} Method\_den$$

$$Method\_den = Public\_method \cup Private\_method$$

$$Method\_response = Oop \times (Oop^*) \times Object\_memory \to Oop \times Object\_memory$$

$$Public\_method :: Meth : Method\_response$$

$$Private\_method :: Meth : Method\_response$$

$$
\begin{array}{l}
MMethod : Method\_desc \to SEnv \to Method\_den \\
MMethod[\![mm]\!]\rho \quad \triangleq \\
\quad \textsf{let } m = Method(mm) \textsf{ in} \\
\quad \textsf{let } md = \textsf{if } m \in Primitive\_method \\
\qquad \textsf{then } m \\
\qquad \textsf{else let } \rho' = \mu(\rho, Class \mapsto Original\_class(mm)) \textsf{ in} \\
\qquad\quad MMethod\_body[\![m]\!]\rho' \\
\quad \textsf{in} \\
\quad \textsf{if } Type(mm) \in \{\textsc{Private}, \textsc{SubV}\} \\
\quad \textsf{then } mk\text{-}Private\_meth(md) \\
\quad \textsf{else } mk\text{-}Public\_meth(md)
\end{array}
$$

The *perform* function makes the necessary check to ensure that external messages cannot invoke private methods (cf. p. 42):

$$
\begin{array}{l}
perform : Selector \times Program\_den \\
\qquad\qquad \times Oop \times Oop^* \times Object\_memory \times \mathbb{B} \\
\qquad\qquad\qquad\qquad \to Oop \times Object\_memory \\
perform(sel, pd, rcvr, args, \sigma, from\_self) \quad \triangleq \\
\quad \textsf{let } class = class(rcvr, \sigma) \textsf{ in} \\
\quad \textsf{if } sel \in public\_selectors(class, pd) \\
\quad\ \vee\ from\_self \wedge sel \in private\_selectors(class, pd) \\
\quad \textsf{then } method(sel, class, pd)(rcvr, args, \sigma) \\
\quad \textsf{else } message\_not\_understood(sel, \rho, \\
\qquad\qquad\qquad\qquad class, rcvr, args, \sigma, from\_self)
\end{array}
$$

$$
\begin{array}{l}
private\_selectors : Class\_name \times Program\_den \to Selector\text{-}\textsf{set} \\
private\_selectors(class, pd) \quad \triangleq \\
\quad \{sel \in \textsf{dom}\, pd(class) \mid pd(class)(sel) \in Private\_method\}
\end{array}
$$

$$
\begin{array}{l}
public\_selectors : Class\_name \times Program\_den \to Selector\text{-}\textsf{set} \\
public\_selectors(class, pd) \quad \triangleq \\
\quad \{sel \in \textsf{dom}\, pd(class) \mid pd(class)(sel) \in Public\_method\}
\end{array}
$$

$$method : Selector \times Class\_name \times Program\_den \rightarrow Method\_response$$

$$method(sel, class, pd) \triangleq Meth(pd(class)(sel))$$

Note that the *message_not_understood* function now serves two purposes: it invokes an error response when a message is received that has no associated method, and also handles messages that are received from other objects that are attempting to invoke private methods.

There are two approaches to ensuring that an object can only invoke its own private methods. First, one could insist that a private method could only be invoked in response to a message sent to self, i.e., enforce the restriction at a syntactic level. Alternatively, a check could be made at run-time to see if the sender and the receiver were the same object. The former requires a change to the *MExpression* meaning function, which inspects the syntax of the send expression (cf. p. 41):

$$MExpression : Expression \rightarrow SEnv \rightarrow DEnv \rightarrow Object\_memory \rightarrow Oop \times DEnv \times Object\_memory$$

$$MExpression\,[\![mk\text{-}Message(rcvr, sel, arglist)]\!]\,\rho\delta\sigma \triangleq$$
$$\quad \text{let } (rcvr\_oop, \delta', \sigma') = MExpression\,[\![rcvr]\!]\,\rho\delta\sigma \text{ in}$$
$$\quad \text{let } (actuals, \delta'', \sigma'') = MExpression\_list\,[\![arglist]\!]\,\rho\delta'\sigma' \text{ in}$$
$$\quad \text{let } (result, \sigma''') = perform(sel, PD(\rho), rcvr\_oop, actuals, \sigma'', rcvr = \text{SELF}) \text{ in}$$
$$\quad (result, \delta'', \sigma''')$$

The latter is based on the values of the current receiver and the receiver of the message:

$$MExpression\,[\![mk\text{-}Message(rcvr, sel, arglist)]\!]\,\rho\delta\sigma \triangleq$$
$$\quad \text{let } (rcvr\_oop, \delta', \sigma') = MExpression\,[\![rcvr]\!]\,\rho\delta\sigma \text{ in}$$
$$\quad \text{let } (actuals, \delta'', \sigma'') = MExpression\_list\,[\![arglist]\!]\,\rho\delta'\sigma' \text{ in}$$
$$\quad \text{let } (result, \sigma''') = perform(sel, PD(\rho), rcvr\_oop, actuals, \sigma'', rcvr\_oop = Rcvr(\delta'')) \text{ in}$$
$$\quad (result, \delta'', \sigma''')$$

As no current language distinguishes between private and public methods (to the best of the author's knowledge), there is no experience to suggest which of these forms will be found superior in practice. Nevertheless, the former semantics suggest both simpler proof rules (based on syntactic characteristics), and the potential for simpler, more efficient implementation.

4.4.2 Static Binding of Messages, Revisited

The use of private methods allows a class designer to decompose her methods without fear that the sub-methods will be invoked from other classes, but this does not prevent a subclass from re-implementing a private method, thereby capturing all messages sent to self, intended to invoke the private method in the superclass. This would allow a subclass to intercept a message at a point when an instance of the class was not in a well-defined state (i.e., when an invariant on the instance did not hold). To avoid this, the form of statically-bound message sends is extended so that methods are called directly, not only in superclasses but also in the same class. In effect, when a class designer designates a method as private, she is saying, "I don't want anyone else to invoke this method"; when statically binding a message she is saying, "I want *this* method to be invoked by this message." Both of these facilities are required to uphold the Principles of Class Encapsulation and Decomposition.

A new form of message-send expression is introduced, the *Self_send*:[4]

$$\begin{aligned} \mathit{Expression}' = \mathit{Expression\_list}' \cup \mathit{Assignment}' \cup \mathit{Object\_name} \\ \cup \mathit{Message}' \cup \mathit{Literal\_object} \cup \mathit{New\_object} \\ \cup \mathit{Super\_send} \cup \mathit{Self\_send} \end{aligned}$$

$$\begin{aligned} \mathit{Self\_send} :: \quad & \mathit{Sel} : \mathit{Selector} \\ & \mathit{Args} : \mathit{Expression}'^{*} \end{aligned}$$

A *Self_send* can be converted into a *Static_send* by tagging it with the name of the class of its enclosing method. This is similar to the processing of *Super_sends* in a single-inheritance hierarchy (§4.2), except that the current

[4] The concrete syntax used in Smalltalk for messages sent to self and *Super_sends* is self foo: bar and super foo: bar; a suitable extension for *Self_sends* might be here foo: bar, for example.

class, instead of the parent class, is propagated down through the processing functions:

$$
\begin{array}{l}
PExpression : Expression' \to Class\_name \to Expression \\
PExpression\,[\![expr]\!]\,this\_class \triangleq \\
\quad \textsf{cases}\ expr\ \textsf{of} \\
\quad mk\text{-}Super\_send(s,a) \to \ldots \\
\quad mk\text{-}Assignment(l,r) \to \ldots \\
\quad mk\text{-}Message(r,s,a) \to \ldots \\
\quad mk\text{-}Self\_send(s,a) \to mk\text{-}Static\_send(s, \\
\qquad\qquad PExpression\_list\,[\![a]\!]\,this\_class, \\
\qquad\qquad this\_class) \\
\quad \ldots \\
\quad \textsf{end}
\end{array}
$$

The well-formedness of a *Self_send*, like that of a *Super_send*, depends on whether a message response is defined for the selector:

$$
\begin{array}{l}
WFExpression : Expression' \to CCEnv \to \mathbb{B} \\
WFExpression\,[\![mk\text{-}Self\_send(sel,args)]\!]\,\theta \triangleq \\
\quad \textsf{len}\,args = nargs(sel) \\
\quad \wedge\ sel \in My\_selectors(\theta) \cup \bigcup \textsf{rng}\,Inherited\_selectors(\theta) \\
\quad \wedge\ \forall arg \in \textsf{rng}\,args \cdot WFExpression\,[\![arg]\!]\,\theta
\end{array}
$$

$$
\begin{array}{lrcl}
CCEnv\ :: & Instvars & : & Id\text{-set} \\
 & Params & : & Id\text{-set} \\
 & Temps & : & Id\text{-set} \\
 & Classes & : & Class\_name\text{-set} \\
 & Inherited\_selectors & : & Class\_name \xrightarrow{m} Selector\text{-set} \\
 & My\_selectors & : & Selector\text{-set}
\end{array}
$$

Note that *CCEnv* is augmented with a *My_selectors* component to record which selectors can be used in a *Self_send* (cf. p. 84). This component is set up in *WFProgram* (cf. p. 103):

$$
\begin{array}{l}
WFProgram : Program' \to \mathbb{B} \\
WFProgram\,[\![class\_map]\!] \quad \triangleq \\
\quad non\_circular(class\_map) \\
\quad \wedge\ \forall class\_id \in \mathsf{dom}\, class\_map \cdot \\
\qquad \mathsf{let}\ par = parents(class\_id, class\_map)\ \mathsf{in} \\
\qquad \mathsf{let}\ inherited\_selectors = \\
\qquad\quad \{p \mapsto \mathsf{dom}\, all\_meths\_type(p, class\_map, \textsc{Public}) \\
\qquad\qquad \cup\, \mathsf{dom}\, all\_meths\_type(p, class\_map, \textsc{SubV}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid p \in par\}\ \mathsf{in} \\
\qquad \mathsf{let}\ sels = local\_selectors(class\_id, class\_map)\ \mathsf{in} \\
\qquad \mathsf{let}\ \theta = mk\text{-}CCEnv(\_, \_, \_, \mathsf{dom}\, class\_map, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad inherited\_selectors, sels)\ \mathsf{in} \\
\qquad par \subseteq \mathsf{dom}\, class\_map \\
\qquad \wedge\ non\_conflicting\_methods(class\_id, par, class\_map) \\
\qquad \wedge\ WFClass\,[\![class\_map(class\_id)]\!]\,\theta
\end{array}
$$

$$
\begin{array}{l}
local\_selectors : Class\_name \times Class\_map' \to Selector\text{-}\mathsf{set} \\
local\_selectors(class\_id, class\_map) \quad \triangleq \\
\quad \mathsf{dom}\, Methods(class\_map(class\_id))
\end{array}
$$

4.5 Inheritance and Primitive Classes

Some of the primitive classes and methods described in Chapter 3 are affected by the presence of inheritance. First, most of the primitive classes cannot be usefully subclassed, because instances of primitive classes are usually created by evaluating literal expressions; therefore, there is no mechanism for creating instances of subclasses. The only exception among the classes mentioned in Chapter 3 is the primitive class of indexable objects; instances are created by a primitive, which would be inherited by subclasses.

Indexable objects show up the deficiencies (mentioned in §4.3.3) in the graph and linear inheritance schemes. If a class's instances may have indexable fields, then so should its childrens' instances, but extending the simple definition of *Plain_object* used for graph and linear inheritance results in indexable fields inherited from multiple parents being aliased to each other. Only

the tree inheritance scheme, with separate name spaces for separate inherited parts, can cope with inheritance of indexable fields properly.

4.6 Summary

This chapter has augmented the model of Chapter 3 with several different inheritance schemes. Single inheritance poses no problems, merely requiring a separate processing stage to transform inheritance-based programs into ones without inheritance. Two multiple inheritance schemes, graph and linear inheritance, can also be handled in this way, but both of these schemes have problems as secure structuring mechanisms. Tree inheritance solves these problems, but requires a change in the semantic domains. To maintain the desired degree of encapsulation, method-hiding and message-hiding features are also added to the language, being smoothly incorporated into the tree inheritance semantics.

Chapter 5

Control Structures

> As Will Rogers would have said, "There is no such thing as a free variable."
>
> *Alan Perlis*

Chapter 3 introduced a simple model of object-oriented languages, and Chapter 4 extended the model to include inheritance. The reader may have observed that the languages defined in these chapters did not have any of the usual features for control, namely branching and looping structures. Conventional control structures such as "if" and "while" statements can be added to the language in the usual manner, as in [Sto77]; there is no unexpected interaction between them and the object-oriented features.

This chapter does not describe conventional control structures, but takes a different approach: it describes the object-oriented control structures provided by Smalltalk-80.[1] In addition to the message-sending mechanism described in

[1] To the author's knowledge, no other object-oriented language has adopted these structures. This is understandable in "hybrid" languages such as Objective-C or the Lisp "family" which have

Chapter 3, Smalltalk provides only one other primitive control structure; all others can be implemented using it. This takes the form of "blocks",[2] which can be considered to be "anonymous methods" or "deferred code." For an introduction to blocks, the reader is referred to [GR83].

Before describing blocks in more detail, it should be pointed out that the language of Chapter 3 is computationally complete. The semantics of message-passing are such that they can be used to provide recursion, and hence iteration, and also branching via dynamic binding. For example, the factorial function on natural numbers can be defined thus (using Smalltalk concrete syntax[3]):

```
class           Integer
factorial
    ↑self isZero restOfFactorial: self

class           True
restOfFactorial: n
    ↑1

class           False
restOfFactorial: n
    ↑n * (n - 1) factorial
```

The isZero method is assumed to invoke a method that returns true if its receiver is zero, false if non-zero. As true and false are instances of different classes, namely True and False, a different method can be invoked depending on the outcome; thus dynamic binding is used to implement conditional branches. This example should also show that this technique is inelegant for expressing non-trivial algorithms; much of the code is pushed into "continuation" methods in the boolean classes.

---

their own control structures, but less so in new, pure, object-oriented languages. The most likely reason that the author can suggest for this is that a certain amount of folklore has built up around these features that considers them to be inefficient.

[2] A most unfortunate choice of name, given that two decades of imperative programming tradition have used the name to mean something else.

[3] This chapter assumes more familiarity with Smalltalk syntax than previous chapters.

5.1 Blocks

Blocks are literals that denote objects which are executable pieces of code. In one view, they can be considered to be anonymous methods. This is clear when the factorial function is rewritten using blocks:

```
class       Integer
factorial
    ↑self isZero
        ifTrue: [1]
        ifFalse: [self * (self - 1) factorial]
```

The blocks are the code fragments delimited by square brackets. A block is executed by sending it a message that is bound to a special primitive (the *value* primitive). This is achieved in the above case by having an ifTrue:ifFalse: message defined in the True and False classes:

```
class       True
ifTrue: trueBlock ifFalse: falseBlock
    ↑trueBlock value

class       False
ifTrue: trueBlock ifFalse: falseBlock
    ↑falseBlock value
```

An obvious difference between the method-based factorial definition and the block-based version is that the recursive code in the false "branch" uses different variable names to refer to the integer argument: in the method version an argument, n, refers to the integer, while self is bound to the receiver, namely false;[4] in the block version self refers to the integer. This illustrates that the environment of the code within the block is the same as the environment of its enclosing text: self and the instance variables of self are in scope. Therefore, the block must be bound to the environment of its enclosing method sometime during the execution of that method, forming a *closure*, with no free variables.

[4] To re-iterate, false denotes the sole instance of the class False.

Objects representing bound blocks are instances of a class Closure.[5]

The analogy of blocks as anonymous methods goes even further: a block may have a number of parameters and temporaries.[6] Parameters are denoted using a preceding colon (to mirror the definition of parameters in methods), and temporaries are enclosed between vertical bars, just as in methods:

```
methodWithParameter: aParam
    | temp1 temp2 |
    ... [ :param1 :param2 | blocktemp |
        ... "body of block with two parameters" ]
```

A block with parameters is activated using variants of the value primitive: value: takes one argument, value:value: takes two, etc., and valueWithArguments: takes one argument, which is an Array (see §3.10.4) of arguments to the value primitive.

The languages of Chapters 3 and 4 are extended to include blocks by extending the abstract syntax of literals thus:[7]

$$Literal\_object = \ldots \cup Block\_body$$

$$\begin{aligned} Block\_body :: \; & Params : Ulist(Id) \\ & Temps : Id\text{-set} \\ & Expr : Expression \end{aligned}$$

Note that the abstract syntax of a block is identical to that of a *Method_body* (cf. p. 33).

The well-formedness rule for blocks is the same as that for methods, except that a modified environment is used:

---

[5] Smalltalk *aficionados* will know that this is not the same as in Smalltalk—the reason for this will be explained in succeeding sections.

[6] The Smalltalk-80 language does not allow blocks to have temporaries. The reason for this is a rather involved technical and historical accident that need not be explained here—the interested reader should consult [Wol87]. This is likely to be changed as part of the current effort to standardise Smalltalk.

[7] As the semantics of blocks and inheritance are independent, the primes used in Chapter 4 to denote unprocessed syntax will be dropped.

$$
\begin{array}{l}
\mathit{WFExpression}\,[\![\mathit{mk\text{-}Block\_body}(\mathit{params}, \mathit{temps}, \mathit{expr})]\!]\,\theta \quad \triangleq \\
\quad \mathsf{let}\ \theta' = \mu(\mu(\theta, \mathit{Params} \mapsto \mathit{Params}(\theta) \cup \mathsf{rng}\,\mathit{params}), \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Temps} \mapsto \mathit{temps})\ \mathsf{in} \\
\quad \mathit{WFExpression}\,[\![\mathit{expr}]\!]\,\theta'
\end{array}
$$

The restriction ensures that any temporaries in the enclosing text are not accessible within the block—the reason for this will be explained in §5.1.1.

The semantic domain of primitive objects is extended to include the denotations of blocks (or, to be exact, closures):

$$
\mathit{Primitive\_object} = \ldots \cup \mathit{Block\_den}
$$

$$
\mathit{Block\_den} = \mathit{Oop} \times (\mathit{Oop}^*) \times \mathit{Object\_memory} \to \mathit{Oop} \times \mathit{Object\_memory}
$$

Again, there is a similarity between methods and blocks (cf. p. 36).

The meaning function for a block expression creates an object (an instance of a primitive class, Closure) that contains the denotation of the block:

$$
\begin{array}{l}
\mathit{MExpression} : \mathit{Expression} \to \mathit{SEnv} \to \mathit{DEnv} \to \mathit{Object\_memory} \to \mathit{Oop} \times \mathit{DEnv} \times \mathit{Object\_memory} \\
\mathit{MExpression}\,[\![b]\!]\,\rho\delta\sigma \quad \triangleq \qquad\qquad (\text{for } b \in \mathit{Block\_body}) \\
\quad \mathsf{let}\ (\mathit{closure}, \sigma') = \\
\qquad \mathit{create}(\mathit{mk\text{-}Object}(\mathsf{Closure}, \mathit{MBlock\_body}\,[\![b]\!]\,\rho\delta), \sigma)\ \mathsf{in} \\
\quad (\mathit{closure}, \delta, \sigma')
\end{array}
$$

$$
\begin{array}{l}
MBlock\_body : Block\_body \rightarrow SEnv \rightarrow DEnv \rightarrow Block\_den \\
MBlock\_body\,[\![mk\text{-}Block\_body(params, temps, expr)]\!]\,\rho\delta \quad \triangleq \\
\quad \lambda closure, args, \sigma \cdot \\
\quad\quad \textsf{if len}\, params \neq \textsf{len}\, args \\
\quad\quad \textsf{then}\, block\_arg\_error(\rho, \delta, closure, args, \sigma) \\
\quad\quad \textsf{else let}\, bindings = Params(\delta) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \dagger\, bind\_args(params, args)\ \textsf{in} \\
\quad\quad\quad \textsf{let}\, \delta' = \\
\quad\quad\quad\quad mk\text{-}DEnv(Rcvr(\delta), bindings, initialise(temps))\ \textsf{in} \\
\quad\quad\quad \textsf{let}\, (result, \delta'', \sigma') = MExpression\,[\![expr]\!]\,\rho\delta'\sigma\ \textsf{in} \\
\quad\quad\quad (result, \sigma')
\end{array}
$$

Points to note:

- Unlike the meaning functions for other literals, the meaning of a block expression always results in the creation of a new object. This is because there is no way to determine, in general, if a new closure is equivalent to a pre-existing closure in the object memory.
- When activated, the block must check that it has been supplied with the correct number of arguments. If not, the *block_arg_error* function is invoked; in Smalltalk this results in an error message being sent. An alternative strategy might be to ignore extra arguments, and supply default values for missing arguments.

The value primitive is defined on instances of class Closure. It applies the denotation of the closure to its arguments:

$$
\begin{array}{l}
value\_primitive : Primitive\_method \\
value\_primitive \quad \triangleq \\
\quad \lambda closure, args, \sigma \cdot body(closure, \sigma)(closure, args, \sigma)
\end{array}
$$

5.1.1 Access to Non-local Variables from within Blocks

The dynamic environment of a block as constructed in *MBlock_body* includes the arguments and temporaries of the block, and arguments of the enclosing

method (or block; blocks can be nested), but not non-local temporaries. The reason for this is that access to non-local temporaries introduces a form of sharing that is extremely complicated to model formally.

Access to non-local temporaries means that a block must be able to alter the dynamic environment of its enclosing method. However, because closures are first-class objects they have an unbounded lifetime, which can exceed the lifetime of the activation that created them. Hence, to model access to non-local temporaries, dynamic environments have to be maintained in a structure indefinitely, and passed around as part of the long-term store with the *Object_memory*.

Although modelling access to non-local temporaries is complex, it is still possible (the problem is the same as that in an imperative language with closures and lexical scoping; a static chain of closures, parallel to the lexical scoping, is constructed). However, a much simpler alternative is espoused in [Wol87]. This is to treat access to non-local temporaries in the same way as access to non-local arguments, fixing the value of the temporary variable when the block is bound. The block has read-only access to non-local temporaries. Consider the following fragment of code:

```
sel: arg1
    | t1 |
    t1 ← "expression 1".
    … "B1" [ :arg2 | t2 | … "p1"
            "B2" [ : arg3 | t3 | … "p2" ]]
```

Block B2 is lexically nested within block B1. Blocks B1 and B2 are bound when they are referenced in an expression, to the environment that is in use at the time. Thus, block B1 has access to arg1 and t1 as well as to its own argument and temporary variable. The value of arg1 is the same as that in the enclosing method, while the value of t1 is whatever it was when B1 was bound, namely the value of "expression 1". The value of t1 is constant throughout the lifetime of the bound block B1, but the code outside can still, of course, modify its own copy of t1. Similarly, block B2 has read-only access to the same value of t1, and also read-only access to arg2 and t2, using the values that were in force at the time B2 was bound. It should be emphasised again that non-local variables are equivalent to arguments: their value does not change during the lifetime of

the closure.

To model this formally, a minor change is made to *MBlock_body* (cf. p. 115):

$$
\begin{array}{l}
MBlock\_body : Block\_body \rightarrow SEnv \rightarrow DEnv \rightarrow Block\_den \\
MBlock\_body\,[\![mk\text{-}Block\_body(params, temps, expr)]\!]\,\rho\delta \quad \triangleq \\
\quad \lambda closure, args, \sigma \cdot \\
\qquad \textsf{if len}\, params \neq \textsf{len}\, args \\
\qquad \textsf{then}\ block\_arg\_error(\rho, \delta, closure, args, \sigma) \\
\qquad \textsf{else let}\ bindings = Temps(\delta) \\
\qquad\qquad\qquad \dagger\, Params(\delta) \dagger\, bind\_args(params, args)\ \textsf{in} \\
\qquad\quad \textsf{let}\ \delta' = \\
\qquad\qquad mk\text{-}DEnv(Rcvr(\delta), bindings, initialise(temps))\ \textsf{in} \\
\qquad\quad \textsf{let}\ (result, \delta'', \sigma') = MExpression\,[\![expr]\!]\,\rho\delta'\sigma\ \textsf{in} \\
\qquad\quad (result, \sigma')
\end{array}
$$

With this model of computation, a shared non-local variable can be simulated using an extra object (see [Wol87] for details). In short, conventional Smalltalk code that uses shared non-local variables can be transformed into equivalent code that uses extra objects to provide the sharing. For example, this code:

```
foo
    | t |
    t ← "value1".
    … [ … t ← "value2" …]
```

becomes:

```
foo
    | t |
    t ← Indirection on: "value1".
    … [ … t value: "value2" …]
```

This transformation can be made by the programmer or by the compilation system.

5.1.2 Uses of Closures

Closures can also be used as *capabilities* [Fab74] for accessing private and subclass-visible methods. By enclosing a send to itself within a block, an object can give to other objects limited and controlled access to its private methods.

5.2 Continuations

The closures described thus far represent deferred actions. When activated, a closure performs its actions, and then returns control to the method that activated it. The basic control structure, therefore, is that of procedure call and return.

In this section the repertoire of abilities of blocks is extended by allowing them to exit in a different way. A conventional closure can be thought of as being passed a continuation sequence when activated, and then exiting by handing control to the continuation sequence. In this extended form, the block is given the current continuation sequence when it is *bound*, activating that sequence when it exits, rather than the one passed to it when activated. This action corresponds to the behaviour of Smalltalk blocks that exit with a "return operator" (↑). Control is handed back to the method in which the block was bound (the so-called *home context*). For example:

```
m1
        "some condition" ifTrue: [self].
        ... "rest of method"

m2
        "some condition" ifTrue: [↑self].
        ... "rest of method"
```

In method m1, when the block is activated it returns the value of self to the place where it was activated. In method m2, if the condition is true the value of self is returned to the method that caused the activation of m2 and the binding of the block; the "rest of method" is not executed.

5.2.1 Formal Semantics

To model the new form of blocks, the method of definition used thus far, i.e., so-called *direct* semantics, has to be abandoned. Because it uses function composition to build denotations, only function-like control structures can be modelled [Sto77]. Instead, *continuation* semantics are needed, with each meaning function being passed an additional argument (known as the continuation), representing the "natural" sequence of actions to be performed after those specified by the current program fragment. To model an unusual control structure, the meaning function for the control structure can choose to ignore the continuation that it has been given, and use a different one.

The conversion from the existing definitions to continuation-based semantics is, for the most part, straightforward. Meaning functions of the form:

$$M: S \to SEnv \to DEnv \to Object\_memory \to Oop \times DEnv \times Object\_memory$$

become:

$$M_c : S \to SEnv \to DEnv \to Object\_memory \to ECont \to Oop \times Object\_memory$$

$$M_c [\![m]\!] \rho\delta\sigma\varepsilon \quad \triangleq \quad \varepsilon(M [\![m]\!] \rho\delta\sigma)$$

where

$$ECont = Oop \times DEnv \times Object\_memory \to Oop \times Object\_memory$$

(The *c* suffix will be dropped from the definitions that follow.) Note that the original functions return a dynamic environment for use in future computation, whereas in the continuation version it is consumed by the continuation ε, and only the result *Oop* and *Object_memory* are used as "answers."

For example, the meaning function for an *Assignment* changes from:

$$
\begin{array}{l}
MExpression : Expression \rightarrow SEnv \rightarrow DEnv \rightarrow \\
\qquad\qquad Object\_memory \rightarrow \\
\qquad\qquad\qquad Oop \times DEnv \times Object\_memory \\
MExpression\,[\![mk\text{-}Assignment(id, rhs)]\!]\,\rho\delta\sigma \;\triangleq \\
\quad \text{let } (result, \delta', \sigma') = MExpression\,[\![rhs]\!]\,\rho\delta\sigma \text{ in} \\
\quad \text{cases } id \text{ of} \\
\quad mk\text{-}Temp\_id(t) \quad \rightarrow (result, update\_temp(t, result, \delta'), \sigma') \\
\quad mk\text{-}Inst\_var\_id(iv) \rightarrow (result, \delta', \\
\qquad\qquad update\_inst\_var(iv, Rcvr(\delta), \\
\qquad\qquad\qquad result, Class(\rho), \sigma')) \\
\quad \text{end}
\end{array}
$$

to

$$
\begin{array}{l}
MExpression : Expression \rightarrow SEnv \rightarrow DEnv \rightarrow \\
\qquad\qquad Object\_memory \rightarrow \\
\qquad\qquad\qquad ECont \rightarrow Oop \times Object\_memory \\
MExpression\,[\![mk\text{-}Assignment(id, rhs)]\!]\,\rho\delta\sigma\varepsilon \;\triangleq \\
\quad MExpression\,[\![rhs]\!]\,\rho\delta\sigma \\
\quad\; (\lambda result, \delta', \sigma' \cdot \\
\quad\quad \text{cases } id \text{ of} \\
\quad\quad mk\text{-}Temp\_id(t) \quad \rightarrow \varepsilon(result, \\
\qquad\qquad\qquad update\_temp(t, result, \delta'), \sigma') \\
\quad\quad mk\text{-}Inst\_var\_id(iv) \rightarrow \varepsilon\,(result, \delta', \\
\qquad\qquad\qquad update\_inst\_var(iv, Rcvr(\delta'), \\
\qquad\qquad\qquad\quad result, Class(\rho), \sigma'))\big) \\
\quad\quad \text{end})
\end{array}
$$

A complete set of formulas can be found in Appendix B.

Once the conversion has been made, a few modifications are necessary to support the extended form of blocks (which, when bound, shall hereafter be called “continuations”). First, the syntax changes are presented:

$$Literal\_object' = Block\_body' \cup Cont\_body' \cup Int\_literal \cup \ldots$$

$$\begin{aligned} Cont\_body' :: \ & Params : Ulist(Id) \\ & Temps : Id\text{-set} \\ & Expr : Expression' \end{aligned}$$

Note that the abstract syntax of a continuation is the same as that for a closure (p. 113). The context conditions are also similar, as are the pre-processing functions; they can be found in Appendix B.

The following change is made to the semantic domain of methods:

$$Method\_den = MCont \to Oop \times (Oop^*) \times Object\_memory \to Oop \times Object\_memory$$

$$Block\_den = Method\_den$$

The types of continuations used are these:

$$MCont = Oop \times Object\_memory \to Oop \times Object\_memory$$

$$ECont = Oop \times DEnv \times Object\_memory \to Oop \times Object\_memory$$

$$LCont = (Oop^*) \times DEnv \times Object\_memory \to Oop \times Object\_memory$$

The *ECont* continuation is used as the continuation of an expression, whereas the *MCont* continuation is used as the continuation of a method—it does not use the dynamic environment. The *LCont* type is used as a continuation for message-sends, where a number of arguments are passed to a method.

The following auxiliary functions convert between *ECont*s and *MCont*s by ignoring or inserting into the continuation a dynamic environment:

$$ignore : MCont \to ECont$$
$$ignore(\nu) \triangleq \lambda r, \delta, \sigma \cdot \nu(r, \sigma)$$

$$insert : DEnv \to ECont \to MCont$$
$$insert\ (\delta)(\varepsilon) \triangleq \lambda r, \sigma \cdot \varepsilon(r, \delta, \sigma)$$

A continuation block does not execute the continuation that it is passed when activated, but the continuation of the method in which it was bound. This continuation is recorded in the dynamic environment of the method (cf. p. 38):

$$\begin{aligned} DEnv :: \quad & Rcvr : Oop \\ & Params : Id \xrightarrow{m} Oop \\ & Temps : Id \xrightarrow{m} Oop \\ & Cont : MCont \end{aligned}$$

The *Cont* field of a dynamic environment is set up in the meaning function for methods (cf. p. 39):

$$\begin{aligned} &MMethod\_body : Method\_body \to SEnv \to Method\_den \\ &MMethod\_body[\![mk\text{-}Method\_body(params, temps, expr)]\!]\,\rho \quad \triangleq \\ &\quad \lambda\nu \cdot \lambda rcvr, args, \sigma \cdot \\ &\qquad \text{let } bindings = bind\_args(params, args) \text{ in} \\ &\qquad \text{let } \delta = mk\text{-}DEnv(rcvr, bindings, initialise(temps), \nu) \text{ in} \\ &\qquad MExpression[\![expr]\!]\,\rho\delta\sigma\ ignore(\nu) \end{aligned}$$

The meaning functions for closure and continuation blocks illustrate the difference between them: a closure block executes the continuation that it was passed when activated, while a continuation block executes the continuation from its dynamic environment:

$$\begin{aligned} &MExpression[\![b]\!]\,\rho\delta\sigma\varepsilon \quad \triangleq \qquad\qquad (\text{for } b \in Block\_body) \\ &\quad \text{let } obj = mk\text{-}Object(\text{Closure}, MBlock\_body[\![b]\!]\,\rho\delta) \text{ in} \\ &\quad \text{let } (closure, \sigma') = create(obj, \sigma) \text{ in} \\ &\quad \varepsilon(closure, \delta, \sigma') \end{aligned}$$

$$\begin{aligned} &MExpression[\![c]\!]\,\rho\delta\sigma\varepsilon \quad \triangleq \qquad\qquad (\text{for } c \in Cont\_body) \\ &\quad \text{let } obj = mk\text{-}Object(\text{Continuation}, MCont\_body[\![c]\!]\,\rho\delta) \text{ in} \\ &\quad \text{let } (closure, \sigma') = create(obj, \sigma) \text{ in} \\ &\quad \varepsilon(closure, \delta, \sigma') \end{aligned}$$

$$
\begin{array}{l}
MBlock\_body : Block\_body \to SEnv \to DEnv \to Block\_den \\
MBlock\_body\,[\![mk\text{-}Block\_body(params, temps, expr)]\!]\,\rho\delta \quad \triangleq \\
\quad \lambda\nu \cdot \lambda closure, args, \sigma \cdot \\
\quad\quad \textsf{if len}\, params \neq \textsf{len}\, args \\
\quad\quad \textsf{then}\ block\_arg\_error(\rho, \delta, closure, args, \sigma, \nu) \\
\quad\quad \textsf{else let}\ \delta' = update\_vars(\delta, \\
\quad\quad\quad\quad\quad Temps(\delta) \dagger Params(\delta) \dagger \\
\quad\quad\quad\quad\quad\quad bind\_args(params, args), \\
\quad\quad\quad\quad\quad\quad\quad\quad initialise(temps))\ \textsf{in} \\
\quad\quad\quad MExpression\,[\![expr]\!]\,\rho\delta'\sigma(ignore\ \nu)
\end{array}
$$

$$
\begin{array}{l}
MCont\_body : Cont\_body \to SEnv \to DEnv \to Block\_den \\
MCont\_body\,[\![mk\text{-}Cont\_body(params, temps, expr)]\!]\,\rho\delta \quad \triangleq \\
\quad \lambda\nu \cdot \lambda closure, args, \sigma \cdot \\
\quad\quad \textsf{if len}\, params \neq \textsf{len}\, args \\
\quad\quad \textsf{then}\ block\_arg\_error(\rho, \delta, closure, args, \sigma, \nu) \\
\quad\quad \textsf{else let}\ \delta' = update\_vars(\delta, \\
\quad\quad\quad\quad\quad Temps(\delta) \dagger Params(\delta) \\
\quad\quad\quad\quad\quad\quad \dagger\ bind\_args(params, args), \\
\quad\quad\quad\quad\quad\quad\quad\quad initialise(temps))\ \textsf{in} \\
\quad\quad\quad MExpression\,[\![expr]\!]\,\rho\delta'\sigma\ ignore(Cont(\delta'))
\end{array}
$$

$$
\begin{array}{l}
update\_vars : DEnv \times Id \xrightarrow{m} Oop \times Id \xrightarrow{m} Oop \to DEnv \\
update\_vars(\delta, params, temps) \quad \triangleq \\
\quad \mu(\mu(\delta, Params \mapsto params), Temps \mapsto temps)
\end{array}
$$

The primitive for activating blocks is the same for both kinds:

$$
\begin{array}{l}
value\_primitive : Primitive\_method \\
value\_primitive \quad \triangleq \\
\quad \lambda\nu \cdot \lambda closure, args, \sigma \cdot body(closure, \sigma)\nu(closure, args, \sigma)
\end{array}
$$

5.2.2 Other Types of Block

The continuation model of blocks described in §5.2 is a more "generous" model than actually provided by Smalltalk, in that some programs that have valid meanings according to the model of §5.2 cause run-time errors in Smalltalk. In Smalltalk, dynamic environments are first-class objects (called *contexts*), and they encode run-time state information (such as the bindings within a *DEnv*), and operational information (such as an instruction pointer). Moreover, the Smalltalk Virtual Machine does not allow one context to return control to a second when the second has already returned control to a third. Hence, activation of a block such as that returned from the following method always causes a run-time error:

```
errorBlock
    ↑[↑self]
```

This is because the home context of the block has already exited when the block is activated.

This restricted form of continuation-blocks can be modelled using a less general form of semantic description than continuations: the "exit" semantics of VDM provide sufficient descriptive power [BJ82]. In an exit semantics, the meaning functions from a direct semantics are modified to return an extra result, which indicates whether that particular construct, or one invoked by it, caused an abnormal exit. Furthermore, the exit value also indicates where processing should resume.

To include the exit model of blocks into the direct semantics of Chapter 3, each dynamic environment is tagged with a unique identifier when it is created, and a set of active environment identifiers is maintained as part of the store. When a method returns, its identifier is removed from the active set, and if another attempt is made to activate its dynamic environment (by returning to it), an error is raised.

This scheme requires that:

- dynamic environments are tagged with identifiers,
- active identifiers are kept in the object memory,

- denotations of methods and expressions are changed to include exit values, and
- exit values are checked when a dynamic environment is exited.

$$
\begin{aligned}
DEnv :: \quad & Rcvr : Oop \\
& Params : Id \xrightarrow{m} Oop \\
& Temps : Id \xrightarrow{m} Oop \\
& Id : DEnv\_id
\end{aligned}
$$

$$
\begin{aligned}
Object\_memory :: \quad & Objects : Oop \xrightarrow{m} Object \\
& Active : DEnv\_id\text{-set}
\end{aligned}
$$

$$Method\_den = Oop \times Oop^* \times Object\_memory \to [ExitV] \times Oop \times Object\_memory$$

$$ExitV = DEnv\_id$$

When a "normal" exit occurs, the exit value is nil; otherwise it is the identifier of the environment in which the continuation block was bound.

$$
\begin{aligned}
& MMethod\_body : Method\_body \to SEnv \to Method\_den \\
& MMethod\_body[\![mk\text{-}Method\_body(params, temps, expr)]\!]\,\rho \quad \triangleq \\
& \quad \lambda rcvr, args, \sigma \cdot \\
& \qquad \textsf{let } (d\_id, \sigma') = make\_new\_denv\_id(\sigma) \textsf{ in} \\
& \qquad \textsf{let } bindings = bind\_args(params, args) \textsf{ in} \\
& \qquad \textsf{let } \delta = mk\text{-}DEnv(rcvr, bindings, initialise(temps), d\_id) \textsf{ in} \\
& \qquad \textsf{let } (exitv, result, \delta', \sigma'') = MExpression[\![expr]\!]\,\rho\delta\sigma' \textsf{ in} \\
& \qquad \textsf{let } \sigma''' = remove\_id(d\_id, \sigma'') \textsf{ in} \\
& \qquad \textsf{if } exitv = d\_id \\
& \qquad \textsf{then } (\textsf{nil}, result, \sigma''') \\
& \qquad \textsf{else } (exitv, result, \sigma''')
\end{aligned}
$$

$$
\begin{aligned}
& make\_new\_denv\_id\ ()\ new\_id\text{:}\ DEnv\_id \\
& \textsf{ext wr } \sigma \ : \ Object\_memory \\
& \textsf{post } new\_id \notin Active(\overleftarrow{\sigma}) \\
& \qquad\qquad \wedge\ \sigma = \mu(\overleftarrow{\sigma}, Active \mapsto Active(\overleftarrow{\sigma}) \cup \{new\_id\})
\end{aligned}
$$

$$remove\_id : DEnv\_id \times Object\_memory \to Object\_memory$$
$$remove\_id(d\_id, \sigma) \triangleq \mu(\sigma, Active \mapsto Active(\sigma) - \{d\_id\})$$

$$MCont\_body : SEnv \to DEnv \to Block\_den$$
$$MCont\_body\,[\![mk\text{-}Cont\_body(params, temps, expr)]\!]\,\rho\delta \triangleq$$
$$\lambda closure, args, \sigma \cdot$$
if len $params \neq$ len $args$
then $block\_arg\_error(\rho, \delta, closure, args, \sigma)$
else let $\delta' = update\_vars(\delta,$
$Temps(\delta) \dagger Params(\delta)$
$\dagger\ bind\_args(params, args),$
$initialise(temps))$ in
let $(exitv, result, \delta'', \sigma') =$
$MExpression\,[\![expr]\!]\,\rho\delta'\sigma$ in
if $exitv \neq$ nil
then if $Id(\delta'') \in Active(\sigma')$
then $(exitv, result, \sigma')$
else $return\_error(result, d\_id, \sigma')$
else $(Id(\delta''), result, \sigma')$

Atomic meaning functions (those that do not invoke other meaning functions) must be modified to generate a normal exit value; non-atomic meaning functions must check for exit values from their component constructs. For example, the meaning functions for assignment and sequential composition are as follows (cf. p. 40):[8]

[8] Much of the extra notational overhead required to check exit values can be relieved by defining suitable combinators, as in [BJ82].

$$
\begin{aligned}
&MExpression : Expression \rightarrow SEnv \rightarrow DEnv \rightarrow \\
&\qquad Object\_memory \rightarrow \\
&\qquad\qquad [ExitV] \times Oop \times DEnv \times Object\_memory \\
&MExpression\,[\![mk\text{-}Assignment(id, rhs)]\!]\,\rho\delta\sigma \;\triangleq \\
&\quad \text{let } (exitv, result, \delta', \sigma') = MExpression\,[\![rhs]\!]\,\rho\delta\sigma \text{ in} \\
&\quad \text{if } exitv \neq \text{nil} \\
&\quad \text{then } (exitv, result, \delta', \sigma') \\
&\quad \text{else cases } id \text{ of} \\
&\qquad mk\text{-}Temp\_id(t) \rightarrow (\text{nil}, result, \\
&\qquad\qquad update\_temp(t, result, \delta'), \sigma') \\
&\qquad mk\text{-}Inst\_var\_id(iv) \rightarrow (\text{nil}, result, \delta', \\
&\qquad\qquad update\_inst\_var(iv, Rcvr(\delta), \\
&\qquad\qquad\quad result, Class(\rho), \sigma')) \\
&\qquad \text{end}
\end{aligned}
$$

$$
\begin{aligned}
&MExpression\,[\![mk\text{-}Expression\_list(exprs)]\!]\,\rho\delta\sigma \;\triangleq \\
&\quad \text{let } (exitv, oop, \delta', \sigma') = MExpression\,[\![\text{hd } exprs]\!]\,\rho\delta\sigma \text{ in} \\
&\quad \text{if len } exprs = 1 \vee exitv \neq \text{nil} \\
&\quad \text{then } (exitv, oop, \delta', \sigma') \\
&\quad \text{else } MExpression\,[\![mk\text{-}Expression\_list(\text{tl } exprs)]\!]\,\rho\delta'\sigma'
\end{aligned}
$$

In fact, this model is not an accurate model of Smalltalk block semantics, as a non-local return "unwinds the stack" between the block and its home context, rendering all intermediate contexts inaccessible; in Smalltalk, the intermediate contexts are still available for use. This can be modelled using modifications to the earlier continuation semantics.

5.3 Dynamic Environments as First-Class Objects

As was mentioned in the previous section, Smalltalk treats every dynamic environment as a first-class object, called a *context*. This has a number of advantages:

- it simplifies the structure of *Object_memory* so that again it has only one structured component,

- each context gains its own identity "for free" so that no special domain of dynamic environment identifiers is required, and
- extra primitives can interrogate contexts facilitating the construction of tools such as debuggers entirely within the language.

It is straightforward to extend the domain of primitive objects to include contexts:

$$Primitive\_object = \ldots \cup Context$$

$$\begin{aligned} Context :: Sender\_ctx &: Oop \\ Rcvr &: Oop \\ Params &: Id \xrightarrow{m} Oop \\ Temps &: Id \xrightarrow{m} Oop \end{aligned}$$

Alternatively, a *Plain_object* can be used to represent a context if the instance variable identifiers of the context are suitably named to avoid conflicts. For example, arguments might be prefixed with arg, temporaries with temp, and the receiver and sender context fields named by self and sender. However, because all contexts have different instance variable names, no class can be defined to access them. Yet another alternative is to access the arguments and temporaries using indices (i.e., use indexable objects).

An additional component of the object memory identifies which object is the "current" or "active" context:

$$\begin{aligned} Object\_memory :: Objects &: Oop \xrightarrow{m} Object \\ Active &: Oop \end{aligned}$$

This component must be modified when a message is sent or when a method or block returns. In effect, the control stack is explicitly represented in the object memory rather than implicitly in the function calls used in the semantic description.

5.4 Primitives

Other useful block primitives can be defined in addition to the "value" primitive (pp. 115, 123). Of the primitives in Chapter 3, only the *equivalence* and

oopOf primitives (§3.9.1) are applicable to blocks, but the argument in §3.9.1 that equivalence between primitive objects should be based on equality of the underlying objects cannot apply to blocks, as showing that two functions are equal is generally undecidable. Also, primitives can be provided to access the internal state of a context (e.g., for debugging).

5.5 Concurrency

As the introduction stated, no attempt is made in this thesis to integrate concurrent features into the semantics. Nevertheless, it can be pointed out that are three obvious points at which concurrency could be introduced. The first occurs when evaluating a block; a primitive can be introduced that evaluates a block in parallel—this is the fork facility in Smalltalk. Second, a message send may be performed concurrently, so that the method invoked by the message runs in parallel with sending method—this is the actor approach (§2.5.1). The third approach is to have each object as an independent process, the concurrency being introduced by the "new" primitive—this is how POOL works (§2.4).

Regarding the semantic definition of these features, the difficulties lie not in the introduction of concurrency, but in the description of synchronisation and interference. Again, there are several approaches. One is to allow multiple processes to access an object simultaneously, interleaving accesses to the object's instance variables (the Smalltalk approach). Another is to treat each object as a monitor (cf. [Hoa74]), queuing processes at each object (the POOL approach). And as in the actor model, one can ban individual assignment to instance variables.

The formal description of these approaches, perhaps by the technique of powerdomains [Plo76], is a topic for future research.

Chapter 6

Conclusions

> Nothing will ever be attempted, if all possible objections must be first overcome.
>
> *Samuel Johnson*

In approaching the task of defining the formal semantics of an object-oriented programming language, this thesis has provided semantic definitions for the features of object-oriented languages that differentiate them from other languages. In Chapter 3, the essential characteristics of object-oriented languages were defined: objects and object identity. Additionally, it was shown that two different and equally valid ways were available to organise objects: classes and prototypes.

In Chapter 4, the organisation of classes was considered, and various inheritance schemes described. It was shown that single inheritance, and the graph and linear forms of multiple inheritance, do not require any changes to the semantic domains used in Chapter 3, indicating that these particular organisations of classes are largely syntactic in nature. However, the tree form of inheritance requires different object structure.

Chapter 5 described an object-oriented control structure, the block, and showed how it could be used, in conjunction with message passing, as a general control structure.

The aim of this thesis was to show that an apposite model of object-oriented languages existed, and that this model is both conceptually simple and sufficiently general. It is the author's belief that this aim has been met. A secondary aim was to use a semantic study to investigate the choices available to the designer of an object-oriented language, and assist her in making design decisions. By describing in detail the differences between classes and prototypes, between single and multiple inheritance, between graph, linear and tree inheritance, and between the various kinds of block, it is hoped that the secondary aim has also been fulfilled. All that remains is to point to future directions of research.

6.1 Future Research

An obvious follow-on from this work is to devise proof rules for a similar language, and show that these rules are sound with respect to denotational semantics [Ame86b]. This would be the next step on the route to proving properties about programs in the language. Furthermore, because of the strong connection between object-oriented systems and persistence, it is important to study how the semantics of programs and data change as programs are modified.

A fertile area for research concerns the definition of type systems for object-oriented languages. The language presented in this thesis types objects rather than storage cells, a fundamentally different approach to that used in conventional languages. Although conventional typing mechanisms have been used for object-oriented languages, it is the author's belief that adding conventional type declaration facilities to the language presented in this thesis will substantially decrease the ability to re-use code by inheritance. It remains to be seen whether useful and convenient typing schemes can be found for such languages.

Appendix A

The Direct Semantics of a Complete Language

> In relation to their systems most systematizers are like a man who builds an enormous castle and lives in a shack close by; they do not live in their own enormous systematic buildings.
>
> *Sören Kierkegaard*

This appendix gathers together the definitions in Chapters 3 and 4, together with the semantics of closures as in §5.1, into a single language. The following selection of features has been made (see the appropriate sections for further explanation):

- Booleans, integers, symbols and nil are available as literals
- a separate class of indexable objects is available (§3.10.4)

- objects that represent classes are present in the object memory; instances are created by sending messages to the appropriate class (§3.11)
- there are no class or global variables (§3.11.1)
- multiple inheritance (the "tree" form) is used (§4.3.3)
- methods may be designated as public, private or subclass-visible (§4.4.1)
- statically-bound sends to parent classes or to self are available (§4.4.2)
- closures and the *value* primitive are provided (§5.1)

A.1 Abstract Syntax

$$Program' = Class\_map'$$

$$Class\_map' = Class\_name \xrightarrow{m} Class\_body'$$

$$\begin{aligned} Class\_body' :: \; & Instvars : Id\text{-set} \\ & Methods : Method\_map' \\ & Parents : Parent\_classes \end{aligned}$$

$$Method\_map' = Selector \xrightarrow{m} Method\_desc'$$

$$Parent\_classes = Class\_name\text{-set}$$

$$\begin{aligned} Method\_desc' :: \; & Method : Method\_body' \cup Primitive\_method \\ & Type : Method\_type \end{aligned}$$

$$Method\_type = \{\textsc{Private}, \textsc{SubV}, \textsc{Public}\}$$

$$\begin{aligned} Method\_body' :: \; & Params : Ulist(Id) \\ & Temps : Id\text{-set} \\ & Expr : Expression' \end{aligned}$$

$$Expression' = Expression\_list' \cup Assignment' \cup Object\_name \cup Message' \cup Super\_send \cup Self\_send \cup Literal\_object'$$

$$Expression\_list' :: Expression'^*$$

$$Assignment' :: LHS : AVar\_id \quad RHS : Expression'$$

$$AVar\_id = Temp\_id \cup Inst\_var\_id$$

$$Var\_id = Arg\_id \cup Temp\_id \cup Inst\_var\_id$$

$$Arg\_id :: Id$$

$$Temp\_id :: Id$$

$$Inst\_var\_id :: Id$$

$$Object\_name' = Var\_id \cup \{\text{SELF}\}$$

$$Message' :: Rcvr : Expression' \quad Sel : Selector \quad Args : Expression'^*$$

$$Super\_send :: Sel : Selector \quad Args : Expression'^* \quad Parent : Class\_name$$

$$Self\_send :: Sel : Selector \quad Args : Expression'^*$$

$$Literal\_object' = Block\_body' \cup Int\_literal \cup Bool\_literal \cup Symbol\_literal \cup Nil\_literal$$

$$Block\_body' :: Params : Ulist(Id) \quad Temps : Id\text{-set} \quad Expr : Expression'$$

$$Int\_literal :: \mathbb{Z}$$

$$Bool\_literal :: \mathbb{B}$$

$Symbol\_literal = Selector$

$Nil\_literal :: \{\}$

$Selector = Unary \cup Binary \cup Keyword$

$Unary :: Id$

$Binary :: \{+, -, *, /, <, \ldots\}$

$Keyword :: Id^*$

$nargs : Selector \to \mathbb{N}$

$nargs(sel) \triangleq$ cases sel of
$mk\text{-}Unary(\_) \to 0$
$mk\text{-}Binary(\_) \to 1$
$mk\text{-}Keyword(ids) \to \text{len } ids$
end

A.2 Context Conditions

$CCEnv ::$
$Instvars : Id\text{-set}$
$Params : Id\text{-set}$
$Temps : Id\text{-set}$
$Inherited\_selectors : Class\_name \xrightarrow{m} Selector\text{-set}$
$My\_selectors : Selector\text{-set}$

$$
\begin{array}{l}
WFProgram : Program' \to \mathbb{B} \\
WFProgram[\![class\_map]\!] \triangleq \\
\quad non\_circular(class\_map) \\
\quad \wedge\ \forall class\_id \in \mathsf{dom}\, class\_map \cdot \\
\qquad \mathsf{let}\ par = parents(class\_id, class\_map)\ \mathsf{in} \\
\qquad \mathsf{let}\ inherited\_selectors = \\
\qquad\quad \{p \mapsto sv\_selectors(p, class\_map) \mid p \in par\}\ \mathsf{in} \\
\qquad \mathsf{let}\ my\_sels = local\_selectors(class\_id, class\_map)\ \mathsf{in} \\
\qquad \mathsf{let}\ \theta = mk\text{-}CCEnv(\_, \_, \_, inherited\_selectors, my\_sels)\ \mathsf{in} \\
\qquad par \subset \mathsf{dom}\, class\_map \\
\qquad \wedge\ non\_conflicting\_methods(class\_id, par, class\_map) \\
\qquad \wedge\ WFClass[\![class\_map(class\_id)]\!]\,\theta
\end{array}
$$

$$
\begin{array}{l}
non\_circular : Class\_map' \to \mathbb{B} \\
non\_circular(class\_map) \triangleq \\
\quad \forall classes \subseteq \mathsf{dom}\, class\_map \cdot \\
\qquad classes \neq \{\} \Rightarrow \\
\qquad\quad \exists c_1 \in classes \cdot \\
\qquad\qquad \forall c_2 \in classes \cdot c_2 \notin parents(c_1, class\_map)
\end{array}
$$

$$
\begin{array}{l}
parents : Class\_name \times Class\_map' \to Parent\_classes \\
parents(class, class\_map) \triangleq Parents(class\_map(class))
\end{array}
$$

$$
\begin{array}{l}
non\_conflicting\_methods : Class\_name \times \\
\qquad\qquad\qquad\qquad Class\_name\text{-}\mathsf{set} \times Class\_map' \to \mathbb{B} \\
non\_conflicting\_methods(c, parents, class\_map) \triangleq \\
\quad \forall sel \in conflicting\_selectors(parents, class\_map) \cdot \\
\qquad sel \in local\_selectors(c, class\_map)
\end{array}
$$

$$conflicting\_selectors : Class\_name\text{-set} \times Class\_map' \rightarrow Selector\text{-set}$$

$$conflicting\_selectors(classes, class\_map) \triangleq$$
$$\quad \text{let } sel\_set = \bigcup\{sv\_selectors(p, class\_map) \mid p \in classes\} \text{ in}$$
$$\quad \{sel \in sel\_set \mid$$
$$\qquad \exists c_1, c_2 \in classes \cdot$$
$$\qquad\quad c_1 \neq c_2 \wedge sel \in sv\_selectors(c_1, class\_map)$$
$$\qquad\qquad \wedge sel \in sv\_selectors(c_2, class\_map)\}$$

$$local\_selectors : Class\_name \times Class\_map' \rightarrow Selector\text{-set}$$
$$local\_selectors(class\_id, class\_map) \triangleq$$
$$\quad \text{dom } Methods(class\_map(class\_id))$$

$$sv\_selectors : Class\_name \times Class\_map' \rightarrow Selector\text{-set}$$
$$sv\_selectors(class\_id, class\_map) \triangleq$$
$$\quad all\_sels\_type(class\_id, class\_map, \text{PUBLIC})$$
$$\qquad \cup all\_sels\_type(class\_id, class\_map, \text{SUBV})$$

$$all\_sels\_type : Class\_name \times Class\_map' \times \{\text{PUBLIC}, \text{SUBV}\} \rightarrow Selector\text{-set}$$
$$all\_sels\_type(class\_id, class\_map, type) \triangleq$$
$$\quad \text{let } others = \bigcup\{local\_sels\_type(class\_id, class\_map, type)$$
$$\qquad\qquad \mid t \in Method\_type - \{t\}\} \text{ in}$$
$$\quad inh\_sels\_type(class\_id, class\_map, type) - others$$
$$\qquad \cup local\_sels\_type(class\_id, class\_map, type)$$

$$local\_sels\_type : Class\_name \times Class\_map' \times Method\_type \rightarrow Selector\text{-set}$$
$$local\_sels\_type(class\_id, class\_map, type) \triangleq$$
$$\quad \{sel \in local\_selectors(class\_id, class\_map)$$
$$\qquad \mid Type(Methods(class\_map(class\_id))) = type\}$$

$$inh\_sels\_type : Class\_name \times Class\_map' \times \{\text{PUBLIC}, \text{SUBV}\} \to Selector\text{-set}$$

$$inh\_sels\_type(class\_id, class\_map, type) \triangleq \bigcup\{all\_sels\_type(p, class\_map, type) \mid p \in parents(class\_id, class\_map)\}$$

$$all\_selectors : Class\_name \times Class\_map' \to Selector\text{-set}$$

$$all\_selectors(class\_id, class\_map) \triangleq \bigcup\{all\_sels\_type(class\_id, class\_map, t) \mid t \in Method\_type\}$$

$$WFClass : Class\_body' \to CCEnv \to \mathbb{B}$$

$$WFClass[\![mk\text{-}Class\_body'(iv, meths, \_)]\!]\,\theta \triangleq \forall sel \in \mathsf{dom}\, meths \cdot \mathsf{let}\ m = Method(meths(sel))\ \mathsf{in}\ m \notin Primitive\_method \Rightarrow nargs(sel) = \mathsf{len}\, Params(m) \wedge WFMethod[\![m]\!]\,\mu(\theta, Instvars \mapsto iv)$$

$$WFMethod : Method\_body' \to CCEnv \to \mathbb{B}$$

$$WFMethod[\![mk\text{-}Method\_body'(params, temps, expr)]\!]\,\theta \triangleq \mathsf{let}\ \theta' = \mu(\mu(\theta, Params \mapsto \mathsf{rng}\, params), Temps \mapsto temps)\ \mathsf{in}\ WFExpression[\![expr]\!]\,\theta'$$

$$WFExpression : Expression' \to CCEnv \to \mathbb{B}$$

$$WFExpression[\![mk\text{-}Expression\_list'(exprs)]\!]\,\theta \triangleq \mathsf{len}\, exprs \geq 1 \wedge \forall e \in \mathsf{rng}\, exprs \cdot WFExpression[\![e]\!]\,\theta$$

$$WFExpression\,[\![mk\text{-}Assignment'(id, rhs)]\!]\,\theta \triangleq$$
$$\quad WFExpression\,[\![rhs]\!]\,\theta \wedge$$
$$\quad \textsf{cases } id \textsf{ of}$$
$$\quad mk\text{-}Inst\_var\_id(iv) \rightarrow iv \in Instvars(\theta)$$
$$\quad mk\text{-}Temp\_id(t) \rightarrow t \in Temps(\theta)$$
$$\quad \textsf{end}$$

$$WFExpression\,[\![mk\text{-}Inst\_var\_id(id)]\!]\,\theta \triangleq id \in Instvars(\theta)$$

$$WFExpression\,[\![mk\text{-}Arg\_id(id)]\!]\,\theta \triangleq id \in Params(\theta)$$

$$WFExpression\,[\![mk\text{-}Temp\_id(id)]\!]\,\theta \triangleq id \in Temps(\theta)$$

$$WFExpression\,[\![\textsc{Self}]\!]\,\theta \triangleq \textsf{true}$$

$$WFExpression\,[\![mk\text{-}Message'(rcvr, sel, args)]\!]\,\theta \triangleq$$
$$\quad WFExpression\,[\![rcvr]\!]\,\theta$$
$$\quad \wedge\ \textsf{len}\,args = nargs(sel)$$
$$\quad \wedge\ \forall arg \in \textsf{rng}\,args \cdot WFExpression\,[\![arg]\!]\,\theta$$

$$WFExpression\,[\![mk\text{-}Super\_send(sel, args, parent)]\!]\,\theta \triangleq$$
$$\quad \textsf{len}\,args = nargs(sel)$$
$$\quad \wedge\ parent \in \textsf{dom}\,Inherited\_selectors(\theta)$$
$$\quad \wedge\ sel \in Inherited\_selectors(\theta)(parent)$$
$$\quad \wedge\ \forall arg \in \textsf{rng}\,args \cdot WFExpression\,[\![arg]\!]\,\theta$$

$$WFExpression\,[\![mk\text{-}Self\_send(sel, args)]\!]\,\theta \triangleq$$
$$\quad \textsf{len}\,args = nargs(sel)$$
$$\quad \wedge\ sel \in My\_selectors(\theta) \cup \bigcup \textsf{rng}\,Inherited\_selectors(\theta)$$
$$\quad \wedge\ \forall arg \in \textsf{rng}\,args \cdot WFExpression\,[\![arg]\!]\,\theta$$

$$WFExpression\,[\![mk\text{-}Block\_body'(params, temps, expr)]\!]\,\theta \triangleq$$
$$\quad \text{let } \theta' = \mu(\mu(\theta, Params \mapsto Params(\theta) \cup \text{rng}\, params \cup Temps(\theta)),$$
$$\qquad\qquad Temps \mapsto temps) \text{ in}$$
$$\quad WFExpression\,[\![expr]\!]\,\theta'$$

$$WFExpression\,[\![l]\!]\,\theta \triangleq \text{true}$$
$$\quad \text{for } l \in Int\_literal \cup Bool\_literal \cup Symbol\_literal \cup Nil\_literal$$

A.3 Processed Abstract Syntax

$$Program = Class\_map$$

$$Class\_map = Class\_name \xrightarrow{m} Class\_body$$

$$\begin{aligned} Class\_body :: \; & Instvars : Instvar\_desc \\ & Methods : Method\_map \end{aligned}$$

$$\begin{aligned} Instvar\_desc :: \; & Local\_instvars : Id\text{-set} \\ & Inherited : Class\_name \xrightarrow{m} Instvar\_desc \end{aligned}$$

$$Method\_map = Selector \xrightarrow{m} Method\_desc$$

$$\begin{aligned} Method\_desc :: \; & Method : Method\_body \cup Primitive\_method \\ & Type : Method\_type \\ & Original\_class : Access\_path \end{aligned}$$

$$Access\_path = Ulist(Class\_name)$$

$$\begin{aligned} Method\_body :: \; & Params : Ulist(Id) \\ & Temps : Id\text{-set} \\ & Expr : Expression \end{aligned}$$

$$\begin{aligned} Expression = \; & Expression\_list \cup Assignment \cup Object\_name \\ & \cup Message \cup Static\_send \cup Literal\_object \end{aligned}$$

$$Expression\_list :: Expression^*$$

$$\begin{array}{rcl} \mathit{Assignment} :: & \mathit{LHS} & : \mathit{AVar\_id} \\ & \mathit{RHS} & : \mathit{Expression} \end{array}$$

$$\begin{array}{rcl} \mathit{Message} :: & \mathit{Rcvr} & : \mathit{Expression} \\ & \mathit{Sel} & : \mathit{Selector} \\ & \mathit{Args} & : \mathit{Expression}^* \end{array}$$

$$\begin{array}{rcl} \mathit{Static\_send} :: & \mathit{Sel} & : \mathit{Selector} \\ & \mathit{Args} & : \mathit{Expression}^* \\ & \mathit{Class} & : \mathit{Class\_name} \end{array}$$

$$\mathit{Literal\_object} = \mathit{Block\_body} \cup \mathit{Int\_literal} \cup \mathit{Bool\_literal} \cup \mathit{Symbol\_literal} \cup \mathit{Nil\_literal}$$

$$\begin{array}{rcl} \mathit{Block\_body} :: & \mathit{Params} & : \mathit{Ulist}(\mathit{Id}) \\ & \mathit{Temps} & : \mathit{Id}\text{-set} \\ & \mathit{Expr} & : \mathit{Expression} \end{array}$$

A.4 Processing functions

$$\mathit{PProgram} : \mathit{Program}' \to \mathit{Program}$$
$$\mathit{PProgram}[\![p]\!] \triangleq \{\mathit{class\_id} \mapsto \mathit{PClass}[\![\mathit{class\_id}]\!]p \mid \mathit{class\_id} \in \mathsf{dom}\, p\}$$

$$\mathit{PClass} : \mathit{Class\_name} \to \mathit{Class\_map}' \to \mathit{Class\_body}$$
$$\mathit{PClass}[\![\mathit{class\_id}]\!]\,\mathit{class\_map} \triangleq \mathit{mk\text{-}Class\_body}(\mathit{inst\_vars}(\mathit{class\_id}, \mathit{class\_map}), \mathit{all\_methods\_of}(\mathit{class\_id}, \mathit{class\_map}))$$

$$\mathit{inst\_vars} : \mathit{Class\_name} \times \mathit{Class\_map}' \to \mathit{Instvar\_desc}$$
$$\mathit{inst\_vars}(\mathit{class\_id}, \mathit{class\_map}) \triangleq \mathit{mk\text{-}Instvar\_desc}(\mathit{Instvars}(\mathit{class\_map}(\mathit{class\_id})), \{p \mapsto \mathit{inst\_vars}(p, \mathit{class\_map}) \mid p \in \mathit{parents}(\mathit{class\_id}, \mathit{class\_map})\})$$

$$all\_methods\_of : Class\_name \times Class\_map' \to Method\_map$$
$$all\_methods\_of(class\_id, class\_map) \triangleq$$
$$\quad local\_meths\_type(class\_id, class\_map, \text{PRIVATE})$$
$$\quad \cup\ all\_meths\_type(class\_id, class\_map, \text{PUBLIC})$$
$$\quad \cup\ all\_meths\_type(class\_id, class\_map, \text{SUBV})$$

$$local\_meths\_type : Class\_name \times Class\_map' \times Method\_type \to Method\_map$$
$$local\_meths\_type(class\_id, class\_map, type) \triangleq$$
$$\quad \textsf{let}\ methods = Methods(class\_map(class\_id))\ \textsf{in}$$
$$\quad \{sel \mapsto mk\text{-}Method\_desc($$
$$\qquad PMethod[\![Method(methods(sel))]\!]class\_id, type, [\,])$$
$$\qquad \mid Type(methods(sel)) = type\}$$

$$all\_meths\_type : Class\_name \times Class\_map' \times Method\_type \to Method\_map$$
$$all\_meths\_type(class\_id, class\_map, type) \triangleq$$
$$\quad \textsf{let}\ other\_types = Method\_type - \{type\}\ \textsf{in}$$
$$\quad \textsf{let}\ others = \textsf{dom} \bigcup\{local\_meths\_type(class\_id, class\_map, t)$$
$$\qquad \mid t \in other\_types\}\ \textsf{in}$$
$$\quad (others \lhd inh\_meths\_type(class\_id, class\_map, type))$$
$$\qquad \dagger\ local\_meths\_type(class\_id, class\_map, type)$$

$$inh\_meths\_type : Class\_name \times Class\_map' \times Method\_type \to Method\_map$$
$$inh\_meths\_type(class\_id, class\_map, type) \triangleq$$
$$\quad \biguplus\{methods\_inherited\_from(p, class\_map, type)$$
$$\qquad \mid p \in parents(class\_id, class\_map)\}$$

$$methods\_inherited\_from : Class\_name \times Class\_map' \times Method\_type \rightarrow Method\_map$$
$$methods\_inherited\_from(class\_id, class\_map, type) \triangleq$$
let $meth\_map = all\_meths\_type(class\_id, class\_map, type)$ in
$$\{sel \mapsto prepend\_class(class\_id, meth\_map(sel)) \mid sel \in \mathsf{dom}\, meth\_map\}$$

$$prepend\_class : Class\_name \times Method\_desc \rightarrow Method\_desc$$
$$prepend\_class(c, mdesc) \triangleq$$
$$\mu(mdesc, Original\_class \mapsto [c]^\frown Original\_class(mdesc))$$

$$PMethod : (Method\_body' \cup Primitive\_method) \rightarrow Class\_name \rightarrow (Method\_body \cup Primitive\_method)$$
$$PMethod\,[\![m]\!]\,this\_class \triangleq$$
if $m \in Primitive\_method$
then m
else let $mk\text{-}Method\_body'(params, temps, expr) = m$ in
$$mk\text{-}Method\_body(params, temps, PExpression\,[\![expr]\!]\,this\_class)$$

$PExpression : Expression' \to Class\_name \to Expression$
$PExpression\ [\![expr]\!]\ this\_class \triangleq$
cases $expr$ of
$mk\text{-}Expression\_list'(e) \to mk\text{-}Expression\_list(\{i \mapsto PExpression\ [\![e(i)]\!]\ this\_class \mid i \in \mathsf{dom}\, e\})$
$mk\text{-}Assignment'(l, r) \to mk\text{-}Assignment(l, PExpression\ [\![r]\!]\ this\_class)$
$mk\text{-}Message'(r, s, a) \to mk\text{-}Message(PExpression\ [\![r]\!]\ this\_class, s, PExpression\_list\ [\![a]\!]\ this\_class)$
$mk\text{-}Super\_send(s, a, p) \to mk\text{-}Static\_send(s, PExpression\_list\ [\![a]\!]\ this\_class, p)$
$mk\text{-}Self\_send(s, a) \to mk\text{-}Static\_send(s, PExpression\_list\ [\![args]\!]\ this\_class, this\_class)$
$mk\text{-}Block\_body'(p, t, e) \to mk\text{-}Block\_body(p, t, PExpression\ [\![e]\!]\ this\_class)$
others $expr$
end

A.5 Semantic Domains

$Object\_memory = Oop \xrightarrow{m} Object$

$Object :: Class : Oop$
$\quad Body : Object\_body$

$Object\_body = Plain\_object \cup Primitive\_object$

$Plain\_object :: Local\_instvars : Id \xrightarrow{m} Oop$
$\quad Inherited : Class\_name \xrightarrow{m} Plain\_object$

$Primitive\_object = \mathbb{Z} \cup Symbol \cup Indexable\_object \cup Class\_obj \cup Block\_den$

$Indexable\_object = \mathbb{N} \xrightarrow{m} Oop$

$Symbol = Selector$

$Class\_obj :: \quad Name : Class\_name$
$\qquad\qquad Instvars : Instvar\_desc$

$Block\_den = Method\_response$

$class : Oop \times Object\_memory \to Class\_name$
$class(oop, \sigma) \triangleq$
$\quad$ let $class\_oop = Class(\sigma(oop))$ in $Name(body(class\_oop, \sigma))$

$body : Oop \times Object\_memory \to Object\_body$
$body(oop, \sigma) \triangleq Body(\sigma(oop))$

$inst\_var : Id \times Oop \times Access\_path \times Object\_memory \to Oop$
$inst\_var(id, oop, path, \sigma) \triangleq$
$\quad Local\_instvars(sub\_body(oop, path, \sigma))(id)$

$sub\_body : Oop \times Access\_path \times Object\_memory \to Plain\_object$
$sub\_body(oop, path, \sigma) \triangleq sub\_obj(body(oop, \sigma), path)$

$sub\_obj : Plain\_object \times Access\_path \to Plain\_object$
$sub\_obj(pobj, path) \triangleq$
$\quad$ if $path = [\,]$ then $pobj$
$\quad$ else $sub\_obj(Inherited(pobj)(\mathsf{hd}\, path), \mathsf{tl}\, path)$

$update\_inst\_var : Id \times Oop \times Oop \times Access\_path \times Object\_memory$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to Object\_memory$
$update\_inst\_var(id, oop, value, path, \sigma) \triangleq$
$\quad \sigma \dagger \{oop \mapsto \mu(\sigma(oop), Body \mapsto$
$\qquad\qquad update\_body(body(oop, \sigma), id, value, path))\}$

$$update\_body : Plain\_object \times Id \times Oop \times Access\_path \rightarrow Plain\_object$$

$$update\_body(pobj, id, value, path) \triangleq$$
$$\quad \text{if } path = [\,] \text{ then } \mu(pobj, Local\_instvars \mapsto Local\_instvars(pobj) \dagger \{id \mapsto value\})$$
$$\quad \text{else let } inh = Inherited(pobj) \text{ in}$$
$$\quad\quad \text{let } new\_part = update\_body(inh(\mathsf{hd}\, path), id, value, \mathsf{tl}\, path) \text{ in}$$
$$\quad\quad \mu(pobj, Inherited \mapsto inh \dagger \{\mathsf{hd}\, path \mapsto new\_part\})$$

A.6 Meaning Functions

$$Program\_den = Class\_name \xrightarrow{m} Class\_den$$

$$MProgram : Program \rightarrow Program\_den$$
$$MProgram[\![p]\!] \triangleq$$
$$\quad \text{let } pd = \{c \mapsto MClass\_body[\![p(c)]\!]pd \mid c \in \mathsf{dom}\, p\} \text{ in}$$
$$\quad pd$$

$$Class\_den = Selector \xrightarrow{m} Method\_den$$

$$Method\_den = Public\_method \cup Private\_method$$

$$Public\_method :: Meth : Method\_response$$

$$Private\_method :: Meth : Method\_response$$

$$Method\_response = Oop \times (Oop^*) \times Object\_memory \rightarrow Oop \times Object\_memory$$

$$SEnv :: PD : Program\_den$$
$$\quad\quad Class : Access\_path$$

$$MClass\_body : Class\_body \to Program\_den \to Class\_den$$
$$MClass\_body\,[\![mk\text{-}Class\_body(iv, meths)]\!]\,pd \triangleq$$
$$\quad \{sel \mapsto MMethod\,[\![meths(sel)]\!]\,pd \mid sel \in \mathsf{dom}\, meths\}$$

$$MMethod : Method\_desc \to Program\_den \to Method\_den$$
$$MMethod\,[\![mm]\!]\,pd \triangleq$$
$$\quad \mathsf{let}\ m = Method(mm)\ \mathsf{in}$$
$$\quad \mathsf{let}\ md = \mathsf{if}\ m \in Primitive\_method$$
$$\qquad \mathsf{then}\ m$$
$$\qquad \mathsf{else\ let}\ \rho = mk\text{-}SEnv(pd, Original\_class(mm))\ \mathsf{in}$$
$$\qquad\quad MMethod\_body\,[\![m]\!]\,\rho$$
$$\quad \mathsf{in}$$
$$\quad \mathsf{if}\ Type(mm) \in \{\textsc{Private}, \textsc{SubV}\}$$
$$\quad \mathsf{then}\ mk\text{-}Private\_meth(md)$$
$$\quad \mathsf{else}\ mk\text{-}Public\_meth(md)$$

$$Primitive\_method = Method\_response$$

$$\begin{array}{lrcl} DEnv & :: & Rcvr & : & Oop \\ & & Args & : & Id \xrightarrow{m} Oop \\ & & Temps & : & Id \xrightarrow{m} Oop \end{array}$$

$$update\_vars : DEnv \times Id \xrightarrow{m} Oop \times Id \xrightarrow{m} Oop \to DEnv$$
$$update\_vars(\delta, params, temps) \triangleq$$
$$\quad \mu(\mu(\delta, Params \mapsto params), Temps \mapsto temps)$$

$$update\_temp : Id \times Oop \times DEnv \to DEnv$$
$$update\_temp(id, value, \delta) \triangleq$$
$$\quad \mu(\delta, Temps \mapsto Temps(\delta) \dagger \{id \mapsto value\})$$

$$MMethod\_body : Method\_body \to SEnv \to Method\_response$$
$$MMethod\_body[\![mk\text{-}Method\_body(params, temps, expr)]\!]\,\rho \triangleq$$
$$\quad \lambda rcvr, args, \sigma \cdot$$
$$\quad\quad \text{let } bindings = bind\_args(params, args) \text{ in}$$
$$\quad\quad \text{let } \delta = mk\text{-}DEnv(rcvr, bindings, initialise(temps)) \text{ in}$$
$$\quad\quad \text{let } (result, \delta', \sigma') = MExpression[\![expr]\!]\,\rho\delta\sigma \text{ in}$$
$$\quad\quad (result, \sigma')$$

$$bind\_args : Ulist(Id) \times Oop^* \to Id \xrightarrow{m} Oop$$
$$bind\_args(formals, actuals) \triangleq$$
$$\quad \{formals(i) \mapsto actuals(i) \mid i \in \text{dom } formals\}$$

$$initialise : Id\text{-set} \to Id \xrightarrow{m} Oop$$
$$initialise(vars) \triangleq \{id \mapsto \text{NILOOP} \mid id \in vars\}$$

$$MExpression : Expression \to SEnv \to DEnv \to$$
$$\qquad Object\_memory \to$$
$$\qquad\quad Oop \times DEnv \times Object\_memory$$
$$MExpression[\![mk\text{-}Expression\_list(exprs)]\!]\,\rho\delta\sigma \triangleq$$
$$\quad \text{let } (oop, \delta', \sigma') = MExpression[\![\text{hd } exprs]\!]\,\rho\delta\sigma \text{ in}$$
$$\quad \text{if len } exprs = 1 \text{ then } (oop, \delta', \sigma')$$
$$\quad \text{else } MExpression[\![mk\text{-}Expression\_list(\text{tl } exprs)]\!]\,\rho\delta'\sigma'$$

$$MExpression[\![mk\text{-}Assignment(id, rhs)]\!]\,\rho\delta\sigma \triangleq$$
$$\quad \text{let } (result, \delta', \sigma') = MExpression[\![rhs]\!]\,\rho\delta\sigma \text{ in}$$
$$\quad \text{cases } id \text{ of}$$
$$\quad mk\text{-}Temp\_id(t) \to (result, update\_temp(t, result, \delta'), \sigma')$$
$$\quad mk\text{-}Inst\_var\_id(iv) \to \big(result, \delta',$$
$$\qquad\qquad update\_inst\_var(iv, Rcvr(\delta'),$$
$$\qquad\qquad\quad result, Class(\rho), \sigma')\big)$$
$$\quad \text{end}$$

$$MExpression\,[\![mk\text{-}Inst\_var\_id(id)]\!]\,\rho\delta\sigma \triangleq$$
$$(inst\_var(id, Rcvr(\delta), Class(\rho), \sigma), \delta, \sigma)$$

$$MExpression\,[\![mk\text{-}Arg\_id(id)]\!]\,\rho\delta\sigma \triangleq (Args(\delta)(id), \delta, \sigma)$$

$$MExpression\,[\![mk\text{-}Temp\_id(id)]\!]\,\rho\delta\sigma \triangleq (Temps(\delta)(id), \delta, \sigma)$$

$$MExpression\,[\![\text{SELF}]\!]\,\rho\delta\sigma \triangleq (Rcvr(\delta), \delta, \sigma)$$

$$MExpression\,[\![mk\text{-}Message(rcvr, sel, arglist)]\!]\,\rho\delta\sigma \triangleq$$
$$\text{let } (rcvr\_oop, \delta', \sigma') = MExpression\,[\![rcvr]\!]\,\rho\delta\sigma \text{ in}$$
$$\text{let } (actuals, \delta'', \sigma'') = MExpression\_list\,[\![arglist]\!]\,\rho\delta'\sigma' \text{ in}$$
$$\text{let } (result, \sigma''') = perform(sel, PD(\rho), rcvr\_oop, actuals, \sigma'', rcvr = \text{SELF}) \text{ in}$$
$$(result, \delta'', \sigma''')$$

$$MExpression\_list : Expression^* \to SEnv \to DEnv \to Object\_memory \to Oop^* \times DEnv \times Object\_memory$$
$$MExpression\_list\,[\![el]\!]\,\rho\delta\sigma \triangleq$$
$$\text{if } el = [\,] \text{ then } ([\,], \delta, \sigma)$$
$$\text{else let } (val, \delta', \sigma') = MExpression\,[\![\text{hd } el]\!]\,\rho\delta\sigma \text{ in}$$
$$\text{let } (val\_list, \delta'', \sigma'') = MExpression\_list\,[\![\text{tl } el]\!]\,\rho\delta'\sigma' \text{ in}$$
$$([val]^\frown val\_list, \delta'', \sigma'')$$

$$
\begin{aligned}
&perform : Selector \times Program\_den \times \\
&\qquad Oop \times Oop^* \times Object\_memory \times \mathbb{B} \\
&\qquad\qquad \rightarrow Oop \times Object\_memory \\
&perform(sel, pd, rcvr, args, \sigma, from\_self) \quad \triangleq \\
&\quad \textsf{let } class = class(rcvr, \sigma) \textsf{ in} \\
&\quad \textsf{if } sel \in public\_selectors(class, pd) \\
&\quad\quad \vee\ from\_self \wedge sel \in private\_selectors(class, pd) \\
&\quad \textsf{then } method(sel, class, pd)(rcvr, args, \sigma) \\
&\quad \textsf{else } message\_not\_understood(sel, pd, \\
&\qquad\qquad class, rcvr, args, \sigma, from\_self)
\end{aligned}
$$

$$
\begin{aligned}
&private\_selectors : Class\_name \times Program\_den \rightarrow Selector\text{-set} \\
&private\_selectors(class, pd) \quad \triangleq \\
&\quad \{sel \in \textsf{dom}\, pd(class) \mid pd(class)(sel) \in Private\_meth\}
\end{aligned}
$$

$$
\begin{aligned}
&public\_selectors : Class\_name \times Program\_den \rightarrow Selector\text{-set} \\
&public\_selectors(class, pd) \quad \triangleq \\
&\quad \{sel \in \textsf{dom}\, pd(class) \mid pd(class)(sel) \in Public\_meth\}
\end{aligned}
$$

$$
\begin{aligned}
&method : Selector \times Class\_name \times Program\_den \rightarrow \\
&\qquad\qquad Method\_response \\
&method(sel, class, pd) \quad \triangleq \quad Meth(pd(class)(sel))
\end{aligned}
$$

$$
\begin{aligned}
&MExpression[\![mk\text{-}Static\_send(sel, args, class)]\!]\, \rho\delta\sigma \quad \triangleq \\
&\quad \textsf{let } (actuals, \delta', \sigma') = MExpression\_list[\![args]\!]\, \rho\delta\sigma \textsf{ in} \\
&\quad \textsf{let } (result, \sigma'') \\
&\qquad\qquad = method(sel, class, PD(\rho))(Rcvr(\delta'), actuals, \sigma') \textsf{ in} \\
&\quad (result, \delta', \sigma'')
\end{aligned}
$$

$$
\begin{aligned}
&MExpression[\![mk\text{-}Int\_literal(int)]\!]\, \rho\delta\sigma \quad \triangleq \\
&\quad \textsf{let } (oop, \sigma') = find\_or\_make\_immutable(int, \textsf{Integer}, \sigma) \textsf{ in} \\
&\quad (oop, \delta, \sigma')
\end{aligned}
$$

$find\_or\_make\_immutable$ ($value$: $Primitive\_object$,
$class$: $Class\_name$) obj: Oop

ext wr σ : $Object\_memory$

post $\sigma(obj) = mk\text{-}Object(class, value) \wedge (\sigma = \overleftarrow{\sigma} \vee \{obj\} \lhd \sigma = \overleftarrow{\sigma})$

$MExpression[\![mk\text{-}Bool\_literal(bool)]\!]\rho\delta\sigma \triangleq$
(if $bool$ then TRUEOOP else FALSEOOP, δ, σ)

$MExpression[\![mk\text{-}Symbol\_literal(s)]\!]\rho\delta\sigma \triangleq$
let $(oop, \sigma') = find\_or\_make\_immutable(s, \text{Symbol}, \sigma)$ in
(oop, δ, σ')

$MExpression[\![b]\!]\rho\delta\sigma \triangleq$ (for $b \in Block\_body$)
let $obj = mk\text{-}Object(\text{Closure}, MBlock\_body[\![b]\!]\rho\delta)$ in
let $(closure, \sigma') = create(obj, \sigma)$ in
$(closure, \delta, store')$

$MBlock\_body : Block\_body \to SEnv \to DEnv \to Block\_den$
$MBlock\_body[\![mk\text{-}Block\_body(params, temps, expr)]\!]\rho\delta \triangleq$
$\lambda closure, args, \sigma \cdot$
if len $params \neq$ len $args$
then $block\_arg\_error(\rho, \delta, closure, args, \sigma)$
else let $\delta' = update\_vars(\delta,$
$Temps(\delta) \dagger Params(\delta)$
$\dagger\ bind\_args(params, args),$
$initialise(temps))$ in
let $(result, \delta'', \sigma') = MExpression[\![expr]\!]\rho\delta'\sigma$ in
$(result, \sigma')$

A.7 Primitives

A.7.1 General primitives

$$class\_primitive : Primitive\_method$$
$$class\_primitive \triangleq \lambda rcvr, \_, \sigma \cdot (Class(\sigma(rcvr)), \sigma)$$

$$oopOf\_primitive : Primitive\_method$$
$$oopOf\_primitive \triangleq$$
$$\lambda rcvr\_oop, \_, \sigma \cdot$$
$$find\_or\_make\_immutable(oop\_of(rcvr\_oop), \text{Integer}, \sigma)$$

where

$$oop\_of \in Oop \xleftrightarrow{m} \mathbb{Z}$$

$$perform\_primitive : Primitive\_method$$
$$perform\_primitive \triangleq$$
$$\lambda rcvr, \text{cons}(sel\_oop, args), \sigma \cdot$$
$$\text{let } sel = body(sel\_oop, \sigma) \text{ in}$$
$$\text{if } sel \in Symbol \wedge nargs(sel) = \text{len } args$$
$$\text{then } perform(sel, pd, rcvr, args, \sigma, \text{false})$$
$$\text{else } perform\_error(sel\_oop, sel, pd, rcvr, args, \sigma)$$

A.7.2 Arithmetic

$$plus\_primitive : Primitive\_method$$
$$plus\_primitive \triangleq$$
$$\lambda rcvr, [arg], \sigma \cdot$$
$$\text{let } addend = body(rcvr, \sigma) \text{ in}$$
$$\text{let } augend = body(arg, \sigma) \text{ in}$$
$$\text{if } augend \in \mathbb{Z}$$
$$\text{then } find\_or\_make\_immutable(addend+augend, \text{Integer}, \sigma)$$
$$\text{else } plus\_error(rcvr, arg, \sigma)$$

A.7.3 Primitives on Class objects

$new\_primitive : Primitive\_method$
$new\_primitive \triangleq$
$\quad \lambda class\_oop, \_, \sigma \cdot$
$\quad\quad$ let $instvars = Instvars(body(class\_oop, \sigma))$ in
$\quad\quad$ let $new\_obj = make\_obj(instvars)$ in
$\quad\quad create(mk\text{-}Object(Name(body(class\_oop, \sigma)),$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad new\_obj), \sigma)$

$create\ (obj: Object)\ new\_oop: Oop$
ext wr σ : $Object\_memory$
post $new\_oop \notin \text{dom}\ \overleftarrow{\sigma} \wedge \sigma = \overleftarrow{\sigma} \cup \{new\_oop \mapsto obj\}$

$make\_obj : Instvar\_desc \to Plain\_object$
$make\_obj(iv\_desc) \triangleq$
$\quad$ let $local = \{id \mapsto \text{NILOOP} \mid id \in Local\_instvars(iv\_desc)\}$ in
$\quad$ let $inh = \{c \mapsto make\_obj(Inherited(iv\_desc)(c))$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid c \in \text{dom}\, Inherited(iv\_desc)\}$ in
$\quad mk\text{-}Plain\_object(local, inh)$

A.7.4 Block primitives

$value\_primitive : Primitive\_method$
$value\_primitive \triangleq$
$\quad \lambda closure, args, \sigma \cdot body(closure, \sigma)(closure, args, \sigma)$

A.7.5 Primitives on Indexable objects

$$
\begin{array}{l}
new\_indexable\_primitive : Primitive\_method \\
new\_indexable\_primitive \quad \triangleq \\
\quad \lambda class\_oop, [size\_oop], \sigma \cdot \\
\qquad \textsf{let } size = body(size\_oop, \sigma) \textsf{ in} \\
\qquad \textsf{if } size \in \mathbb{N} \\
\qquad \textsf{then let } body = \{i \mapsto \textsc{NilOop} \mid i \in \{0, \ldots, size - 1\}\} \textsf{ in} \\
\qquad\quad \textsf{let } class\_name = Name(body(class\_oop, \sigma)) \textsf{ in} \\
\qquad\quad create(mk\text{-}Object(class\_name, body), \sigma) \\
\qquad \textsf{else } create\_error(class\_oop, size\_oop, \sigma)
\end{array}
$$

$$
\begin{array}{l}
size\_primitive : Primitive\_method \\
size\_primitive \quad \triangleq \\
\quad \lambda obj, [\,], \sigma \cdot \\
\qquad find\_or\_make\_immutable(\textsf{max dom } body(obj, \sigma), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textsf{Integer}, \sigma)
\end{array}
$$

$$
\begin{array}{l}
at\_primitive : Primitive\_method \\
at\_primitive \quad \triangleq \\
\quad \lambda obj, [index\_oop], \sigma \cdot \\
\qquad \textsf{let } index = body(index\_oop, \sigma) \textsf{ in} \\
\qquad \textsf{if } index \in \mathbb{N} \wedge in\_bounds(obj, index, \sigma) \\
\qquad \textsf{then } (body(obj, \sigma)(index), \sigma) \\
\qquad \textsf{else } bound\_error(obj, index\_oop, \sigma)
\end{array}
$$

$$
\begin{array}{l}
atput\_primitive : Primitive\_method \\
atput\_primitive \quad \triangleq \\
\quad \lambda obj, [index\_oop, value], \sigma \cdot \\
\quad\quad \textsf{let } index = body(index\_oop, \sigma) \textsf{ in} \\
\quad\quad \textsf{if } index \in \mathbb{N} \wedge in\_bounds(obj, index, \sigma) \\
\quad\quad \textsf{then let } new\_obj = Body(\sigma(obj)) \dagger \{index \mapsto value\} \textsf{ in} \\
\quad\quad\quad (obj, \sigma \dagger \{obj \mapsto \mu(\sigma(obj), Body \mapsto new\_obj)\}) \\
\quad\quad \textsf{else } bound\_error(obj, index\_oop, \sigma)
\end{array}
$$

$$
\begin{array}{l}
in\_bounds : Oop \times \mathbb{N} \times Object\_memory \rightarrow \mathbb{B} \\
in\_bounds(obj, index, \sigma) \quad \triangleq \quad index \in \textsf{dom } body(obj, \sigma)
\end{array}
$$

$$
\begin{array}{l}
grow\_primitive : Primitive\_Method \\
grow\_primitive \quad \triangleq \\
\quad \lambda obj, [size\_oop], \sigma \cdot \\
\quad\quad \textsf{let } size = body(size\_oop, \sigma) \textsf{ in} \\
\quad\quad \textsf{if } size \in \mathbb{N} \wedge body(obj, \sigma) \in Indexable\_object \\
\quad\quad \textsf{then } (obj, grow(obj, size, \sigma)) \\
\quad\quad \textsf{else } grow\_error(obj, size\_oop, \sigma)
\end{array}
$$

$$
\begin{array}{l}
grow : Oop \times \mathbb{N} \times Object\_memory \rightarrow Object\_memory \\
grow(oop, size, \sigma) \quad \triangleq \\
\quad \textsf{let } new\_body = \textsf{if } (size - 1) \in \textsf{dom } body(oop, \sigma) \\
\quad\quad\quad\quad\quad\quad \textsf{then } \{0, \ldots, size - 1\} \lhd body(oop, \sigma) \\
\quad\quad\quad\quad\quad\quad \textsf{else } \{i \mapsto \textsc{NilOop} \mid i \in \{0, \ldots, size - 1\}\} \\
\quad\quad\quad\quad\quad\quad\quad \dagger\ body(oop, \sigma) \\
\quad\quad \textsf{in} \\
\quad \sigma \dagger \{oop \mapsto \mu(\sigma(oop), Body \mapsto new\_body)\}
\end{array}
$$

Appendix B

The Continuation Semantics of a Complete Language

This appendix extends the language of the previous appendix to include continuation blocks (§5.2). Some changes are necessary to semantic domains, and all meaning functions are altered to use continuations. The context conditions and pre-processing functions of the previous appendix still apply.

B.1 Changes to Abstract Syntax

$$\begin{aligned} \mathit{Literal\_object'} = {} & \mathit{Block\_body'} \cup \mathit{Cont\_body'} \cup \mathit{Int\_literal} \\ & \cup \mathit{Bool\_literal} \cup \mathit{Symbol\_literal} \cup \mathit{Nil\_literal} \end{aligned}$$

$$\begin{aligned} \mathit{Cont\_body'} :: \ & \mathit{Params} : \mathit{Ulist}(\mathit{Id}) \\ & \mathit{Temps} : \mathit{Id}\text{-set} \\ & \mathit{Expr} : \mathit{Expression'} \end{aligned}$$

B.2 New Context Condition

$$
\begin{array}{l}
WFExpression\,[\![mk\text{-}Cont\_body'(params, temps, expr)]\!]\,\theta \quad \triangleq \\
\quad \text{let } \theta' = \mu(\mu(\theta, Params \mapsto \\
\qquad\qquad Params(\theta) \cup \text{rng}\, params \cup Temps(\theta)), \\
\qquad\qquad\qquad\qquad Temps \mapsto temps) \text{ in} \\
\quad WFExpression\,[\![expr]\!]\,\theta'
\end{array}
$$

B.3 Changes to Processed Abstract Syntax

$$
\begin{array}{l}
Literal\_object = Block\_body \cup Cont\_body \cup Int\_literal \\
\qquad\qquad \cup Bool\_literal \cup Symbol\_literal \cup Nil\_literal
\end{array}
$$

$$
\begin{array}{rcl}
Cont\_body :: & Params & : Ulist(Id) \\
& Temps & : Id\text{-set} \\
& Expr & : Expression
\end{array}
$$

B.4 Changes to Processing Functions

$$
\begin{array}{l}
PExpression : Expression' \to Class\_name \to Expression \\
PExpression\,[\![expr]\!]\,this\_class \quad \triangleq \\
\quad \text{cases } expr \text{ of} \\
\quad \ldots \\
\quad mk\text{-}Cont\_body'(p, t, e) \to mk\text{-}Cont\_body(p, t, \\
\qquad\qquad PExpression\,[\![e]\!]\,this\_class) \\
\quad \text{others } expr \\
\quad \text{end}
\end{array}
$$

B.5 Changes to Semantic Domains

$$
\begin{array}{l}
Method\_response = MCont \to \\
\qquad Oop \times (Oop^*) \times Object\_memory \to \\
\qquad\qquad Oop \times Object\_memory
\end{array}
$$

$$MCont = Oop \times Object\_memory \to Oop \times Object\_memory$$

$$ECont = Oop \times DEnv \times Object\_memory \to Oop \times Object\_memory$$

$$LCont = (Oop^*) \times DEnv \times Object\_memory \to Oop \times Object\_memory$$

$$ignore : MCont \to ECont$$
$$ignore(\nu) \triangleq \lambda r, \delta, \sigma \cdot \nu(r, \sigma)$$

$$insert : DEnv \to ECont \to MCont$$
$$insert\ (\delta)(\varepsilon) \triangleq \lambda r, \sigma \cdot \varepsilon(r, \delta, \sigma)$$

B.6 Meaning Functions

$$Program\_den = Class\_name \xrightarrow{m} Class\_den$$

$$MProgram : Program \to Program\_den$$
$$MProgram[\![p]\!] \triangleq$$
$$\quad \mathsf{let}\ pd = \{class\_id \mapsto MClass\_body[\![p(class\_id)]\!]pd \mid class\_id \in \mathsf{dom}\, p\}\ \mathsf{in}$$
$$\quad pd$$

$$Class\_den = Selector \xrightarrow{m} Method\_den$$

$$Method\_den = Public\_method \cup Private\_method$$

$$Public\_method :: Meth : Method\_response$$

$$Private\_method :: Meth : Method\_response$$

$$SEnv :: PD : Program\_den$$
$$\quad Class : Access\_path$$

$$MClass\_body : Class\_body \to Program\_den \to Class\_den$$
$$MClass\_body\,[\![mk\text{-}Class\_body(\_, meths)]\!]\,pd \triangleq$$
$$\quad \{sel \mapsto MMethod\,[\![meths(sel)]\!]\,pd \mid sel \in \mathsf{dom}\, meths\}$$

$$MMethod : Method\_desc \to Program\_den \to Method\_den$$
$$MMethod\,[\![mm]\!]\,pd \triangleq$$
$$\quad \mathsf{let}\ m = Method(mm)\ \mathsf{in}$$
$$\quad \mathsf{let}\ md = \mathsf{if}\ m \in Primitive\_method$$
$$\qquad \mathsf{then}\ m$$
$$\qquad \mathsf{else\ let}\ \rho = mk\text{-}SEnv(pd, Original\_class(mm))\ \mathsf{in}$$
$$\qquad\quad MMethod\_body\,[\![m]\!]\,\rho$$
$$\quad \mathsf{in}$$
$$\quad \mathsf{if}\ Type(mm) \in \{\textsc{Private}, \textsc{SubV}\}$$
$$\quad \mathsf{then}\ mk\text{-}Private\_meth(md)$$
$$\quad \mathsf{else}\ mk\text{-}Public\_meth(md)$$

$$Primitive\_method = Method\_response$$

$$\begin{array}{lrcl} DEnv & :: & Rcvr & : & Oop \\ & & Params & : & Id \xrightarrow{m} Oop \\ & & Temps & : & Id \xrightarrow{m} Oop \\ & & Cont & : & MCont \end{array}$$

$$update\_temp : Id \times Oop \times DEnv \to DEnv$$
$$update\_temp(id, value, \delta) \triangleq$$
$$\quad \mu(\delta, Temps \mapsto Temps(\delta) \dagger \{id \mapsto value\})$$

$$update\_vars : DEnv \times Id \xrightarrow{m} Oop \times Id \xrightarrow{m} Oop \to DEnv$$
$$update\_vars(\delta, params, temps) \triangleq$$
$$\quad \mu(\mu(\delta, Params \mapsto params), Temps \mapsto temps)$$

$$
\begin{array}{l}
MMethod\_body : Method\_body \to SEnv \to Method\_response \\
MMethod\_body[\![mk\text{-}Method\_body(params, temps, expr)]\!]\,\rho \quad \triangleq \\
\quad \lambda\nu \cdot \lambda rcvr, args, \sigma \cdot \\
\qquad \mathsf{let}\ bindings = bind\_args(params, args)\ \mathsf{in} \\
\qquad \mathsf{let}\ \delta = mk\text{-}DEnv(rcvr, bindings, initialise(temps), \nu)\ \mathsf{in} \\
\qquad MExpression[\![expr]\!]\,\rho\delta\sigma\ ignore(\nu)
\end{array}
$$

$$
\begin{array}{l}
bind\_args : Ulist(Id) \times Oop^* \to Id \xrightarrow{m} Oop \\
bind\_args(formals, actuals) \quad \triangleq \\
\quad \{formals(i) \mapsto actuals(i) \mid i \in \mathsf{dom}\, formals\}
\end{array}
$$

$$
\begin{array}{l}
initialise : Id\text{-}\mathsf{set} \to Id \xrightarrow{m} Oop \\
initialise(vars) \quad \triangleq \quad \{id \mapsto \textsc{NilOop} \mid id \in vars\}
\end{array}
$$

$$
\begin{array}{l}
MExpression : Expression \to SEnv \to DEnv \to \\
\qquad\qquad Object\_memory \to \\
\qquad\qquad\qquad ECont \to Oop \times Object\_memory \\
MExpression[\![mk\text{-}Expression\_list(exprs)]\!]\,\rho\delta\sigma\varepsilon \quad \triangleq \\
\quad MExpression[\![\mathsf{hd}\, exprs]\!]\,\rho\delta\sigma \\
\qquad \lambda oop, \delta', \sigma' \cdot \\
\qquad\quad \mathsf{if\ len}\, exprs = 1\ \ \mathsf{then}\ \varepsilon(oop, \delta', \sigma') \\
\qquad\quad \mathsf{else}\ MExpression[\![mk\text{-}Expression\_list(\mathsf{tl}\, exprs)]\!]\,\rho\delta'\sigma'\varepsilon
\end{array}
$$

$$
\begin{array}{l}
MExpression[\![mk\text{-}Assignment(id, rhs)]\!]\,\rho\delta\sigma\varepsilon \quad \triangleq \\
\quad MExpression[\![rhs]\!]\,\rho\delta\sigma \\
\qquad \lambda result, \delta', \sigma' \cdot \\
\qquad\quad \mathsf{cases}\ id\ \ \mathsf{of} \\
\qquad\quad mk\text{-}Temp\_id(t) \qquad \to \varepsilon(result, update\_temp(t, result, \delta'), \sigma') \\
\qquad\quad mk\text{-}Inst\_var\_id(iv) \to \varepsilon\big(result, \delta', \\
\qquad\qquad\qquad\qquad update\_inst\_var(iv, Rcvr(\delta'), \\
\qquad\qquad\qquad\qquad\quad result, Class(\rho), \sigma')\big) \\
\qquad\quad \mathsf{end}
\end{array}
$$

$$MExpression[\![mk\text{-}Inst\_var\_id(id)]\!]\rho\delta\sigma\varepsilon \triangleq$$
$$\varepsilon(inst\_var(id, Rcvr(\delta), Class(\rho), \sigma), \delta, \sigma)$$

$$MExpression[\![mk\text{-}Arg\_id(id)]\!]\rho\delta\sigma\varepsilon \triangleq \varepsilon(Args(\delta)(id), \delta, \sigma)$$

$$MExpression[\![mk\text{-}Temp\_id(id)]\!]\rho\delta\sigma\varepsilon \triangleq \varepsilon(Temps(\delta)(id), \delta, \sigma)$$

$$MExpression[\![\text{SELF}]\!]\rho\delta\sigma\varepsilon \triangleq \varepsilon(Rcvr(\delta), \delta, \sigma)$$

$$MExpression[\![mk\text{-}Message(rcvr, sel, arglist)]\!]\rho\delta\sigma\varepsilon \triangleq$$
$$MExpression[\![rcvr]\!]\rho\delta\sigma$$
$$\lambda rcvr\_oop, \delta', \sigma' \cdot$$
$$MExpression\_list[\![arglist]\!]\rho\delta'\sigma'$$
$$\lambda actuals, \delta'', \sigma'' \cdot$$
$$perform(sel, PD(\rho), rcvr\_oop, actuals, \sigma'',$$
$$rcvr = \text{SELF}, insert\ \delta''\varepsilon)$$

$$MExpression\_list : Expression^* \to SEnv \to$$
$$DEnv \to Object\_memory \to LCont \to$$
$$Oop^* \times Object\_memory$$
$$MExpression\_list[\![el]\!]\rho\delta\sigma\kappa \triangleq$$
$$\text{if } el = [\,]$$
$$\text{then } \kappa([\,], \delta, \sigma)$$
$$\text{else } MExpression[\![\text{hd } el]\!]\rho\delta\sigma$$
$$\lambda r, \delta', \sigma' \cdot$$
$$MExpression\_list[\![\text{tl } el]\!]\rho\delta'\sigma'$$
$$\lambda rl, \delta'', \sigma'' \cdot$$
$$\kappa([r]^\frown rl, \delta'', \sigma'')$$

$$
\begin{aligned}
&perform : Selector \times Program\_den \times \\
&\qquad Oop \times (Oop^*) \times Object\_memory \\
&\qquad \times \mathbb{B} \times MCont \to Oop \times Object\_memory \\
&perform(sel, pd, rcvr, args, \sigma, from\_self, \nu) \triangleq \\
&\quad \text{let } class = class(rcvr, \sigma) \text{ in} \\
&\quad \text{if } sel \in public\_selectors(class, pd) \\
&\quad\ \vee (from\_self \wedge sel \in private\_selectors(class, pd)) \\
&\quad \text{then } method(sel, class, pd)\nu(rcvr, args, \sigma) \\
&\quad \text{else } message\_not\_understood(sel, pd, class, rcvr, args, \\
&\qquad\qquad \sigma, from\_self, \nu)
\end{aligned}
$$

$$
\begin{aligned}
&private\_selectors : Class\_name \times Program\_den \to Selector\text{-set} \\
&private\_selectors(class, pd) \triangleq \\
&\quad \{sel \in \text{dom}\, pd(class) \mid pd(class)(sel) \in Private\_meth\}
\end{aligned}
$$

$$
\begin{aligned}
&public\_selectors : Class\_name \times Program\_den \to Selector\text{-set} \\
&public\_selectors(class, pd) \triangleq \\
&\quad \{sel \in \text{dom}\, pd(class) \mid pd(class)(sel) \in Public\_meth\}
\end{aligned}
$$

$$
\begin{aligned}
&method : Selector \times Class\_name \times Program\_den \to \\
&\qquad\qquad Method\_response \\
&method(sel, class, \rho) \triangleq Meth(pd(class)(sel))
\end{aligned}
$$

$$
\begin{aligned}
&MExpression[\![mk\text{-}Static\_send(sel, args, class)]\!]\,\rho\delta\sigma\varepsilon \triangleq \\
&\quad MExpression\_list[\![args]\!]\,\rho\delta\sigma \\
&\qquad \lambda actuals, \delta', \sigma' \cdot \\
&\qquad\quad method(sel, class, PD(\rho))(insert\ \delta'\varepsilon)(Rcvr(\delta'), actuals, \sigma')
\end{aligned}
$$

$$
\begin{aligned}
&MExpression[\![mk\text{-}Int\_literal(int)]\!]\,\rho\delta\sigma\varepsilon \triangleq \\
&\quad \text{let } (oop, \sigma') = find\_or\_make\_immutable(int, \text{Integer}, \sigma) \text{ in} \\
&\quad \varepsilon(oop, \delta, \sigma')
\end{aligned}
$$

$find\_or\_make\_immutable$ (*value*: *Primitive_object*,
class: *Class_name*) *obj*: *Oop*

ext wr σ : *Object_memory*

post $\sigma(obj) = mk\text{-}Object(class, value) \wedge (\sigma = \overleftarrow{\sigma} \vee \{obj\} \lhd \sigma = \overleftarrow{\sigma})$

MExpression $[\![mk\text{-}Bool\_literal(bool)]\!]\,\rho\delta\sigma\varepsilon \triangleq$
ε(if *bool* then TRUEOOP else FALSEOOP, δ, σ)

MExpression $[\![mk\text{-}Symbol\_literal(s)]\!]\,\rho\delta\sigma\varepsilon \triangleq$
let $(oop, \sigma') = find\_or\_make\_immutable(s, \text{Symbol}, \sigma)$ in
$\varepsilon(oop, \delta, \sigma')$

MExpression $[\![b]\!]\,\rho\delta\sigma\varepsilon \triangleq$ (for $b \in Block\_body$)
let $obj = mk\text{-}Object(\text{Closure}, MBlock\_body\,[\![b]\!]\,\rho\delta)$ in
let $(closure, \sigma') = create(obj, \sigma)$ in
$\varepsilon(closure, \delta, \sigma')$

MExpression $[\![c]\!]\,\rho\delta\sigma\varepsilon \triangleq$ (for $c \in Cont\_body$)
let $obj = mk\text{-}Object(\text{Continuation}, MCont\_body\,[\![c]\!]\,\rho\delta)$ in
let $(closure, \sigma') = create(obj, \sigma)$ in
$\varepsilon(closure, \delta, \sigma')$

$MBlock\_body : Block\_body \to SEnv \to DEnv \to Block\_den$
$MBlock\_body\,[\![mk\text{-}Block\_body(params, temps, expr)]\!]\,\rho\delta \triangleq$
$\lambda\nu \cdot \lambda closure, args, \sigma \cdot$
if len *params* ≠ len *args*
then $block\_arg\_error(\rho, \delta, closure, args, \sigma, \nu)$
else let $\delta' = update\_vars(\delta,$
$Temps(\delta) \dagger Params(\delta)$
$\dagger\ bind\_args(params, args),$
$initialise(temps))$ in
MExpression $[\![expr]\!]\,\rho\delta'\sigma(ignore\ \nu)$

$$
\begin{array}{l}
MCont\_body : Cont\_body \to SEnv \to DEnv \to Block\_den \\
MCont\_body\,[\![mk\text{-}Cont\_body(params, temps, expr)]\!]\,\rho\delta \triangleq \\
\quad \lambda\nu \cdot \lambda closure, args, \sigma \cdot \\
\qquad \mathsf{if}\ \mathsf{len}\, params \neq \mathsf{len}\, args \\
\qquad \mathsf{then}\ block\_arg\_error(\rho, \delta, closure, args, \sigma, \nu) \\
\qquad \mathsf{else}\ \mathsf{let}\ \delta' = update\_vars(\delta, \\
\qquad\qquad Temps(\delta) \dagger Params(\delta) \\
\qquad\qquad\qquad \dagger\ bind\_args(params, args), \\
\qquad\qquad\qquad\qquad initialise(temps))\ \mathsf{in} \\
\qquad\quad MExpression\,[\![expr]\!]\,\rho\delta'\sigma\ ignore(Cont(\delta'))
\end{array}
$$

B.7 Primitives

B.7.1 General primitives

$$
\begin{array}{l}
class\_primitive : Primitive\_method \\
class\_primitive \triangleq \lambda\nu \cdot \lambda rcvr, \_, \sigma \cdot \nu(Class(\sigma(rcvr)), \sigma)
\end{array}
$$

$$
\begin{array}{l}
oopOf\_primitive : Primitive\_method \\
oopOf\_primitive \triangleq \\
\quad \lambda\nu \cdot \lambda rcvr\_oop, \_, \sigma \cdot \\
\qquad \nu(find\_or\_make\_immutable(oop\_of(rcvr\_oop), \mathsf{Integer}, \sigma))
\end{array}
$$

where

$$
oop\_of \in Oop \xleftrightarrow{m} \mathbb{Z}
$$

$$
\begin{array}{l}
perform\_primitive : Primitive\_method \\
perform\_primitive \triangleq \\
\quad \lambda\nu \cdot \lambda rcvr, \mathsf{cons}(sel\_oop, args), \sigma \cdot \\
\qquad \mathsf{let}\ sel = body(sel\_oop, \sigma)\ \mathsf{in} \\
\qquad \mathsf{if}\ sel \in Symbol \wedge nargs(sel) = \mathsf{len}\, args \\
\qquad \mathsf{then}\ perform(sel, pd, rcvr, args, \sigma, \mathsf{false}, \nu) \\
\qquad \mathsf{else}\ perform\_error(sel\_oop, sel, pd, rcvr, args, \sigma, \nu)
\end{array}
$$

B.7.2 Arithmetic

$plus\_primitive : Primitive\_method$
$plus\_primitive \triangleq$
$\lambda\nu \cdot \lambda rcvr, [arg], \sigma \cdot$
let $addend = body(rcvr, \sigma)$ in
let $augend = body(arg, \sigma)$ in
if $augend \in \mathbb{Z}$
then $\nu(find\_or\_make\_immutable(addend + augend,$ Integer, $\sigma))$
else $plus\_error(rcvr, arg, \sigma, \nu)$

B.7.3 Block primitives

$value\_primitive : Primitive\_method$
$value\_primitive \triangleq$
$\lambda\nu \cdot \lambda closure, args, \sigma \cdot body(closure, \sigma)\nu(closure, args, \sigma)$

B.7.4 Primitives on Class objects

$new\_primitive : Primitive\_method$
$new\_primitive \triangleq$
$\lambda\nu \cdot \lambda class\_oop, \_, \sigma \cdot$
let $instvars = Instvars(body(class\_oop, \sigma))$ in
let $new\_obj = make\_obj(instvars)$ in
let $class\_name = Name(body(class\_oop, \sigma))$ in
$\nu(create(mk\text{-}Object(class\_name, new\_obj), \sigma))$

B.7.5 Primitives on Indexable objects

$$
\begin{array}{l}
new\_indexable\_primitive : Primitive\_method \\
new\_indexable\_primitive \quad \triangleq \\
\quad \lambda\nu \cdot \lambda class\_oop, [size\_oop], \sigma \cdot \\
\qquad \mathsf{let}\ size = body(size\_oop, \sigma)\ \mathsf{in} \\
\qquad \mathsf{if}\ size \in \mathbb{N} \\
\qquad \mathsf{then}\ \mathsf{let}\ body = \{i \mapsto \textsc{NilOop} \mid i \in \{0, \ldots, size - 1\}\}\ \mathsf{in} \\
\qquad\quad \mathsf{let}\ class\_name = Name(body(class\_oop, \sigma))\ \mathsf{in} \\
\qquad\quad \nu(create(mk\text{-}Object(class\_name, body), \sigma)) \\
\qquad \mathsf{else}\ create\_error(class\_oop, size\_oop, \sigma, \nu)
\end{array}
$$

$$
\begin{array}{l}
size\_primitive : Primitive\_method \\
size\_primitive \quad \triangleq \\
\quad \lambda\nu \cdot \lambda obj, [\,], \sigma \cdot \\
\qquad \nu(find\_or\_make\_immutable(\mathsf{max}\,\mathsf{dom}\, body(obj, \sigma), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{Integer}, \sigma))
\end{array}
$$

$$
\begin{array}{l}
at\_primitive : Primitive\_method \\
at\_primitive \quad \triangleq \\
\quad \lambda\nu \cdot \lambda obj, [index\_oop], \sigma \cdot \\
\qquad \mathsf{let}\ index = body(index\_oop, \sigma)\ \mathsf{in} \\
\qquad \mathsf{if}\ index \in \mathbb{N} \wedge in\_bounds(obj, index, \sigma) \\
\qquad \mathsf{then}\ \nu(body(obj, \sigma)(index), \sigma) \\
\qquad \mathsf{else}\ bound\_error(obj, index\_oop, \sigma, \nu)
\end{array}
$$

$$
\begin{array}{l}
atput\_primitive : Primitive\_method \\
atput\_primitive \quad \triangleq \\
\quad \lambda \nu \cdot \lambda obj, [index\_oop, value], \sigma \cdot \\
\qquad \textsf{let } index = body(index\_oop, \sigma) \textsf{ in} \\
\qquad \textsf{if } index \in \mathbb{N} \wedge in\_bounds(obj, index, \sigma) \\
\qquad \textsf{then let } new\_obj = Body(\sigma(obj)) \dagger \{index \mapsto value\} \textsf{ in} \\
\qquad\quad \nu(obj, \sigma \dagger \{obj \mapsto \mu(\sigma(obj), Body \mapsto new\_obj)\}) \\
\qquad \textsf{else } bound\_error(obj, index\_oop, \sigma, \nu)
\end{array}
$$

$$
\begin{array}{l}
in\_bounds : Oop \times \mathbb{N} \times Object\_memory \rightarrow \mathbb{B} \\
in\_bounds(obj, index, \sigma) \quad \triangleq \quad index \in \textsf{dom}\, body(obj, \sigma)
\end{array}
$$

$$
\begin{array}{l}
grow\_primitive : Primitive\_Method \\
grow\_primitive \quad \triangleq \\
\quad \lambda \nu \cdot \lambda obj, [size\_oop], \sigma \cdot \\
\qquad \textsf{let } size = body(size\_oop, \sigma) \textsf{ in} \\
\qquad \textsf{if } size \in \mathbb{N} \wedge body(obj, \sigma) \in Indexable\_object \\
\qquad \textsf{then } \nu(obj, grow(obj, size, \sigma)) \\
\qquad \textsf{else } grow\_error(obj, size\_oop, \sigma, \nu)
\end{array}
$$

$$
\begin{array}{l}
grow : Oop \times \mathbb{N} \times Object\_memory \rightarrow Object\_memory \\
grow(oop, size, \sigma) \quad \triangleq \\
\quad \textsf{let } new\_body = \textsf{if } (size - 1) \in \textsf{dom}\, body(oop, \sigma) \\
\qquad\qquad\qquad\qquad \textsf{then } \{0, \ldots, size - 1\} \lhd body(oop, \sigma) \\
\qquad\qquad\qquad\qquad \textsf{else } \{i \mapsto \textsc{NilOop} \mid i \in \{0, \ldots, size - 1\}\} \\
\qquad\qquad\qquad\qquad\qquad \dagger\, body(oop, \sigma) \\
\quad\; \textsf{in} \\
\quad \sigma \dagger \{oop \mapsto \mu(\sigma(oop), Body \mapsto new\_body)\}
\end{array}
$$

Appendix C

Summary of Notation

Here is a list of VDM symbols used in this thesis with which the reader may not be familiar.

$[type]$	$type \cup \{\mathsf{nil}\}$
x-set	(finite) set type constructor
card s	cardinality of set s
$a :: c_1 : t_1, \ldots$	a is a composite type, with components c_1 of type $t_1, \ldots$
$\mu(a, c \mapsto v)$	the composite value a, updated at c to v
$x \xrightarrow{m} y$	map type constructor from x to y
$x \overset{m}{\longleftrightarrow} y$	one-one mapping from x to y
$\{1 \mapsto 2, \ldots\}$	mapping from 1 to 2,…
dom m	the domain of map m
rng m	the range (codomain) of map m
$s \lhd m$	map m restricted to domain s
$s \lhd\!\!\!\!- m$	map m with set s removed from the domain
$\{\ \}$	the empty set or empty map
$\dagger$	map overwite operator
x^*	sequence type constructor
$[a, b, \ldots]$	the sequence $a, b, \ldots$
len s	the number of elements in sequence s

$\mathsf{dom}\, s$	the indices of sequence s ($= \{1, \ldots, \mathsf{len}\, s\}$)
$\mathsf{rng}\, s$	the set of elements in sequence s
$s_1 \frown s_2$	sequences s_1 and s_2 concatenated
$\mathsf{hd}\, s$	the head (car) of sequence s
$\mathsf{tl}\, s$	the tail (cdr) of sequence s
$\mathbb{B}, \mathbb{N}, \mathbb{N}_1$	booleans, natural numbers, natural numbers without zero

Appendix D

Index to Types and Functions

D.1 Index to Types

D.2 Index to Functions

Bibliography

[ABC+83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, November 1983.

[AdBKR86] P. America, J. de Bakker, J. Kok, and J. Rutten. Operational semantics of a parallel object-oriented language. In *Proceedings of the Thirteenth ACM Symposium on the Principles of Programming Languages*, pages 194–208, St. Petersberg Beach, Florida, January 1986.

[Ado85] Adobe Systems Inc. *PostScript Language Reference Manual*. Addison-Wesley, 1985.

[Agh86] G. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[Ame85] P. America. Definition of the programming language POOL-T. ESPRIT Project 415 document 0091, Philips Research Laboratories, September 1985.

[Ame86a] P. America. POOL-T: A parallel object-oriented language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1986.

[Ame86b] P. America. A proof theory for a sequential version of POOL. ESPRIT Project 415 document 0188, Philips Research Laboratories, October 1986.

[AW82] E. A. Ashcroft and W. W. Wadge. Ṛfor semantics. *ACM Transactions on Programming Languages and Systems*, 4(2):283–294, April 1982.

[BDMN73] G. M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Input Two-Nine, 1973.

[BI82] A. H. Borning and D. H. H. Ingalls. Multiple inheritance in Smalltalk-80. In *Proceedings of National Conference on Artificial Intelligence*, pages 234–237, Pittsburgh PA, 1982.

[BJ82] D. Bjørner and C. B. Jones. *Formal Specification and Software Development*. Prentice-Hall, 1982.

[BKK+86] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. CommonLoops: Merging Lisp and object-oriented programming. *ACM SIGPLAN Notices*, 21(11):17–29, November 1986. Proc. OOPSLA.

[Bla83] A. P. Black. Exception handling: The case against. Technical Report 82-01-02, University of Washington, Computer Science Department, 1983.

[Bor77] A. H. Borning. ThingLab—An object-oriented system for building simulations using constraints. In *Proceedings of the Fifth International Joint Conference Artificial Intelligence*, pages 497–498, Cambridge, Mass., 1977.

[Bor81] A. H. Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[BS83] D. G. Bobrow and M. Stefik. The LOOPS manual. Technical report, Xerox PARC, December 1983.

[Cot84] I. D. Cottam. Extending Pascal with one-entry/multi-exit procedures. Technical report, Department of Computer Science, University of Manchester, December 1984.

[Cox86] B. J. Cox. *Object Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1986.

[Cri84] F. Cristian. Correct and robust programs. *IEEE Transactions on Software Engineering*, SE-10(2):163–174, March 1984.

[CWB86] P. J. Caudill and A. Wirfs-Brock. A third generation Smalltalk-80 implementation. *ACM SIGPLAN Notices*, 21(11):119–130, November 1986. Proc. OOPSLA.

[Dah87] O.-J. Dahl. Object-oriented specification. In P. Wegner and B. Shriver, editors, *Research Directions in Object-Oriented Programming*, chapter 16, pages 561–576. MIT Press, 1987.

[DG87] L. G. DeMichiel and R. P. Gabriel. The Common Lisp Object System: An overview. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings of the 1987 European Conference on Object-Oriented Programming*, volume 276, pages 151–170. Springer-Verlag, Paris, June 1987.

[Don77] J. E. Donahue. Locations considered unnecessary. *Acta Informatica*, 8:221–242, 1977.

[Fab74] R. S. Fabry. Capability-based addressing. *Comm. ACM*, 17(7):403–412, July 1974.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[GS82] L. J. Guibas and J. Stolfi. A language for bitmap manipulation. *ACM Transactions on Graphics*, 1(3):191–214, July 1982.

[Hew79] C. Hewitt. Control structure as patterns of passing messages. In P. H. Winston and R. H. Brown, editors, *Artificial Intelligence: An MIT Perspective*, volume 2, pages 433–465. MIT Press, 1979.

[HN87] B. Hailpern and V. Nguyen. A model for object-based inheritance. In P. Wegner and B. Shriver, editors, *Research Directions in Object-Oriented Programming*, chapter 6, pages 147–164. MIT Press, 1987.

[Hoa74] C. A. R. Hoare. Monitors: an operating system structuring concept. *Comm. ACM*, 17(10):549–557, October 1974.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[Ing78] D. H. H. Ingalls. The Smalltalk-76 programming system: Design and implementation. pages 9–16, Tucson, Arizona, 1978.

[Ing83] D. H. H. Ingalls. The evolution of the Smalltalk Virtual Machine. In G. Krasner, editor, *Smalltalk-80: Bits of history, words of advice*, pages 9–28. Addison-Wesley, 1983.

[Jon86] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.

[KC86] S. N. Khoshafian and G. P. Copeland. Object identity. *ACM SIGPLAN Notices*, 21(11):406–416, November 1986. Proc. OOPSLA.

[LGFT86] D. M. Lewis, D. R. Galloway, R. J. Francis, and B. W. Thomson. Swamp: A fast processor for Smalltalk-80. *ACM SIGPLAN Notices*, 21(11):131–139, November 1986. Proc. OOPSLA.

[Lie86a] H. Lieberman. Concurrent object-oriented programming in Act 1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press, 1986.

[Lie86b] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. *ACM SIGPLAN Notices*, 21(11):214–223, November 1986. Proc. OOPSLA.

[Mas86] I. A. Mason. *The Semantics of Destructive Lisp*, volume 5 of *CSLI Lecture Notes*. Center for the Study of Language and Information, 1986.

[Mey87] B. Meyer. Eiffel: Programming for reusability and extendibility. *ACM SIGPLAN Notices*, 22(2):85–94, February 1987.

[Moo86] D. A. Moon. Object-oriented programming with Flavors. *ACM SIGPLAN Notices*, 21(11):1–8, November 1986. Proc. OOPSLA.

[Plo76] G. D. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5:452–487, 1976.

[RC86] J. Rees and W. Clinger (eds). Revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, December 1986.

[Ren82] T. Rentsch. Object oriented programming. *ACM SIGPLAN Notices*, 17(9):51–57, September 1982.

[SCB+86] C. Schaffert, T. Cooper, B. Bullis, M. Kiliam, and C. Wilpolt. An introduction to Trellis/Owl. *ACM SIGPLAN Notices*, 21(11):9–16, November 1986. Proc. OOPSLA.

[Sch78] R. W. Scheifler. A denotational semantics of CLU. Technical Report MIT/LCS/TR-201, MIT Laboratory for Computer Science, May 1978.

[Sch86] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

[Sho79] J. F. Shoch. An overview of the programming language Smalltalk-72. *ACM SIGPLAN Notices*, 14(9):64–73, September 1979.

[Smi82] B. C. Smith. Reflection and semantics in a procedural language. Technical Report MIT/LCS/TR-272, MIT Laboratory for Computer Science, January 1982.

[Sny85] A. Snyder. Object-oriented programming for Common Lisp. Technical Report ATC-85-1, Application Technology Center, Hewlett-Packard Labs, July 1985.

[Sny86a] A. Snyder. CommonObjects: An overview. *ACM SIGPLAN Notices*, 21(10):19–28, October 1986.

[Sny86b] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. *ACM SIGPLAN Notices*, 21(11):38–45, November 1986. Proc. OOPSLA.

[Sny87] A. Snyder. Inheritance and the development of encapsulated software components. In P. Wegner and B. Shriver, editors, *Research Directions in Object-Oriented Programming*, chapter 7, pages 165–188. MIT Press, 1987.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[Str87] B. Stroustrup. What is "object-oriented programming"? In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings of the 1987 European Conference on Object-Oriented Programming*, volume 276, pages 51–70. Springer-Verlag, Paris, June 1987.

[SUH86] A. D. Samples, D. Ungar, and P. Hilfinger. SOAR: Smalltalk without bytecodes. *ACM SIGPLAN Notices*, 21(11):107–118, November 1986. Proc. OOPSLA.

[Ten73] R. D. Tennent. Mathematical semantics of SNOBOL4. In *Proceedings of the First ACM Symposium on the Principles of Programming Languages*, pages 95–107, Boston, Massachusetts, 1973.

[Ten77] R. D. Tennent. Language design methods based on semantic principles. *Acta Informatica*, 8:97–112, 1977.

[Ten81] R. D. Tennent. *Principles of Programming Languages*. Prentice-Hall, 1981.

[The83] D. G. Theriault. Issues in the design and implementation of act 2. Memo. 728, MIT Lab. for Artificial Intelligence, June 1983.

[Tou86] D. Touretzky. *The Mathematics of Inheritance Systems*. Morgan Kaufmann/Pitman, Los Altos, California, 1986.

[Uni80] United States Department of Defense. *The Programming Language Ada: Reference Manual*, volume 106 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Weg86] P. Wegner. Perspectives on object-oriented programming. Technical Report CS-86-25, Department of Computer Science, Brown University, December 1986.

[Wir83] N. Wirth. *Programming in Modula-2*. Springer-Verlag, 1983.

[WM80] D. Weinreb and D. Moon. Flavors: Message passing in the Lisp machine. Memo. 602, MIT Lab. for Artificial Intelligence, 1980.

[Wol86] M. Wolczko. Specifications of four garbage collectors. Internal report, University of Manchester, Dept. of Computer Science, 1986.

[Wol87] M. Wolczko. Semantics of Smalltalk-80. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings of the 1987 European Conference on Object-Oriented Programming, Lecture Notes in Computer Science*, volume 276, pages 108–120. Springer-Verlag, Paris, June 1987.

[Yel87] P. M. Yelland. Denotational semantics for reflection in a procedural language. Unpublished manuscript, 1987.