Implementation Techniques for Object-Oriented Systems

Martin Charles Atkins

Submitted for the degree of Doctor of Philosophy

University of York

Department of Computer Science

June 1989

Contents

Figures

Acknowledgements

First I would like to thank my parents, Pam and John, for the many ways in which they have helped to make everything possible! Thanks also to my supervisor, Ian Wand, for encouraging me to defect from Mathematics, and for his help and support over the subsequent years.

All my colleagues, both past and present, at York have contributed to making it a wonderful environment in which to think and work. Particular thanks go to Charles Forsyth, Peter Westlake, and Alan Dix, with whom countless discussions have shaped my understanding of Computer Science. Special thanks are also due to John McDermid, whose support, encouragement, and numerous comments were so important while I was writing up.

This work was supported by the Science and Engineering Research Council, via a studentship and subsequent grants under which I was a Research Associate. I would also like to thank IBM Corporation for funding my work at the Thomas J. Watson Research Center in 1987, and everyone who helped me while I was there — particularly the members of the Manufacturing Research Department.

My colleagues on the Ten15 project at RSRE have also been most supportive, and have shown me what I now believe to be the way forward. They allowed me to take time, at an inconvenient moment, to finish writing up, for which I am most grateful.

Declaration

Destroy methods were developed by Lee Nackman and others in Design Automation Systems (Manufacturing Research Department) at the IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, New York. The techniques in chapter 5 were developed in collaboration with Lee whilst visiting Yorktown Heights.

The material in this chapter first appeared as a paper, entitled ‘‘*The Active Deallocation of Objects in Object-Oriented Systems*’’, co-authored with Lee, in *Software — Practice and Experience*, volume 18, number 11, November 1988. It is Copyright 1988, John Wiley & Sons, Ltd. In particular Lee was responsible for the design of figure 5.1, for which I am most grateful.

Chapter 2 has also been submitted to *Software — Practice and Experience*, as a paper entitled ‘‘*Strongly-Typed Subclassing using First-Class Procedures*’’. All this material is reprinted by permission of John Wiley & Sons, Ltd.

Abstract

Most current object-oriented systems are constructed specifically to support a particular view of objects, and their implementation. This thesis describes two aspects of object-oriented systems, and discusses how these can be supported using general-purpose constructs, rather than special-purpose primitives.

First the implementation of objects, constructed simply or using inheritance, is described using first-class functions in strongly-typed languages. It is seen that the type system is crucial in determining the techniques that can be used, and a less flexible type system can make objects very inefficient. However a sufficiently powerful type system does allow objects of comparable efficiency to those which might be provided as primitives by a system.

A new approach to subtype relationships is presented for use in systems that do not directly support subtypes, and for those that do, a new subtype relationship between existentially quantified types is presented. Both of these are important in the construction of objects, and they can be used together in the implementation of multiple inheritance.

The second part explores techniques to allow an object to have some control over the events that occur when it is no longer needed by the system. A direct implementation is described, but then again it is seen that an implementation by the composition of lower-level primitives is preferable, and in fact in this case is also easier.

Introduction

This thesis discusses some techniques for the construction of *object-oriented systems*. In this context an ‘‘object-oriented system’’ is ‘‘a system that supports the construction and execution of objects’’. This could range from the run-time environment for an object-oriented programming language, to an object-oriented operating system — although the emphasis is at the smaller-scale end of this spectrum. The use of the word ‘‘system’’ is intended to imply that, to a large extent, language-independent, and ‘‘system-level’’ concerns are of interest.

Object-oriented design and implementation techniques are a popular approach to addressing problems which are manifest in the so-called ‘‘software crisis’’, caused by the ever-increasing scope and complexity of software systems. The approach is less revolutionary than those of, say *functional*, or *declarative* programming, being imperative in nature. However the traditional separation of program and data is replaced by a close relationship between data, and the operations that manipulate it. This change of emphasis from code to data is fundamental to the object-oriented approach.

The intention is that the structure of a software system should reflect the structure of the problem that is being solved, and object-oriented techniques, which were originally proposed for simulations, provide a way of achieving this. This approach makes it easier to separate abstract behaviours from each other, which encourages the collection of such behaviours into program libraries, promoting the re-use of existing code.

Object-oriented systems usually provide *inheritance*, or an equivalent mechanism, for the incremental construction of the behaviours of objects. This is particularly important since it allows general-purpose behaviours, found in a library, say, to be specialized to the task at hand — a problem not adequately addressed by other systems, which do not help the programmer to resolve the problems of re-using complex routines, which are seldom exactly what is required, leading to the limited re-use of programs.

Most existing object-oriented systems have been designed specifically to execute programs written using object-oriented techniques. The theme of this thesis is that explicit support is not needed, if sufficiently powerful primitives are available, and the system allows these to be composed in the correct ways.

The first part of the thesis describes techniques for the support of objects in strongly-typed systems, and shows how more expressive type systems allow us to capture progressively more of the essence of what objects really are, and so implement them more efficiently. This means that explicit support for objects is not needed for them to be implemented safely and efficiently, which is useful if one wishes to provide objects in an existing system. It is also a useful technique in new systems, since it requires fewer primitives to be implemented, and verified to be ‘‘safe’’ and ‘‘correct’’.

A further advantage of constructing objects is that it makes the system less specialized towards any particular model of objects, or of their implementation. This is particularly important if the system is intended to support a range of object-oriented languages — especially since there is little agreement on exactly what an object-oriented system is, and how some features, such as multiple inheritance, should be defined.

Chapter 2 discusses the provision of objects in statically strongly-typed systems that provide *First-class*, or *Higher-order*, function values, such as Standard ML. The technique of *type-coercion* is introduced to allow subtype relationships, and hence objects, to be defined.

Chapter 3 discusses how the relaxation of the type system to directly allow *subtype* relationships between values, aids in the construction of objects. However it is not until *existential quantification* of types is allowed by the type system (in chapter 4) that objects can be defined with similar run-time, and space, complexities to those seen in systems providing primitive support for objects.

Chapters 5 and 6, look at a different aspect of object-oriented systems — the ‘‘active deallocation of objects’’. This is when an object is given an opportunity to take some action when it becomes unreferenced by, and hence of no further use to, the rest of the system. Chapter 5 discusses how an object-oriented system might provide *destroy methods*, which are routines in objects that are invoked by the memory system to prepare the object for its deallocation. Chapter 6 then shows how a similar effect can be achieved using primitives which are also of more general use. Surprisingly this leads to a simpler overall mechanism for active deallocation, continuing the theme of the first part, that appropriate primitives are desirable in preference to direct system support.

Chapter 1

Object-Oriented Systems

1.1 What Does "Object-Oriented" Mean?

A good way of thinking of an object-oriented system is of a space which contains many independent *objects*. Each object provides a *behaviour* which is a set of operations that the object can be requested to carry out. It does this by internal computation and by requesting other objects, which it can name, to carry out operations in turn.

Each object contains some state, and so the decomposition of a problem into a collection of objects must reflect the relationships between parts of the state of the system. This means that ''object-oriented'' inherently implies a ''data-oriented'' view of the system.

An alternative is to view each object as presenting a *resource* to the rest of the system. Usually this resource is simply the state in the object, but objects can also represent real devices, or resources outside the system. The object's operations are provided to allow the rest of the system to use the resource, but limit access so that it can only be manipulated in ways that the object chooses to allow.

Nothing has been said of the relative sizes of the objects that make up a system, and this is the essential difference between many object-oriented systems. In an object-oriented programming language, such as Smalltalk-80 or C++, objects take the place of data structures in traditional languages, and can be very small. In Smalltalk-80 everything is an object, even the integers. More commonly, objects take the traditional role of structures (or records), and so often use a few words of memory. In contrast object-oriented operating systems, such as Eden, use objects at a much coarser grain. Here objects provide the behaviour of files, processes, devices, and other services that are in the system. Despite this disparity in scale, these systems have many things in common, and this is what makes them ''object-oriented''.

There have been many attempts to define the properties that make a system ''object-oriented'',[1, 2, 3] but apart from a consensus that the system should contain ''objects'', there is little agreement on an exact meaning. This means that it is best simply to define what the phrase is intended to convey in this thesis — no guarantees are made that this is the same as definitions elsewhere, but it most resembles that presented by Blair et. al.[4]

An ''object-oriented system'' is one that contains *objects* with the following properties:

1 Encapsulation — an object's state is only accessible using its nominated operations.

2 Dynamic lifetimes — objects can be created as the system executes.

3 Identity — each object has a name, which can be used to refer to it.

4 Substitution — objects that provide compatible operations can be used interchangeably.

The encapsulation provided by objects is just what is normally meant by saying that an object is an *instance* of an *abstract data-type*.[5] This does not necessarily mean that an object cannot make its internal state visible from outside, but rather that this is a choice which was made by the designer of the object, which the rest of the system cannot circumvent. This means that the possible interactions of the object with the rest of the system are limited to those which the designer decides to allow, making it possible for objects to be designed and implemented independently of each other, and the rest of the system, to a greater extent than is possible with traditional techniques.

Most modern programming languages provide some support for abstract data-types — examples are *packages* in Ada,[6] and *modules* in Modula-2.[7] However these are instantiated (or in Ada, *elaborated*) statically, at compile time.† An object-oriented system is more dynamic, and objects have dynamic lifetimes. One of the actions that an object can carry out as part of its behaviour is to create another object. Conversely as the system executes objects become unnecessary, and the resources associated with them can be reclaimed by the system. This might occur by the explicit action of the programmer, or automatically using some form of *garbage-collection*.[8] Since the reclamation of objects' resources is purely a performance issue this is not explicit in the definition given above, however real systems must address this in some way, and this leads to the topic of *active deallocation*, which is discussed in chapters 5 and 6.

One of the possible actions of an object is to request some action by another object. This means that it must be possible for an object to contain references to other objects, that is, in some way to *name* other objects. Some constancy is implied in this naming. Subsequent uses of a name are expected to refer the same object, at least in some abstract sense.

Finally, and most importantly, it should be possible to substitute one object for another in contexts where the substituted object can provide the necessary operations. The substituted object might also allow additional operations, but so long as it provides all the operations that are needed in the context in which it is used, the system will continue to function. Of course the system might actually *do* something quite different from what it would have done without the substitution, since the substituted object might have a completely different effect, but the substitution is allowed. Careful use of this property is how much of the power of object-oriented systems is gained.

Allowing substitutions seems strange at first, but is actually quite familiar in the guise of the Unix file system,[9] which provides several operations on open files, such as `read`, `write`, and `close`. However we know that a file descriptor for a

† Although run-time initialization is allowed.

terminal, or some other device, can be substituted for a file descriptor for a file, and these operations will continue to work. However the routines executed, and the resulting effects will be those appropriate to the actual object being used.

The substitution property introduces a form of *polymorphism*[10] into object-oriented systems, allowing parts of the system to work successfully on a range of different kinds of object. It is also the property that introduces almost all the implementation difficulties associated with object-oriented systems.

1.2 Terminology for Object-Oriented Systems

Like any other development, object-oriented techniques have introduced their own terminologies, some of which will be used here. The introduction of terminology specifically related to inheritance and delegation will be postponed until these concepts are described in the next section.

Behaviour

Objects will often be said to display *behaviour*. This is simply the combination of operations with some hidden state, which different operations on an object can use to communicate. Thus consecutive invocations of an operation might return different results, depending on other operations that have occurred on the object.

Message

The request to an object to carry out some operation is usually called a *message* to the object. This may, or may not, imply an underlying communication, depending on the type of system that is involved. However some transfer of data is normally involved even if this is just in the form of parameters included in the request.

Following this, the request is often referred to as the *sending of a message to the object*, and the object is said to *receive* the message. This is roughly equivalent to calling a function in traditional systems, since the flow of control goes with the message from the sending, to the receiving object. When the request has been acted on by the object, a reply is returned to the sender of the message, and it can continue to execute.

Actor languages have a different model of message passing, which will be discussed briefly later.

Method

The code describing the actions to occur when an object is sent a message is called a *method*, or less often, a *script*. In C++ it is called a *member function*, but *method* will be used uniformly here to avoid a proliferation of equivalent

terminology. An object is said to *understand* a message, if it has a method corresponding to that message. Some systems can check that all the messages will be understood statically, at compile-time, while other systems postpone this check until runtime, so that the reception of a message that is not understood becomes a run-time error.

Receiver, Self

The difference between a method and a traditional *function* (or *procedure*) is that a method executes ''in the context of'' a particular object, that is the object to which the message that requested its execution was sent. This object is called the *receiver* of the message, and the method has access to its concrete representation. Most languages allow this object to be referred to in the method as ''`self`'', or in C++ ''`this`''.

Class, Instance

Many objects in a system will differ only in their names and current state, but use the same methods. This means that it is often useful to collect together all the common methods, so that they can be referred to collectively. Such a collection of methods is called a *class*, since it defines the behaviour for a *class* of objects. The objects are then said to be *instances* of the class which contains their methods.

Other information that can be shared by all the instances of a class will also be stored in the class. In particular the class usually contains a description of the construction of the object, and provides operations that use this information to construct new instances.

In some systems, such as Smalltalk-80, classes are themselves objects, and the system is self-describing at this level.

Instance variable

The state contained in an object usually consists of a collection of named variables — in this way an object is very much like a traditional structure. These are called *instance variables*. In LISP-based object oriented systems they are also known as *slots*, but these can also (at least in concept) contain the methods associated with the object.

1.3 Inheritance and Delegation

The substitution property of object oriented systems makes it very common for many objects in the system to have similar behaviours, so that they can be used interchangeably. Thus it is important for systems to provide some support for the construction of similar behaviours. Most systems that organise behaviour in classes use *inheritance* for this, *delegation* is commonly used in systems that do not use classes.

1.3.1 Inheritance

When a class is defined it can, and in some systems must, inherit the behaviour of another class. This means that all the methods in the inherited class, known as its *superclass*, are implicitly provided in the inheriting class, or *subclass*. Consequently any message understood by instances of the superclass will also be understood by instances of the subclass. It also means that any state described by the superclass will be present in instances of the subclass.

This is sufficient, but not necessary, to guarantee that an instance of the subclass can be substituted for instances of the superclass, and we shall see later that this forms the basis of the type system in many object-oriented languages.

Of course, if we are defining a new class, then we do not want only to inherit the behaviour of a superclass, but also wish to add behaviour, or modify the inherited behaviour in some way. Inheritance mechanisms provide for this by allowing the inheriting class to introduce new behaviour that is added to that obtained from the superclass. They also allow the subclass to *override* inherited methods, so that new definitions for inherited methods can be provided. Thus messages associated with the old behaviour can cause parts of the new behaviour to be used instead.

It is usually expected that the overriding behaviour will be, in some sense, an extension of the inherited behaviour, and it is common for overriding methods to invoke the methods that they override as part of their actions. This means that it should be possible for a method to use inherited behaviour that is overridden in its class — this is often called *sending a message to super*, the phrase coming from the Smalltalk notation for this action.

However the most subtle way in which behaviour can be extended is by *sending messages to self*. It is common for the evaluation of a method to make use of the actions of other methods defined for the same object. At its simplest this is the same as calling a local function in conventional languages. However inheritance opens up a new possibility in this situation, which is that a method might cause the invocation of a method defined in a subclass of the class in which it is itself defined. This method will usually have overridden an inherited version that would otherwise have been invoked.

This means that a method's effect can be modified by overriding the definitions of other methods that it uses, allowing parameterization of behaviours. It also means that the interface specification of a class cannot simply consist of the behaviour that it makes available in instances, but must also include the behaviour that it uses by sending messages to self, which might be altered in inheriting classes. The specification of the semantics of a class must provide a way for inherited behaviour to be altered by the inheritance process — this has been described by Cook.[11]

This parameterization of behaviour can be used in a way that is similar to that provided by higher-order functions in functional languages, but is generally notationally less convenient.

Together these possibilities make inheritance a very powerful way of composing existing, and new, behaviours to construct the desired behaviour. This is the foundation which encourages a high degree of code re-use in object-oriented systems. However it can also make the design process more difficult, since the designer of a class should now consider not just the behaviour that is required for the task at hand, but also the interfaces that should be made available to allow this behaviour to be re-used by inheritance.

It is often useful to distinguish two different kinds of classes:

- *Normal* classes — which describe the behaviour of objects in the system, and

- *Abstract classes* — which describe common behaviour to be inherited by other classes, which is not in some sense ''complete''. Such classes are not expected to be instantiated directly.

Some object-oriented languages such as Trellis/Owl,[12, 13] Eiffel[14] and C++ provide mechanisms allowing the programmer to assert that a class is abstract, and the system does not allow instances of these classes to be created. Other languages, such as Smalltalk-80, document abstract classes as a programming technique, but do not provide explicit support for them.

To sum up, there are five important properties expected of inheritance mechanisms:

1 Inherit behaviour — previously defined behaviour can be used as part of the definition of new behaviour.

2 Additional behaviour — new behaviour can be added to that which was inherited.

3 Overriding operations — inherited operations can be replaced by new behaviour.

4 Access to overridden behaviour — new behaviour should be able to make use of inherited behaviour that it, or some other new behaviour, hides by overriding.

5 Access to overriding behaviour — inherited behaviour should be able to use overriding behaviour defined when it is inherited.

1.3.2 Multiple Inheritance

The discussion above assumes that a class is built by extending the behaviour of one other class. In practice there are many situations where the behaviour that is desired is the synthesis of the behaviours of several other classes. The restriction that a class can only have one superclass is inconvenient, and can force an unnatural structure on the classes.

The ability to inherit the behaviours of more than one class is called *multiple inheritance* and has been offered by several object-oriented systems, including most of the LISP-based systems, the traits system used for the software in the Xerox Star office system,[15] and more recently in Trellis/Owl, Eiffel, and C++.

In most situations where multiple inheritance is useful there are ways in which the inheritance relationships between classes could be changed to remove the need for multiple inheritance. However these changes make the classes less general, and so less likely to be useful elsewhere, making multiple inheritance particularly important for improving the possibility of code re-use. With multiple inheritance it is possible to construct classes describing relatively small behaviours, which are then combined in a ‘‘mix and match’’ style, which is not possible with single inheritance.

With single inheritance there are no possibilities of *conflicts*, where the inheriting class is inconsistent because of the behaviour it inherits — local definitions of methods clearly override those that are inherited. However with multiple inheritance the possibility exists that several superclasses might provide methods for the same message, without any ‘‘clearly correct’’ way to decide between them, giving a conflict.

Similar conflicts arise with inherited instance variables, but here overriding is not useful, rather the question is whether two instance variables with the same name and type, inherited from different classes, should be the same instance variable in the object, or whether there should be two instance variables in the object, and a syntax to allow them to be accessed individually.

Different systems have chosen to resolve ambiguities in different ways, which can lead to profound differences in the ways in which inheritance is used in these systems. For example some approaches encourage the mix and match approach, while others encourage approaches more like those used with single inheritance.

Two example inheritance graphs will be used to demonstrate the resolution of conflicts:

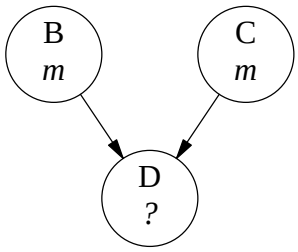


Figure 1.1: Conflicting Inheritance of a Message

This shows class D inheriting behaviour for the message m, from both of its superclasses, B and C. If there is also a definition for m in class D, overriding both inherited methods, then there is no conflict, but without this we need a way to decide which of the inherited methods is invoked when the message m is sent to an instance of class D.

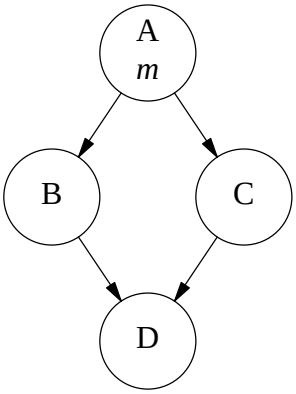


Figure 1.2: Conflicting Inheritance of a Class

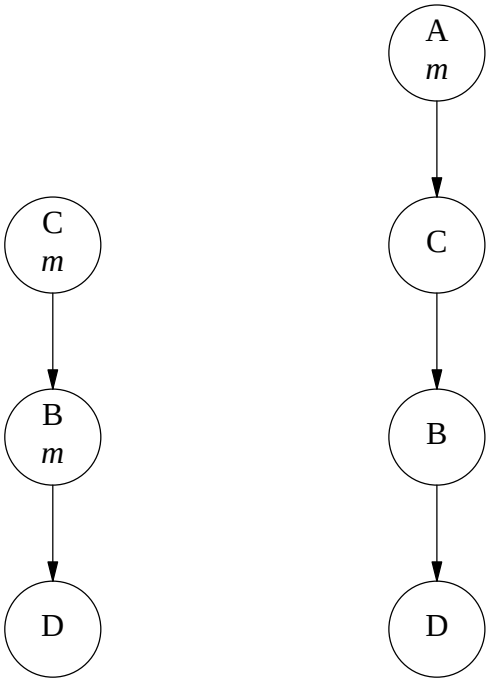
In this case the entire behaviour of the class A is inherited twice by class D, and so every message that is understood by instances of A will be in conflict in the definition of class D, unless it is overridden.

There are three main ways to resolve inheritance ambiguities, Snyder[16] called these: *linear*, *graph-oriented*, and *tree* resolutions of ambiguity.

Linear resolution

In the linear resolution of conflicts the system attempts to find a linear ordering for the inheritance of the superclasses, which preserves as many of the relationships between the classes as possible, and then (in effect) uses single inheritance to construct a class hierarchy with no conflicts.

Given the example class definitions the system would construct the following inheritance hierarchies:



Message conflict Class conflict

Figure 1.3: Linear Conflict Resolution

This approximates a breath-first search of the inheritance hierarchy. The problem with this is that it can only be an approximation, since it introduces inheritance relationships between classes which were not in the original inheritance graph, here between classes B and C. This might interfere with attempts by B to communicate with its superclasses.

The advantage of linear resolution is that any inheritance pattern can be accepted by the system, which is claimed to be useful for the rapid prototyping of systems. This technique is used by the object-oriented extensions to LISP, Symbolics' Flavors,[17] Common LOOPS,[18] and CLOS.[19]

Graph resolution

In graph resolution of inheritance conflicts, the general philosophy is that a conflict is a programmer error, like a type error, and so rather than attempting to resolve the conflict the system should refuse to accept classes that introduce conflicts. However there are some special conflicts that the system will resolve automatically. In particular if the *same* behaviour is inherited more than once, it is not regarded as a conflict, and the resulting class implements the behaviour *once* — in the sense that instances of the class will only contain one set of instance variables for the behaviour, regardless of how often it is inherited. This means that classes that are inherited might communicate via common ancestor classes, and it reflects the view of objects as modelling real-world behaviour — a real-world object can only have a behaviour once!

Tree resolution

Snyder suggested tree-based resolution of inheritance to avoid problems with the other solutions: the multiple inheritance of a method is an error unless it is explicitly overridden in the inheriting class. But in contrast to graph resolution, inherited state is always distinct.

This means that both the examples given above are illegal without overriding, and that the conflicting class example is treated as if it were of the form:

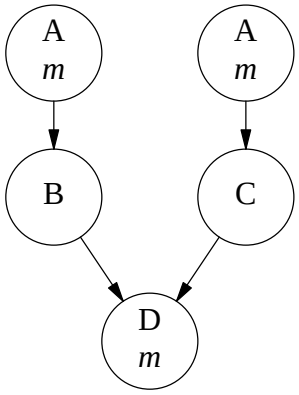


Figure 1.4: Tree-based Multiple Inheritance of a Class

as far as storage is concerned, when the overriding method is added to class D.

The advantages of this are:

- No unexpected sharing can occur between classes because of their choice of instance variable names, or their choice of which classes to inherit. This means that knowledge of the classes that were inherited no longer need to be part of the specification of a class. With graph resolution this was needed so that communication via common ancestor classes could be predicted by the writer of an inheriting class.

- All inherited methods can see their instance variables in the same relative positions. This allows more efficient code sharing between classes than would otherwise be possible.

Tree resolution was first used in Snyder's CommonObjects extension to LISP. This was designed with particular emphasis on the encapsulation of objects and classes, the intention being that classes did not have to include any information about their construction in their externally visible specification. Version 2 of C++ also uses tree resolution for most classes, but acknowledges that graph resolution is sometimes useful, by allowing classes to explicitly declare which of their superclasses can be shared when they are inherited.

1.3.3 Delegation

An alternative to inheritance, which has been used particularly in Actor languages is *delegation*.[20] The idea is that an object, when it is asked to carry out some action for which it does not have a method, simply forwards the request to

another object, called the *delegate*. This means that the behaviour of the delegate is visible through those objects which delegate to it. Delegation introduces more possibilities for the sharing of behaviour than are possible with inheritance, since several objects may all delegate messages to a single object, making changes to its state visible through them all. An object can also delegate different messages to different objects, which is the delegation equivalent of multiple inheritance.

There has been much discussion about the exact relationship between inheritance and delegation, in particular see Lieberman[20] and Stein,[21] but a fundamental difference is in the value of `self` as seen by a method reached by delegation. Here `self` refers to the delegate, rather than referring to the object to which the original request was made, making inheritance property 5 (access to overriding behaviour) very difficult to achieve.

If delegation is allowed to use a slightly different mechanism from normal message sends this problem can be avoided, which is the approach used in Ungar and Smith's language *self*.[22]

The efficient implementation of delegation is difficult, particularly in systems where messages are relatively expensive, simply because it increases the frequency of message sends.

1.4 Some Object-Oriented systems

This section gives a brief survey of some of the more important object-oriented systems that have been developed. The survey is by no means exhaustive — there now being many dozens of systems that are claimed to be ''object-oriented''. However the systems mentioned have either been particularly influential, or are typical of a particular approach to ''object-orientedness''.

Simula-67

Object-oriented programming was first supported by Simula-67,[23] an extension of Algol-60 specifically intended for the programming of simulations. Simula-67 introduced the *Class* construct, which was intended to reflect program organisation techniques that Kristen Nygaard first used in simulations for the Norwegian Nuclear power program.[24] The idea being that the program structure should parallel that of the system which is being modelled. The class allowed objects to be defined to model each real-world object on a one-to-one basis.

Simula-67 is most important for its influences on other languages. It introduced most of the concepts central to object-oriented programming, but was never very widely used — perhaps because the system support required by other language features (particularly co-routines) was perceived as being too expensive for a general-purpose language.

A class could be defined with another as its *prefix*, which is the Simula mechanism for the inheritance of implementations. The term ''prefix'' reflects the underlying implementation, where each object contains an instance of its prefix class

as its first part. This is the most efficient implementation strategy for object-oriented languages, and is described in detail later.

Being based on Algol-60, Simula-67 is a statically strongly-typed language. The types of objects are defined by their classes, each class being implicitly a *subtype* of all its prefix classes — this approach has formed the basis of the type systems for most of the current strongly-typed object oriented languages, such as C++, and Eiffel.

The one weak area in Simula's support for object-oriented programming is that it did not support the encapsulation of objects in that, while operations can be attached to objects, there was no way to restrict the access that was allowed to an object's state from outside. A directive similar to C++'s `private` has been added more recently to provide control of the visibility of object's state.

Kristen Nygaard is now involved in the design of a successor to Simula, called *Beta*.[25, 26] This language provides a general abstraction mechanism, called a *pattern*, which can provide the effect of a class, but is more general. For example, functions and co-routines are also described by patterns in Beta.

C++

C++[27] is an extension of C[28] which was (and continues to be) developed by Bjarne Stroustrup of AT&T Bell Laboratories, at Murray Hill. C++ was directly influenced by Simula-67, and development started when Stroustrup added classes to C using a pre-processor,[29] to help him to program some simulations.

Although highly compatible with C, C++ is a language in its own right, and in addition to classes, adds operator overloading, stricter type checking, in-line functions, a less strict declaration syntax, and user-controlled storage management. At the same time many strengths of C are kept, including the possibility of simple efficient implementations, and low-level access to the machine. Version 2 of C++ (due out in the Summer of 1989) adds multiple inheritance, and some other less important facilities, to the previously distributed version.

Being based on C, and having a similar efficiency, C++ has found a widespread acceptance that has not been enjoyed by other object-oriented systems, and many sites are increasingly using it in preference to C. Objects are implemented in the same way as in Simula, and this is extended to multiple inheritance using techniques that were first proposed for Simula by Krogdahl.[30, 31]

Smalltalk

The Smalltalk systems developed in the Learning Research Group † at the Xerox Palo Alto Research Center have been most influential in developing the field of object-oriented systems.

† Which later became the ''Software Concepts Group'', or SCG.

Smalltalk was intended to be the software component of the *Dynabook*, a powerful, portable, personal computer, which turned out to be far more ahead of its time than was perhaps appreciated. Indeed technologies such as compact high resolution, touch sensitive displays, and batteries that can provide the power for a powerful system with sufficient memory, are only now beginning to become available. It will probably still be several years before the cost becomes low enough for a Dynabook to be feasible.

Fortunately Smalltalk developed independently of the Dynabook hardware:[32]

> The very first Smalltalk evaluator was a thousand-line BASIC program which first evaluated 3+4 in October 1972. It was followed in two months by a Nova assembly code implementation which became known as the Smalltalk-72 system.

Several iterations from this initial design culminated in the Smalltalk-80 system,[33] that was distributed outside Xerox PARC, and is the version of Smalltalk that many people now have access to.

Smalltalk was conceived from the start as a complete system, rather than simply a language, and took many ideas from LISP systems which were also developing in this way. In particular Smalltalk presents a very dynamic environment and encourages the incremental development of programs. This led to run-time type-checking, and a powerful, but potentially expensive mechanism for message dispatch, which were some of the causes of the slow acceptance of Smalltalk-80 outside research centres. It is only quite recently that personal computer systems have become generally available which are powerful enough to support Smalltalk, and more sophisticated techniques for the fast execution of Smalltalk have been developed.[34, 35]

The integrated environment provided by Smalltalk-80 is simultaneously both its strongest and its weakest feature, since it provides a highly productive environment for programming, but also restricts Smalltalk applications to situations that can support the whole environment. In particular it is difficult for Smalltalk tools to be written that interact conveniently with other tools as is common, say in Unix, and it is difficult for companies to distribute Smalltalk programs, independently of a particular Smalltalk system.

LISP-based systems

Many object-oriented systems have been implemented in LISP. There are several reasons for this:

- LISP provides much of the low-level support that is required, and allows the user to construct his own higher-level constructs upon those that already exist. Furthermore this can be done in such a way that the additional constructs are no less convenient to use than those in the original system. This means that LISP is an ideal system for investigating things like object-oriented ideas, and many of these prototypes developed into practical tools that people used.

• LISP does not provide an abstraction mechanism at an intermediate granularity between the function, and the package (where these are provided). Classes and objects fitted nicely into this gap.

• Many influential people at Xerox PARC worked on both Smalltalk and InterLISP-D, leading to much cross-fertilization of ideas.

The dominant object-oriented extensions to LISP were Symbolics' *Flavors*[17] and *LOOPS*[36] (the LISP Object-Oriented Programming System) for InterLISP-D developed at Xerox PARC.[37] LOOPS developed into *Common LOOPS*[18] based on Common LISP.[38] Recently there has been a standardization effort directed at object-oriented extensions to Common LISP as part of its standardization. The result of this is *CLOS* (The Common LISP Object System),[19] which is most closely related to Common LOOPS.

All these object-oriented extensions allow the use of multiple inheritance. LOOPS also introduced *multi-methods*, which generalize message dispatch, so that the choice of which method is executed for a given message can depend on the types of the parameters as well as the type of the receiver of the message. Message dispatch is normally achieved in these systems by dynamic lookup similar to that in Smalltalk (this is further complicated by multi-methods). As in Smalltalk the interactive nature of LISP systems, makes the global analysis that would be needed to optimize message lookup very difficult.

Some other LISP-based object-oriented systems have a rather different philosophy. The best example of this is Snyder's CommonObjects[39] which stresses encapsulation over immediate flexibility.

Other systems deserve a mention in passing: *T*[40] and *Oaklisp*[41] are based on the *Scheme* dialect of LISP, which is a natural choice for experimenting with novel language constructs.[42] *T* is interesting for its unformity and run-time efficiency. *Oaklisp* and *ObjVlisp*[43] provide more flexibility in the construction of classes than other systems, by making the equivalent of the Smalltalk *Metaclass hierarchy* manipulable by the programmer.

Actors

Actor languages model the view of an object-oriented system as a group of communicating objects more closely. In an actor system all objects execute concurrently, and messages between the objects either synchronize the objects' execution, or are queued until the receiving object next attempts to receive a message. Like Smalltalk-80, actor systems are usually *uniformly object-objected*, so that everything in the system is represented by an object — this unfortunately rules out many opportunities for optimization.

Actor systems were first proposed by Hewitt[44] and this has led to several actor-based languages, including the Act family,[45] Plasma and Acore.[46]

Object-Oriented Operating Systems

Many recent research operating systems have adopted an object-oriented structure, where each object is the equivalent of a process in conventional operating systems. This makes these systems most like actor systems, but with a much larger granularity of objects. The advantages of this approach are seen to be:

- Message-passing communication between objects can be generalized transparently to communication over a network, leading to transparent distribution.

- Parts of the functionality of the system can be put into objects. This allows the decomposition of the majority of an operating system into independent parts, which can be designed, developed, and tested independently. For example, it is possible for an experimental version of the file system to be tested, while the old (working) version is still in use by other users of the system.

- The trusted part of the system, which runs in kernel-mode and directly manages the machine, can be made very small, so that it is less likely to contain a mistake. The effects of errors elsewhere are limited by the same memory-protection mechanisms that protect programs from each other.

- The kernel is less committed to any particular kind of the system, and could support a range of different systems (possibly at the same time), since this specialization occurs in the server processes outside it. This means that, from the users' point of view, several different operating systems could co-exist on one machine. This is rather like the *virtual machine* techniques that have been used for many years, but the ''virtualization'' takes place at a higher level, and includes the services provided by the kernel.

Examples of systems that have taken this approach are Mach,[47] Amoeba,[48] Minix,[48] and Clouds.[49] Eden[50] took the object-oriented model further by making files and other user-visible parts of the system into objects, and encouraging the user to interact with them on this basis rather than using a traditional imperative command language.

Object-Oriented Databases

''Object-oriented'' databases are also becoming available, although again we find little agreement on precisely what this means. Some of these systems are ''simply'' persistent systems similar to Smalltalk, an example is GemStone,[51] while others provide ''object-oriented'' extensions to more conventional databases, as in Postgres.[52] The relationship of these systems to other object-oriented systems is not always clear, but a common theme is that these systems provide much better support for fine grained and highly structured data. Also they normally allow user-defined abstract datatypes, in contrast to traditional databases that usually limit users to a small number of built in types.

1.5 Type Systems and Typed Object-Oriented Systems

A type system, in any language or system, is a framework for describing the items that are to be manipulated. This information is then used in two different ways:

1 To ensure that the system is *type-correct*, that is to verify that values are always used, or combined in ways that are deemed to be permissible. This removes the possibility of certain kinds of error existing in programs that have been successfully type-checked, which is useful when it is necessary to trust the system's correctness.

2 To give the system more information about the described items. Sometimes this information is vital for any kind of execution, for example the system must be able to determine how to access the fields in a structure (or record). In other situations this information allows execution to be more efficient, for example some run-time checks might be proved to be unnecessary.

There is an important distinction depending on *when* the type information is used, here we are particularly interested in *statically* typed systems, which are systems where the type information is calculated, and used, during static analysis of the program — usually during compilation. Other systems, particularly those which are interactive or stress properties other than run-time efficiency, such as LISP, awk,[53] and icon,[54] manipulate type information at run-time to ensure that values are not misused, but do not use the information statically. This is more flexible, but makes the types less useful for the purposes given above, and will not be considered further.

There are several views of what exactly types are — see Donahue and Demers[55] for a less conventional view, based on the ideas of Reynolds[56] — but in some sense this does not matter. The important properties of a type system, beyond the necessary self-consistency and safety, are the amount of useful information that it can tell us about the system, and the extent to which it restricts those things that can be done.

All type systems restrict what the programmer can do to some extent. Some of these restrictions, such as not allowing a boolean value to be used as the address of a structure, are quite reasonable, but others may not be. There may always be a need for languages that either do not have types,† such as BCPL,[57] or allow the programmer to explicitly ignore the rules of the language's type system, such as C and Algol 68. However two developments are encouraging more and more software to be developed using *strongly-typed* languages, which do not allow such loop-holes.

First there is much concern about the increasing cost, complexity, and failure rate of software, and strongly typed languages are seen as one step in improving this situation. At the same time the theory underlying type systems has progressed rapidly, and it is now possible to type-check many systems that could not have been checked before. This means that the penalty accrued by using strongly-typed languages is decreasing, and consequently the range of applications in which they

† Or more accurately, have a single type — usually a machine integer.

can be used is increasing.[58]

The items that must be described by a type system usually consist of the data that is manipulated by the program, and the operations which are used to manipulate it. Higher-order languages (or languages with *first-class functions*) regard the operations as being values in their own right, and so type-correctness is a property of the application of values to other values.

Most systems also require type information about variables that will hold the data, so that the memory requirements of these variables can be determined statically, and allowing operations on the variables to also be checked. Many more recent languages, such as Standard ML and Russell,[59] take this one stage further. Types are given to the bindings of names to values, rather than to ''variables'', and updatable values are simply values that allow their contents to be updated, and have types which reflect this. Of course a name can be bound to an updatable value, just as it can be bound to any other value, allowing the traditional view of ''variables'' as named updatable locations.

Some examples of types and notation

There are many different notations for types. Most examples in this thesis will use that of the language *fun*, which was introduced for didactic purposes by Cardelli and Wegner,[10] so they could discuss types and related issues. The major exception is in chapter 2, where all examples are given in Standard ML,[60] and have been successfully executed.

The simplest types are those which have no internal structure such as `Integer`, `Character`, and `Real`. The type with a single value, called `()`, `Unit`, or `void` is also surprisingly useful. These are often called *type-constants*. Different systems make different decisions about exactly what type-constants should be provided, for example Pascal[61] has a subrange type of the form `a..b` rather than the type-constant `integer`. However these choices are essentially arbitrary, and while they effect the expressiveness of the type system, they have little impact on the mechanisms it must use.

A type system should also provide *type-constructors* which allow more complex data structures to be built from type constants, and other constructed types. These take the form of compile-time functions from types to types, but sometimes a more convenient notation is used. For example, the most important type-constructor is that for depicting the types of functions; the type of a function that takes a parameter of type `Real`, and will return a result of type `Integer` might be written:

`Real` $\rightarrow$ `Integer`

and a function `trunc` of this type might be declared by:

`trunc: Real` $\rightarrow$ `Integer`

It is also useful to be able to name types, here we will say:

```
type convert = Real → Integer
```

which would allow the declaration:

```
trunc: convert
```

Other interesting and useful type constructors are for Cartesian products (for grouping values), structures (for grouping labelled values), and tagged unions (for allowing values that can take one of several types). Examples of these might be:

```
type IntPair = Integer × Integer
type Point = {x: Integer, y: Integer}
type IntOrReal = [a: Integer, b: Real]
```

Types constructed in this way can become very complex. Many types have similar structure, so that it is convenient if the programmer can define his own type-constructors, or *type-functions*, for example

```
typefn Pair[T] = T × T

type IntPair = Pair[Integer]
```

In this case we could regard type-functions as being *macros* that are expanded when they are applied to type parameters, but in general this is not possible since type functions can be recursive, and it is best to regard a type function as the definition of a new type-constructor, mapping types to types.[10] In practice this means that inference rules for type-checking the application of type-functions must be defined.

A useful example of a recursive type-function, defines types for lists of values, all of the same type.

```
typefn List[T] = [nil: (), cell: (T × List[T])]
```

Type-checking

Type-checking is the static verification that values in a system are only used in ways that are allowed by the type system. This normally guarantees that a value will not be used inconsistently, and so removes the possibility of some kinds of error. It also verifies that the types that have been given to objects accurately describe those objects, so that this information can be safely used by the compiler when it chooses data representations, and when it is looking for program transformations, and optimizations.

The programmer can specify the types of values explicitly, by including a type declaration for a value, or implicitly by using an operation that requires values of particular types. The type-checker must verify that there are no inconsistencies between the types that the programmer has specified, and those which are inferred for other values. Thus the inference of types for values becomes bound up with

type-checking.

If we consider the type checking of an application of a function to a value we will see these activities:

```
value f: Integer → Real =
          fun(x: Integer)
             ...x...;

f(10)
```

The system knows that the type of the literal constant `10` is `Integer`. The type-checker verifies that the application of `f` to this value is type-correct. It then infers that the type of the result of the application is `Real`, so that it can check uses of the result.

A type-system is described by two things: the inference rules which it can use to relate the types of objects which interact, and the type-compatibility rules which define how the system checks if a value can be legally used in the context in which it appears.

Some systems such as Standard ML, attempt to deduce the type of functions from the operations that they contain, and the types of other functions which are in scope. This can only be done for some restricted type systems — in particular it is important that every item can be given a unique *most general type*.[62] Other systems have chosen to provide a more sophisticated type system, that is one that allows more programs to be type-checked, but can no longer infer types for groups of functions. They can only check the type consistency of a function, and must be given the types of all the other functions that are used, and sometimes the types of some intermediate values also. Examples of systems that have resulted from this choice are Fairbairn's Ponder,[63] and RSRE's Ten15.[64]

Polymorphism

A very significant way in which modern type systems are more expressive than those of languages like Pascal, and Ada, is with respect to their treatment of *polymorphism*. This allows a function to be defined which can be applied to values of several different types. For example the `cons` function which creates values of the list type shown above should be able to work with lists of values of any kind. Its type would be given by:

```
cons: ∀ t . (t × List[t]) → List[t]
```

This says that `cons` can be given a value of any type, and a list of values of the same type, and it will return a list of values of that type. ''∀'' is pronounced ''for all'', and expresses *universal* quantification over types — the type variable `t` ranges over all the types in the type universe.

Polymorphic types of this kind are extremely useful, for example they allow types to be given to higher-order functions of the style first used in LISP, without requiring run-time type-checking. Common examples are:

```
map:  ∀ s,t . (s → t) → List[s] → List[t]
fold: ∀ s,t . ((s × t) → t) → t → List[s] → t
```

Existential quantification over types is another form of polymorphism which is useful. Its use in defining objects forms the subject of chapter 4 where it will be described in detail.

1.5.1 The Types of Objects

Many object-oriented languages have used run-time type-checking to ensure that the messages that are sent to an object are understood by that object. However the static use of types to prove that errors of this sort can never occur is the domain of type-checking, and is as useful here as it is in traditional languages. The question arises as to what type-model should be used to represent the types of objects.

The important property of objects in this context is the property of substitution (property 4) which says that objects that provide a common set of operations should be able to be used interchangeably in contexts that only require the common behaviour. Inheritance provides a way in which we can guarantee that the inheriting class will have all the behaviour of its superclass. This suggests that the type of an object should be associated with its class, and that instances of a subclass should be able to be used in all the places where instances of the superclass are expected. This has formed the basis of almost every typed object-oriented language, starting with Simula-67, and ranging through C++, Trellis/Owl, and Eiffel.

Unfortunately there is a fundamental flaw with this type system; it is based around an implementation technique for classes, namely inheritance, and not directly around the set of operations that the objects provide — which is ultimately what is important for the substitution property to work.

A clear example of this was first described by Snyder,[16] when he considered objects representing stacks and double-ended queues: a double-ended queue is like a stack, except that `push` and `pop` operations are provided for both ends, so that data can be pushed on at one end, and popped off at the other. A stack provides these operations only for one end. It follows that, with suitable naming of the messages, a queue object should be able to be used wherever a stack object is expected, which would suggest that the queue class should inherit from the stack class.

We are now presented with a system that provides an efficient queue implementation, and we have to define our own stacks. We would like to be able to implement the stack class by inheriting the queue implementation, and hiding the methods that access one end. This means that the inheritance we desire for the implementation is the reverse of that which is needed for the types to be correct!

More generally the type of an object should have no connection with the way in which it is implemented. This can be achieved in (at least) two different ways:

1 Two different sorts of inheritance could be used, one to define the type of a class, and the other to define the implementation. Where it is convenient these inheritance hierarchies could be constructed in parallel, giving the same effect as is offered by current systems, but this would be a choice that was made by the implementor of the class, on a class-by-class basis.

2 The compatibility check between the types of objects to see if the substitution rule can be used, could be based directly on their signatures, rather than the way they were constructed.[65] This is an extension of the use of *structural equivalence* of types, as in Algol 68,[66] to the type checking of objects.

Both these solutions can continue to work with the efficient message dispatch scheme used by C++ which is described in the next section, but the second approach is used in the rest of this thesis. It is more general, and can be easier to implement, since explicit mechanisms for defining the type hierarchy do not need to be provided.

This approach, based on structural equivalence or the *conformity* of types is used by Emerald, which also replaces primitive classes by an *object constructor* primitive. A function that returns the result of executing an object constructor is similar to a class in other systems, but Emerald does not appear to have explicit mechanisms for the inheritance or composition of object implementations.[67, 68]

1.6 The Implementation of Object-Oriented Systems

Here we shall review the implementation techniques that are commonly used in class-based object-oriented systems. In particular the implementations of Smalltalk-80 and virtual functions in C++ will be described, as these are representative of the techniques that have been used in run-time typed, and statically typed languages, respectively. C++ has probably the lowest overhead of any object-oriented language, and so represents a good target for efficient implementations. To be fair though, C++ achieves this by using the static type information, which places some restrictions on the programmer. The additional problems caused by multiple inheritance will be discussed later.

There are two related problems that are faced by any implementation of this kind of object-oriented system:

1 Allowing the substitution of objects with related behaviours.

2 Implementing inheritance, which in turn implies that we should allow the five properties of inheritance given previously.

Both these problems are, in practice, solved by the same mechanism — the dynamic binding of messages to methods, and in some sense it is the presence of this dynamic binding that makes methods different from the procedures exported by abstract data types. Dynamic binding allows the object that receives a message to

decide which method is to implement that behaviour.

Substitution is now allowed because the context into which an object is substituted does not need to know anything about its methods, only about the messages that it can send to the object. This requires a run-time representation of messages, and for the dynamic binding to be inexpensive since it will occur with a frequency similar to that of function calls in a conventional system.†

An object in these systems is represented by a pointer to the block of memory which contains the object's state, rather like a structure in conventional languages. The first word in this block is provided by the system, and is used to implement the dynamic binding of messages to methods that occurs in message dispatch. This word is called the *class pointer* since it contains a pointer to information about the class of the object. The rest of the block contains the object's instance variables, which are arranged so that, for every class, the state of its superclass precedes its own state in the memory block.

For example consider the class hierarchy shown in figure 1.5, which shows two classes, C and D, inheriting from class B, which itself inherits from class A. Figure 1.6 shows how instances of these classes would be organised in memory.

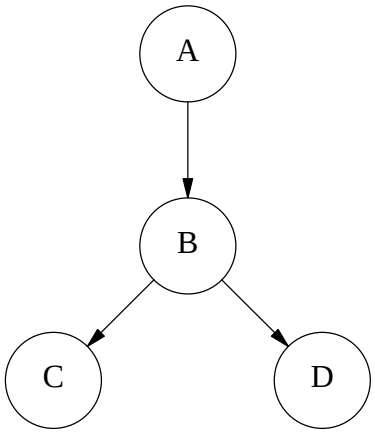


Figure 1.5: An Example Class Hierarchy

† Actually message sends are likely to occur more frequently than this would suggest, because the encapsulation of objects implies that many things will need to be accessed by a message that would be accessed directly, as a structure field say, in a conventional system. It is a major challenge to these systems to show that this extra overhead can be acceptable, especially as the dynamic binding makes it difficult for such functions to be compiled in-line.

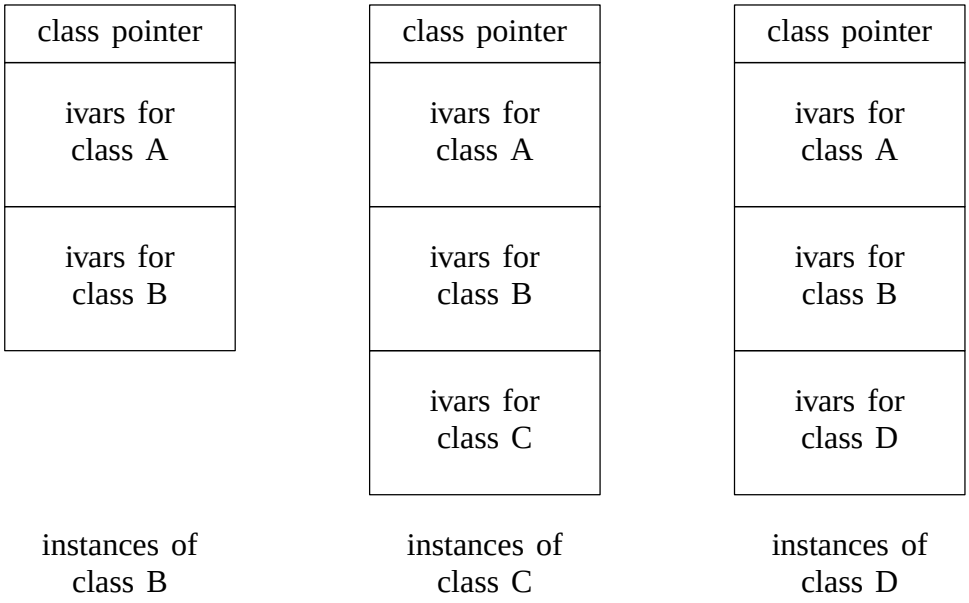


Figure 1.6: Memory Layout of Instances

Note that the instance storage for classes A and B is in the same place in all these instances. This is important because, in general, it means that all the instance variables defined by a class are at the same positions in all the instances, not just of that class, but also of all the classes that inherit from it. Thus methods can have the offsets of instance variables compiled into them, and continue to work with instances of subclasses of their class.

The static and dynamic typed systems differ most in the implementation of message dispatch, which will be discussed next.

Dynamic systems

In Smalltalk-80 the class pointer in an object points to another object that represents the object's class. One field in the class object points to the *message table* for that class, which is a hash table directly implementing the message to method mapping function for the messages defined by this class. Messages are also represented by objects — they are `symbols` which are string-like objects that are guaranteed to be unique in the system. This means that equality of object pointers can be used to efficiently check for the equality of messages.

When a message is sent to an object, it is looked up in the message table found in the object's class. Two things can then happen, either a method is found, in which case it can be invoked, or the lookup fails because the message is not defined in this class. In this case another pointer in the class object is followed, which points to the object representing the superclass of this class, and the lookup is repeated. This continues until either a method is found in one of the classes, or the superclass pointer is found to be nil, which means that the object does not understand the message, and an error is reported.

Notice that a message table may contain a mapping for a message that it inherits from a superclass, this allows the class to insert its own definition overriding the inherited one.

When the method is invoked it is given access to a pointer to the object for which it is executing. If it sends messages to this object the lookup process starts again at the bottom of the inheritance chain, giving the method access to overriding behaviour. It is also possible for a method to send a message to ''super'', which means that the lookup process for a message starts in the class immediately inherited by the class in which the method is defined, so that the superclass's behaviour for the message can be accessed.

Using this technique several hash tables might have to be searched during each message send — an operation that one would not expect to be very quick. However in practice messages usually exhibit considerable dynamic locality. If a message is sent to an object, then it is likely that the next send of the message will also be to an instance of the same class. This means that a cache of the results of message dispatch operations can considerably reduce the total cost of message dispatch, and the resulting composite mechanism is efficient enough for practical use. Caching the result of message dispatches in the code, at the point at which the message is sent, further exploits this locality to good effect.[34] Unfortunately the cost of any particular message dispatch is still unpredictable, and will sometimes be worse than before, since the cache must now be updated with the results of searches. Even so, the best case remains significantly slower than a conventional procedure call.

The dynamic development environment which makes Smalltalk such a productive programming environment also, unfortunately, makes the kind of global analysis that is necessary to optimize message dispatch very difficult, since a change to a class would have to be propagated to all the classes that inherit from it, which might then need to be re-analysed.

Static systems

When C++ is compiled more information is available than when a Smalltalk class is compiled, and this allows a much more efficient message dispatch mechanism. In particular it is known from the object's type that no message lookup can fail. Single inheritance allows messages to be allocated linearly, superclass-first, in just the same way as instance variables are allocated to positions in instances of the class. A message in C++ is represented at runtime by an index into the object's message table which contains pointers to the methods arranged for that object. The run-time action of sending a message to an object consists of following its class pointer, which points to the message table, and indexing from this with the message index. This is sufficiently simple that it can be compiled in-line at each message send, and means that message sends require only a few memory cycles of overhead in addition to that for a normal function call.

An example will clarify this procedure. Consider the C++ classes in program 1.1

```
 1:class A {
 2:      int ivar_a;
 3:public:
 4:      virtual int getval();
 5:};
 6:class B : public A {
 7:      int ivar_b;
 8:public:
 9:      virtual int getval();
10:      virtual int set_b(int new_b);
11:};
```

Program 1.1

Here class `B` inherits the behaviour of class `A`, possibly overriding the definition of the inherited method `getval`, and adding some new behaviour of its own. Instances of `A` and `B` (called `an_a` and `a_b`) would appear in memory as shown in figure 1.7.

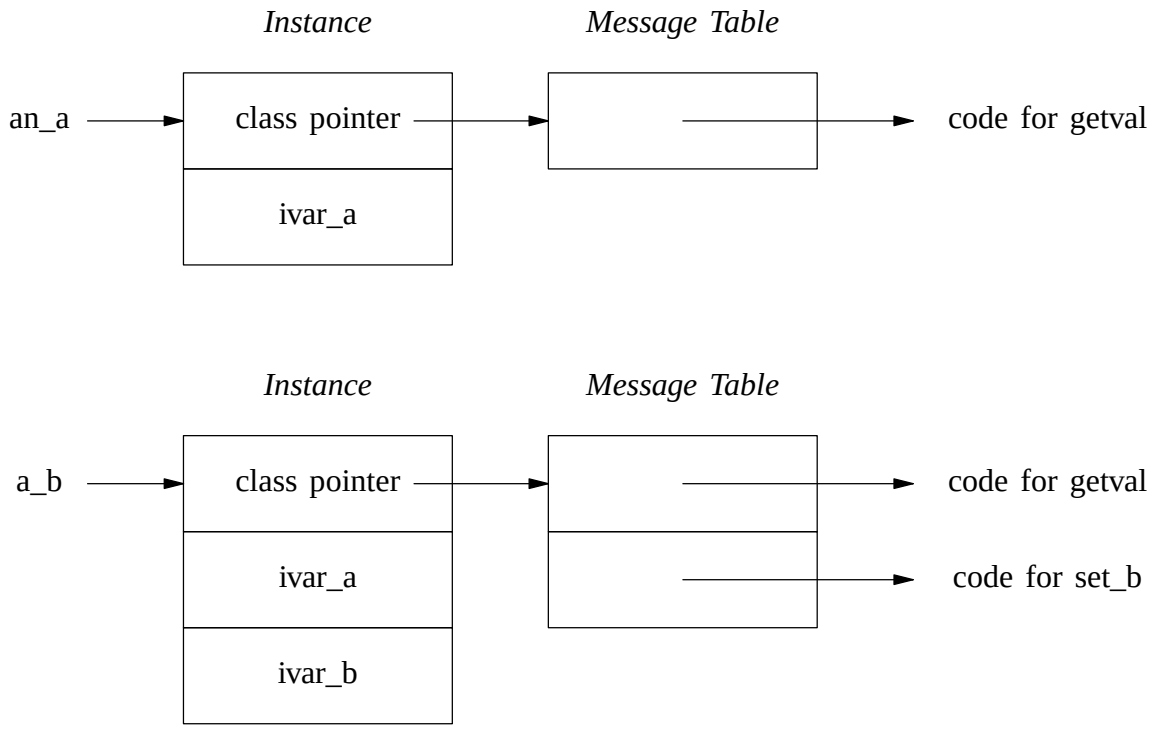


Figure 1.7: Object Implementation in C++

Clearly the structure of the instance of `B` is an extension (in several places) of the structure of `A`. Thus any code that is given a pointer to `a_b` instead of a pointer to `an_a` will continue to work. However the message table entry for `getval` can point to different methods in the different tables, so that the method that is actually invoked by a `getval` message to `a_b` might not be the same as that which would be invoked by the message being sent to `an_a`. The extra fields that are present in the structures for `a_b` are not accessible using pointers declared to be to objects which are instances of class `A`, since instances of `A` do not have these fields.

When the method found in the message table is invoked it is given, as an extra parameter, a pointer to the receiving object, which the programmer can access in the method using the pseudo-variable `this`. This pointer is also used to access instance variables, and to obtain the class pointer for the object when messages are sent to self. Thus messages to self can invoke overriding methods referred to by this message table, that might have been defined by inheriting classes. Access to overridden behaviour inherited from a superclass is simple since it can be determined statically which method will be invoked by any invocation, and so a normal C function call can be used.

It would appear from this description that additional behaviour provided by an object cannot be accessed if the object can only be accessed by pointers declared to be to instances of its superclasses, but this is not quite true. Consider an overriding method for `getval`. Since this was defined in class `B`, when it is invoked the parameter giving its self pointer is known to point to an instance of `B`, even though the code that sent this message could only assume that it was an instance of class `A`. This widening of the type is rather restricted, in comparison with the freedom allowed by the run-time checking of the validity of message sends to objects, but can nevertheless be used to good effect.

Multiple Inheritance

Multiple inheritance makes it impossible to allocate object resources in the strict order of inheritance (unless the ‘‘linear’’ technique is used) which was used, both for instance variables and messages in the previous techniques. There is a simple solution to this problem, which is to let the offsets for fields of structures change. This means that a run-time mapping from a structure tag, to its position in the structure must be made. This is the approach adopted by Amber,[69] which uses caching techniques, similar to those for message lookup, to make this acceptably efficient.

A more subtle approach, first suggested by Stein Krogdahl[31, 30] is used to implement multiple inheritance in version 2 of C++.[70] The idea is to include in the message table, for each method, the offset of the instance variables that it uses. Since all these instance variables must have been declared when the class was defined, they will occur as a single block of storage in the object. Thus a single offset can adjust the self pointer to get a pointer to the instance variables. Careful adjustments must then be made to these object pointers whenever coercions take place.

Each method expects the first word of the instance storage it sees to be a class pointer. This is important since it allows overriding behaviour that is in the message table (there are now several message tables for each class) to contain different offsets which will maintain the self pointer even in the most complicated cases, where messages to self invoke methods with access to different subsets of the object’s state. Unfortunately this means that an object might now contain several message table pointers, giving a small space overhead, but there seems to be no way to avoid this.

Chapter 2

Strongly-Typed First-Class Functions

Many recent languages and systems have explored the effect of removing arbitrary, or implementation-driven restrictions. In particular Scheme,[71] a dialect of LISP, has promoted the use of *first-class functions* with lexical scope in the context of conventional imperative languages.† First-class, or *higher-order* functions are also common in *functional*, or *applicative* programming languages, as promoted by Backus.[73]

First-class functions can be stored in variables, given as parameters to other functions, and returned in the results of a function. The use of lexical scope means that these functions carry with them the environment in which they were defined. The combination implies that function activation records cannot always be allocated on a stack, since an activation record might have to out-live the execution of the function for which it was created in order to retain the environment for a function defined within it. In the most general situations garbage collection is needed to reclaim stack frames.

Scheme is a runtime typed language. More recently several languages with first-class functions have been developed that are statically type checked. Examples are Standard ML,[60] Poly,[74] and Russell.[59] These languages have *polymorphic* type systems, which are much less restrictive than the type systems of languages such as Pascal, but are not sufficiently expressive for the efficient implementations of objects.

In principle the choice of type system used by an implementation language should have little bearing on the efficiency of the resulting implementation. In practice however, this is far from the case. One aim of type systems is to provide additional information that can be used by compilers so that they can verify the validity of code improvements. An underlying theme of these chapters on the implementation of objects, is a demonstration of the converse: unsophisticated type systems can unduly restrict an implementation, forcing it to use less efficient techniques. This is because sufficiently subtle relationships between functions and values cannot be formally expressed, and thus the safety of applications of these functions cannot be demonstrated to the system.

The combination of a function with hidden state is very similar to an object. The purpose of this chapter is to demonstrate, in an informal but practical way, that first-class functions, and objects are abstractions that can exhibit similar properties and can be used to model each other. The use of first-class functions for this, in run-time typed languages, is well understood,[42] but here we are concerned with statically strongly-typed languages.

---

† Scheme is also important for allowing the use of *first-class continuations*.[72]

First the implementation of simple first-class functions by objects will briefly be discussed. However objects have a richer structure than first-class functions, and so most of the chapter will examine the modelling of objects by first-class functions. First simple objects are described, introducing the basic techniques. Inheritance is then discussed, and the implications for the type system examined. Finally the costs of various ways of providing objects are discussed, before considering some of the ways in which the current techniques for defining and using objects might be extended using ideas from previous sections and the flexibility of first-class functions.

C++ and Standard ML are the languages used for examples. C++ being used for examples in an object-oriented language, and Standard ML for those requiring first-class functions.

2.1 Objects Can Implement Simple First-Class Functions

Objects are, in a sense, generalisations of functions, since an object ‘packages together’ the methods provided by it, which then share the state stored in the object. Thus it is quite easy to use an object to get the same effect as a first-class function. A function that does not access non-local variables is trivially mapped onto a simple class, consider:

```
1:fun twice x = 2*x
```

Program 2.1

which is modelled by sending the message `doit` to an instance of the C++ class:

```
1:class twice {
2:      int doit(int x);
3:};

4:int twice::doit(int x)
5:{
6:      return 2*x;
7:}
```

Program 2.2

Now consider a function which, each time it is called, returns one more than its previous value — as might be used to generate unique symbols. The normal technique for maintaining this kind of local state is to store it in non-local variables, viz:

```
1:val counter =
2:  let val value = ref 0
3:  in
4:    fn () => ( value := !value + 1;
5:               !value )
6:  end
```

Program 2.3

Line 1 introduces the definition of a value (in this case a function) called `counter`, whose body is defined by lines 4 and 5. These occur in a scope that includes `value`, defined on line 2 to be a cell initially containing the value zero. Cells are explicitly de-referenced by the prefix operator "`!`", and are assigned new values by "`:=`". The only reference in the system to the cell is from this function, so that calls of `counter` return the values `1,2,3,...`, etc. Thus we have defined a single counting function. Usually more than one such function is required, and so a more realistic example defines a function which returns a new counting function each time it is called:

```
1:fun make_counter first_value =
2:  let val value = ref (first_value - 1)
3:  in
4:    fn () => ( value := !value + 1;
5:               !value )
6:  end
```

Program 2.4

```
1:class counter {
2:      int value;
3:public:
4:      counter(int first_value);
5:
6:      int next_value();
7:};

8:counter::counter(int first_value)
9:{
10:     value = first_value - 1;
11:}

12:int counter::next_value()
13:{
14:     value = value + 1;
15:     return value;
16:}
```

Program 2.5

The equivalent using objects, written in C++, is given in program 2.5. Each time the *constructor* `counter` is executed, a new counter object is created. Given a counter object, called `o`, this can then be 'called' by evaluating "`o.next_value()`", which calls the method `next_value`.

In effect the programmer is explicitly manipulating the non-local state now stored in the object, which would have been maintained automatically by first-class functions.

Groups of first-class functions, which share some non-local state are naturally mapped onto an object, holding the state, and a method for each first-class function. However more complex sharing between functions might be difficult to model, since each function may have access to a different subset of the non-local data, different items of which might have different lifetimes.

First-class functions also allow more flexible control over the visibility of methods since references to those might also be caught as non-local data. Object-oriented languages that allow such visibility to be controlled must do so by language primitives, such as the `private`, `public`, and `protected` keywords in C++, or by convention, as in Smalltalk-80.

Both ML and C++ are strongly typed languages, but no discussion of this was required in these examples. In general monomorphic first-class functions can be represented by C++ instances. No attempt has been made to map polymorphic functions onto the more limited *inclusion polymorphism* provided by C++, and in general this cannot be achieved without subverting the C++ type system.

2.2 First-Class Functions Can Implement Objects

It is obvious that the techniques of the previous section can be used 'in reverse' so that simple objects are implemented by first-class functions, this approach is discussed in references,[42, 65] and elsewhere. The techniques will be reviewed through an example, described by the C++ class definition given in program 2.6.

```
 1:class accumulator {
 2:      int acc;
 3:public:
 4:      accumulator();
 5:
 6:      virtual void add(int value);
 7:      virtual int  total();
 8:      virtual char *image();
 9:};

10:accumulator::accumulator()
11:{
12:      acc = 0;
13:}

14:void accumulator::add(int value)
15:{
16:      acc += value;
17:}

18:int accumulator::total()
19:{
20:      return acc;
21:}
```

Program 2.6

A new accumulator object ("`a`" say) would be created by the constructor "`accumulator()`". It then keeps a running total of all the values given to it by calls such as "`a.add(10)`", which can be obtained by the rest of the program by calling "`a.total()`". This leads quite simply to the ML equivalent in program 2.7 (`image` will be introduced in program 2.10).

```
1:fun accumulator () =
2:  let val acc = ref 0
3:  in
4:    { add   = fn value => acc := !acc + value,
5:      total = fn ()    => !acc
6:    }
7:  end
```

Program 2.7

Some observations should be made at this point. Firstly, the ML program returns a record, the elements of which are the first-class functions implementing the methods of the C++ object. It could equally well have returned a tuple, but it is useful to be able to associate names with the functions. However it would not be possible to return a list, since the functions do not all have the same type.

The functions both refer to the non-local cell `acc`, which is otherwise invisible to the rest of the program. The value in the cell is increased by calls such as "`#add a 10`", and the total is obtained by "`#total a ()`". The ML notation "`#xxx`" provides a field selector function for the field `xxx`, and function application binds to the left, so that "`#add a 10`" is equivalent to "`(#add a) 10`" and calls the function stored as field `add` in the structure `a`, with the parameter `10`.

Program 2.7 is very much shorter than its C++ equivalent, this is because the type declaration can be elided in ML, but must be given in full in C++ (The C++ layout style also tends to make it longer). Both programs have well-formed types, the type of `accumulator` in program 2.7 being:

```
unit -> {add:   int  -> unit,
         total: unit -> int
        }
```

It is possible to separate the definition of the methods, from that of the record that is returned, which is sometimes useful, and is shown in program 2.8:

```
1:fun accumulator () =
2:  let
3:    val acc = ref 0
4:    fun add value = acc := !acc + value
5:    and total ()  = !acc
6:  in
7:    {add=add, total=total }
8:  end
```

Program 2.8

A very basic facility in object-oriented programming is the ability to *send a message to self*,† that is, to call another method defined for the same object as the

† In C++ a pointer to self is denoted by the *pseudo variable* `this`.

method doing the call. Consider the C++ definition of the method `image` as declared in program 2.6:

```
1:char *accumulator::image()
2:{
3:       return itos(total());
4:}
```

Program 2.9

(Here `itos` is a (hypothetical) library function that returns newly allocated memory containing the string representation of its number parameter.)

Since this is a method of the class `accumulator`, `image` could refer directly to the instance variable `acc`. Instead the method intended for public access to this value was used, which makes the `image` routine more independent of the rest of the implementation of the class (something that only becomes important in larger, more realistic classes), and interacts with inheritance, as will be seen later. The fact that this is a message to self could have been made explicit by replacing line 3 by the equivalent:

```
3:   return itos(this->total());
```

but this is not necessary, since other methods of a class are directly in scope inside a method, in the same way as the instance variables.

The complete ML implementation of this might be:

```
1:fun accumulator () =
2:  let
3:    val acc = ref 0
4:    fun add value = acc := !acc + value
5:    and total ()  = !acc
6:    and image ()  = makestring(total())
7:  in
8:    {add=add, total=total, image=image }
9:  end
```

Program 2.10

and this retains the simple syntax for the call of `total`. However this is not a true implementation of the C++ routine, since the call of `total` on line 6 is statically bound to the implementation of the function, and so is not a message to self. In contrast `total` in the C++ function (line 3, program 2.9) is resolved to the actual method (by an array access) at run time.

It is worth noting in passing, that this ML program would correctly implement the C++ program if the declaration of the method `total` (line 7, program 2.6) had not included the keyword `virtual`, which would have directed the compiler to bind calls to the method statically.

The dynamic binding of messages to self only becomes important when we consider classes that inherit from `accumulator`, but it will be demonstrated in this simpler case to ease the presentation of inheritance which follows. References to `this` are a form of recursion,[75] and introducing the recursion explicitly allows

us to more closely model the C++ semantics.

First, it is convenient to define a type and a function to aid in the construction of recursive references — ML pointers do not allow a *Null-pointer* value, as is common in other languages, and so we construct this using a union:

```
 1:datatype 'a pointer = none | ptr of 'a;
 2:exception NotSet;

 3:fun deref (ref(ptr x)) = x
 4:  | deref _            = raise NotSet;
```

Program 2.11

Typically a `pointer ref` will be created with the initial value `none`, and will later be assigned the recursive reference. This can then be dereferenced by calling `deref`. A correctly constructed program will never call `deref` before a value has been assigned to the reference, so that the exception `NotSet` should never be raised. In this way we can construct recursive data structures, which are not allowed statically by ML.

This is used in the new definition of `accumulator` which includes a message to self on line 14:

```
 1:type accumulator_type = {add:    int -> unit,
 2:                         total: unit -> int,
 3:                         image: unit -> string
 4:                        };

 5:fun accumulator () =
 6:  let
 7:    val acc  = ref 0
 8:    val self = ref (none: accumulator_type pointer)
 9:
10:    val myself =
11:          {add   = fn value => acc := !acc + value,
12:           total = fn ()    => !acc,
13:           image = fn ()    =>
14:                       makestring(#total (deref self) ())
15:          }
16:  in
17:    self := ptr myself;
18:    myself
19:  end;
```

Program 2.12

Here the type of the `accumulator` object must be defined, so that the initial value of the variable `self` can be given the correct type — only rather limited type inference is possible when `ref`s are used.

So far we have seen that first class functions can be used to implement simple objects, the next section will discuss the ways in which inheritance can be introduced into this scheme.

2.3 First-Class Functions and Inheritance

To discuss inheritance we will introduce some new C++ classes, which inherit from the class `accumulator` defined by program 2.6. Inheritance is most often used to extend the protocol of an existing class, as an example the class `s_accumulator` extends the protocol of `accumulator` by adding a new method called `sub` which subtracts its parameter from the value in the object. This is done indirectly, by calling the inherited `add`, since the cell `acc` is not visible outside `accumulator`. All the methods defined by `accumulator` are implicitly defined for `s_accumulator` objects.

```
1:class s_accumulator : public accumulator {
2:public:
3:      virtual void sub(int value);
4:};

5:void s_accumulator::sub(int value)
6:{
7:      add(-value);
8:}
```

Program 2.13

This can be implemented simply in ML by having the object-creation routine, call the routine representing the inherited class, and patch together the result and the new method:

```
 1:fun s_accumulator () =
 2:  let
 3:      val sup = accumulator ()
 4:  in {
 5:      add   = #add sup,
 6:      total = #total sup,
 7:      image = #image sup,
 8:      sub   = fn value => #add sup (~value)
 9:  }
10:  end
```

Program 2.14

This has the well-formed type:

```
unit -> {add:   int   -> unit,
         total: unit  -> int,
         image: unit  -> string,
         sub:   int   -> unit
        }
```

While this captures admirably the inheritance of the implementation of `accumulator`, the type of this function is unrelated to the type of `accumulator`. In contrast the type of C++ `s_accumulator` objects is a *subtype* of the type `accumulator`. This means that wherever an `accumulator` instance is expected, an `s_accumulator` instance may be used instead. In

particular a variable declared as (pointing to) an `accumulator` might in fact be (pointing to) an `s_accumulator`, or indeed any instance whose class inherits from `accumulator`. This limited polymorphism is both cheap to implement, and surprisingly powerful, but the lack of any subtype relationship in ML does not allow it. It is safe, since any subtype must include (from the way it is constructed) all the operations, and visible data items, of the inherited type.

Before discussing possible solutions to this problem, several more C++ examples will be presented to demonstrate the properties that are required from an implementation. The first example shows the way in which this limited polymorphism can be used.

```
 1:class averager : public accumulator {
 2:      int items;
 3:public:
 4:      averager();
 5:
 6:      virtual int average();
 7:};

 8:averager::averager()
 9:{
10:      items = 0;
11:}

12:int averager::average()
13:{
14:      return total() / items;
15:}

16:void averager::add(int value)
17:{
18:      accumulator::add(value);
19:      items++;
20:}
```

Program 2.15

Here the new method `average` has been defined, and the inherited function `add` has been redefined, or *overridden*. The new version calls the old, resolving the name using the scope operator "`::`". It also updates a counter, so that the average can be calculated later.

Since this class is a subtype of `accumulator` an instance of it can be given to a routine expecting an `accumulator` object, and then the average of the numbers given to it can be found (but only using a reference to the object declared as an `averager`).†

† This is a simplistic, and as a result slightly dangerous example. Note that it exposes implementation details of the routines which use it — specifically, the number of times they call `add` affects the result. Since these routines believe they are using an `accumulator` they may not call this function in the simple way assumed.

It only remains to see what happens when a method in an inherited class refers to self.

```
 1:class scaled_accumulator : public accumulator {
 2:      int scale;
 3:public:
 4:      scaled_accumulator(int s);
 5:
 6:      virtual int total();
 7:};

 8:scaled_accumulator::scaled_accumulator(int s)
 9:{
10:      scale = s;
11:}

12:int scaled_accumulator::total()
13:{
14:      return scale * accumulator::total();
15:}
```

Program 2.16

`Scaled_accumulator` overrides `total` so that the value returned is scaled by the factor (`scale`) set when the object was created. The new version of `total` calls the overridden version, since the inherited instance variable is not visible. Consider now the inherited method `image`, which returns a string containing the representation of the value returned by total. When this executes "`this->total()`"(line 3 program 2.9), the method defined in the subclass will be called, so that `image` will be given the scaled result.

This dynamic binding of self in superclasses to the instance of the subclass provides the access to overriding behaviour that was the fifth property required by inheritance in the previous chapter.

2.3.1 Using Dynamic Type-checking

Of course LISP-like languages have been used to implement objects many times. However these have always relied on the runtime-typed nature of the languages,[42, 76] which allows the use of objects as if they were instances of a superclass, making program 2.14 a satisfactory implementation of `s_accumulator`. However, to achieve recursive references to self a more dynamic mechanism is usually used.

The following examples are presented in a (hypothetical) runtime typed language, with an ML-like syntax. The usual approach is to introduce a *dispatch* function, which is called whenever a message is sent to an object. This function then invokes the appropriate method. First we must re-implement `accumulator` using this style:

```
 1:fun accumulator () =
 2:  let
 3:    val acc = ref 0
 4:    fun add value = acc := !acc + value
 5:    fun total = !acc
 6:    fun image = makestring (!self "total")
 7:
 8:    val self = ref fn message . params =>
 9:             case message of
10:                 "add"   => add params
11:               | "total" => total params
12:               | "image" => image params
13:               | _       => raise does_not_understand
14:  in
15:   ( !self, (fn newself => self := newself) )
16:  end
```

Program 2.17

Here "`message . params`" is intended to associate `message` with the first parameter (which is expected to be a `string`), and `params` with the remaining parameters, or with "`()`" if there are none. The dispatch function (lines 8-13) attempts to match the requested operation against those which are provided. If found it applies the operation to its parameters, otherwise it raises the exception `does_not_understand`, to communicate the runtime type error to the system.

Note that in a "real" implementation each operation would be named by a unique *symbol*, so that the tests would simply be identity tests, rather than the (expensive) string equality above. In classes with many methods the dispatch function could use hashing, or some other search technique, to improve on the linear search.

As before new objects are created by calling the function `accumulator`, which now returns a pair. In normal use only the first element of this pair, the dispatch function, is used, and the other function should be discarded. Local references to the dispatch function are made through the variable `self`, for example on line 6. The second function in the result pair allows the extended dispatch function defined by a subclass to replace the one defined locally. Then a message to self may be mapped by the dispatch function to a member function defined in the inheriting class.

```
 1:fun scaled_accumulator scale =
 2:  let
 3:    val (super, set_super_self) = accumulator ()
 4:    fun total = scale * (super "total")
 5:
 6:    val self = ref fn message . params =
 7:               case message of
 8:                   "total"  => total params
 9:                 | _        => super message . params
10:  in
11:   set_super_self self;
12:   ( !self, (fn newself => set_super_self newself;
13:                           self := newself) )
14:  end
```

Program 2.18

This implementation of `scaled_accumulator` inherits from `accumulator` by calling its creation function on line 3. This sets two variables — `super` is used where the superclass' un-overridden behaviour is required, such as on line 4, and `set_super_self`, is used on line 11 to make the super object's self variable refer to the local dispatch function. A similar function is defined on lines 12 and 13, so that a class that inherits this one can also install an even more specific dispatch function. Thus the dispatch function of the ultimate class will be the one used by all of its superclasses when they refer to self.

Runtime type checking has been used for two distinct purposes here, firstly it has been used to allow the use of an instance of a subclass, where an instance of the superclass was expected, and secondly it has been used in the definition of the dispatch function. Specifically, it is needed to allow the communication of parameters to member functions, and the returning of their results. It might appear that this use could be avoided by changing the dispatch function to return the method, which could then be applied to its parameters, however this would involve an equivalent type insecurity, since the dispatch function would need to be able to return functions of differing types.

It should be noted that these requirements for runtime type checking are not inherent in the object-oriented technique — C++ is a sufficient counter-example, where objects can be used within the requirements of strong type checking. The problem comes from the type system of ML, and similar languages, which while very expressive, do not have inclusion polymorphism. The type system proposed by Cardelli[65] shows one way in which these type systems might be extended to include subtypes, and this will be discussed further in the next two chapters. In contrast the following technique allows objects, with inheritance, to be constructed within the constraints of an ML-like type system, albeit with a greater runtime cost.

2.3.2 Using Type-coercion

The requirement for runtime type checking to allow the definition of the dispatch function is avoided by returning to the previous mechanisms for dispatching to methods, so that an object is again represented by a record of the methods. This leaves the essential problem, the use of inheriting objects where instances of the superclass are expected, which is removed by introducing a different, but equivalent way of thinking about subtypes and inheritance.

If a class `B` inherits from the class `A` it is traditional to think of instances of `B` as more specific versions of instances of `A`. Let `a` and `b` be instances of `A` and `B`, respectively. Then `b` *is a* `A`, and can be used wherever `a` can be used. A different way of looking at this is that whenever an instance of `A` is required, `b`'s type can be changed (possibly automatically) to be indistinguishable from `A`. This will be called *type-coercion*, note that this does not involve any change to the object (as might be implied by *coercion*). Now inheritance becomes the specification of which type-coercion operations should be provided. In this example a type-coercion which will change `b` into an `A` should be provided, but not one type-coercing `a` to be a `B`. Programs 2.19 and 2.20 use this technique to provide type-correct implementations of the C++ classes `accumulator` and `scaled_accumulator`.

```
 1:type accumulator_type = {add:    int -> unit,
 2:                         total: unit -> int,
 3:                         image: unit -> string
 4:                        };

 5:fun accumulator () =
 6:  let
 7:    val acc = ref 0
 8:    val self = ref (none: accumulator_type pointer)
 9:
10:    fun add value = acc := !acc + value
11:    fun total ()  = !acc
12:    fun image ()  =
13:                 makestring(#total (deref self) ())
14:
15:    val myself =
16:          {add   = add,
17:           total = total,
18:           image = image
19:          }
20:  in
21:    self := ptr myself;
22:    ( myself, (fn newself => self := ptr newself) )
23:  end;
```

Program 2.19

`Accumulator` now has the type:

```
unit -> ( accumulator_type, accumulator_type -> unit )
```

As before the object creation function returns a pair, the second element of which should be discarded, except by another creation routine that is inheriting this behaviour. Inheriting objects use the second function to install their version of self, this time a method record, containing references to the overriding functions.

```
 1:type scaled_accumulator_type =
 2:                 {add:    int -> unit,
 3:                  total: unit -> int,
 4:                  image: unit -> string,
 5:                  accumulator_protocol: accumulator_type
 6:                 };

 7:fun scaled_accumulator scale =
 8:  let
 9:    val (super, set_super_self) = accumulator ()
10:    val self =
11:             ref (none: scaled_accumulator_type pointer)
12:    fun total () = scale * (#total super ())
13:
14:    val myself =
15:          {add   = #add super,
16:           total = total,
17:           image = #image super,
18:           accumulator_protocol =
19:                       {add   = #add super,
20:                        total = total,
21:                        image = #image super
22:                       }
23:          }
24:  in
25:    self := ptr myself;
26:    set_super_self (#accumulator_protocol myself);
27:    ( myself,
28:      (fn newself =>
29:          (self := ptr newself;
30:           set_super_self (#accumulator_protocol newself))) )
31:  end;
```

Program 2.20

Which has the type:

```
int -> ( scaled_accumulator_type,
         scaled_accumulator_type -> unit )
```

Thus an instance `o`, returned by the function `scaled_accumulator` can be type-coerced to an `accumulator` by evaluating the expression "`#accumulator_protocol o`", which may then be used wherever an instance of `accumulator` may occur. However if the message `total` is sent to it, the function defined in `scaled_accumulator` will be executed. Also the inherited function `image` will use the overriding version of `total`.

Similarly `averager` might be defined by program 2.21, giving it the type:

```
unit -> (averager_type, averager_type -> unit )
```

Here we see that as well as overriding the `add` function, an extra method `average` is introduced that can be used wherever the true type of the instance is known.

```
type averager_type =
          {add:      int -> unit,
           total:   unit -> int,
           image:   unit -> string,
           average: unit -> real,
           accumulator_protocol: accumulator_type
          };

fun averager () =
  let
    val nitems = ref 0;
    val (super, set_super_self) = accumulator ()
    val self = ref (none: averager_type pointer)

    fun average () =
          real(#total (deref self)()) / real(!nitems)
    fun add value  = ( #add super value;
                       nitems := !nitems + 1)
    val myself =
          {add     = add,
           total   = #total super,
           image   = #image super,
           average = average,
           accumulator_protocol =
                       {add   = add,
                        total = #total super,
                        image = #image super
                       }
          }
  in
    self := ptr myself;
    set_super_self (#accumulator_protocol myself);
    ( myself,
      (fn newself =>
          (self := ptr newself;
           set_super_self (#accumulator_protocol newself))) )
  end
```

Program 2.21

In this way all the properties of inheritance given previously can be reproduced in a strongly-typed language with first-class functions. Classes can be defined that allow instances to be used as if they were instances of a superclass, and methods defined in the subclass override those inherited from the superclass, while

still being able to use the overridden methods themselves. Finally methods in a superclass call overriding methods from the subclass, if they exist, in preference to those in their own class.

The notational penalty for this is the requirement that instances are explicitly type-coerced to the correct type. However there are already many other notational overheads involved in all the definitions of objects we have seen, and an automatic tool which generated these definitions would almost certainly be able to insert the correct type-coercions.

Since each object-creation function calls the function returned by its super object to re-bind self in the super object, and this in turn calls the function it received previously, it is clear that the superclass chain can be arbitrarily long, and general single inheritance can be used. But to achieve the inclusion polymorphism a type-coercion value needs to be included for each class from which the object inherits, both directly and indirectly.

2.3.3 Implementing Multiple-inheritance

This may be extended directly to multiple inheritance, where a class has more than one superclass. The introduction of multiple inheritance to systems such as Smalltalk-80 normally causes three problems:

- With single inheritance storage for instance variables can be allocated linearly in instances, so that the storage associated with each class is always at the same offset in instances regardless of the classes that might inherit it and also contribute their own instance variables. This is not possible with multiple inheritance, and so it is normally necessary for the offsets of a class' instance variables to be determined at runtime.

- In a similar way, systems such as C++ (before multiple inheritance was added) allocated space in the arrays of method pointers linearly. So that the offset of a particular method was always the same. Again this is not possible with multiple inheritance.

- Multiple inheritance introduces possibilities for conflicts, for example when methods for the same message, are inherited from more than one superclass. Different systems have chosen to resolve these ambiguities in different ways, so that programming in Eiffel[77] is quite different from say, Flavors.[78]

Implementing inheritance by type-coercion the first two problems do not occur. Instance variables are not stored in the object, but rather stored in closures and accessed via references in methods, so they do not compete for offsets in object storage. Also since the object record of a class is a different type from that of a superclass there is no constraint that methods occur in the same positions in both, only that one record can be made from the other.

Solutions to the third problem depend on the way in which multiple inheritance is to be defined, but since the object records are constructed by hand (or by a pre-compilation source-to-source translation) the programmer is free to resolve

conflicts in any way that is desired, and indeed is free to experiment with different resolution strategies. However this technique is biased towards the tree resolution of conflicts, and other strategies would require more effort to model.

It is interesting to note that the type-coercion mechanism introduced here is similar to that used by version 2 of C++, to allow multiple inheritance. However in C++ the coercions are used to overcome the first two difficulties outlined above, rather than attempting to allow inclusion polymorphism in a way that is type-correct in the underlying implementation language.[70]

2.4 The Cost of Objects

The previous section showed how objects, with inheritance, can be constructed in a strongly-typed language with first-class, lexically scoped functions. Here we shall consider briefly the relative costs of the techniques presented. Both the memory cost, and the runtime cost are of interest.

2.4.1 Memory usage

To put this discussion into context we will first consider the costs associated with objects in a language providing explicit support, in this case C++.† In C++ an object is simply a C structure that contains fields for each instance variable defined in its class, and all inherited classes. If any of these classes include the declaration of any `virtual` methods (as do all the examples here), a single *class pointer* is added to the structure pointing to an array of pointers to all the virtual methods. This array is shared by all instances of the class in the system, so the space overhead of inheritance is a pointer per object, and a single array with one pointer for each virtual method. This array could easily be a structure, like the records we introduce in ML, which would allow its type to be more fully described, but this is not important since it is not visible to the programmer.

When a message, corresponding to a virtual method, is sent to an object, the object's class pointer is followed, and the array indexed by a constant, which was generated at compile-time to represents the message. The result is the pointer to the code for the method. A pointer to the object is inserted as the first parameter of the function. Thus the overhead in calling a method in addition to that for a 'normal' call is, pushing the extra parameter, indirecting to get the class pointer, and indexing from this with a constant. The cost of objects in C++ is very small!

Next consider objects constructed using first-class functions in ML, but without inheritance, as in program 2.7 defining `accumulator`, and reproduced here:

---

† The following does not include the (small) extra overhead of multiple inheritance in more recent research versions of C++.[70]

```
 1:fun accumulator () =
 2:  let val acc = ref 0
 3:  in
 4:    { add   = fn value => acc := !acc + value,
 5:      total = fn ()    => !acc
 6:    }
 7:  end;
```

Program 2.7

In the following discussion a *cell* is the smallest unit of storage that can hold either an integer value, or a reference to another cell — typically this will be four bytes. Each time `accumulator` is called a cell, `acc`, is created. This is the storage for the instance variables. In addition a record is created containing two references to closures, one for each of the methods `add` and `total`, each of which contains a reference to the non-local cell `acc`.

The exact amount of storage used for these data structures will depend on the system, but a lower bound may be estimated. We will assume that a closure requires one cell for each object to which the function has a reference, plus a cell containing a reference to the function's code. Thus each time `accumulator` is called two cells are needed for the result record, two cells for each closure and one cell for the updatable value `acc`. Giving a total of $2+2\times2+1=7$ cells.

It is assumed that all values, apart from integers, are *boxed*, that is, are represented by pointers to storage, rather than being represented directly in the containing data structure. If the closures could be included directly in the result record, only $2\times2+1=5$ cells would be needed. Note that the in-line allocation of data structures introduces many other problems, and the author knows of no system that achieves it for user-defined data objects. It is not easy for `acc`'s storage to also be included, since both closures require references to the same cell.

None of this storage can be shared by all instances of the class, and so the implementation compares quite badly with the equivalent C++ requirement of 2 cells per object, and 2 cells shared by all instances of the class.

An alternative would be to allocate the instance variables as fields in the returned record, so that `accumulator` might become:

```
 1:fun accumulator () =
 2:  let
 3:    val self =
 4:          {acc   = ref 0,
 5:           add   = fn value => acc := !acc + value,
 6:           total = fn => !acc
 7:          }
 8:  in
 9:    self
10:  end;
```

Program 2.22

This also requires $1 + 2 \times 2 = 5$ cells per object (with in-line contexts).† By removing the recursive references, and adding a self parameter to each method the contexts for the member functions can be shared — giving an implementation very similar to that of C++. However neither approach has been pursued since the encapsulation of objects is missing, and they are not amenable either to single inheritance in a strongly-typed implementation language, or to multiple inheritance.

In the description of the costs of inheritance the following symbols will be used:

M^o_{class} = Memory cells per instance of *class*
M^c_{class} = Memory cells shared by all instances of *class*
F_{class} = Number of methods in *class*
I_{class} = Number of instance variables defined by *class*
$A(class)$ = The set of ancestor classes of *class*

Each closure returned by `accumulator` captures a reference to `acc`. In the majority of classes there will be many instance variables, and most closures must capture references to all or many of these. Thus, for a class c, the closure might need $I_c + 1$ cells, giving a worst case total of:

$$\begin{aligned} M^o_c &= F_c && \text{for returned record of closures} \\ &+ F_c(I_c + 1) && \text{for closures} \\ &+ I_c && \text{for instance variables} \end{aligned}$$

If several methods all have references to many instance variables it is possible for these to be allocated in a separate closure, to which each method's closure has a single reference. This is called *closure hoisting*, and is done by systems such as T.[79] In the ideal case, where all instance variables are allocated in-line in a single closure the memory requirements might be reduced to:

$$\begin{aligned} M^o_c &= F_c && \text{for returned record of closures} \\ &+ 2F_c && \text{for closures} \\ &+ I_c && \text{for shared closure of in-line instance variables} \end{aligned}$$

If the correct closures can be allocated in-line (in the returned record) this might be further reduced to:

$$\begin{aligned} M^o_c &= 2F_c && \text{for returned record of in-line closures} \\ &+ I_c && \text{for shared closure of in-line instance variables} \end{aligned}$$

which is still substantially more than C++'s:

† Actually the references to `acc` on lines 5 and 6 would have to be constructed using pointers, to be legal. This artificially increases the space overhead, and so is not shown here.

$$M_c^o = I_c + 1 \qquad \text{and} \qquad M_c^c = F_c$$

Assuming all the instance variables require a single cell.

If inheritance is now included the memory costs for C++ become:

$$M_c^o = 1 + \sum_{d \in A(c)} I_d \qquad \text{and} \qquad M_c^c = \sum_{d \in A(c)} F_d$$

The analysis for first-class functions is too complex to present in full. An object requires all the storage which its superclasses define. Space is also needed for the type coercion values, to which the object records must include references. Object records returned by superclasses must, in general, be kept, since these are used when methods use inherited, but locally overridden behaviour. However the `set_super_self` functions can be discarded after they have been used — indeed they should not be kept since they provide a way in which the consistency of the object could be destroyed.

To put these discussions into context we might consider a ‘typical’ class, defining 3 instance variables, 5 methods, and inheriting from 1 other ‘typical’ class. Each C++ instance of this class requires 7 cells, with 10 cells shared by all instances (and even fewer if methods override inherited behaviour). The worst case procedural implementation may use as many as 64 cells per object, which might be improved to ‘only’ 34 cells by closure hoisting and unboxed closures.

2.4.2 Run-time costs

The run-time costs of the procedural implementation of objects are more encouraging. Sending a message to an object only requires that the correct function is obtained from the object record, which is then invoked with the parameters. So the overhead in addition to that of a normal function call is the record access — almost the same as for C++. Object creation is more expensive however, both because the object is constructed by iteratively invoking all the superclasses, and since much more memory must be allocated, and initialised. A new overhead introduced by this technique is the execution of the type-coercions, but these are also simple record accesses so the overhead is small. However it should be remembered that the type-coercions must be constructed when the object is created, so the real cost is hidden here. It might be possible to construct some type-coercions when they are first needed, which would be more efficient in many situations.

2.4.3 Comparison with Dynamic Type-checking

An implementation based on dispatch functions typically uses less memory. The context for the dispatch function need only contain references to methods defined in its class, together with references to the dispatch function from the superclass, and a reference to its own code. This replaces the object record, and the type-coercion values are not required. Constants, such as the names of the methods,

can be shared by all instances of the class, as is the code of the dispatch function.

However the cost of a message send is much higher. When a message is sent to an object, its dispatch function must search for the method to be executed. A simple linear search will require, on average, half as many tests as there are methods defined in the class. This can be improved by hashing, but this requires more memory, none of which can be shared by all the instances of the class, unless an extra level of indirection is introduced.

Furthermore, the search through the superclasses to find a method defined in a superclass and not overridden must execute one dispatch function for each class. All except the last of these dispatch functions will determine that the method is not defined in that class, which requires the worst case search, testing against all the local methods. This could be avoided by 'copying down' inherited methods into the dispatch functions of inheriting classes. However this makes these tables as big as the object record, or even larger if the tables are sparse for hashed searching.

Thus the type-coercion technique for implementing objects, as well as being type-safe, makes sending messages to objects more efficient than using dispatch functions, at the expense of greater per-object memory requirements.

2.5 Extensions to Inheritance

New ways of thinking about constructs provided by programming languages often suggest extensions to the existing mechanisms, and new ways in which the mechanisms might be used. Considering inheritance as a mechanism for automatically providing type-coercion operations up the sub/super-type hierarchy (ie. towards the root), suggests *downwards* type-coercion. This would allow an instance to be type-coerced to be an instance of a subclass of its actual class.

This addresses an important weakness in existing object-oriented systems when using libraries of existing classes. Inheritance is intended to promote the use of libraries, since it allows libraries to be specialised for the specific problem being solved, however a difficulty occurs when code in the library creates instances of other classes, since the client has no opportunity to provide its own class, inheriting from the library class. Consider an integer class in a library that does not provide a method for the message `factorial`. It is easy to define a new integer class, inheriting from the one in the library, which adds this method, however `factorial` is still not defined on any of the integer objects that are returned by other library routines.

What is needed is a way to type-coerce an instance of a class into an instance of a class which inherits from it. For this to be safe, the type-coercion must provide methods for all messages that the subclass adds to those inherited from the superclass. It might also override inherited methods, but care must be taken since it is not clear whether these overriding methods will, or indeed should, be visible to those in superclasses. As an example of this, consider the following function which type-coerces an instance of `accumulator` into an instance of `scaled_accumulator`.

```
1:fun scale_acc acc scale =
2:     {add   = #add acc,
3:      total = fn () => scale * (#total acc ()),
4:      image = #image acc,
5:      accumulator_protocol = acc: accumulator_type
6:     }
```

Program 2.23

This function returns an object of the same type as instances of `scaled_accumulator`, and so it could be used wherever an instance of `scaled_accumulator` is expected, however the behaviour of the object is not quite the same. In particular the `image` method returns the string representation of the value stored in the instance of `accumulator`, rather than the scaled value, since the overriding of `total` is not visible to the superclass. An alternative type-coercion function might be:

```
1:fun scale_acc acc scale =
2:     {add   = fn inc => #add acc (scale * inc),
3:      total = #total acc,
4:      image = #image acc,
5:      accumulator_protocol = acc: accumulator_type
6:     }
```

Program 2.24

and this correctly duplicates the `scaled_accumulator` behaviour, by changing the value held by `accumulator`. However this would not be possible if, for example, `accumulator` made other use of its value, or the scale factor could be changed. Another alternative would be to override `image`, but this implies that its definition is known, and that the state required for its implementation is visible through the other methods.

A more general solution would be for all records representing objects to contain references to the `set_super_self` function returned when the object is created, in much the same way that they contain the type-coerced versions of the object, unfortunately this would increase the size of all objects, and provide a mechanism that would allow an object's behaviour to be modified from outside.

Downwards type-coercion is quite simple if no methods from the object are overridden. This only leaves the possibility of adding new protocol to an object, an ability which is still extremely useful. More general downwards type-coercion can usually be achieved by the creation of a new instance of the target class, as is presently necessary for all downward type-coercion, however this does not preserve sharing relationships with other parts of the program that might have references to the object, and may be changing or accessing its state. Thus, while downwards type-coercion is not a complete solution to these problems, it does provide an additional capability which may useful in many practical situations.

When objects are defined using inheritance and first class functions, the description must explicitly include the type-coercions to the super-types. What would it mean if these type-coercions were not given? It would not be possible to use such an object in place of an instance of its superclass. Thus this is a technique for hiding the details of the implementation of the object, it no longer being

possible to determine that the object was constructed by inheriting from another.

Conversely it would be possible for type-coercions to be present that return objects that appear to be instances of classes that are not actually superclasses of the original object. Any object represented by a record with methods of the same type as those defined by a class may be used in place of instances of that class — of course, it is also helpful if the *behaviour* is in some sense compatible.

Together these possibilities allow the classes from which an object appears to inherit, to be different from the actual way in which it is defined. This would allow the separation of the conceptual and implementation hierarchies, as is proposed by Halbert and O'Brien,[80] and by Snyder.[16]

Using type coercion it is also very easy to provide *or*-inheritance† as proposed by LaLonde, Thomas, and Pugh.[81] Since this only involves subsetting an object's protocol, it can be implemented directly, and does not suffer from the difficulties of downwards type-coercion.

More generally, several type coercions might exist which provide unrelated subsets of an object's protocol. For example, the operations on the writing end of a queue object could be separated from those acting on the reading end. This would allow the ends to be disseminated separately to different parts of a system. In a conventional object-oriented system extra objects would have to be introduced to represent the ends, which is less efficient, and requires more forethought when the classes are designed.

2.6 Conclusions

First-class functions and objects are each powerful enough to provide many of the facilities of the other. In particular it is possible to use first-class functions in a statically strongly-typed language, such as Standard ML, to provide objects by programming in a particular style. This style defines the way in which objects are created, and messages sent to them. In a strongly-typed language it must also make explicit the conversions where an object is to be regarded as an instance of one of its superclasses.

First-class functions are less specific, and hence more general, programming constructs than classes and objects, so implementing objects using first-class functions allows the advantages of object-oriented programming, while also allowing the use of more expressive first-class functions when they are needed. The underlying system does not have to directly support both mechanisms, and ensure compatibility between them. This can be compared with Smalltalk-80 whose *block* construct is similar to a first-class function, with semantics which cannot easily be defined in terms of objects and message-passing.

---

† This is probably better described as *intersection*-inheritance in contrast to 'normal' multiple inheritance where the resulting protocol is the union of those of its superclasses.

Regrettably the space required by objects constructed in this way is much greater than that which might be expected from a system providing objects directly. Thus it is unlikely that the techniques described could get more than limited use in practical programs. However a basic duality between the constructs is demonstrated, and the development of the technique helps to identify the areas where the strong typing of Standard ML is inconvenient. The following chapters investigate progressively more expressive type systems, which allow objects to be constructed more efficiently.

Despite possible inefficiencies, objects which are constructed in this way can provide a more flexible environment in which to experiment with different ways of defining objects, than is possible in a system which provides objects directly.

Chapter 3

Subtype Relationships

There have been several suggestions that subtype relationships between structure types can be introduced into type systems to allow the construction of *objects*.[10][65], This chapter investigates how subtype relationships can be used, but shows that they are still not sufficient to enable the efficient construction of objects.

3.1 The Subtype Relationship

The *raison d'être* of subtype relationships is the *substitution rule*, viz:

```
A is a subtype of B if, and only if,
wherever an instance of type B is expected,
an instance of type A can be given instead.

                The Substitution Rule
```

This rule both guides us in the definition of subtype relationships, and shows us how they can be used. It allows a value to be substituted for a value of a supertype at run-time without causing a type inconsistency, or run-time error. This is reminiscent of the substitution property of objects defined in chapter 1, and indeed the intention is that subtype rules can be used to enable systems to support the substitution property.

The subtype relationship becomes the type compatibility rule of the type system, previously this allowed an object to be used only if its type was *equal* to the expected type. Now a value whose type is a subtype of the expected type can also be used, which means that more programs are now type-correct, and so we have a more flexible type system.

In particular an instance of type A can be assigned to a variable declared to hold instances of type B, or given as an actual parameter when calling a function which is declared with a formal parameter of type B. The subtype relation can also be thought of in terms of the generality of the type, so that if A is a subtype of B, then B is *more general* than A.

The phrase ''x *is a subtype of* y'' will often be denoted by ''$x \leq y$''. The subtype relationship is a partial ordering on types, and if the type system includes ''bottom'' and ''top'' types, makes the type system a *lattice*. The reverse relationship will also be used:

```
A is a supertype of B if, and only if, B is a subtype of A.
```

and ''x *is a supertype of* y'' denoted by ''$x \geq y$''.

Subrange types

Languages such as Pascal and Ada allow *subranges* of the integer types. Subranges are subtypes of the `integer` type, and other subrange types. For example the type `t: 3..10` is a subtype of both the type `s: 2..20` and the type `integer`. This allows a variable of type `t` to be used wherever values of type `s`, or type `integer` are required, such as assignments to variables of these types. In contrast an arbitrary integer value cannot be assigned to a variable of type `t` without some guarantee that it will be within the correct range of values.

It is clear that a value with a more restricted range may always be used where values from a greater range are expected. Thus the subtype relationship between subrange types can be formalized as follows:

Let s = s_l .. s_h
and t = t_l .. t_h
then
s ≤ t if, and only if, $s_l \geq t_l$ and $s_h \leq t_h$

Subtype Rule 1

and we can see that this guarantees the substitutability of values of the subtype. A more general, but in this case equivalent, way of defining this would be to say that s is a subtype of t, if and only if, the set of all the values allowed by s is a subset of the set of all values allowed by t.

Structure types

If we now consider the structure type, there are at least two useful ways in which subtype relationships might be defined. The important properties that we expect of structures are that they contain several *fields* each having a type, and identified by a *tag*, which is unique in the structure. The value of a field can be extracted from a structure given the tag, and (in languages which allow assignment) the value of a field in a structure can be changed, again given the tag. The tags are usually part of the type of the structure, and new tags cannot be added, or existing tags removed from structures without changing their type.

This suggests that in order to satisfy the substitution rule, an instance of a subtype structure must contain all the tags of the supertype, and that the values associated with the tags must themselves be usable wherever the corresponding value from the supertype can be used — the substitution rule again. Formally, we have:

Let $s = \{ m_1: s_1, m_2: s_2, \ldots m_i: s_i \}$
and $t = \{ n_1: t_1, n_2: t_2, \ldots n_j: t_j \}$
then
$s \leq t$ if, and only if,
$\forall\ k,\ 0 < k \leq j,$
$\exists\ l$ s.t. $m_l = n_k$
and $s_l \leq t_k$

Subtype Rule 2

This implies that i ≥ j, ie that the subtype structure can have more, but not fewer fields than the supertype. To illustrate this with some examples,

{ a: int, b: character} ≤ { a: int }

{ a: int, b: character} ≤ { b: character }

{ a: int, b: character} ≤ { b: int }
if, and only if,
character ≤ int

but

{ a: int, b: character} ≰ { c: int }

This definition of the subtype relationship between structure types is very powerful, and we shall return to it later. However many languages have chosen a more restrictive definition, which allows a more efficient implementation. The restriction is on the order in which the fields appear in the structure, so that the supertype must be a prefix of the subtype, viz:

Let $s = \{ m_1: s_1, m_2: s_2, \ldots m_i: s_i \}$
and $t = \{ n_1: t_1, n_2: t_2, \ldots n_j: t_j \}$
then
$s \leq t$ if, and only if,
$\forall\ k,\ 0 < k \leq j,$
$m_k = n_k$
and $s_k \leq t_k$

Subtype Rule 3

which means that

{ a: int, b: character} ≤ { a: int }

but now

```
{ a: int, b: character}   ⋦     { b: character }
```

The advantage of this definition is that the position of each field in instances of the supertype is the same as its position in any instance of any subtype. Thus if a structure is used where an instance of one of its supertypes is expected, all the fields that can be named are at the same position in the structure as they would have been if an instance of the supertype had been used. These positions can be calculated at compile-time, so that allowing the possibility of the substitution of a structure by a subtype requires no additional run-time cost.

Some systems, such as (current) Ten15,[82] have an even more restricted subtype relation between structure types, which does not allow the subtype to introduce fields not appearing in the supertype. This is too restrictive for the discussions below, and will not be considered further.

So far the subtype relationship has been described as a relationship between existing types. In contrast most object-oriented systems use the subtype relationship as a mechanism by which types are constructed. New types are defined by *inheriting* properties from supertypes, and then adding more properties. As a result the new type is a subtype of the inherited types *by construction*. The choice of a relationship between existing types, or a property that is true because of the method of construction of types, is the same as the choice between *structure* or *name* equivalence of structure types in languages such as Pascal, and Algol-68. In effect the definitions of the subtype relationship given above are simply weaker statements of the rule for structure equivalence in Algol-68, viz:

```
Let  s = { m1: s1, m2: s2, ... mi: si }
and  t = { n1: t1, n2: t2, ... ni: ti }
then
     s = t if, and only if,
                i = j
          and  ∀ k, 0 < k ≤ i,
                     sk = tk
```

Algol-68 Structure Compatibility Rule

The requirement that the fields' tags are the same is also removed here, so that the type of a field is associated with its position in the structure, and the tag strings are removed entirely from the global meaning of the type.

It is important to note that the subtype relationship provides a partial order on types, while the structure equivalence rule is defining equivalence classes of types. However the rules are related by:

```
A = B if, and only if, A ≤ B and B ≤ A
```

so that structure equivalence can be stated using the subtype relationship.

Structure equivalence is weaker than name equivalence, in the sense that more structures are equivalent using structure equivalence. As a result, types that are structure equivalent but not name equivalent are of some interest. Name equivalence, together with *opaque types* can be used to hide the information in a structure. An opaque type is a type for which only the name is known, so that even if the actual type is known, the contents of instances of the type cannot be accessed. In contrast, if the programmer has an instance of a type, and knows, or guesses its structure, a system based on structure equivalence will allow unlimited access to the contents. In most languages this is not an issue, since structure types normally provide access operations to their fields automatically, and inseparably from the type itself. But it is the mechanism used to hide the representation of data structures defined by modules in Modula-2 using opaque pointers.[7] Also R. T. House has implemented a particularly elegant package mechanism for Algol-60 based on this hiding mechanism.[83]

Programming environments offer, through persistent objects and their types, other opportunities to use this kind of encapsulation mechanism. The type of a structure can be used like a *capability*, allowing a program to have references to a value, without necessarily allowing access to its contents. Systems which use structure equivalence must provide this encapsulation using other mechanisms such as lexical scope, as in chapter 2, or existential quantification over types, which will be described in chapter 4.

Function types

The substitution rule can be applied to function types, where the usual subtype rule is:

Let s = $s_p \rightarrow s_r$
and t = $t_p \rightarrow t_r$
then
s $\leq$ t if, and only if,
$s_r \leq t_r$
and $t_p \leq s_p$

Subtype Rule 4

Note that while the result type of s can be any subtype of the result type of t, the relationship must be reversed for the parameter types. This phenomenon is called the *contra-variance* of the parameter type — in contrast the result type is said to display *co-variance*.

Contra-variance allows s to have a parameter type which is a supertype of the parameter type of t, but since we can use the substitution rule when the function is called we can provide a value with any subtype of s_p, which might also be a subtype of t_p. What s cannot do is to restrict the range of parameter types which are acceptable more than t does.

This defines the subtype relationship between functions of a single parameter. Rather than generalizing this rule it is convenient to allow the parameter to be a *Cartesian product* type, which also provides a natural extension to multiple return values.

Cartesian products

The subtype relation for Cartesian product types is similar to the most restrictive relation for structures seen above:

Let s = $s_1 \times s_2 \times \ldots s_i$
and t = $t_1 \times t_2 \times \ldots t_j$
then
s ≤ t if, and only if,
$i = j$
and $\forall\ k,\ 0 < k \leq i,$
$s_k \leq t_k$

Subtype Rule 5

Cartesian products have several advantages over structures, but are less flexible. They are typically syntactically easier to manipulate, so they can be used conveniently for function parameters and results. Also this subtype rule means that the sizes of Cartesian products are known at compile time, so space can be allocated for them more efficiently. In practice many systems do not explicitly construct a product containing the parameters to a function, instead it is implicitly represented by the contents of registers, and other scattered storage.

Updatable values

Following Ten15,[82] the type of an updatable value, or *Ref*, will be written ''Ref[A⇒B]''. Values of this type are created by the `ref` constructor, and allow two main operations:

Assign: $\forall$ a,b . Ref[a ⇒ b] × a → ()
Contents: $\forall$ a,b . Ref[a ⇒ b] → b

A is the type of the items that can be assigned to the Ref, and B is the type of the items that can be read back. To ensure type-safety we must have that A ≤ B, so that communicating a value through a Ref can force us to use the value in a more general way, but cannot allow the value to be used in places where the original value could not be used.

Like function types, Refs display contra-variance in the first parameter, making the subtype rule:

$$\texttt{Ref}[s_i \Rightarrow s_o] \leq \texttt{Ref}[t_i \Rightarrow t_o] \text{ if, and only if, } s_o \leq t_o \text{ and } t_i \leq s_i$$

Subtype Rule 6

This should not be confused with the type of the value which can be assigned to a Ref, which can obviously take any subtype of the Ref's assignment type.

Most of the references that we see are not involved in interesting subtype relationships, having the same type for values being assigned and taken out. We shall use the notation:

$$\texttt{Ref}[a] \equiv \texttt{Ref}[a \Rightarrow a]$$

which implies the degenerate subtype relationship:

$$\texttt{Ref}[s] \leq \texttt{Ref}[t] \text{ if, and only if, } s = t$$

Subtype Rule 7

There is also a coercion available on Refs:

$$\texttt{Read\_only: } \forall\ a,b\ .\ \texttt{Ref}[a \Rightarrow b] \rightarrow \texttt{Ref}[\bot \Rightarrow b]$$

where $\bot$ (or ''bottom'') is the type containing no values. Of course it is not possible to create a value of this type (since there are none), and so it is not possible to use the `Assign` function on the resulting Ref, i.e. it is a read-only Ref. The read-only version of a Ref is a supertype of the original Ref, since $\bot$ is a subtype of every type, so we can always use a writable Ref where a read-only one is expected.†

It is important that this is not creating a new Ref value, but rather creating another view of the same Ref value, which has a different (but related) type. This means that views of the Ref value may still exist that allow assignment to it, and so successive calls of `Contents` on a read-only Ref are not guaranteed to return the same result.

This leads us to an interesting interaction between the read-only coercion and the subtype relationships between Ref types. This can be illustrated as follows:

Consider types s and t, such that s ≤ t, and define

---

† The coercion making a Ref write-only can also be useful, since it formalizes the communication protocol through a shared Ref. A write-only Ref might also be used to represent an uninitialized value.

```
type S = Ref[s ⇒ s];
type T = Ref[t ⇒ t];
```

then S ≰ T, since Refs are contra-variant in their assignment type. However if we create some values of these types, and read-only views of them

```
value sval: S = ref(...);
value tval: T = ref(...);

value ro_sval: Ref[⊥ ⇒ s] = Read_only(sval);
value ro_tval: Ref[⊥ ⇒ t] = Read_only(tval);
```

we now see that Ref[⊥⇒ s] ≤ Ref[⊥⇒ t] and so ro_sval can be substituted for ro_tval. This allows us to get some of the benefits of a subtype rule on Refs that was co-variant in the assignment types, without the insecurities this would introduce. There are parallels here with the way that problems due to the contra-variance of function parameter types can be avoided by putting the parameters into the non-locals of the function, which will also be used later.

Type-functions

Since type-functions are not simply macro-expanded, the subtype relationships involving them must be defined explicitly. In the following Ψ(s, t) and Ω(s, t) denote type expressions, possibly containing free occurrences of s and t.

```
Let typefn F[t] = Ψ(t)
then
       F[x] ≤ Ω if, and only if,
                    Ψ(x) ≤ Ω
              given  F[x] ≤ Ω
and
       Ω ≤ F[x] if, and only if,
                    Ω ≤ Ψ(x)
              given  Ω ≤ F[x]
```

Subtype Rule 8

Notice that when we are checking if a type is the subtype of another, we are only concerned that the subtype relationship is consistent. The rules given here use this fact to allow recursive type-functions to be checked by allowing the result to be assumed during the verification of the relationship with the body of the type-function.

This leads to the corollary:

```
Let typefn F[s] = Ψ(s)
and typefn G[t] = Ω(t)
then
       F[x] ≤ G[y] if, and only if,
                    Ψ(x) ≤ Ω(y)
              given  F[x] ≤ G[y]

                        Subtype Rule 9
```

However it is important to remember that $x \leq y$ does not necessarily imply that $F[x] \leq F[y]$, since the formal parameter might occur in a contra-variant context, say as the parameter type of a function type. Type-functions for which this property can be proved are said to be *monotonic*.

3.2 Subtype Relationships for Defining Objects

It has been shown in the previous chapter how objects can be defined using static scope to provide encapsulation, and *type-coercion* to allow the substitution of instances of different types. In this chapter we will discuss the ways in which objects can be constructed when the underlying language provides subtype mechanisms such as those described in the previous section, and the implications of different implementation techniques.

As before we shall discuss various implementations of *accumulator*-like objects — that is objects that accumulate the values which they are given. These are convenient since simple examples can be given which display the whole range of properties that we expect from an object-oriented system. In particular we can show the encapsulation of objects, and mechanisms for their construction. When we look at subclasses we see the inheritance of implementations, using both the extension and overriding of inherited behaviour. This leads to the most subtle problems involving the use of inherited behaviour in the definition of overriding methods, and access to overriding behaviour, via the *self reference* from the superclass.

It should be remembered that the class is really a mechanism for ‘‘programming in the large’’. This can make the notational overhead for simple examples appear unreasonable, however the overhead is fixed, and so is less important in realistic classes.

Using scope for encapsulation

We start with a simple accumulator class, which will be used as the superclass for the classes that follow. It may be defined by:

```
 1:type Accumulator =
 2:       {  add:   Integer → (),
 3:          total: () → Integer,
 4:          image: () → String
 5:       };

 6:value make_accumulator:
 7:           () → (Accumulator × Accumulator → ()) =
 8:    fun()
 9:          let acc: Ref[Integer] = ref 0
10:          and self: Ref[Accumulator] = ref
11:              {  add   = fun(value: Integer)
12:                             acc := !acc + value,
13:                 total = fun()
14:                             !acc,
15:                 image = fun()
16:                             itos(!self.total())
17:              }
18:          in
19:             (!self, fun(ns: Accumulator) self := ns)
20:          end;
```

Program 3.1

This implementation of the type `Accumulator` uses lexical scope as the encapsulation mechanism, to hide the concrete implementation and actual run-time state of an accumulator from the rest of the program. The creation routine returns a pair, the first element of which is the created object, the second element being used by inheriting classes so that they can change the binding of self, allowing the methods defined in `make_accumulator` access to overriding methods. An alternative approach to this problem will be presented later.

As an example of a class that inherits this behaviour, consider the definition of `make_scaled_accumulator` in program 3.2, which creates `Accumulator`-like objects, which automatically scale the total by a given factor.

```
1:type Scaled_Acc =
2:       {  add:       Integer → (),
3:          total:     () → Integer,
4:          image:     () → String,
5:          set_scale: Integer → ()
6:       };
```

```
 7:value make_scaled_accumulator:
 8:            () → (Scaled_Acc × Scaled_Acc → ()) =
 9:    fun()
10:          let scale: Ref[Integer] = ref 0
11:          and (super, ss_self):
12:                  (Accumulator × Accumulator → ()) =
13:                                        make_accumulator()
14:          and self:  Ref[Scaled_Acc] = ref
15:              {  add       = super.add,
16:                 total     = fun()
17:                                  !scale × super.total(),
18:                 image     = super.image,
19:                 set_scale = fun(ns: Integer)
20:                                 scale := ns
21:              }
22:          in
23:             ss_self(!self);
24:             (!self, fun(ns: Scaled_Acc)
25:                       self := ns;
26:                       ss_self(ns) )
27:          end;
```

Program 3.2

Here we see that an instance of the superclass is created, on line 13. Some of the inherited methods from this instance are then returned in the object result structure, together with the local overriding method definition for `total`, and the additional behaviour of `set_scale`. Line 23 installs the new view of the object's behaviour in the superclass' self reference, using the function it provided for this purpose, and lines 24-26 provide a new function which another inheriting class can use in turn.

`Scaled_Acc` ≤ `Accumulator` using either Subtype Rule 2, or 3, which means that the calls on lines 23 and 26 are type-correct, as is the call on line 16 of program 3.1, which would now invoke the overriding definition of `total`.

Separating the representation

It is important that this method of construction for objects does not require the object receiving a message to be given as a parameter in the invocation of a method. This is important because the subtype rule on functions (Subtype Rule 4) does not allow this in the 'obvious' way. To explain this we will consider a simpler representation of classes, and show where it is unsatisfactory. The construction follows more closely the way in which objects are implemented in C++ — an object is represented by its storage, together with a structure of its methods. Implementations of `Accumulator` and `Scaled_accumulator` are shown in programs 3.3 and 3.4. ("&" is the structure concatenation operator, which can be used in the construction of structure types, and values.)

```
type Accumulator =
      { funcs: {  add:   Accumulator × Integer → (),
                  total: Accumulator → Integer,
                  image: Accumulator → String
               },
        store: {
                  acc:   Ref[Integer]
               }
      };
value make_accumulator: () → Accumulator =
    fun()
       { funcs =
            {  add   = fun(self: Accumulator,
                           value: Integer)
                       self.store.acc :=
                                !self.store.acc + value,
               total = fun(self: Accumulator)
                         !self.store.acc,
               image = fun(self: Accumulator)
                         itos(self.funcs.total())
            },
         store =
            {  acc = ref 0
            }
       };
```

Program 3.3

```
type Scaled_Acc =
      { funcs: {  add:       Scaled_Acc × Integer → (),
                  total:     Scaled_Acc → Integer,
                  image:     Scaled_Acc → String,
                  set_scale: Scaled_Acc × Integer → ()
               },
        store: {
                  acc:   Ref[Integer],
                  scale: Ref[Integer]
               }
      };
```

```
12:value make_scaled_accumulator: () → Scaled_Acc =
13:    fun()
14:       let super = make_accumulator()
15:         in
16:         { funcs =
17:              {  add   = super.funcs.add,
18:                 total = fun(self: Scaled_Acc)
19:                           !self.store.scale
20:                                ×  super.funcs.total(self),
21:                 image = super.funcs.image,
22:                 set_scale = fun(self: Scaled_Acc, ns: Integer)
23:                               self.store.scale := ns
24:              },
25:           store = super.store & { scale = ref 0 }
26:         };
```

Program 3.4

The problem is that `Scaled_Acc` is no longer a subtype of `Accumulator`. This is caused by the definition of the methods on lines 18 and 22 which need to declare `self` with the correct type so that references to the instance variable `scale` (such as that on line 19) are possible. The contra-variance of function parameter types means that the types of these functions are not subtypes of the corresponding types in `Accumulator`.

The previous implementation of these classes did not suffer from this problem since the instance variables of the object were bound into the non-locals of the methods, rather than being passed as a parameter. This has the added advantage that the instance variables cannot be accessed directly, providing the encapsulation that is expected of an object. However it is much more expensive in terms of the amount of memory used, since new closures must be created for every method in every object. In the faulty approach the same closure could be used for a method in every instance of the class, and with care even the structure of methods could be shared in this way. This is an example of a situation where we can prove that a construction is safe, but are not able to express this in the type system, and so cannot ‘‘convince’’ the system to accept the program.

Using less memory

The subtype relationship has allowed us to save memory elsewhere though. Previously it was necessary for a type-coercion structure to be included in every object for every superclass of that object’s class. This is not needed here because the subtype relationship and the substitution rule allowed the types to be compatible.

It is important to note that while the examples here were constructed using methods from the superclasses, the fact that the resulting classes’ types were subtypes of the superclasses did not depend on this method of construction. This means that any class with the same type as `Scaled_Acc` from program 3.2 would also be a subtype of `Accumulator`.

In the common case where the subclass is implemented in terms of the superclass, we can use subtype relationships to further reduce the memory requirements of objects. In programs 3.1 and 3.2 the closure of each method would usually contain a pointer to each of the instance variables to which the method refers. Some systems, most notably the *Orbit* compiler for T,[79] will attempt to share some of this storage by an optimization called *closure hoisting*, but a subtype relationship allows the programmer to force the maximum amount of sharing to occur, by allowing the instance variables to be brought together into a single structure, to which each method's closure needs only to keep a single reference.

A different way of providing the self reference will also be introduced here. This requires a different view of what is going to constitute a class. Instead of the single object-creation function that we saw in programs 3.1 and 3.2, and in the previous chapter, a class definition will now export three functions, as well as the necessary types. These will be called the ''make'' function, the ''mk'' function, and the ''close'' function. Normal clients of the class will only need access to the make function, which creates new instances of the class, but classes that inherit the implementation of this class do so by using the other two functions. The mk function, creates the representation of the state of an instance of the class, and the close function, creates a structure of methods with references to a given representation hidden in their closures.

To construct the recursive references to self it is convenient to define a variant of Ref, which we will call ''ChoiceRef''. This is very similar to the type constructor, `pointer`, which was used in the previous chapter, and is introduced for the same reasons. It differs from a Ref, in that a newly created ChoiceRef does not need to be given an initial value. Any attempt to get the contents of a ChoiceRef that has not been explicitly assigned a value will cause a run-time error, but this should never occur in well-formed programs. The subtype rules for ChoiceRef are the same as those for Ref, and the coercion to allow the creation of read-only versions can also be used.

To demonstrate the improved sharing, and the use of ChoiceRef, we must first re-implement the class `Accumulator`. (The classes' types are the same as in programs 3.1 and 3.2)

```
1:type Acc_rep = { acc:  Ref[Integer] };
2:type Acc_selfrep =
3:      { self: ChoiceRef[⊥ ⇒ Accumulator] } & Acc_rep;

4:value mk_accumulator: () → Acc_rep =
5:      fun()
6:         { acc = ref 0 };
```

```
 7:value close_accumulator: Acc_selfrep → Accumulator =
 8:       fun(me: Acc_selfrep)
 9:            {  add   = fun(value: Integer)
10:                          me.acc := !me.acc + value,
11:               total = fun()
12:                          !me.acc,
13:               image = fun()
14:                          itos((!me.self).total())
15:            };

16:value make_accumulator: () → Accumulator =
17:       fun()
18:          let cr: ChoiceRef[Accumulator ⇒ Accumulator] =
19:                                      make_choice_ref();
20:          and me = { self = Read_only(cr) } &
21:                                      mk_accumulator();
22:          and res  = close_accumulator(me);
23:          in
24:             cr := res;
25:             res
26:          end;
```

Program 3.5

This organisation removes the necessity for the function result that was previously attached to each object that was created, so that an inheriting class could insert an appropriate value for self. Now an inheriting class' mk and close functions use those from the class it is inheriting, and all classes provide their own make function that creates an appropriate self reference, using a ChoiceRef, and calls the other functions appropriately.

Now the inheriting class `Scaled_Acc` becomes:

```
1:type SAcc_rep = Acc_rep & { scale:  Ref[Integer] };
2:type SAcc_selfrep =
3:       { self: ChoiceRef[⊥ ⇒ Scaled_Acc] } & SAcc_rep;

4:value mk_scaled_accumulator: () → Acc_rep =
5:       fun()
6:          mk_accumulator() & { scale = ref 1 };
```

```
 7:value close_scaled_accumulator:
 8:                        SAcc_selfrep → Scaled_Acc =
 9:       fun(me: SAcc_selfrep)
10:          let super: Accumulator = close_accumulator(me);
11:          in { add   = super.add,
12:               total = fun()
13:                        !me.scale × super.total(),
14:               image = super.image,
15:               set_scale = fun(ns: Integer)
16:                        me.scale := ns
17:             }
18:          end;
19:value make_scaled_accumulator: () → Accumulator =
20:       fun()
21:          let cr: ChoiceRef[Scaled_Acc ⇒ Scaled_Acc] =
22:                                  make_choice_ref();
23:          and me = { self = Read_only(cr) } &
24:                                  mk_scaled_accumulator()
25:          and res  = close_scaled_accumulator(me)
26:          in
27:             cr := res;
28:             res
29:          end;
```

Program 3.6

The most important point to make about this is that ChoiceRef[⊥⇒Scaled_Acc] ≤ ChoiceRef[⊥⇒Accumulator] since Scaled_Acc ≤ Accumulator, so that SAcc_selfrep ≤ Acc_selfrep. This means that the representation of the subclass can be used where the representation of the superclass is expected, which is particularly important on line 10, where the representation of the subclass is bound into the closures of the inherited operations. The subtype rule on ChoiceRefs is also important since it allows access to overriding methods.

Objects constructed in this way can be expected to use somewhat less memory than using the previous technique. Each method's closure now need only contain a reference to the object's representation, rather than a separate reference to each instance variable to which it refers, and possibly a reference to self also. However the closure still needs to keep a reference to each inherited method that it calls directly, such as on line 13 — the equivalent of a *super call* in Smalltalk — since these references are not in the version of self stored in the representation.

This separation of the clients', and the inheritors' interfaces to a class reflects the fact that an inheriting class can often manipulate its inherited behaviour in ways that are not allowed to normal clients. For example in Smalltalk-80 an inheriting class has access to the instance variables that it inherits, while C++ provides an *ad hoc* mechanism, in the form of the `public`, `private`, and `protected` keywords whereby the access allowed to different kinds of clients can be specified. CommonObjects[39] takes the other extreme view, and only allows inheriting classes the same access to their superclasses as would be allowed to any other client.

The mechanisms introduced here make this distinction between kinds of clients obvious, but since it is not possible to hide the objects' representation, do not give the programmer more control of visibility. This can be achieved using *existential quantification* of types, which is the topic of chapter 4.

3.3 Multiple Inheritance

Multiple inheritance is very useful, particularly when the behaviour of objects is more finely sub-divided so that there are more opportunities for behaviour to be shared. However it introduces several problems that are not seen with single inheritance, and different object-oriented languages have chosen different solutions to these problems. An advantage of constructing objects from other language primitives, as is described here, is that it leaves the programmer free to resolve these problems in the way that is deemed most appropriate for the task in hand. It is even possible, with care, for several different solutions to be used in different parts of a single system.

The examples in the previous section are all correct using the more restrictive subtype rule for structures (Rule 3), which requires that the supertype must be a prefix of the subtype. This is always possible when we want single inheritance, since we can always arrange for the superclass' fields to occur first in the subclass, but when a class is to have several superclasses it is no longer possible to put all their fields first. Thus for multiple inheritance the less restrictive subtype rule (rule 2) is needed. Unfortunately this means that the position of a field within a structure can no longer be calculated at compile-time, and some kind of dynamic look-up is needed. This is done in Amber[69] which uses a cache of the offsets calculated for recent structure field accesses to decrease the overhead of dynamic look-up.

The techniques used are an obvious extension of those described in the previous section. We will extend the previous example to create a subclass of the `Accumulator` class as defined in program 3.1, which is displayed as a graphic object. The resulting class might form the basis of a simple dial. Problems with using the class organisation of program 3.5 will be discussed later.

First we outline a class providing the appropriate behaviour for displayed objects, this is given in Program 3.7.

```
1:type Point = Integer × Integer

2:type Displayed =
3:       {  make_image: () → Bitmap,
4:          draw:       () → (),
5:          undraw:     () → (),
6:          move:       Point → ()
7:       }
```

```
 8:value make_displayed:
 9:               Point → (Displayed × Displayed → ()) =
10:    fun(iposn: Point)
11:         let posn: Ref[Point] = ref iposn
12:         and bits: Bitmap = ref NullBitmap
13:         and on_screen: Boolean = False
14:         and self: Ref[Displayed] = ref
15:             {  make_image = fun()
16:                      raise "Should not be called",
17:                draw   = fun()
18:                      if !on_screen then !self.undraw();
19:                      on_screen := true;
20:                      bits := !self.make_image();
21:                      xor_to_bitmap(screen_bitmap,
22:                                    !posn, !bits),
23:                undraw = fun()
24:                      if !on_screen then (
25:                            xor_to_bitmap(screen_bitmap,
26:                                          !posn, !bits);
27:                         on_screen := False
28:                      ),
29:                move   = fun(np : Point)
30:                      let old_on_screen = !on_screen
31:                      in
32:                        !self.undraw();
33:                        posn := np;
34:                        if old_on_screen then
35:                                !self.draw()
36:                      end
37:             }
38:         in
39:            (!self, fun(ns: Displayed) self := ns)
40:         end;
```

Program 3.7

This is an *abstract class* which means that it provides some behaviour, but it does not provide enough for direct instances of it to be useful. Instead the class is used as a receptacle for a set of related behaviours that are expected to be useful to inheriting objects. These are expected to redefine (at least) the `make_image` method so that the object displays an appropriate image when it receives the `draw` message.

A local reference to the bitmap actually put on the screen is kept by the object so that it can always remove the correct image in the `undraw` method. There is a problem with this, which is that the displayed object might be garbage collected before it can remove its image from the screen bitmap — problems of this kind are the subject of chapters 5 and 6.

Note that the variable `screen_bitmap` is non-local to the class. This could be a global variable, but it is possible that the reference bound into the non-locals of `make_displayed` is the only reference in the system. This would mean that all access to `screen_bitmap` would have to be through classes derived from `displayed`. This technique of forcing access to a device or datastructure through a class is very useful since it means that it cannot be misused.

In this case a class with a non-local is similar to class with a *class variable*, or access to a *pool* in Smalltalk-80. The important difference is that no special support was needed for the mechanism, either from the object-oriented veneer that we are discussing, or from the underlying language, rather it is a natural consequence of the presence of first-class functions, which are fundamental to the whole discussion. This makes the system much more flexible, since the patterns of use do not need to be decided in advance by the language designer, and is one of the reasons why we wish to implement object-oriented techniques on top of a system with first-class functions, rather than simply using one of the many existing object-oriented languages. In this case we are led towards the concept of *first-class classes*, and particularly classes that are returned in the results of a function — there are many possibilities in this direction to be explored.

We can now inherit `Displayed` to define a subclass of `Accumulator` from program 3.1 which maintains its value on the screen.

```
1:type Disp_Acc =
2:       {  add:   Integer → (),
3:          total: () → Integer,
4:          image: () → String,
5:          make_image: () → Bitmap,
6:          draw:  () → (),
7:          undraw:     () → (),
8:          move:  Point → ()
9:        }
```

```
10:value make_disp_acc: Point → (Disp_Acc × Disp_Acc → ()) =
11:    fun(posn: Point)
12:        let (supacc, ss_supacc):
13:                Accumulator × Accumulator → () =
14:                            make_accumulator()
15:        and (supdisp, ss_supdisp):
16:                Displayed × Displayed → () =
17:                            make_displayed(posn)
18:        and self: Disp_Acc =
19:            {  add    = fun(value: Integer)
20:                            supacc.add(value);
21:                            !self.draw(),
22:               total  = supacc.total,
23:               image  = supacc.image,
24:               make_image = fun()
25:                            string_to_bitmap(!self.image()),
26:               draw   = supdisp.draw,
27:               undraw = supdisp.undraw,
28:               move   = supdisp.move
29:            }
30:        in
31:            ss_supacc(!self);
32:            ss_supdisp(!self);
33:            (!self, fun(ns: Disp_Acc)
34:                        self := ns;
35:                        ss_supacc(ns);
36:                        ss_supdisp(ns) )
37:        end;
```

Program 3.8

There are few surprises here — using subtype rule 2 it is clear that `Disp_Acc` ≤ `Displayed`, and `Disp_Acc` ≤ `Accumulator`, as we require, and that the techniques generalize in the obvious way to more than two superclasses.

The essential semantic problem introduced by multiple inheritance, but not seen here, is that of conflicts when methods with the same name are inherited from more than one superclass. A solution to this is to tag each field name with a representation of its type, so that two inherited methods can only conflict if they have the same type, in which case the conflict must be resolved by providing an overriding method. This may be practical in a pre-processor which is producing code from an explicit class syntax, but not when the classes are being written by hand. It also requires a form of overload resolution at the point of call which will not be trivial in the presence of subtypes. A more satisfactory solution can be provided by using type-coercion, but that removes the need for the subtype relationship entirely.

Languages with multiple inheritance must also solve this problem for instance variables. The most convenient solution in the system presented here is to uses Snyder's tree resolution. This is easy to achieve using the techniques of program 3.1, but not with the instance variables collected together in a structure, as in

program 3.5, even with the modified subtype rule, since several fields with the same tag would be needed. This is the same problem as with inheriting conflicting methods, and similar solutions based on renaming might be considered.

3.4 Conclusions

Possible ways in which objects might be constructed in a system with appropriate subtype relationships have been discussed. These implementations of objects are more efficient than those in chapter 2 which do not use subtypes, but still cannot approach the efficiency, particularly in terms of memory usage, that can be achieved by systems defining objects as primitives.

However the techniques do not appear to provide an adequate implementation of multiple inheritance, even when the less efficient subtype rule that was specifically intended for multiple inheritance is used. The problems are caused by conflicting method declarations, which are inherited from more than one class, and in general make it difficult to define a class that is a subtype of several other arbitrarily-chosen classes.

These remaining issues will be addressed in the next chapter.

Chapter 4

Existential Types

Mitchell and Plotkin first introduced *existential* quantification over types to model abstract types,[84] and this has since been extended to the types of modules.[10] Cardelli also introduced the use of *subtypes* to represent the use of inheritance, which is discussed in the previous chapter. However it was shown, in the example of program 3.4, that the subtype rule appropriate for functions does not have the properties that one would like to allow the construction of objects. As a result the construction of objects is much less efficient (particularly in terms of space) than can be achieved in systems with direct support for objects, such as C++ or Smalltalk-80. This chapter shows that an efficient implementation of objects can be achieved using a combination of existential quantification over types, and type compatibility based on an appropriate subtype rule.

4.1 Existential Quantification

Existential quantification over types is similar (and indeed related to) the more common universal quantification over types. It allows us to manipulate values without knowledge of the exact types that are involved, instead we can say ‘‘There is a value, of some type’’, and manipulate this value in a limited way.

Unlike other type constructors, simple existential types are not very useful; it is necessary to have a certain amount of structure in an existential type before it becomes useful. The simplest existential type is the type

```
∃ t . t
```

which simply says ‘‘there is a value, which has a type’’. However since no information about the properties of the value can be gained from the type, there are no non-trivial operations which can be applied to the value, and so it can never be used. In fact, this is the type `Top`, or ⊤, which forms the biggest type in the type lattice, when we have a subtype relationship. A simple existential type with which we can compute is:

```
∃ t . t × (t → Integer)
```

which describes a type for a pair, the second element of which is a function which can be applied to the first element of the pair to yield an Integer. Pairs having this type include:

```
        (1, succ),
        (2, lambda x . x-1),
        ([1,2,3], length),
and     (1.5, truncate_to_int)
```

using obvious names for common functions. Once these values are coerced to the

existential type given above they (by definition) all have the same type, and so it is possible for a non-homogeneous list of these values to be constructed, while remaining type-safe, and without requiring run-time interpretation of types.

Like subtypes, existential types introduce the possibility that a value can have several types, for example consider the pair:

```
(3, 1.5) : Integer × Real
```

This also has the existential types:

```
(3, 1.5) : ∃ t . t × Real
(3, 1.5) : ∃ t . Integer × t
(3, 1.5) : ∃ t,s . t × s
```

and of course

```
(3, 1.5) : ∃ t . t
```

Less obviously in a type system that includes a subtype relationship, it also has the type:

```
(3, 1.5) : ∃ t . t × t
```

This is because all types are subtypes of Top, so

```
(3, 1.5) ≤ (⊤, ⊤)
and
(⊤, ⊤): ∃ t . t × t
```

Pack and Open

The multiplicity of types for values means that it is usually necessary to explicitly coerce a value to give it an existential type. The `pack` and `open` operators from Cardelli and Wegner's language *fun*[10] will be used here. The syntax for `pack` is:

```
pack[s = t in Ψ(s)]v
```

Where we again use Ψ(s) to represent a type expression with free occurrences of s.

This returns a value with type ∃ a . Ψ(a), and is type correct if the value `v` has the type Ψ(t). This can be thought of as hiding the occurrences of the type t in the type expression Ψ(t) behind a new existential type. For example:

```
pack[a = Integer in a × (a → Integer)] (2, succ)
```

gives the value discussed before, with the type ∃ a . a × (a → Integer).

`Open` is used when we wish to compute with a value of existential type.

```
open p as x[t] in expression... end;
```

introduces two new tokens whose scope is the in...end block. `X` is a new name for `p`, the value that was being opened, which can be used as a normal value, parts of which have the (unknown) type `t`. Thus if p is the result of the pack statement above, then

```
open p as x[t] in (second x)(first x) end
```

yields the result `3`. Many systems should be able to type-check expressions involving values with existential types without the explicit use of open, in which case it will only be needed if the local type identifier is required. However to avoid possible confusion, open will always be used explicitly in the examples that follow.

4.2 The Subtype Relationship for Existential Types

We have seen that existential types allow two values of different types to be coerced to the same type, in a way that still allows them to be involved in computations. This is similar, but not the same as, the way that a subtype rule allows values of one type to be substituted for values of a supertype. The subtype relationship allows a value with *sufficient* properties to be used instead, while existential quantification specifies what is not known about a value, but we can use information that can be deduced from the context around the unknown types. The question arises as to how these kinds of substitution properties might interact, and so we wish to find subtype relationships involving existentially quantified types.

To motivate the subtype rule, it is useful to first examine the substitution property of existential types: the type compatibility rule for such types could be stated (rather informally) as:

$$\exists\ a\ .\ \Psi(a) \quad = \quad \exists\ b\ .\ \Omega(b)$$

if, and only if

$$\Psi = \Omega$$

But to avoid problems with the meaning of the equality of type expressions we shall actually use:

$$\exists\ a\ .\ \Psi(a) \quad = \quad \exists\ b\ .\ \Omega(b)$$

if, and only if

$$\text{for all } t \in \text{Types}\ .\ \Psi(t) = \Omega(t)$$

where ‘Types’ is the set of all possible types in the system.†

† Note that ‘for all’ is used for the logical statements, rather than ‘∀’, to avoid confusion with the notation for universal quantification.

The proposal is that a subtype relationship between existential types is defined analogously, viz:

$$\exists\ a\ .\ \Psi(a) \quad \leq \quad \exists\ b\ .\ \Omega(b)$$

if, and only if

$$\text{for all } t \in \text{Types}\ .\ \Psi(t) \leq \Omega(t)$$

A subtype rule such as this must not introduce the possibility that a value can be misused. Two justifications of it will be given, first an informal one, and then one based on the model of types as sets of values.

It is interesting that this subtype rule does not directly involve the existential type, but only the type expression in which it appears. This means that it should not conflict with other subtype relationships involving existential types that might be provided by a type system. In particular this means that it could be used with Cardelli and Wegner's *bounded existential* types.

Informal justification

Informally it is convenient to think of the type expressions Ψ and Ω as defining the *shapes* of the values that the existential types describe, while parts within this shape are hidden behind the existential type tokens a and b.

If we have some operation that is expecting a value with the type $\exists$ b . Ω(b) then it can only rely on the shape Ω in its operation, b being unknown. Thus the substitution of a value with a shape Ψ should be possible if we could guarantee that Ψ(b) was a subtype of Ω(b). Our subtype rule states that $\Psi(t) \leq \Omega(t)$ for all possible types that t could take — which must include the type that is hidden behind the existential b! However we wish to make the substitution not with Ψ(b), but with Ψ(a) — a type with the same shape, but a different existential. But no use can actually be made directly of values of type b, so changing from one unknown type to another can not affect the execution of the program. The new unknown values can only be manipulated in ways consistent with the shape of the whole value, and we have already argued that this is unaffected by the substitution of a value of the subtype's shape.

Unfortunately the reverse implication might be stronger than is necessary. If there is a type t, such that $\Psi(t) \not\leq \Omega(t)$, given a value of the type $\exists$ a . Ψ(a) we do not know if the type hidden by the existential is t or not, and so cannot be sure if the existential types are subtypes. A static type system must assume the worst in this situation, in order to be safe in the presence of the substitution rule, and so is forced to conclude that the existential types are not subtypes. Thus the ''and only if'' is the result of pessimism forced on the type system rather than an inference.

Justification if types are sets

If we regard types as being sets of values, a more satisfactory justification of this subtype rule can be made. In fact, not all sets represent reasonable types, and in practice types are represented by special kinds of sets, called *ideals*. However since here we are comparing types, rather than constructing new types, this does not affect the following discussion.

Here T(t) will mean 'the set of values that are of the type t'. An existential type is represented by the union of the type sets of all the types it could possibly hide, i.e.

$$T(\exists\ t\ .\ \Psi(t)) \quad = \quad \bigcup_t\ T\ \Psi(t)$$

This is because a value of existential type could actually be a value of any of the types formed by Ψ(t). In contrast the type set of a universal quantification is the intersection of the type sets of the types formed by the type expression, since a universally quantified value must have all these types simultaneously.

The subtype relationship can be defined very simply using the set representation of types,

$$s \le t \quad \text{if, and only if,} \quad T(s) \subseteq T(t)$$

which says that all values of the subtype are also values of the supertype.

Now

$$T(\exists\ a\ .\ \Psi(a)) \quad = \quad \bigcup_s\ T\ \Psi(s)$$

and

$$T(\exists\ b\ .\ \Omega(b)) \quad = \quad \bigcup_t\ T\ \Omega(t)$$

Thus

$$\exists\ a\ .\ \Psi(a) \quad \le \quad \exists\ b\ .\ \Omega(b)$$

becomes

$$\bigcup_s\ T\ \Psi(s) \quad \subseteq \quad \bigcup_t\ T\ \Omega(t) \qquad (1)$$

and the condition

$$\text{for all t } \varepsilon \text{ Types . } \Psi(t) \le \Omega(t)$$

becomes

$$\text{for all t } \varepsilon \text{ Types . } T\ \Psi(t) \subseteq T\ \Omega(t) \qquad (2)$$

The argument is straight-forward: Each element of $\bigcup_s\ T\ \Psi(s)$ from (1) must be an element of at least one set $T\ \Psi(t)$, for some t. Thus, by (2), it is an element of at least the corresponding set $T\ \Omega(t)$, and so also a member of $\bigcup_t\ T\ \Omega(t)$.

Checking the subtype rule

Even if a subtype rule does not introduce any inconsistencies into the type system it is of little use if it cannot be implemented reasonably easily. The subtype rule suggested here is more complicated than the established ones presented earlier,

but it is reasonable to expect type-systems to achieve the necessary inferences. The problem can be split into two stages. In the first stage the type expressions that are to be compared must be identified. This may involve moving existential quantifiers through unrelated type declarations to get both types into a canonical form, which would be where all quantifiers have been moved to the innermost possible scope. For example, if we are checking the subtype relationship:

```
      ∃ t . Integer × { a: t, b: t → Integer }
  ≤   Integer × ∃ s . { a: s, b: s → Integer, c: s → Real }
```

it is easiest if this is first transformed into the equivalent:

```
      Integer × ∃ t . { a: t, b: t → Integer }
  ≤   Integer × ∃ s . { a: s, b: s → Integer, c: s → Real }
```

which is then simplified to:

```
      ∃ t . { a: t, b: t → Integer }
  ≤   ∃ s . { a: s, b: s → Integer, c: s → Real }
```

In the second stage we check the condition that the type expressions are in the subtype relationship for all types. This can be achieved by substituting a unique, unevaluated type identifier for the existential type identifiers in both type expressions, and then comparing them. If they can be proved to be subtypes without the need to introduce constraints on the unevaluated type identifier (apart from the obvious "t ≤ t" and "t ≥ t") then the condition is satisfied for all types, and the subtype relationship is proved. So in the above example, substituting x for t and s, we have to verify:

```
      { a: x, b: x → Integer }
  ≤   { a: x, b: x → Integer, c: x → Real }
```

which is obviously true for all types x.

Recursive types

A remaining subtlety is that the condition might depend on the original rule that we wish to prove, which is acceptable, since we are only attempting to prove the consistency of the subtype relationship. This is similar to what was seen before in the definition of subtype rule 8 for recursive type-functions, and will only be needed when type-functions are used with existential quantification, since this is the only way that recursive types can be introduced. This is not a problem, but care must be taken to avoid unbounded recursion when this occurs. For example consider the subtype test:

```
typefn A[x] = T × x × (∃ y . A[y])
type A = ∃ z . A[z]
```

and

```
typefn B[x] = S × x × (∃ y . B[y])
type B = ∃ z . B[z]
```

prove A ≤ B.

It is clear that A ≤ B if, and only if, T ≤ S and A ≤ B, and so the subtype relationship is self-consistent. This means that the final, and most general, version of the subtype rule is:

```
∃ a . Ψ(a)    ≤    ∃ b . Ω(b)

if, and only if
       for all t ε Types . Ψ(t) ≤ Ω(t)
                     given ∃ m . Ψ(m) ≤ ∃ n . Ω(n)
```

Other subtype rules

Other subtype rules are suggested by this, and it might be possible for a system to also allow these. In particular the subtype can impose more structure on the existential value, viz:

```
∃ a . Ψ(Φ(a))    ≤    ∃ b . Ω(b)

if, and only if
       for all t ε Types . Ψ(t) ≤ Ω(t)
```

Verifying this subtype relationship would be substantially more difficult, since the structure of Φ must be discovered. It is possible that future systems might allow the programmer to supply directions to the type checker, so that it does not have to discover a proof, but can simply verify that the programmer's proof is correct. This could allow significantly more general type systems, and might be required in this case. However since these extended subtype rules are not required by the techniques for the construction of objects that follow, this will not be pursued further.

4.3 Defining Objects

The previous chapter showed that the contra-variance in the subtype relationship for function types makes the construction of objects difficult. This becomes apparent in the 'obvious' construction technique for objects, which is to give an object a type of the form:

```
o × (m₁: o → r₁, m₂:...)
```

Here o is the type of the representation of the object, and is paired with a structure that contains the functions implementing the methods. Two classes of this form might be:

```
type A = o × (m1: o → r1)
type B = p × (m1: p → s1, m2: (p×b2) → s2)
```

Now if p ≤ o and $r_1 \leq s_1$, we would like B ≤ A, since B only provides more behaviour than A. However closer examination reveils that B ≰ A because

```
p → s1      ≤      o → r1
```

requires that p ≥ o.

However we could use existential types to hide the representation of objects. This is highly desirable since the objects as defined above allow unlimited access to their representation, and so do not provide the *encapsulation* that is normally expected of objects. We now have:

```
type A = ∃ o . o × (m1: o → r1)
type B = ∃ p . p × (m1: p → s1, m2: (p×b2) → s2)
```

and we see that B ≤ A if:

```
for all types t,
     t × (m1: t → r1)
≤    t × (m1: t → s1, m2: (t×b2) → s2)
```

i.e. if (by Subtype Rule 5 and t ≤ t)

```
for all types t,
       (m1: t → r1)
≤      (m1: t → s1, m2: (t×b2) → s2)
```

i.e. if (by Subtype Rule 3)

```
for all types t,
       t → r1    ≤    t → s1
```

i.e. if (by Subtype Rule 4)

```
for all types t,
       t    ≤    t
and    r1   ≤    s1
```

Now the contra-variance of the function parameters does not affect the subtype computation at all. The subtype can introduce new methods, and so long as the common methods' return types are subtypes, the objects' types will also be subtypes.

If we have an object, a say, whose type is (possibly a subtype of) A, then we can invoke the m_1 message after first opening the object, viz:

```
open a as x[z]
in
       let (rep, mt) = x
       in
              mt.m1(rep)
       end
end
```

(The let-clause uses pattern matching to take the value x apart conveniently.)

This formulation of objects makes them representation independent. Previously the representation used by the subtype had to be a subtype of the representation of the supertype, but now arbitrary representations can be used, since once they are hidden behind an existential type they are not distinguishable. This is similar to what can be achieved by storing the object's state in non-local variables accessed through the closures of the functions that are the object's methods, as is described in the previous chapters. However this approach requires much less space, since the member functions' closures can be shared by all instances of a class, as the closures do not contain any object-specific state.

This representation independence gives the implementor of a class much more freedom since it is only necessary for the subclass to be a subtype of the appropriate superclass, and it is not necessary for it to inherit the implementation or have the same instance variables. In particular the subclass might require much less state than the superclass, since it might not require such general-purpose data structures. An example of this would be a class for rectangles with horizontal and vertical sides, which would only require two points as instance variables, but which it might be convenient to make a subtype of general quadrilateral and polygon classes.

Of course in practice many classes will be defined by extending an existing class in the traditional way, adding new methods, and overriding some of the existing methods. But while systems like Smalltalk-80[33] and C++[27] require classes to be constructed in this way, this scheme allows it as one of many possibilities. This is the separation of type and implementation inheritance that is discussed by Snyder.[16]

However one problem remains with this scheme for constructing objects. This is that the methods receive the object's representation as a parameter, but not the structure containing the methods. This means that the method cannot send messages to self (the current object), going through the normal dynamic binding mechanism that would allow a method defined in an inherited class to invoke a method defined by the subclass. The normal approach to this is to make the entire object a parameter to the methods, as shown in the recursive type declaration

```
type A = ∃ o . o × (m1: (o × Integer × A) → r1,
                    m2: (o × Integer) → r2)
```

where we see method m_1 taking the whole object as its third parameter. This gives it access to the function that is present in the object in the m_2 field, which might be the function that was defined at the same time as m_1, but could alternatively be an

overriding function from a subclass. If we now look at a subtype of this

$$\texttt{type B} = \exists\ \texttt{o}\ .\ \texttt{o} \times (m_1\texttt{: (o} \times \texttt{Integer} \times \texttt{A)} \rightarrow s_1\texttt{,}\ m_2\texttt{: (o} \times \texttt{Integer)} \rightarrow s_2\texttt{,...)}$$

we see that the parameter of the method must still have type A, rather than the new type B. This is unfortunate, since it means that new classes cannot make use of more behaviour from their subclasses than they were declared with in the original class.

Any attempt to substitute B instead of A here, reintroduces the previous problems caused by the contra-variance of function parameter types. This means that the dependence of a method on other methods that may be redefined would have to be declared when the method is first given a type. In practice this is too restrictive and an alternative is needed. In the previous chapter we saw how updatable values could be used to construct the recursive data structures that include a self field. The subtype relationship on existential types allows this to be used here also, and we shall see this used in the next section.

In these types the method m_1 was also given the concrete representation of the object's state as a parameter. This might appear to be unnecessary, since the whole object is the third parameter, but the method cannot access the state using this, since the recursion in the type occurs outside the introduction of the existential type hiding the state. That is, the existentials in the recursive uses of the type are not the same as those used directly.

This can be seen more clearly if the recursion of the type is introduced explicitly — this will be done using the ''paradoxical'' Y-combinator, to construct the least fixed point of a type function. Using this notation, the previous definition of A becomes

$$\texttt{type A} = \texttt{Y[}\ \lambda\ \texttt{x}\ .\ \exists\ \texttt{o}\ .\ \texttt{o} \times (m_1\texttt{: (o} \times \texttt{Integer} \times \texttt{x)} \rightarrow r_1\texttt{,}\ m_2\texttt{: (o} \times \texttt{Integer)} \rightarrow r_2\texttt{)]}$$

and it is obvious that each application of the type function introduces a new existential type. However the type

$$\texttt{type A} = \exists\ \texttt{o}\ .\ \texttt{Y[}\ \lambda\ \texttt{x}\ .\ \texttt{o} \times (m_1\texttt{: (o} \times \texttt{Integer} \times \texttt{x)} \rightarrow r_1\texttt{,}\ m_2\texttt{: (o} \times \texttt{Integer)} \rightarrow r_2\texttt{)]}$$

uses the same existential type throughout. Now the first parameter of m_1 is redundant, and can be removed, since its third parameter's concrete representation is known.

Previously any object with the correct type could have been given as the third parameter of m_1, but now the value must have the same existential type as the object in which the method was found, which can only be satisfied by using the receiving object.

4.4 Example Classes

We shall use the same example classes as in previous chapters so that the different techniques may be compared more easily. As stated in the previous section this technique does not require that a class is constructed by inheriting from classes that are its supertypes, however this is what will be shown since this is a common situation that we wish to adequately support — especially if we are attempting to emulate the techniques of other object-oriented systems.

First we present the simple `Accumulator` class.

```
 1:typefn A(x) = {
 2:        add:   x × Integer → (),
 3:        total: x → Integer,
 4:        image: x → String
 5:  };

 6:type Accumulator = ∃ t . t × A(t);

 7:type Arep = { value: Ref[Integer]};
 8:type Aselfrep = { self: ChoiceRef[⊥ ⇒ Accumulator]}
 9:                                    & Arep;

10:value mt_accumulator: A[Aselfrep] = {
11:       add   = fun(rep: Aselfrep, value: Integer)
12:             rep.value := !rep.value + value,
13:       total = fun(rep: Aselfrep)
14:             !rep.value,
15:       image = fun(rep: Aselfrep)
16:             open !rep.self as me[z]
17:             in
18:                let (srep, sops) = me
19:                in
20:                   itos(sops.total(srep))
21:                end
22:             end
23:       };

24:value mk_accumulator: Integer → Arep =
25:       fun(ivalue: Integer)
26:          { value = ref ivalue};
```

```
27:value make_accumulator: Integer → Accumulator =
28:       fun(ivalue: Integer)
29:          let cr: ChoiceRef[Accumulator ⇒ Accumulator] =
30:                                        make_choice_ref();
31:          and self = { self = Read_only(cr) } &
32:                                        mk_accumulator(ivalue);
33:          and res  = pack[r = Aselfrep in r × A(r)]
34:                 (self, mt_accumulator);
35:          in
36:             cr := res;
37:             res
38:          end;
```

Program 4.1

Clients of this class come in two varieties, those that just create and manipulate instances of it, and secondly those that are going to inherit its behaviour. The first type of clients only need access to the object creation routine `make_accumulator`, and knowledge of the type `Accumulator`.

An instance of any subtype of `Accumulator` can be put into the self ChoiceRef, which allows a view of the same object that overrides some messages to be inserted. When instances are created the correct value for self is inserted on line 36, so that obtaining the contents of a self field, as occurs on line 16, should never fail in any well-formed program.

Classes that inherit the behaviour will use `mk_accumulator` and the *message table* `mt_accumulator`. We shall see this in the definition of `Scaled_accumulator` which behaves like `Accumulator`, but scales the total by a given amount:

```
 1:typefn SA(x) = {
 2:        add:   x × Integer → (),
 3:        total: x → Integer,
 4:        image: x → String,
 5:        set_scale: x → ()
 6:   };

 7:type Scaled_Acc = ∃ t . t × SA(t);

 8:type SArep = Arep & { scale: Ref[Integer] };
 9:type SAselfrep = { self: ChoiceRef[⊥ ⇒ Scaled_Acc]}
10:                                     & SArep;

11:value mt_scaled_accumulator: SA[SAselfrep] = {
12:       add   = mt_accumulator.add,
13:       total = fun(rep: SAselfrep)
14:            !rep.scale × mt_accumulator.total(rep),
15:       image = mt_accumulator.image,
16:       set_scale = fun(rep: SAselfrep, ns: Integer)
17:            rep.scale := ns
18:       };
```

```
19:value mk_scaled_accumulator: () → SArep =
20:       fun()
21:          mk_accumulator(0) & { scale = ref 1};

22:value make_scaled_accumulator: () → Scaled_Acc =
23:       fun()
24:          let cr: ChoiceRef[Scaled_Acc ⇒ Scaled_Acc]
25:                   = make_choice_ref();
26:          and self = { self = Read_only(cr) } &
27:                                    mk_scaled_accumulator();
28:          and res  = pack[r = SAselfrep in r × SA(r)]
29:                                 (self, mt_scaled_accumulator);
30:          in
31:             cr := res;
32:             res
33:          end;
```

Program 4.2

This requires some explanation, and in the absence of language implementations, careful justification. First note that

`Scaled_Acc` ≤ `Accumulator`

since `A` is a prefix of `SA`. This implies that

`SAselfrep` ≤ `Aselfrep`

as ChoiceRef is co-variant in its result type, and `Arep` is a prefix of `SArep`. However type functions are, both in general and in this example, non-monotonic, so that:

`SA[SAselfrep]` ⋦ `A[Aselfrep]`

This means that the subtype relationship between the results returned by `make_accumulator` and `make_scaled_accumulator` is not established until the representation has been hidden behind the existential quantifications, by the coercions in lines 33 and 28, respectively.

In contrast the inherited methods in the message table for the `add` and `image` messages have types that are subtypes of the types required for these methods. For example the type of `mt_accumulator.add`:

```
        Aselfrep × Integer → ()
≤       SAselfrep × Integer → ()
```

since

`Aselfrep` ≥ `SAselfrep`

So that their uses on lines 12 and 15 are type-correct.

One potential problem with this is that the implementation of a class is visible in classes which inherit it. This might be considered undesirable,[16] but is similar to the encapsulation provided by Smalltalk-80 classes. However it is possible, with some reorganisation of these definitions, for the inherited state to be hidden behind an existential type, so that the implementation of a class is only visible at the point of its definition.

4.5 The Cost of Objects

So what have we gained by using existentially quantified types in our definitions of objects? The principle advantage, and indeed the motivation of this approach is that the cost of providing objects is now similar to the costs incurred in systems that provide objects as primitive constructs. The principle saving is that most of the overhead of an object can now be shared amongst all the instances of the class. The approaches of both previous chapters required the instance variables of an object to be stored in the non-locals of the functions that implemented methods. This was used both to provide the encapsulation of objects, and also to remove the requirement for a subtype relationship between records of instance variables. However the result was that the storage requirement of an object was roughly proportional to $i \times m$, where i is the number of instance variables, and m the number of methods.† In chapter 2 it was also necessary to construct type-coercion operators for all the superclasses of an object, this could use considerable extra space, again for every instance.

In contrast the storage requirements of programs 4.1 and 4.2 are $i + 2$ words per-object. One overhead word being needed for the ChoiceRef containing the self reference, and one for the reference to the message table. The message table itself, and all the functions making up the methods of the object can be shared by all instances of the class, and do not contain any extravagant data structures. This compares favourably with the cost of objects in C++, and Smalltalk-80, both of which require $i + 1$ words per object.

Sending a message to an object defined using this technique is actually more expensive than using the previous approaches, since at least one extra dereference is needed to get to the method, and the representation must be given as a parameter. However neither of these extra operations are very expensive. On the other hand, the creation of objects is much cheaper, since the object creation process is simpler, and much less memory must be allocated and initialized. This also means that the system will create less garbage, and so place less strain on the memory system.

4.6 Multiple Inheritance

Existential types can also be used to decrease the cost of objects constructed using type-coercion as described in chapter 2. This is useful if the type system does not allow the subtype relationship discussed in the previous sections. It can also be used to allow an efficient implementation of multiple-inheritance.

† The actual cost is much more complicated, since many methods only refer to a small number of instance variables, and there are overridden methods which still require storage.

Type-coercion involves actively converting an object to an instance of its supertype, but the run-time cost of these conversions is usually much less than that required for the run-time mapping of structure field names, to data positions that is implied by changing the subtype rule. Consider three classes:

```
type A = ∃ o . o × (m1: o → r1)
type B = ∃ p . p × (m2: p → r2, m3: (p×...) → r3)
type C = ∃ q . q × (m4: q → r4, to_A: q → A, to_B: q → B)
```

where C is intended to be a subtype of both A and B. Using type-coercion this means that operators coercing an instance of C into an instance of A and B must be provided — here they are called `to_A` and `to_B`. It would be reasonable to expect that an instance of C would contain within its representation instances of A and B that would be returned by the type-coercion operators, but the results will often need to include more state for use by overriding methods, and this might mean that other methods require *wrappers* to access their instance variables. The self references in these interior objects can be arranged to contain coerced versions of the enclosing object, using the same techniques as before, so that overriding methods can be accessed.

Again the cost of this is small. Each interior object must have its own self reference, which means that there will be a self reference for every superclass, rather than one which is shared by them all. However multiple-inheritance in C++[70] also requires this, so the overhead is not unreasonable. As before all other data structures are part of the class, and so are shared by all the instances of the class.

In fact this implementation of multiple-inheritance is very similar to that in C++. In C++ when a message is found by indexing the message table, an offset is also obtained which is added to the pointer to the receiving object before it is given as the parameter to the method. This means that the method's pointer to self is actually referencing the instance variables inherited from the parent within the object, rather than the object itself. The execution of the type-coercion operators is achieving the same effect as the addition of this offset, and it would be possible to package this up in the message table in the same way. Such type-coercions could be inserted automatically by the system, or left explicitly to the programmer, if desired.

An alternative implementation of multiple inheritance would be to use subtype rule 3 to allow the re-ordering of the fields in subtypes, and using existential types for objects' types. However this is not just much less efficient, but does not allow the hiding of inherited instance variables behind an existential type, because the field names must be visible for the subtype relationship to be verifiable.

4.7 Conclusions

A new subtype rule has been proposed that relates the types of type expressions using existential quantification, and it has been shown how this can be used to construct classes and objects. Objects constructed in this way use much less storage than was required for previous techniques, to the extent that it would appear

to be feasible to use objects defined in this way in practical systems, rather than their use being limited to purely theoretical interest.

Many of the types constructed are recursive, with the recursion being hidden behind an existential type, rather than being explicitly introduced. As such, the existential types often take the rôle of "`like Current`" in Eiffel, but since they do not have their own *ad hoc* type rules, cannot introduce similar type insecurities.[85]

Using a technique such as this, rather than providing objects as primitives in the system, is less committed to any one particular view of objects. This is particularly important for systems such as Ten15,[64] which are intended to be a target system for many languages, some of which might have conflicting definitions of objects. It also means that the programmer is not limited to the definition of objects provided by any particular language, but can choose techniques that are appropriate for the task at hand.

More generally these three chapters have demonstrated how more expressive type systems can allow more efficient implementations of objects to be used. This is not a surprising result, but the fact that strongly-typed languages can make it impossible to use well-founded algorithms has often been ignored. Research on type systems will hopefully continue to reduce the number of algorithms falling into this category.

Chapter 5

Active Deallocation of Objects

As an object oriented system executes, objects are created, and other objects become redundant. Many languages require the programmer to deallocate each object explicitly when it is no longer needed, but it is increasingly common for the deallocation to be carried out automatically by the system, using garbage collection.[8] This makes the programmer's task easier and less error-prone, but is particularly important if it should not be possible for the integrity of the system to be affected by mistakes (or indeed, malicious actions) made by the programmer.

Objects are used to represent all kinds of data. Many correspond to structures in other languages, but some present a resource, such as an open file or terminal, to the rest of the system. Use of this resource is then achieved by sending messages to the *guardian-object*, which can carry out arbitrary checks implementing access rights, or to guarantee the integrity of the resource.

In this case, it may be important that a specific action occurs when the resource is no longer needed. A typical action would be to deallocate the resource, for example close a file, but could also involve more complex actions, such as flushing buffered output or rewinding a magnetic tape. Since this deallocation action is part of the object's behaviour as an abstract data type, it should be implemented by the object itself. Moreover, encapsulation usually prevents access from outside the object to the data that would be needed for the deallocation. This suggests that a method should be called to carry out this action. This chapter introduces special methods, that will be called *destroy methods* to achieve this, and discusses how they might be implemented.

Destructors in C++ are similar to destroy methods, as is the part of a block following the "`****`" statement in PascalPlus.[86] However, in these languages the programmer must explicitly deallocate heap objects, so it is simple for the compiler to arrange for the deallocation code to be executed immediately before deallocation. Similarly, the compiler can arrange for their execution prior to exiting a block, to deallocate stack allocated objects. The *inner* statement in Simula-67[23] is more general, but its implementation, (beyond the fact that reference counting was the principle garbage collection technique) is not widely documented.

It is important that destroy messages be sent automatically, otherwise, the programmer has to decide when the destroy message should be sent. A mistake could result in a resource becoming both unreachable and unavailable for reuse, this is particularly undesirable when the resource is scarce, such as a file descriptor or a physical device. The other possibility is that the destroy message is sent while the resource is still in use, which may cause future attempts to use the resource to fail. Both of these scenarios are more important when we consider an object-oriented system, which might run for an extended period, rather than simply an object-oriented program.

It can be argued that errors such as these are mistakes by the programmer and can be fixed. However in large systems, particularly those in which multiple processes share such resource objects, it can be impossible to determine statically when an object becomes inaccessible. Determining accessibility dynamically implies, in the worst case, that the programmer must write a garbage collector for the resource, duplicating the garbage collector provided by the system, most likely without access to the low-level primitives that it can use to manipulate references. The presence of a garbage collector implies that it has already been decided that the programmer should not be required to carry out this task, and so it is inconsistent to require it for these special objects. Other language features, such as exceptions, can further complicate matters.

5.1 Uses of Destroy Methods

A common example use of destroy methods is to manage resources external to the object-oriented system. This is the situation referred to above, in which the object represents something that cannot be garbage collected by the system. Examples are file descriptors, devices, terminal windows, and operating system resources.

Most such destroy methods explicitly return the resource, making it available for future reuse, or cause the deallocation of data structures, that is, they do what a garbage collector would do if it understood the resource. However, more complicated actions may also be necessary, such as rewinding a magnetic tape, flushing a buffer to a file, or restoring parts of a screen obscured by a deallocated window object.

Another use of destroy methods is to allocate objects from a pool. When an object becomes garbage, its destroy method can return it to a pool of free objects, this creates a new reference to the object and it should not actually be deallocated. This technique is particularly useful if the object's current state is important, or if recreating the object is difficult or expensive. Again, the more obvious examples concern devices.

Consider, for example, a system with a collection of line printers. An object is associated with each printer; these are created when the system is initialized and might contain information about the corresponding physical printer. Another object maintains a pool of free printers, each represented by its associated object. When a request for a printer is received, it is allocated by deleting the associated object from the pool and returning it to the requester. When the associated object becomes unreachable, the allocator is informed by the execution of a destroy method for the object. Instead of allowing deallocation to proceed, it returns the associated object to the pool. Many operating systems implement similar allocators via special purpose code that provides the function of a destroy method for a specific kind of object, such as a file descriptor. Incorporating destroy methods in a language provides a similar function in a more general setting.

An important use of destroy methods is in building object-oriented interfaces to systems implemented in procedural languages. When a system is composed of parts written in an object-oriented language and (possibly pre-existing) parts written

in procedural languages — particularly those without garbage collection — it will often be necessary for garbage collection of objects to cause other-language data structures to be explicitly deallocated or returned to a pool. These are actually the same problems described previously, but occur in different situations not necessarily involving physical resources.

This embedding of code written in traditional languages in objects, so that it can be used as if it had been written in the object-oriented language, has been called *gift wrapping*, and is a good way of making existing libraries available to users of object-oriented languages.

For example, the object-oriented language AML/X[87] was used in the *Tiered Geometric Modelling System* (or TGMS),[88] to provide an object-oriented interface to GDP — a solid modelling system written in PL/1.[89] TGMS provides a class called `solid`. An instance of `solid` corresponds to a polyhedron data structure in GDP. TGMS programs often create instances of solids, and hence of GDP polyhedron data structures, as intermediate results. Destroy methods are used to explicitly deallocate the GDP data structure and remove the polyhedron from the graphics screen when an instance of `solid` is no longer accessible. This use of destroy methods relieves the TGMS programmer from having to manage GDP storage and the display of intermediate results.

Of course not all objects in a system will require a destroy method, in the following discussion an ‘‘object with a destroy method’’ will be called an *OWDM*. To be most useful, destroy methods should have the following properties:

Property 1

Each time the last reference to an OWDM (other than from the stack frame of the object’s destroy method) is removed, its destroy method should be called exactly once before continuing the computation that deleted the last reference.

Note that this prevents the destroy method from being called recursively on the same object because the destroy method has an implicit reference to the object (i.e, `self`). However, it does allow the destroy method to be called more than once during an object’s lifetime, as occurs when allocating objects from a pool.

Property 2

A destroy method should be able to do any operation that is possible in an ordinary method. Specifically, the destroy method should be able to create new objects and to send messages to objects to which it has references, including `self`. By Property 1, such objects will not have had their own destroy methods invoked because references to them still exist.

Property 3

The presence of destroy methods should not increase the running time of other methods, for example, other methods should not have to determine if they are creating new references to an object, since this places an additional overhead on the system regardless of whether a destroy method is executing.

Property 4

The implementation should be *safe*, that is, it should not be possible to obtain a reference to an object that has been deallocated or to an object whose destroy method has executed, unless the destroy method created a new reference.

It is particularly difficult to satisfy this if destroy methods are invoked on objects in disconnected, self-referential groups, as might result from cycles. This is difficult because the execution of the destroy methods for two (or more) objects may be mutually dependent on the state of the other object(s).

5.2 Implementing Destroy Methods

The implementation of destroy methods depends on the garbage collection algorithm employed by the underlying language implementation. A particularly simple implementation is possible when reference counting is used. An implementation will also be presented using mark-scan garbage collection, this solution is less satisfactory for a variety of reasons. Implementations using other garbage collection algorithms, such as multiple-space, real-time algorithms[90, 91] or generation scavenging[92] should be possible, but have not been investigated in detail, particularly given the alternative techniques that are presented in the next chapter. Parallel garbage collectors[93] present additional problems.

5.2.1 Reference Counting Garbage Collection

Traditional reference counting associates a *reference count* with each object, which is incremented each time a new reference to the object is created. Whenever a reference is destroyed, the count is decremented. When it becomes zero, no other references to the object can exist, so the object is deallocated. This is shown in the following function, using the C programming Language:

```
 1:decrease_refs_to(object)
 2:object_ptr object;
 3:{
 4:       object->reference_count--;
 5:       if(object->reference_count == 0) {
 6:            for(j in objects referenced by object)
 7:                 decrease_refs_to(j);
 8:            deallocate(object);
 9:       }
10:}
```

Program 5.1

This function can be modified to call the object's destroy method, as follows:

```
 1:decrease_refs_to(object)
 2:object_ptr object;
 3:{
 4:       object->reference_count--;
 5:       if(object->reference_count == 0) {
 6:            if(has_destroy_method(object)) {
 7:                 object->reference_count++;
 8:                 call_destroy_method(object);
 9:                 object->reference_count--;
10:
11:                 if(object->reference_count == 0) {
12:                      for(j  in objects referenced by object)
13:                           decrease_refs_to(j);
14:                      deallocate(object);
15:                 }
16:            } else {
17:                 for(j  in objects referenced by object) {
18:                      decrease_refs_to(j);
19:                 deallocate(object);
20:            }
21:       }
22:}
```

Program 5.2

The second test of the count is needed because new references to the object may have been created during execution of the destroy method by the assignment or passing of `self`. Property 1 is assured because each time the last reference is removed, the count goes to zero and the destroy method is called. Incrementing the reference count before calling the destroy method prevents recursive calls of the destroy method, which would otherwise occur when references from the destroy method's stack frame (e.g. `self`) are deleted, or when other references created during execution of the destroy method are deleted.

The additional cost for objects not having destroy methods is the cost of the `has_destroy_method` test. In a strongly typed language, this might be done at compile time; otherwise, it is the cost of one method look-up, or the space for a `has_destroy_method` flag bit and the time to check it.

As with normal reference count garbage collection, groups of objects with circular references will not be identified as garbage and any destroy methods associated with the objects will not be called. Likewise, if reference counts are allowed to *saturate*,[33] an object's destroy method will not be called unless the system uses a non-reference counting garbage collector to reclaim saturated objects; in this case, the call to the destroy method may be delayed, as described below. In other respects, the algorithm satisfies the properties given above.

Variations on reference counting, such as deferred reference counting,[34] are changed in a similar way. However, destroy methods will not be called until the next garbage collection occurs. Any algorithm in which garbage collection takes place as a separate phase of execution requires that Property 1 be replaced by:

Property 1a

Each time the last reference to an OWDM (other than from the stack frame of the object's destroy method) is removed, its destroy method must eventually be called exactly once.

Unlike Property 1, Property 1a allows the destroy method to be executed in the next garbage collection, which may be some considerable time after the last reference to the object is destroyed. Even so, Property 1a is difficult to achieve when separate execution and garbage collection phases exist. Consider an object whose destroy method is called in a garbage collection. Three possible situations may follow: no new references to the object are created; a new reference is created, but is destroyed before the next garbage collection; and a new reference is created, and still exists at the next garbage collection. The next garbage collection cannot distinguish between the first two possibilities, but the first possibility should cause the object to be deallocated, whereas the second and third possibilities should both cause the destroy method to be called again. The next section introduces a primitive operation that allows the programmer to resolve this ambiguity.

5.2.2 Mark-scan Garbage Collection

Mark-scan, which is one of the oldest forms of garbage collection, usually takes a lower total amount of CPU time than reference counting, since it does not require action on every reference operation. However, since all useful computation is suspended during garbage collection, it can cause long pauses in the execution of the system. Mark-scan is often used as a secondary garbage collection algorithm by systems that use reference counting to obtain predictable real-time performance, but need another algorithm to collect self-referential structures.

Mark-scan garbage collection consists of two phases. First, all reachable objects are traversed starting from all the references in the interpreter (or some other *system roots*). A mark bit is associated with each object in the system, and is set on each object reached during the traversal, indicating that the object is reachable. In the second phase, every object is visited, and all unmarked objects are deallocated:

```
 1:for(i in system roots)
 2:        mark_from(i);

 3:mark_from(o)
 4:object_ptr o;
 5:{
 6:        if(not_marked(o)) {
 7:              set_mark(o);
 8:              for(j in objects referenced by o)
 9:                    mark_from(j);
10:       }
11:}
```

Program 5.3: Mark Phase

```
1:for(i in all allocated objects) {
2:       if(not_marked(i))
3:            deallocate(i);
4:       reset_mark(i);
5:}
```

Program 5.4: Scan Phase

Note that the scan phase resets the mark bit on all reachable objects, leaving the system ready for the next garbage collection. All objects are created with the bit reset, so that it is never necessary to scan all the objects to initialize the mark bits.

An important difference between mark-scan garbage collection and reference counting is that in mark-scan the roots of a group of objects (e.g. a tree) are not identified, whereas reference counting always processes the roots first. This becomes important with the introduction of destroy methods because the order in which the destroy methods are called may be significant. Suppose there is an OWDM a that refers to an OWDM b. If destroy methods are called in other than root-first order, the destroy method for a could attempt an operation on b after b's destroy method has executed. Thus, the operation might attempt to use object b when it is in an inconsistent state.

If the group of objects includes a cycle of more than one OWDM, there is no *a priori* correct order in which to execute the destroy methods. In fact there might not be an order which does not cause a destroy method to use an object whose destroy method has already executed — thus contravening Property 4. This is not an issue with reference count garbage collection because it does not identify such object groups as garbage. However if the objects are ever to be collected the cycle must be broken in some way. One rather *ad hoc* solution, is to introduce the following additional property dealing explicitly with cycles, and to note, with some distaste, that it can cause the violation of Property 4:

Property 6

If an object a is unreachable from the system roots, but is reachable from another object b (also unreachable from the system roots), then a's destroy method will be called only if b's destroy method has already been called or if b is also reachable from a. In the later case, the destroy method will be called for one object in the cycle containing a and b.

The mark-scan garbage collection algorithm can be modified to call destroy methods in the correct order. Besides the mark bit, two additional flag bits are needed. One of them, called the *destroy* bit (D), is set when it is determined that the object's destroy method must be, or has been called. This bit is only needed in objects that have destroy methods, but for simplicity we shall assume that it is defined for all objects. The other bit, called *colour* (C), identifies objects that are reachable from objects whose destroy methods are to be called. All objects are created with all three bits set to zero. The modified algorithm also requires a mechanism for enumerating OWDMs. This can be efficiently achieved by chaining all such objects together in a list. The space overhead of this is small because most objects do not have destroy methods.

The modified mark-scan algorithm consists of five phases, as follows:

1. Determine those objects that are reachable from the system roots.

2. Determine which OWDMs are not reachable from the system roots. These are candidates for having their destroy methods invoked later in this garbage collection. Such objects are called *destroy candidates*. Also, identify objects reachable from destroy candidates, because such objects should not be deallocated yet (Property 2).

3. Identify the destroy candidates that should not actually have their destroy methods called because they may be used in the execution of another object's destroy method.

4. Scan all allocated objects, deallocating those that were not found to be reachable in phases 1 and 2.

5. Call the destroy methods for each remaining destroy candidate.

Thus, phases 1 and 2 determine the objects that are garbage, and these are deallocated in phase 4. Phases 2 and 3 find the objects whose destroy methods should be called in phase 5.

The five phases are described in more detail below. Figure 5.1 illustrates all possible object states that can occur during garbage collection. The state of an object is determined by a combination of its flag bits and the phase of the algorithm that has completed execution. The ovals in each row represent all possible object states between consecutive phases of the mark-scan garbage collection. States that are not shown cannot occur.

Phase 1 — Mark

This phase marks all objects reachable from the system roots. The algorithm is identical to the previous mark phase, except that the destroy bit is reset on all reachable objects.

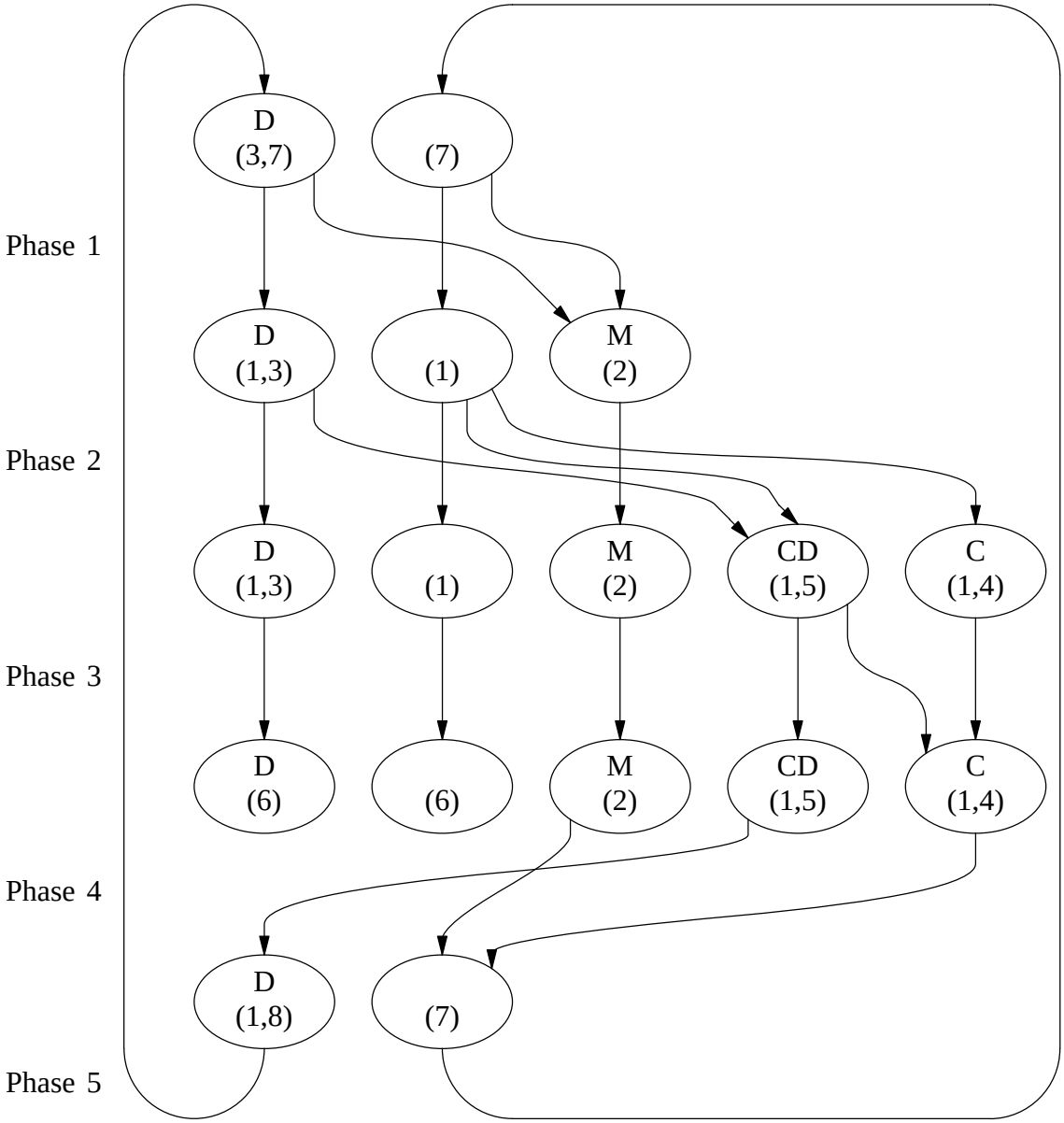
```
 1:for(i in system roots)
 2:       mark_from(i);

 3:mark_from(o)
 4:object_ptr o;
 5:{
 6:       reset_destroy(o);
 7:       if(not_marked(o)) {
 8:            set_mark(o);
 9:            for(j in objects referenced by o)
10:                 mark_from(j);
11:       }
12:}
```

Program 5.5: Phase 1

All objects enter this phase with their mark and colour bits set to zero. Only those objects reachable from the system roots are marked; unreachable objects do not change state. Note that D is set if and only if the destroy method was called in a previous garbage collection, and the object remains unreachable from the system roots.

Phase 2 — Identify candidates

Phase 2 finds destroy candidates and sets their destroy and colour bits. Objects that have their destroy bit set prior to phase 2 are not destroy candidates unless they are reachable from another destroy candidate (indicating that a new reference to the object was created when its destroy methods executed during the last garbage collection).



Key:

(1)	Not reachable from system roots
(2)	Reachable from system roots
(3)	Destroy method was called previously
(4)	Reachable from destroy candidate
(5)	Destroy candidate
(6)	To be deallocated
(7)	Reachability from system roots unknown
(8)	Destroy method will be called

M	Mark bit set
C	Colour bit set
D	Destroy bit set

Figure 5.1: Object State Transitions

```
 1:for(o in objects with destroy method)
 2:       if(not_marked(o)
 3:        &&not_destroy_bit_set(o)
 4:        &&not_coloured(o)) {
 5:            colour_from(o);
 6:            set_destroy(o);
 7:       }

 8:colour_from(o)
 9:object_ptr o;
10:{
11:       if(not_coloured(o) && not_marked(o)) {
12:            set_colour(o);
13:            for(j in objects referenced by o)
14:                 colour_from(j);
15:       }
16:}
```

Program 5.6: Phase 2

In addition, `colour_from` visits destroy candidates and all objects that are reachable from them. These objects are coloured (C) to prevent them from being deallocated in phase 4 (which is required by Property 2). There is no need to visit objects that have already been coloured in this phase since such objects are reachable from destroy candidates and therefore will no longer be destroy candidates after phase 3.

Phase 3 — Prune

This phase ensures that Property 2 holds by identifying destroy candidates whose destroy methods should not be called because they are reachable from other destroy candidates. To do this it is necessary to visit all objects reachable from destroy candidates, and reset their destroy bits. This traversal is limited by objects that are marked, since, by definition, there can be no references back from a reachable object to objects accessible only from objects to be destroyed. The mark bits in the objects are used to restrict the recursion, as in the original mark phase, but are restored as the recursion unwinds ready for the next traversal, namely:

```
 1:for(i in objects with destroy method)
 2:       if(is_destroy_bit_set(i) && is_coloured(i))
 3:            reset_reachable_destroy_bits(i);
```

```
 4:reset_reachable_destroy_bits(o)
 5:object_ptr o;
 6:{
 7:       if(not_marked(o)) {
 8:            set_mark(o);
 9:            for(j  in objects referenced by o)
10:                 if(not_marked(j) {
11:                      reset_destroy(j);
12:                      reset_reachable_destroy_bits(j);
13:                 }
14:            reset_mark(o);
15:       }
16:}
```

Program 5.7: Phase 3

Note that since this routine will traverse every path reachable from each destroy candidate, some objects may be visited more than once.

In any cycle that contained one or more destroy candidates at the start of phase 3, exactly one destroyable object (the first one encountered) will remain a destroy candidate (i.e. have D set) after the execution of this phase has completed.

Phase 4 — Scan

Scan, and deallocate objects that are neither marked nor coloured:

```
 1:for(i  in all allocated objects) {
 2:       if(not_marked(i) && not_coloured(i))
 3:            deallocate(i);
 4:       reset_mark(i);
 5:       reset_colour(i);
 6:}
```

Program 5.8: Phase 4

Note that this phase resets the mark and colour bits on all reachable objects, but leaves the destroy bits intact. Objects that are unmarked but have the destroy bit set are objects for which the destroy method was called in the last garbage collection, and for which no new reference has been created. They are deallocated together with other unreachable objects. Since the scan takes place before the destroy methods are executed, the maximum amount of free space will be available for objects that might be allocated by the destroy methods.

Phase 5 — Call

Call the destroy method on all allocated objects that have the destroy bit set:

```
1:for(i in objects with destroy method)
2:        if(is_destroy_bit_set(i))
3:              call_destroy_method(i);
```

Program 5.9: Phase 5

As noted in the previous section, the memory system cannot always determine if an object that has had its destroy method called, in the previous garbage collection, should have it called again. As written, the destroy method will not be called again unless the object is reachable from a destroy candidate or the system roots in at least one garbage collection before it again becomes a destroy candidate.

This behaviour could be altered by a new primitive operation, `destroy_again`, which could be made available for use in destroy methods. Calling `destroy_again` would assert to the memory system that the object should be eligible to have its destroy method called again. If this assertion is not made, the object would cease to be regarded as having a destroy method, and its lifetime would be determined strictly by its reachability from the system roots. This could be implemented by a revised phase 5:

```
1:for(i in objects with destroy method)
2:        if(is_destroy_bit_set(i)) {
3:              call_destroy_method(i);
4:              if(is_destroy_bit_set(i))
5:                    remove_destroy_method(i);
6:        }
```

Program 5.10: Revised Phase 5

Lines 4 and 5 interact with `destroy_again`. `Destroy_again` resets the destroy bit so that `remove_destroy_method` is not called, otherwise this routine would remove the object from whatever data structures are used to enumerate all OWDMs, thus removing it from consideration in future garbage collections. Resetting the destroy bit guarantees that the destroy method would be called in the next garbage collection if no references to the object are found.

Care should be taken if the execution of a destroy method results in another (recursive) garbage collection. In particular, the destroy bits on all destroy candidates not yet enumerated in phase 5 should be reset so that the invariants expected at the completion of phase 5 hold.

A collection of objects with *k layers* of destroyable objects will be completely deallocated in *k+1* garbage collections. Each garbage collection identifies, and then calls, the destroy methods of the least-referenced layer; an extra garbage collection is then needed to deallocate normal objects which were only referenced by the last layer of OWDMs.

5.3 Examples of Deallocation

Consider the sequence of operations when the structure shown in Figure 5.2 becomes unreferenced.

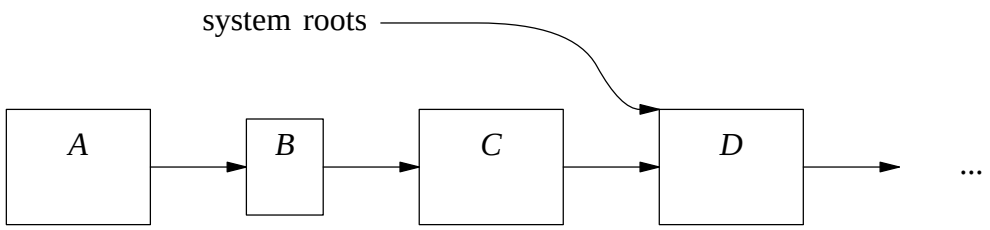


Figure 5.2

In all these diagrams, large boxes represent OWDMs and small boxes represent other objects. The objects are referred to by labels in the top of the boxes. Any of the garbage collection bits that are set in the object are represented by the letters M, C and D at the bottom of the box.

When garbage collection begins, the objects have the states shown in figure 5.2. The mark phase (phase 1) follows the pointer from the system roots, setting the mark bit on object D and all objects reachable from it. Phase 2 then colours all objects reachable from A and C. Note that the result of this phase depends on the order in which A and C are enumerated. Figure 5.3 shows the system state if A is enumerated first,

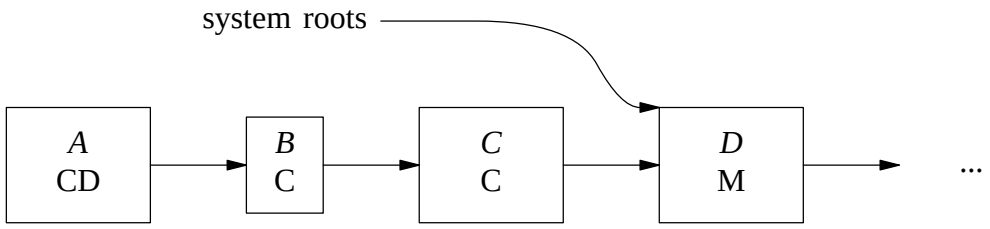


Figure 5.3

and figure 5.4 shows the system state if C is enumerated first.

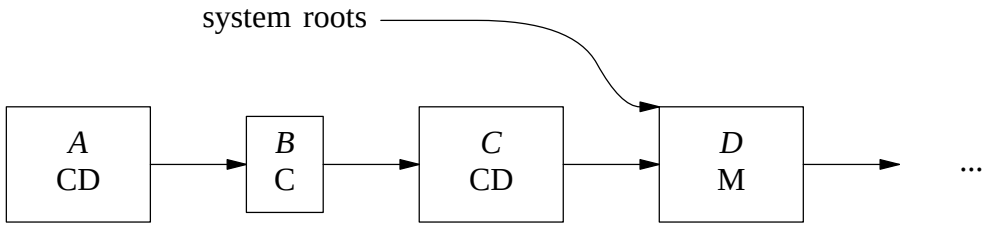


Figure 5.4

Phase 3 then visits all the objects reachable from A and C. Regardless of the order in which A and C are enumerated, the destroy bit in object C is reset in the traversal starting at object A and the traversal starting at C has no effect. Both traversals stop when object D is reached, since it is marked. The result, which is independent of the enumeration order, leaves the destroy bit set only on object A.

None of the objects shown are deallocated in the scan phase (phase 4) since all the objects are either marked or coloured. Finally, the destroy method on object A is called in phase 5. If `destroy_again` was called A's destroy bit was reset, and a similar sequence will occur in the next garbage collection. Otherwise no further destroy methods are required for object A, and it becomes a normal object for the next garbage collection. In this case, the destroy method on object C will be called in the next garbage collection, unless A's destroy method created new references to B or C.

Now consider the more complex structure that includes a loop of references, in figure 5.5.

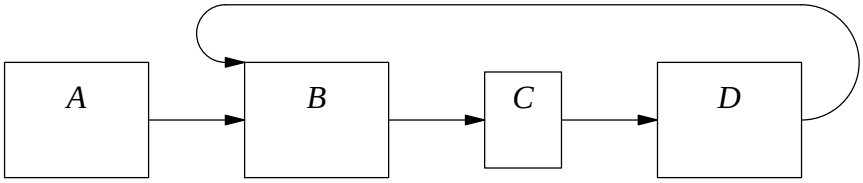


Figure 5.5

Again, the structure is shown before the mark phase. In this case, the mark phase will not reach any of the objects shown, since there are no pointers into this structure. Therefore, this is also the state before phase 2. Phase 2 visits and colours A, B, D, and all the objects reachable from them (C in this case). Thus all the objects shown will be coloured at the start of phase 3. Also, the destroy bits on object A and, depending on the order in which destroyable objects are enumerated, possibly one of B or D will be set.

Phase 3 resets the destroy bits on objects B and D. This occurs regardless of the order in which destroyable objects are enumerated, because all unmarked objects reachable from A are visited by `reset_destroy_bits`. Thus, none of the objects shown is collected in the scan phase, and the destroy method defined on object A is called in phase 5.

If A's destroy method calls `destroy_again` the structure returns to its initial state. Otherwise, at the next garbage collection the objects will be as shown in Figure 5.6:

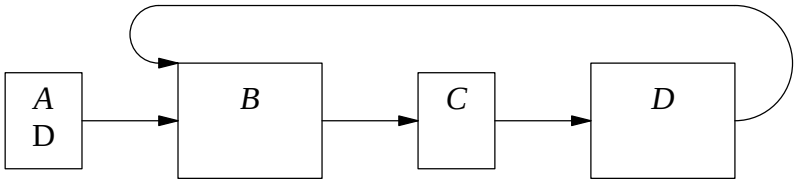


Figure 5.6

Notice that object A is now shown as a small object since it no longer has an active destroy method — It will no longer occur in the enumeration of destroyable objects.

In the next garbage collection, the mark phase again does not reach this collection of objects. In phase 2, the objects reachable from objects B and D are coloured, so that we obtain either Figure 5.7 or Figure 5.8 depending on the order

of enumeration of destroyable objects.

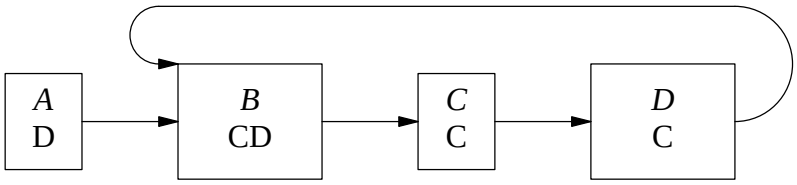


Figure 5.7

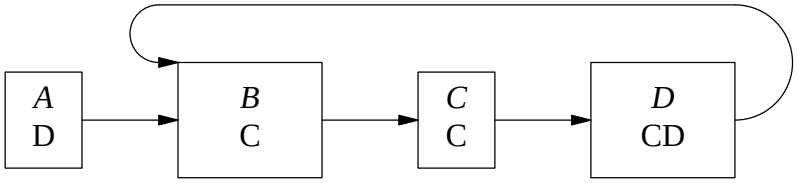


Figure 5.8

In either case, phase 3 has no effect since only one object has its destroy bit set. In the scan phase, object A will be deallocated since it is neither marked, not coloured, and the colour bits are reset in the other objects. Thus, the destroy method on one of object B or object D is executed in the last phase. The effect of phases 2 and 3 is to choose arbitrarily which of the destroy methods for the OWDMs in the cycle will be called.

Since object A no longer exists both cases are equivalent, and we will assume that the destroy method on object B is the one that is executed. Again there are two possibilities, depending on whether or not the destroy methods calls `destroy_again`. If it is not called, the state at the next garbage collection is as shown in Figure 5.9.

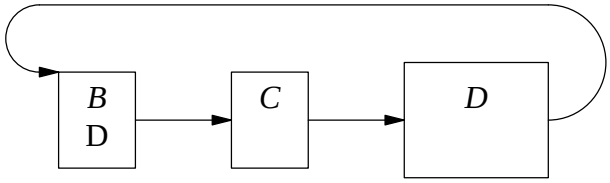


Figure 5.9

Again, this is unaffected by the mark phase. Phase 2 will colour all the objects and set the destroy bit in object D. Phases 3 and 4 have no effect since all objects are coloured. Thus, in phase 5, D's destroy method will be called and it is able to break the loop.

5.4 Program Termination

When a program terminates, all its storage becomes garbage. If the language does not have destroy methods, there is no need to collect this storage because it will be returned to the host operating system. The introduction of destroy methods places additional meaning on the deallocation of objects and the destroy methods associated with the objects should be invoked. Naturally, this would not be appropriate if the objects existed in a persistent store,[94] where the system roots exist independently of the execution of any one program.

With mark-scan garbage collection, it would probably be acceptable to execute one or more iterations of garbage collection. This would not be as time consuming as at other times, since the system roots will no longer contain valid references, and so the mark phase is trivial. However it may need to be repeated several times, until all objects have been deallocated.

It would be useful here for a parameter to be associated with destroy methods to indicate if the garbage collection is due to a lack of free space, or program termination. This would allow destroy methods that normally create new references to their object to have a different behaviour at program termination. A decision must be made about the behaviour of a system with objects that persistently re-create references to themselves at program termination. If this is regarded as a programmer error, the resulting infinite loop may be the correct behaviour.

Other categories, such as a stack-allocated object going out of scope, and garbage collection explicitly invoked by the program may also be useful. In systems with combined garbage collection algorithms, such as reference-counting combined with mark-scan, the type of garbage collection causing a destroy method to be invoked could also be useful information for the destroy method.

In systems that use reference counting, it would be very expensive to count down the references from the system roots, since this potentially involves visiting all objects in the system many times. Also, saturated reference counts, if they are allowed, create objects that cannot be deallocated in this way. If mark-scan garbage collection is also available, this would be more appropriate at program termination.

5.5 Experience

Destroy methods have been available in the language AML/X[87] for some time, where they have been found to be invaluable — especially in the construction of TGMS. However, the existing implementation is not robust in the face of cycles of destroyable objects, and the more complex actions that might be taken by destroy methods. This originally prompted the development of the proposals presented here.

The semantics of destroy methods are much more complex than was originally thought, and it is unfortunate that their exact behaviour depends on the garbage collection algorithm employed, thus allowing implementation to intrude into language semantics.

Experience has shown that very simple destroy methods predominate. However, to simplify the algorithms significantly it is probably necessary to disallow most accesses to instance variables and to `self`. This might still allow the use of destroy methods to manage external resources, but would not allow managing a pool of objects. Nevertheless, it should be possible for a compiler to conservatively identify such *safe* destroy methods, which could then be implemented more simply and efficiently.

Reference counting garbage collection gives a particularly simple way to implement destroy methods. Mark-scan garbage collection presents more problems, but an implementation is possible, albeit with some delay before unreferenced data

is deallocated. The ability to enumerate all destroyable objects and two extra mark bits on the objects are required.

It is unclear what the correct behaviour should be when one or more OWDMs appear in a cycle. Perhaps this alone should suggest that a simpler type of destroy method should be used, but the restrictions seem too great.

The next chapter describes an alternative technique that allows the programmer to provide the effects of destroy methods using other, lower-level, primitives. This overcomes most of the problems with the techniques of this chapter.

Chapter 6

Weak Pointers

In the previous chapter it was shown that there are circumstances in which it is very useful if an object can arrange to be warned of its impending deallocation. This warning appeared as a message to the object from the garbage collector, causing the invocation of a *destroy method* defined in the object. Modifications to the well-known *reference-counting* and *mark-scan* garbage collection algorithms to support destroy methods were described.

The modifications to reference-counting are simple, suffering only from reference-counting's inability to detect unreachable groups of objects containing circular references. In contrast the modifications for mark-scan garbage collection are very complex, and in the absence of a formal proof of the correctness of the resulting algorithm it is difficult to feel confident of its correctness. Unfortunately both these garbage collection algorithms have been replaced by more efficient algorithms in most modern systems, and the modifications described give few hints about ways in which many of these algorithms might also be modified to support destroy methods.

Sometimes complex algorithms are caused by an inappropriate choice of the primitive operations on which the algorithm is based. The primitives might be at too high a level of abstraction, making the desired control difficult to obtain, or they might be at too low a level, requiring the algorithm to deal with too many irrelevant details. The purpose of this chapter is to demonstrate that the effect of destroy methods can be provided quite simply, by using more appropriate low-level primitives. These are more widely useful, and the entire resulting system is significantly simpler, and hence easier to implement, more trust-worthy, and hopefully also more efficient, than the schemes described before.

A further advantage is that fewer details of the underlying garbage collection algorithm are made visible to the programmer in the definition of the destroy method mechanisms. However some differences, such as those caused by the choice of synchronous or asynchronous garbage collectors cannot be completely hidden.

Much of the code to control destroy methods is moved from the memory system, into libraries which could be normal user code. This has obvious advantages for the development and maintenance of both the destroy method code, and the memory system.

The primitives used for this formulation of destroy methods, in order of importance, are *weak pointers*, *forwarding objects*, and *garbage collection notification*. There are several variations but weak pointers underly these as the fundamental mechanism.

6.1 The Primitives

Weak pointers

Weak pointers are references to objects that are ''not strong enough'' to stop the object from being deallocated by a garbage collection. A weak pointer can be created to refer to any object. This can later be *firmed* to obtain a new 'normal' reference to that object, provided that, in the intervening period, a garbage collection did not occur which collected the object because at that time there were no 'normal' references to it. If the object has been collected, any attempt to firm the weak pointer will produce an exceptional value, say *nil*, raise an exception, or signal its failure in some other way appropriate in the language being used.

It is important that a weak pointer must be firmed explicitly before the object to which it refers can be used, since this is when a new 'normal' reference to the object is created, and also when the previous collection of the object might become apparent and require special action by the user program. The dereferencing operation must be indivisible — in particular it must not be interrupted by a garbage collection.

Weak pointers are not new, being available in the T system,[95] on the Flex machine,[96] where they are called *shaky pointers*, and elsewhere. Some of the uses that are made of weak pointers in these systems will be discussed later.

Pointers that are weaker even than weak pointers might be proposed. These pointers would be cleared by any garbage collection, regardless of whether the object to which they refer has been deallocated or not, however these do not seem to be sufficiently useful to justify their inclusion in a system, despite their simpler implementation.

Forwarding Objects

Forwarding objects are simply objects that achieve their behaviour by forwarding all the messages that they receive, to another object, and then returning the result from that object. They represent indirect references to objects. Similar constructs have been used in Smalltalk-80 to provide transparent access to objects on other systems over a network,[97] here the forwarding can be achieved by an arbitrarily complex action — a remote procedure call in this case. Transparent forwarding objects which do no processing of forwarded messages have not previously been seen to be useful, since if they are transparent, why have them there at all? Here we provide an example of their utility.

Forwarding objects are not strictly necessary as primitives of the system, as will be shown later, but the abstraction is useful, and simplifies the description of the implementation of destroy methods.

Garbage collection notification

Garbage collection notification is the simplest primitive of all, we simply require a mechanism that allows routines provided by the user, to discover when a garbage collection has occurred. There are several different ways in which this might be achieved, ranging from a semaphore signaled by the garbage collector, to a list of functions that are to be called automatically after each garbage collection is completed.

6.2 Implementing Destroy Methods

In its simplest form (some variations will be discussed later) destroy methods are implemented by a library which interacts in a simple way with the memory system using weak pointers. The library maintains a list with one element for each object in the system that has a destroy method. As before these objects will be called *OWDM*s, and this list will be called the *destroy list*. Each element of the destroy list contains a normal pointer to the OWDM, and a weak pointer to a forwarding object to the OWDM. It is arranged that no other normal pointers to the OWDM exist in the system, instead all other references are to the forwarding object, but since forwarding is transparent this is not visible to the rest of the system. Thus the destroy list has the following structure, where weak pointers are represented by dashed arrows:

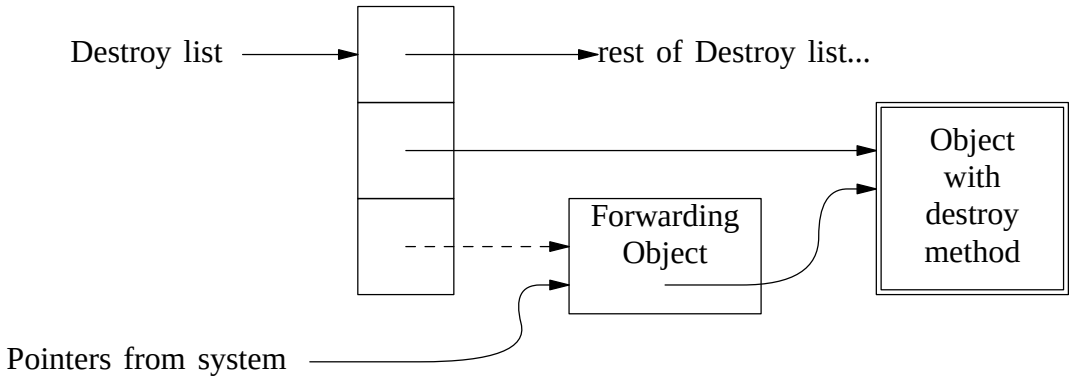


Figure 6.1: An Element of the Destroy List

After each garbage collection the destroy list is traversed and each weak pointer is tested to see if it has been cleared. This would imply that the forwarding object has been garbage collected, because there are no references to it from the rest of the system. But as far as the rest of the system is concerned the forwarding object *is* the OWDM, and so this tells us that the destroy method in the OWDM should be called.

When an OWDM is created it must be inserted into the destroy list by calling a routine from the destroy method library, which will be called "`make_destroyable`". This routine creates a forwarding object, updates the destroy list, and returns the forwarding object as its result. It is important that a reference to the forwarding object is returned by the OWDM's creation routine, rather than a reference to the OWDM itself, since we are relying on all references

to the object being via its forwarding object to determine when it becomes unreachable.

Similarly methods in the OWDM should not export references to `self`, but should use references to the forwarding object instead. A library routine `get_forwarding_object` will obtain a reference to the forwarding object which can be used in these situations. If this carries too high an overhead an OWDM could remember, in an instance variable a weak pointer to its forwarding object, which can be firmed each time a pointer to self is required. Obviously the object must not remember a normal pointer to its forwarding object!

While this complicates the definition of OWDMs, it is important to note that the rest of the system is not aware if any particular object is an OWDM or not, so that the notational cost is limited to the objects requiring the additional functionality of destroy methods.

When the library calls a destroy method in an OWDM, it is removed from the destroy list, and so will be collected by the next garbage collection. However it is possible for the destroy method to call `make_destroyable` again so that a new entry is made for the object in the destroy list, and its destroy method will be called once more before the object can be deallocated. Of course a reference to a forwarding object which is returned by `make_destroyable` can be given to other objects by the destroy method, which is how an OWDM would return itself to a pool of free objects, as was described previously.

Here care must be taken that the insertion of this new element into the destroy list does not conflict with the traversal of the list which is currently taking place by the library. This can be achieved by creating all new destroy list elements on a second list which is appended to the destroy list when its traversal finishes. Similarly the insertion of a new element into the destroy list must be indivisible, so that it cannot be interrupted by a garbage collection which then causes the library to start a traversal.

6.2.1 Variations on the Algorithm

The exact mechanism of garbage collection notification which causes the execution of the destroy method library has not been described in detail, since there are several alternatives depending on other properties of the system. The first, and simplest approach is if the last action of the garbage collector is to call a user-supplied routine — several systems, such as T, maintain a list of actions to be carried out before, and after each garbage collection, and allow libraries to add their own actions to these lists. This has several disadvantages, in particular the time a garbage collection takes may be increased by possibly many arbitrary computations in the destroy methods. This is probably unacceptable in systems that aim for interactive response even in the presence of garbage collections.

More fundamentally garbage collection, and hence the execution of destroy methods, occurs at an unpredictable point in the execution of the system so that care must be taken if the destroy method interacts with other objects. Even in a system that provides semaphores, or other concurrency control mechanisms, these are not

useful in this situation, unless it can be guaranteed that the execution of destroy methods cannot block (since this would, in effect block the garbage collector!), or a sensible meaning of blocking can be provided. An even worse situation occurs if the execution of a destroy method causes another garbage collection, since concurrent access to the destroy list must then be allowed.

In a system that provides (pseudo-)concurrent processes many of these problems can be addressed. In this type of system it is natural for the destroy method library to define a process, rather than a routine, which discovers unreachable OWDMs and executes their destroy methods. This process would normally be blocked, but the last action of the garbage collector would be to make it runnable, by signaling a semaphore, or calling a Unix-style *wake-up* routine. The process traverses the destroy list, calling the relevant destroy methods, and then loops back, blocking until another garbage collection occurs.

The cost added to garbage collection is now very small, as it only has to make the destroy method process runnable. If the system allows priorities to be associated with processes then the destroy process can be safely given a relatively low priority — the user has little control over when destroy methods are executed, since this depends on a garbage collection occurring, thus the extra delay caused by executing destroy methods at a low priority should not be important. If the system has some idle time this can be used to execute pending destroy methods, although some care is needed that large numbers of such objects do not accumulate, since their memory cannot be reclaimed until the destroy methods have been executed. The previous routine-based solution always ran the destroy methods at a very high (i.e. garbage collection) priority.

Now that we have a separate process, it is also possible for a destroy method to use normal techniques to synchronise access to shared data structures. If the destroy method process blocks while executing a destroy method, this might delay the execution of other destroy methods, but is preferable to the previous situation.

Now a garbage collection occurring during the evaluation of a destroy method, or before the destroy method process has completed its scan of the destroy list and blocked again, does not require special handling since there is no possibility of the destroy method routine being active more than once.

It is possible that the resetting of weak pointers for some objects will not be discovered until the next garbage collection occurs at which the destroy method process is blocked awaiting a garbage collection, since this is the only time at which the destroy method process goes back to the start of the destroy list. This leads to another variation in the algorithm, which removes the requirement for the garbage collection notification primitive altogether! This is simply to have the destroy method process never block, but simply always cycle through the destroy list at a low priority.

In a system that provides very cheap processes another possibility exists; there could be a destroy method process for every OWDM in the system! Instead of the garbage collector calling the wake-up function on a global ‘‘there has been a garbage collection’’ variable, it could wake-up processes waiting on variables associated with each weak pointer that is cleared. This is more generally useful,

and so is an obvious extension to the concept of weak pointers,[98] it fixes the problem of a destroy method sleeping to get access to a shared variable, and delaying the evaluation of other destroy methods, but might incur a large cost in the process scheduler, both from the large number of waiting processes that it might cause, and also because (potentially) many processes could all become runnable at the same time.

6.3 Weak Pointers

In the previous section we have seen how destroy methods can be conveniently implemented using weak pointers. It remains to show that the implementation of weak pointers themselves is feasible, and indeed much simpler than the implementations of destroy methods in the previous chapter.

To recap, a weak pointer is a reference to an object that is ''not strong enough'' to stop the object from being deallocated by garbage collection. A weak pointer can be created to any object, and then later a 'normal' reference to the object can be obtained, if the object has not been deallocated, by firming the weak pointer. In a garbage collection an object's storage is reclaimed if it is not referenced by a normal pointer. Any weak pointers that reference an object that is reclaimed are *cleared* so that future attempts to firm them will return a null pointer, raise an exception, or signal the non-availability of the object in some other way appropriate to the language.

No other operations are allowed on weak pointers — this limits the operations which must interact with the memory system, and should not be interrupted by a garbage collection. Successfully firming a weak pointer creates an ordinary reference to the object, which will stop it from being collected in the normal way.

6.3.1 The Implementation of Weak Pointers

The implementation of weak pointers must be done with some cooperation from the garbage collector. Unfortunately there are now a diverse range of garbage collection techniques[8] which cannot all be considered here. Instead algorithms will be given for Mark-scan, and Fenichel-Yochelson semi-space[99, 90, 100] garbage collection algorithms, which are representative of many others. In both cases recursive algorithms with be presented for simplicity, although both algorithms can be re-cast into non-recursive versions that reverse the pointers they traverse so that they do not need a stack, or other unbounded auxiliary storage.

The algorithms follow broadly the same scheme: a slightly modified version of the underlying garbage collection algorithm, is followed by a pass through all the weak pointers in the system giving them their correct value. The underlying garbage collection algorithm must identify weak pointers, and processes them differently from normal pointers.

The extra pass on each garbage collection should not be very expensive — the work being proportional only to the number of weak pointers in the system. Weak pointers are expected to form only a small proportion of all pointers since they are relatively special-purpose. In some other cases it is possible to combine the extra

pass with the last pass of a multi-pass garbage collection algorithm.

This requires that weak pointers are distinguishable from normal pointers and scalar data. Many systems use tags to recognise pointers to objects, and it may be possible for weak pointers to be given a special tag. However tags are usually a very scarce resource, and it might not be desired to use a tag for this purpose, especially if weak pointers are relatively uncommon. In the VAX Ten15 system[64] weak pointers are given the same tag as normal pointers, but contain the negation of the address of the object to which they refer. This is equivalent to using the sign bit as a third tag bit in pointers, but has the advantage that the extra tag bit is not needed in other data words.

An alternative that can be used when extra tags are not available, is to make weak pointers objects in memory, effectively single-word records referenced by normal pointers, which contain the address of the weakly referenced object, or Null. Weak pointer objects are then given their own object type, so that the garbage collector can identify the object as a weak pointer.† This is similar to the way that 'Standard ML of New Jersey' represents mutable variables (*refs*) as memory objects, even though they could be represented by a single word.[101]

By making weak pointers into memory objects it becomes possible for them to use more than one word of memory. In particular, if a weak pointer object is two words long (plus whatever object header the system requires) the second word can be used to keep a list of all the weak pointers in the system, so that they can be conveniently enumerated by the garbage collector. This list must be treated specially by the garbage collector, if it is not to keep weak pointer objects alive which can only be reached from it. In fact we shall see that it is usually convenient for the garbage collector to construct the list itself, the extra words not being valid between garbage collections, and existing only so that the garbage collector does not need to obtain storage to remember the list of weak pointers, the size of which cannot be predicted beforehand.

6.3.2 Mark-Scan Garbage Collection

A Mark-Scan garbage collection involves making two passes through the object memory. Each object has a single bit of memory associated with it (called the *mark bit*) which is normally clear. In the first pass all reachable objects are visited by recursively following all pointers reachable from the *system root pointers*. Then in the second pass the whole of object memory is scanned (ie. every object is visited). If an object is marked, its mark bit is reset ready for the next invocation of the garbage collector, otherwise the object is unreachable and is reclaimed for future re-use, normally by adding it to a free list. In a C-like pseudo-code this becomes something like:

```
1:for(i in system roots)
2:        mark_from(i);
```

† Most systems already distinguish several types of memory object, so that the garbage collector can decide if the object might contain pointers to other objects, and how they might be arranged.

```
 3:mark_from(o)
 4:object_pointer o;
 5:{
 6:       if(not_marked(*o)) {
 7:            set_mark(*o);
 8:            for(j  in object pointers in *o)
 9:                 mark_from(j);
10:       }
11:}
```

Program 6.1: Mark Phase

```
 1:for(i  in all allocated objects) {
 2:       if(not_marked(i))
 3:            deallocate(i);
 4:       reset_mark(i);
 5:}
```

Program 6.2: Scan Phase

First we modify the meaning of ‘‘objects referenced by o’’ in `mark_from` to exclude references from weak pointers, this ensures that an object is not marked if it can only be reached by weak pointers. After the mark phase a new phase is added which resets the weak pointers pointing to unmarked objects, viz:

```
 1:for(w  in all existing weak pointers) {
 2:       if(!is_clear(w) && not_marked(firm(w)))
 3:            clear(w);
 4:}
```

Program 6.3: Weak-fixup phase

Note that null weak pointers may be left from previous garbage collections.

If we now assume that weak pointers are objects in memory we can modify the mark phase so that the enumeration in the weak-fixup phase is trivial. A weak pointer object is a structure of the form:

```
struct weak_pointer_object {
       object_header hdr;
       object_pointer weakp;
       object_pointer next;
};
```

where `object_header` is the one-word field that occurs at the top of all memory objects, and contains information for the garbage collector. This usually includes the mark bit, the length of the object, and its *kind*, which might be `POINTERS`, `NO_POINTERS`, or in this case `WEAK_POINTER`. `Weakp` contains the address of the object to which the weak pointer refers, and `next` is used for the enumeration. The first two garbage collection phases now become:

```
 1:object_pointer weak_list_head;
 2:...
 3:weak_list_head = Null;
 4:for(i in system roots)
 5:       mark_from(i);
 6:...

 7:mark_from(o)
 8:object_pointer o;
 9:{
10:       if(not_marked(o->hdr)) {
11:            set_mark(&o->hdr);
12:            switch(object_kind(o->hdr)) {
13:            case POINTERS:
14:                 for(j in object pointers in *o)
15:                      mark_from(j);
16:                 break;
17:            case NO_POINTERS:
18:                 break;
19:            case WEAK_POINTER:
20:                 o->next = weak_head_list;
21:                 weak_head_list = o;
22:                 break;
23:            ...
24:       }
25:}
```

Program 6.4: Mark phase

```
 1:for(w = weak_list_head; w != Null; w = w->next) {
 2:       if(w->weakp != Null && not_marked(w->weakp->hdr))
 3:            w->weakp = Null;
 4:}
```

Program 6.5: Weak-fixup phase

This shows more detail than before, including a test to stop the scanning of objects that are known not to contain object pointers; these might be things such as strings, bitmaps, or arrays of numbers. The field `next`, used to make the list of weak pointer objects, is neither accessible by the user program, nor valid between garbage collections. The scan phase is unchanged by the addition of weak pointers.

6.3.3 Fenichel-Yochelson Semi-space Garbage Collection

A system using Fenichel and Yochelson's copying garbage collection algorithm divides the object memory into two *semi-spaces*, only one of which is used as the system executes normally. New objects are allocated in the current semi-space until insufficient space remains to satisfy an allocation. At this point all the reachable objects in the current (*old*) space are copied into the other (*new*) space, the roles of the spaces are then interchanged and the system continues to execute. This has several advantages:

• Most importantly, the work required is proportional to the amount of memory in use at the time of the garbage collection, rather than the number of objects that have been allocated (as is the case with mark-scan garbage collection).

• Only a single pass is made through the objects in the old space, and the new space is written to sequentially, which usually gives better performance when virtual memory is used.

• After a garbage collection all the live objects are collected together (ie. it is *compacting*), this also gives better paging performance, and allows a simpler allocation algorithm, where new objects are simply ‘‘carved off’’ the free memory block, without the need for free lists, etc.[101]

The only complication required is to ensure that several pointers to the same object, still point to the same object after a garbage collection. When an object is copied into new-space, its header in old-space is changed to a *forwarding pointer*,† and the address of the object in new-space is recorded in the old object. Whenever the garbage collector follows a pointer it checks to see if it references a forwarding pointer, and if so the address of the copy in new-space can be used directly.

The complete algorithm takes the following form. It is assumed for simplicity that the forwarding address can be contained in the header together with the indication that this is not a normal header.

```
1:Word *ffw; /* pointer to first free word */
2:...
3:ffw = base of new space;
4:for(i in system roots)
5:       i = copy(i);
6:swap new and old spaces
7:...
```

† Not to be confused with the forwarding objects described in the next section. Forwarding pointers are not visible to the programmer, and only exist while garbage collection is in progress.

```
 8:object_pointer copy(o)
 9:object_pointer o;
10:{
11:       Word *posn, *result;
12:
13:       if(is_forwarded(o->hdr))
14:            return get_forward(o->hdr);
15:
16:       result = posn = ffw;
17:       ffw += get_size(o->hdr);
18:       switch(object_kind(o->hdr)) {
19:       case POINTERS:
20:            *posn++ = o->hdr;
21:            for(j  in object pointers in *o)
22:                 *posn++ = copy(j);
23:            break;
24:       case NO_POINTERS:
25:            fast_copy(result, o);
26:            break;
27:       ...
28:       }
29:       o->hdr = forwarder_to(result);
30:       return result;
31:}
```

Program 6.6

The `fast_copy` routine does not need to examine the words it copies into new-space, since it is known that the object does not contain pointers.

The modifications to the mark phase of mark-scan garbage collection can be used (almost without change) in `copy`, but we can do better — there is no need for each weak pointer object to contain a spare word for the list of weak pointers to be constructed! Instead we can use the fact that, after it has been copied to new-space, a two-word object (now including the header) such as a weak pointer object is actually represented by four words, two in new-space, and two in old-space. However the old-space copy is only needed to forward other references to the weak pointer, so the other field can be used to construct the list for the weak-fixup phase.

A weak pointer object is now represented by:

```
struct weak_pointer_object {
       object_header hdr;
       object_pointer weakp;
};
```

and the semi-space algorithm becomes:

```
 1:Word *ffw;           /* pointer to first free word */
 2:weak_pointer_object *weak_list_head;
 3:...
 4:ffw = base of new space;
 5:weak_list_head = 0;
 6:for(i in system roots)
 7:       i = copy(i);
 8:/* Weak-fixup phase: */
 9:for(i = weak_list_head; i != 0; i = i->weakp) {
10:       object_pointer o = get_forward(i);
11:       if(is_forwarded(o->weakp->hdr))
12:            o->weakp = get_forward(o->weakp->hdr);
13:       else
14:            o->weakp = Null;
15:}
16:swap new and old spaces

17:object_pointer copy(o)
18:object_pointer o;
19:{
20:       Word *posn, *result;
21:
22:       if(is_forwarded(o->hdr))
23:            return get_forward(o->hdr);
24:
25:       result = posn = ffw;
26:       ffw += get_size(o->hdr);
27:       switch(object_kind(o->hdr)) {
28:       case POINTERS:
29:            *posn++ = o->hdr;
30:            for(j in object pointers in *o)
31:                 *posn++ = copy(j);
32:            break;
33:       case NO_POINTERS:
34:            fast_copy(result, o);
35:            break;
36:       case WEAK_POINTER:
37:            *posn++ = o->hdr;
38:            *posn++ = o->weakp;
39:            o->weakp = weak_list_head;
40:            weak_list_head = o;
41:            break;
42:       ...
43:       }
44:       o->hdr = forwarder_to(result);
45:       return result;
46:}
```

Program 6.7

6.3.4 Other uses of Weak Pointers

The T system[95] provides weak pointers, as well as several other data structures which provide weak versions of conventional sets, lists, associations, and tables (key to value mappings). These could have been provided using weak pointers, by any user of the system, but are available as part of the standard environment.

Weak pointers were also developed independently for the Flex system,[102] where they are called *shaky pointers*. Flex encourages the use of first-class function values, and shaky pointers were originally developed to decrease the load placed on the memory allocation system by the heap allocation of function activation records.

Each function value includes a shaky pointer to an activation record which could be used when it is next executed. This would have been left by a previous execution of a return instruction, which verifies that no other references to the activation record have been produced, which could have been exported to a wider scope. The result is that repeated calls of a function will usually re-use the same activation record, making function calls cheaper on average, and decreasing the rate at which activation records are allocated, and hence the frequency with which the garbage collector must be executed. When a garbage collection occurs all the remembered activation records are reclaimed, since the chances are that many of them are attached to functions that will not be called again for some time.

The most common use of weak pointers by normal programs, is to cache a value whose recalculation is possible, but undesirable. Using a weak pointer a reference to the value can be kept, but in the event of space becoming scarce, its storage can be reclaimed by the garbage collector. There are many variations on this, ranging from traditional caches, to memo functions.

However weak pointers have uses beyond the construction of caches. In the Flex system, objects on the disk are represented by *disk capabilities* which are main memory data structures containing all the information that is required to access the value from the disk. It is important that two copies of a disk capability do not exist in one main memory, so that a consistent view of the disk is maintained. Thus when a disk capability is read from the disk, as a field of another disk object which is being read, the system must check to see if another copy of this capability is already in memory, in which case the existing copy should be used instead of the new copy. Of course the disk subsystem cannot keep references to all the disk capabilities that it has read, or these would never become unreachable, and so would eventually fill the whole of memory. Instead a weak pointer is kept to each disk capability.

It would have been possible for the disk sub-system to have been written in a way so that it was not necessary to keep disk capabilities unique, and so use weak pointers. However Flex also provides *remote capabilities* which are capabilities to functions and datastructures on other Flex computers, and the implementation of these depends more crucially on the unique copies of remote capabilities in each machine's main memory, and hence on the use of weak pointers.[96]

6.4 Forwarding Objects

The implementation of destroy methods uses *forwarding objects* which are objects that resend all the messages that they receive to another object. Similar objects have been proposed as a mechanism for providing distributed object-oriented systems,[97] where they were called *Proxy objects*. Here the forwarding of a message could involve its transmission to another computer system, where the appropriate method is to be executed. The forwarding mechanism can be arbitrarily complex, and it is likely that only the final delivery will use the normal message send mechanism. The result of executing the method is returned in some way to the forwarding object, and thence back to the initiating object, unless the system allows the sender of a message to also nominate the object to which the reply should be sent, in which case it might be returned directly to the originating object.[45] This would be similar to passing the *continuation* in the message.

Forwarding objects are related to *delegation* which is the alternative to inheritance used by *actor*-based languages, such as Act,[45] and Acore.[46] In a system that uses delegation an object can nominate another object as its *delegate*, which means that any message sent to the object, for which it does not have a method, is sent to the delegate instead. This should be compared with the search through behaviours defined by the class, and then the superclasses of an object receiving a message, which implements the sharing of behaviours that is fundamental to inheritance. The exact relationship between delegation and inheritance is explored elsewhere,[20, 21] but it appears that forwarding objects are objects in a system with inheritance, which implement a less general form of delegation, where all messages are delegated. A system which provides delegation can provide forwarding objects trivially.

Forwarding objects might be provided as primitive constructions by an object-oriented system, or can be programmed explicitly, in a ‘‘brute-force’’ manner by the user. Smalltalk-80 also allows a more elegant approach. It is interesting to consider the type of forwarding objects, and the implications this might have on appropriate type systems for object oriented systems.

6.4.1 Forwarding Objects in Smalltalk-80

Smalltalk-80 allows the ‘‘brute-force’’ implementation of forwarding objects. If a forwarding object is required to an object of class `c`, then we can define another class, `c'`. This defines objects with a single instance variable to contain a reference to the instance of `c` to which messages are to be forwarded. A method is defined for each message understood by `c`, which simply returns the result of sending the same message to the object in its instance variable.

There are several problems with this approach. Most objects in Smalltalk-80 systems respond to a large number of different messages, so that defining the forwarding messages would be tedious, also care would have to be taken that the addition or deletion of methods in the client were reflected in the definition of the forwarding object. Both these problems would be decreased to some extent by automating the production of the classes of forwarding objects.

More fundamentally, it seems wrong that so many forwarding classes and methods must be provided, which all do essentially the same thing. Obviously some way of abstracting the forwarding behaviour is required, and we really wish to define a class `Forwarder` which could provide forwarding objects for any objects in the system.

This can be done in Smalltalk-80 since there is no static type checking, and because the system exposes the underlying message send mechanism in a useful way. The class `Forwarder` defines objects which only provide a behaviour for a single message called `doesNotUnderstand:`, which is a special message which can be generated by the Smalltalk virtual machine. When a message is sent to an object the virtual machine looks for a method associated with that message in the *message table* stored in the object's class. If a method is not found the search is repeated in the class' superclass, and so on, up the superclass chain until a method for the message is found. This search directly implements inheritance.

It is possible for a message not to be understood by any of the classes in this chain, which is discovered if a search for the message fails in a class that does not have a superclass (this will usually be the class `Object`, which is the root of the inheritance hierarchy). When this happens the virtual machine creates an object representing the failed message, together with its parameters, and sends the message `doesNotUnderstand:` to the original receiving object, with this packaged up message as the parameter. A method implementing `doesNotUnderstand:` is then searched for, in just the same way as before. Usually the implementation in class `Object` is found, which signals an error to the user, and allows the debugger to be used to diagnose the failure. However two other possibilities can occur. Any class can provide a new implementation for this message, so that object-specific error behaviour can be provided. Alternatively the search may fail a second time, which is detected as a fatal error by the virtual machine.

Forwarding objects can be constructed by creating objects that only have a behaviour for the `doesNotUnderstand:` message. Any other message sent to the object will fail to find a corresponding method, and so will be packaged up as a message object and given as a parameter to a `doesNotUnderstand:` message. The `doesNotUnderstand:` method simply invokes the message on its target object using `perform:`, inherited by normal objects from class `Object`.

A problem remains with objects whose structure is known by the Smalltalk virtual machine. This is the only point at which objects are not used in the object-oriented way, by sending messages. Instead the virtual machine extracts fields directly from parameter objects, which will of course fail if the object is a forwarding object. It is not clear how, or if, the virtual machine might be restructured to avoid this problem, without introducing large additional overheads, but fortunately it is unusual to wish to forward messages to such objects, and so the mechanism described can be used for most practical purposes.

6.4.2 Forwarding Objects in Typed Languages

Both the approaches to defining forwarding objects in Smalltalk relied on being able to treat an object as if it were an instance of the class of the object to which it was forwarding messages. This is allowed in Smalltalk since checks of the validity of messages are made when the messages are sent, at runtime. Many other object-oriented languages, notably C++,[27] Trellis/Owl,[12] and Eiffel,[14] are type-checked at compile-time, and so the forwarding objects must be type-compatible with the objects to which they are forwarding, if they are to be made invisible to the rest of the system.

In these languages this means that the forwarding object's class must be a subclass of the object to which they are forwarding messages. This can be achieved using the ''brute-force'' approach above, but unfortunately instance variables are inherited by a subclass as well as all the messages of the superclass. Usually this is necessary to allow inherited methods to continue to access their instance variables, but here we are overriding all the inherited methods, and so the instance variables are superfluous.

This is another example of classes where independent control of the types of objects, and hence the sub-type relationships into which they partake, and their implementation, possibly by inheritance, is necessary in order to create clean definitions. We wish to define the forwarding object to have a sub-type of the object to which it forwards messages, so that it can appear wherever the target object could have appeared. But we do not want to implement it by inheriting the implementation of the target object, since this behaviour has nothing to do with the mechanisms of forwarding. Rather we would prefer to simply provide all the methods that are needed. This problem was first discussed by Snyder,[16] and the forwarder could be constructed in this more direct way in his language, CommonObjects.[39]

6.4.3 Primitive Support for Forwarding Objects

In languages which carry out the run-time resolution of messages to methods by indexing into a *message table*, such as C++, it may be possible to have a single implementation of the forwarding class, which can be shared by all forwarded objects.

In C++ the first field of each object is a pointer to an array of pointers to the functions that implement the methods for all the instances of its class. This array is the message table associated with the class — also called a *vtable* in C++ literature. When a message send is compiled, code is generated to call the nth function pointer in the message table of the receiving object. The value of n is a constant calculated by the compiler, from the message, and the type of the receiving object. This allows subclasses to override inherited methods by pointing elements in their message tables to their own definitions of methods. They can also provide methods for extra messages, by extending the message table, but these messages cannot be used in places where the object is being used as an instance of one of its superclasses, since these messages are not defined by the superclass.

It is possible for a forwarding object to be implemented by an object containing the pointer to the target object, with its method table pointer referring to a special *forwarding message table*. The aim is for this message table to be shared by all forwarding objects in the system, the only constraint is that the table must contain entries for every message that might be sent to a forwarded object, and so must be at least as big as the biggest message table of a forwarded object.

The methods in this object table are identical, except that each knows its own position in the table. When a message with offset n is sent to a forwarding object, the forwarding method simply calls the nth method in the message table of its target object.

Care must be taken to ensure that the method receives the correct parameters. The nth forwarding method must be able to forward messages to any object that can accept a message with offset n, many of which will probably require different parameters. On most systems this can be achieved by the forwarding method jumping to the target method, rather than calling it, without rearranging the parameters. Thus the target method sees the parameters as they were arranged by the caller. Note that this implies that the forwarding methods would probably have to be written in assembler, since this cannot be expressed directly in C++.

In fact the forwarding method must change the value of self that will be seen by the target method, this is passed by C++ as an extra parameter, but (at least in the present AT&T C++ translator) its position is always the same, so it can be altered by the forwarding method, without further knowledge of the message, or the target object, being needed.

If it were known that an implementation always left the method table index — n in the previous discussion — in a standard place after the call of a method, all the forwarding methods could be the same. However this is not usually the case, since there is no other reason for the index to be kept, and changing the C++ compiler to do this would probably add a small additional cost to all message sends.

The problems with the inadequacy of the C++ type system for forwarding objects can be overcome when all the objects involved are referenced by pointers, since C++ allows arbitrary conversions using casts. A forwarding object to an instance of class `c` can be created, and then a pointer to it cast to be a pointer to an instance of `c`. Thus the forwarding object can be hidden as far as the rest of the system is concerned. This would not be possible in languages that do not allow such freedom with conversions, where the brute-force approach would probably be needed, at least for the definition of the type of the forwarding object.

It should be noted that these techniques for forwarding objects only work for *purely-object oriented* classes. Classes that expose any part of their instances' implementation to external access cannot be forwarded. Such exposure might be through visible instance variables, or in C++ the use of non-virtual member functions — that is messages whose resolution does not go through the message table, but is carried out statically, at compile time.

6.4.4 Alternatives to Forwarding Objects

We have seen that forwarding objects can be provided in many object oriented languages, but their implementation is often rather unsatisfactory, requiring support from the language, or non-trivial effort from the programmer. There are also some unresolved issues about the semantics of forwarding objects — in particular the meaning of object-equality must be re-examined.

In this situation it is appropriate to ask if the full power of forwarding objects is required for the implementation of destroy methods. For many OWDMs we do not need the whole object to survive its deallocation, and be valid when the destroy method executes, rather this was required so that the destroy method could execute in the context of a consistent object. However in practice usually only a small part of the state of an OWDM is needed for the execution of the destroy method, and thus this is the only state that need be kept in the OWDM.

Recognition of this leads us to a different allocation of the behaviour between the objects. What was previously the forwarding object now contains most of the state, and defines most of the methods. This will be called the *primary* object, since it is also the object to which the rest of the system has references. The other object, now called the *secondary* object, contains only the information that is needed for the execution of the destroy method. It defines the destroy method, and service methods to be used by the primary object. As before the destroy list keeps a weak pointer to the primary object, and a normal pointer to the secondary object.

The methods in the primary object fall into three categories. First those that can be entirely implemented with the state in the primary object. Next are those that can carry out most of their behaviour locally, but may need to use simple behaviours from the secondary object, either to access or update its state. Finally there are the methods that are easier to implement entirely in the secondary object, and so the corresponding messages must be forwarded by the primary object. In effect the previous approach put all methods into this third category. By splitting the behaviour in this way we have reduced the number of messages that need to be forwarded, so that the 'brute force' approach can be used much more easily.

Carrying this to its ultimate conclusion we can see that objects should be designed from the start with the execution of destroy methods in mind. If the overhead of the primary object keeping values in the secondary object is thought to be too great, then copies of those values could be kept in both objects, and it only becomes necessary for the secondary object's value to be kept up to date. A library could still be provided which uses weak pointers to determine when the primary object becomes unreachable, and send the destroy method to the secondary object. There is no longer any worry that the secondary object might not be consistent, since it is an independent object. Of course the secondary object must not contain a reference to the primary object as this would stop it from ever becoming unreachable, and so the weak pointer to it would never be cleared.

This approach solves many of the problems seen in the last section with the implementation of forwarding objects, since we no longer require a large number of messages to be forwarded. Also it will be easier to ensure that references to the internal object are not accidentally exported, since these will not have a generally

useful behaviour and so it will be less likely that it could be used by accident. The disadvantage is that it is more difficult to add a destroy method to an existing class, and that the programmer must be more aware of the details — however these details are probably easier for the programmer to understand.

6.5 Conclusions

Several improvements in previous techniques for the definition and implementation of destroy methods have been proposed. In particular weak pointers provide the mechanism by which the reachability of objects is determined. Since weak pointers have a well defined behaviour, can be provided for a variety of garbage collection algorithms, and are more general than destroy methods, a strong case can be made for their inclusion in any system with garbage collection.

Forwarding objects are, in contrast, less well understood constructs, but where they are available, they can be used with weak pointers to provide destroy methods. However re-structuring objects that require active deallocation can achieve the desired effect without requiring full-blown forwarding objects, and it is expected that this approach would be preferable in most systems.

Some other system support is also preferable, in particular multi-tasking allows destroy methods to be executed in the background, rather than increasing the time for garbage collection. Notification of the occurrence of a garbage collection can help in the scheduling of this background process. Again these are features that would be desirable in most systems for more general use.

Semantic problems, such as the correct order of destruction of collections of OWDMs including cycles, remain. In the systems described such structures will never be deallocated, much as was seen previously with reference-counting, however the ‘solution’ to this given in the modifications for mark-scan garbage collection was at best *ad hoc*, and so its loss is not seen as a problem. On balance it is probably best to force the programmer to explicitly consider this difficult situation, and decide on the appropriate mechanism to resolve it. This is much easier now that destroy methods are not primitives of the system.

It is particularly interesting that choosing the correct level of abstraction, has led to a much simpler implementation of destroy methods, while also introducing primitives that are more generally useful. This is not surprising since there is no substitute for ‘‘getting it right’’,[103] but getting it right first time is not always easy!

Chapter 7

Conclusions and Further Work

This thesis has examined some aspects of the support required for object-oriented systems, mostly in the context of statically strongly-typed systems. The emphasis has been on primitives that allow the appropriate support to be constructed, rather than providing the support directly.

The advantages of this approach are that a single system can support a range of different views of objects, without requiring special support for each view. Thus the result is more flexible, and gives the programmer more scope to choose techniques that are thought to be appropriate in any particular situation. It also means that fewer primitive operations need be provided by the system, and these are usually simpler than the operations for direct support would have been. This makes the basic system easier to implement, debug, and possibly prove correct, which in turn means that higher-level operations can be built on a more trustworthy base.

A possible disadvantage for strongly-typed systems is that simpler type-systems often do not allow primitives to be composed in the ways that are needed. This implies that more powerful type systems should be used, and this certainly has a generally beneficial effect on the utility of a system, but can limit the use of the techniques in existing systems.

7.1 The Introduction

The introduction described the concepts of object-oriented systems, giving a definition of ‘‘object-oriented’’ that is based on the properties of objects, rather than the way in which they are constructed. Consequently this can be used to characterize many systems that are claimed in the literature as supporting an ‘‘object-oriented’’ approach, such as object-oriented operating systems, but are not included in other definitions. Some of the more important, and influential object-oriented systems were then described.

Modern type systems were then briefly described, along with the basis for the types of objects, that is commonly used in strongly-typed object-oriented languages. Problems with a type system based on the construction of objects’ classes were outlined.

Finally the common implementation techniques for objects, using both dynamic and static method dispatch techniques were described.

7.2 Implementing Objects

First-class functions

Chapter 2 first showed how simple objects can be implemented in strongly-typed systems with first-class functions. It then introduced the technique of *type-coercion*, and showed how this can be used to allow the substitution property of objects, specifically when objects and classes are implemented using inheritance. The technique can be extended trivially to implement multiple inheritance.

An extension to normal inheritance, which was called *downwards* type-coercion, was also discussed as a solution to problems caused by objects being created by libraries.

Closures are used to provide the encapsulation and type-compatibility of objects. Unfortunately objects constructed in this way are very inefficient, particularly in terms of space, because a closure is needed for each method an object provides. The data-structures needed for type-coercion can also be rather large, and the type-coercion operations themselves introduce a run-time overhead.

Subtype relationships

Chapter 3 described subtype rules, and showed how these can be used in the construction of objects. Given that subtype rules were proposed specifically to describe objects and inheritance, it is disappointing that objects constructed in this way are still relatively space-inefficient. The principle cause of this inefficiency is the contra-variance of function parameter types, which does not allow objects to be constructed in the ‘‘natural’’ way, but forces the use of non-local storage for instance variables.

Subtype relationships are even less helpful in allowing efficient implementations of multiple inheritance, since the general subtype relationship for structures introduces a substantial overhead to the operations that access members, and does not adequately address problems caused by conflicting inheritances of names.

Existential types

Chapter 4 showed how the existential quantification of types, together with a new subtype rule for existential types, solves the problems caused by function parameter-type contra-variance, and allows objects to be constructed with a similar overhead to those in C++, entirely within a strongly-typed environment. Here both the space requirements of objects, and the time complexity of message dispatch are similar. The use of a combination of these techniques with type-coercion to efficiently implement multiple inheritance is also outlined.

7.3 Active Deallocation

Direct support

Chapter 5 introduced the concept of ‘‘active deallocation’’. It described how this might be used, and how some current systems address these requirements. Two ‘‘direct’’ implementations of destroy methods were then given, for systems using reference-count, and mark-scan garbage collection algorithms. While the former is simple, adding destroy methods to mark-scan garbage collection is extremely complex.

Unfortunately neither garbage collection algorithm is used extensively in modern systems, and there are several other problems. At the system level, many details of the garbage collection algorithm can become visible to the programmer through the behaviour of destroy methods. More fundamentally there are problems caused by the fact that destroy methods are called during garbage collection, asynchronously with the execution of the rest of the system. Consequently, at least for new systems, these algorithms are superseded by those described next.

Primitive support

Chapter 6 shows how much more satisfactory implementations of active deallocation can be achieved by the programmer, given some basic support in the form of *weak pointers*, by the system. *Forwarding objects* and *garbage collection notification* are also convenient for this, but it is seen that these are not essential.

Weak pointers are both simpler to implement, and more widely useful than destroy methods. This strongly suggests that systems should provide them in preference to destroy methods, which could then be implemented by a library. This organisation also allows the programmer to structure objects so that forwarding objects are not required for active deallocation. This makes the design decisions that are required more obvious, and is likely to be more efficient.

7.4 Comments and Further work

The provision of objects, using first-class functions, has been common where object-oriented extensions have been made to LISP. It is described particularly clearly by Abelson and Sussman.[42] The advantages of static type checking made it attractive to attempt this in strongly-typed languages, however it is seen that even the relatively powerful type system of Standard ML is not flexible enough to allow efficient objects to be constructed. The addition of subtype rules to the type system helps a little, but it was only with the further addition of existential quantification over types that techniques with acceptable efficiency are found.

While these techniques cannot solve the problems inherent in multiple inheritance they do allow various implementations. It is particularly useful here that the programmer has more control over the form of multiple inheritance implemented, since there are more choices here than with other primitives in object-oriented programming, and less agreement on which solutions are the ''correct'' ones for different applications.

Unfortunately, few current systems have a type system as expressive as this requires. One that is powerful enough is Ten15,[64] and it was this system that stimulated some of the ideas presented here. It is intended to use these techniques as one starting-point for a project investigating the support for objects in Ten15, when a suitable implementation of Ten15 becomes available.

Ten15 also provides mechanisms that allow a message dispatch mechanism more like that in Smalltalk-80, while retaining static type checking elsewhere. It will be interesting to compare these techniques, to gain a better understanding of the situations in which each are appropriate. There will also be opportunities to investigate further the various techniques for providing multiple inheritance that have been described.

The first part of the thesis concentrated on allowing inheritance — it would be interesting to look at similar ways of providing support for delegation. It might be hoped that this would ultimately provide a way in which a system might allow inheritance and delegation to co-exist, so that each might be used where it is more appropriate. Some work along these lines has already been done by the author, when support for inheritance was added to AML/X,[104] but this work differed in several respects, and further investigation is needed.

Support for other features of systems should also be investigated. Techniques allowing the efficient execution of operations that require simultaneous access to several objects are needed. This is particularly important in the implementation of operations like matrix multiply. C++ allows a method to access the representation of all the instances of its class, in addition to that of self. It also allows a class to define other classes as *friend* classes, which can then access the representation of its instances. The schemes presented here give a much stricter view of encapsulation, which is usually desirable, but is over-restrictive in cases such as these. This tension between encapsulation and efficiency remains an open issue in the design of object-oriented systems, but here again it is hoped that constructing objects using primitives allows the programmer more flexibility.

Similarly when existential quantification is used, information about the exact type of an object is lost forever. This is usually appropriate for the implementation of an object-oriented language, but might be too severe, for example if the objects are persistent. The long lifetimes of persistent objects implies that mechanisms might be needed to allow objects to evolve, so that they can use new behaviours that have been added to their class. The general problem of what happens to persistent objects when their definitions are changed is very much a topic for current and future work.

Recent work by Canning, Cook et al.[105] introduces a more expressive form of bounded universal quantification, which they call *F-bounded quantification*. Like the use of existential types here, F-bounded quantification allows the construction of the recursive types with subtype relationships that are needed for objects, but it does not require the representation to be hidden. The combined use of F-bounded and existential types is still to be explored, but there is every reason to believe that the synthesis would be beneficial.

The active deallocation of objects is an extremely useful technique in some parts of a system, particularly those providing low-level access to the machine. However interactions with the normal execution of a system must be considered very carefully, and here it is particularly useful that special-purpose primitives are not used, since their interactions with other parts of the system (for example, tasking) would probably be complex and inconvenient.

A general theme has been that the provision of appropriate, and general, primitives is preferable to specific support. It provides the programmer with a more flexible system, less committed to a particular view of how things should be done. For the implementation of objects, first-class functions are sufficient active support, however a sophisticated type system is needed before the approach is viable in terms of cost. Similarly, weak pointers are all that are needed to get the effects of the active deallocation of objects, but some other support can be helpful.

Partly the aim has been to discover what it is that must truly be primitive in systems. In retrospect it seems obvious that one answer to this is that the more expressive the type system is, the fewer primitives are needed. Again this is a question of interest to Ten15, since it is intended to support a range of languages by providing sufficiently flexible primitives, rather than providing the union of all the primitives of the supported languages. At the linguistic level, syntactic issues have been largely ignored, however in practice it is important that baroque syntaxes do not make this approach impractical.

A long-term aim would be to collect a library of techniques such as these — unfortunately they are often less than obvious — which programmers could use when they are faced with similar problems. This is similar in spirit to the concept of using ‘‘little languages’’[106] to solve problems, rather than attempting everything in a general-purpose language. However here we are saying that it is the system language itself that should be tailorable to a variety of approaches. Again this concept is more common in the LISP community, but it is hoped that it has been demonstrated that this is now also viable in strongly-typed systems.

References

1. P. Wegner, ‘‘Dimensions of Object-Based Language Design’’, *ACM SIGPLAN Notices*, Orlando, Florida, USA **22**(12), pp. 168-182 (December 1987). Proceedings of OOPSLA ’87.

2. Bjarne Stroustrup, ‘‘What is ‘‘Object-Oriented Programming’’?’’, *Proceedings of ECOOP*, Paris (June 1987).

3. T. Rentsch, ‘‘Object Oriented Programming’’, *ACM SIGPLAN Notices* **17**(9), pp. 51-57 (September 1982).

4. Gordon S. Blair, John J. Gallagher, and Javad Malik, ‘‘Genericity vs Inheritance vs Delegation vs Conformance vs ... (Towards a Unifying Understanding of Objects)’’, Research Report, Department of Computing, University of Lancaster (1988).

5. B. Liskov and S. Zilles, ‘‘Programming with Abstract Data Types’’, *ACM SIGPLAN Notices* **9**(4), pp. 50-59 (April 1974).

6. J. D. Ichbiah et. al., ‘‘Reference Manual for the Ada Programming Language’’, ANSI/MIL-STD-1815A-1983, United States Department of Defense (February 17th, 1983).

7. N. Wirth, ‘‘Modula-2 (second edition)’’, ETH Institut fur Informatik Report 36 (1980).

8. Jacques Cohen, ‘‘Garbage Collection of Linked Data Structures’’, *ACM Computing Surveys* **13**(3), pp. 341-367 (September 1981).

9. B. W. Kernighan and J. R. Mashey, ‘‘The UNIX Programming Environment’’, *Software - Practice and Experience* **9**(1), pp. 1-16 (January 1979).

10. Luca Cardelli and Peter Wegner, ‘‘On Understanding Types, Data Abstraction, and Polymorphism’’, *ACM Computing Surveys* **17**(4), pp. 471-523 (December 1985).

11. William Cook, ‘‘A Denotational Semantics of Inheritance’’, Ph.D. Thesis, CS-89-33, Dept. of Computer Science, Brown University, Providence, Rhode Island (May 1989).

12. Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt, ‘‘An Introduction to Trellis/Owl’’, *ACM SIGPLAN Notices* **21**(11), pp. 9-16 (November 1986). Proceedings of OOPSLA ’86.

13. Craig Schaffert, Topher Cooper, and Carrie Wilpolt, ‘‘Trellis: Object-Based Environment - Language Reference Manual’’, DEC-TR-372, Digital Equipment Corporation, Hudson, MA (25 November, 1985).

14. Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall, Inc, Englewood Clifs, New Jersey (March 1988).

15. G. Curry, L. Baer, D. Lipkie, and B. Lee, ‘‘Traits - An Approach to Multiple-Inheritance Subclassing’’, in *Proceedings of the Conference on Office Automation Systems* (June 1982).

16. Alan Snyder, ‘‘Encapsulation and Inheritance in Object-Oriented Programming Languages’’, *ACM SIGPLAN Notices* **21**(11), pp. 38-46 (November 1986). Proceedings of OOPSLA ’86.

17. Daniel Weinreb and David Moon, ‘‘Objects, Message Passing, and Flavors’’, pp. 245-275 in *Lisp Machine Manual*, Massachusetts Institute of Technology (March 1981).

18. Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel, ‘‘CommonLoops: Merging Lisp and Object-Oriented Programming’’, *ACM SIGPLAN Notices* **21**(11), pp. 17-29 (November 1986). Proceedings of OOPSLA ’86.

19. D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. Keene, G. Kiczales, and D. A. Moon, ‘‘Common Lisp Object System Specification’’, ANSI X3J13 Document 87-002, American National Standards Institute, Washington, DC (March 1987).

20. H. Lieberman, ‘‘Delegation and Inheritance: Two Mechanisms for sharing Knowledge in Object-Oriented Systems’’, pp. 79-89 in *3ème Journées d’Etudes Langages Orientés Objets*, ed. J. Bezivin and P. Cointe, AFCET, Paris (1986).

21. Lynn Andrea Stein, ‘‘Delegation is Inheritance’’, *ACM SIGPLAN Notices*, Orlando, Florida, USA **22**(12), pp. 138-146 (December 1987). Proceedings of OOPSLA ’87.

22. David Ungar and Randall B. Smith, ‘‘Self: The Power of Simplicity’’, *ACM SIGPLAN Notices*, Orlando, Florida, USA **22**(12), pp. 227-242 (December 1987). Proceedings of OOPSLA ’87.

23. G. Bertwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard, *Simula Begin*, Auerbach, Philadelphia, PA (1973).

24. Kristen Nygaard, ‘‘History and Basic Concepts’’, *Object Oriented Programming Society - meeting 3* (20th March 1986).

25. B. B. Kristensen, O. L. Madsen, B. Moller-Pederson, and K. Nygaard, ‘‘Multi-Sequential Execution in the Beta Programming Language’’, *ACM SIGPLAN Notices* **20**(4), pp. 57-70 (April 1985).

26. Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard, ‘‘Abstraction Mechanisms in the Beta Programming Language’’, *Proceedings of the 10th Annual ACM Symposium on the*

Principles of Programming Languages, pp. 285-298 (January 24-26, 1983).

27. Bjarne Stroustrup, ‘‘The C++ Programming Language’’, ISBN 0-201-12078-X, Addison Wesley Publishing Company (March 1986).

28. B. W. Kernighan and D. M. Richie, *The C Programming Language*, Prentice-Hall, Inc, Englewood Clifs, New Jersey (1978).

29. Bjarne Stroustrup, ‘‘Classes: An Abstract Data Type Facility for the C Language’’, *ACM SIGPLAN Notices* **17**(1), pp. 42-52 (January 1982).

30. Stein Krogdahl, ‘‘Multiple inheritance in Simula-like Languages’’, *Bit* **25**, pp. 318-326 (1985).

31. Stein Krogdahl, ‘‘An Efficient implementation of Simula Classes with Multiple Prefixes’’, 83, Institute of Informatics, University of Oslo (June 1984).

32. Daniel H. H. Ingalls, ‘‘The Evolution of the Smalltalk Virtual Machine’’, pp. 9-28 in *Smalltalk-80: Bits of History, Words of Advice*, ed. Glenn Krasner, Addison-Wesley Publishing Company (1983).

33. Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley Publishing Company (1983).

34. L. Peter Deutsch and Allan M. Schiffman, ‘‘Efficient Implementation of the Smalltalk-80 System’’, *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages*, Salt Lake City, Utah, pp. 297-302 (January 1984).

35. R. E. Johnson, J. O. Graver, and L. W. Zurawski, ‘‘TS: An Optimizing Compiler for Smalltalk’’, *ACM SIGPLAN Notices*, San Diego, California, USA **23**(11), pp. 18-26 (November 1988). Proceedings of OOPSLA ’88.

36. Mark J. Stefik, Daniel G. Bobrow, and Kenneth M. Kahn, ‘‘Integrating Access-Oriented Programming into a Multiparadigm Environment’’, *IEEE Software* **3**(1), pp. 10-18 (January 1986).

37. Warren Teitelman and Larry Masinter, ‘‘The Interlisp Programming Environment’’, *IEEE Computer* **14**(4), pp. 25-33 (April 1981).

38. Guy L. Steele, Jr., *Common LISP: The Language*, Digital Press, Bedford, MA (1984).

39. Alan Snyder, ‘‘CommonObjects: An Overview’’, *ACM SIGPLAN Notices* **21**(10), pp. 19-28 (October 1986).

40. Stephen Slade, ‘‘The T Programming Language: A Dialect of LISP’’, ISBN 0-13-881905-X, Prentice-Hall, Inc, Englewood Clifs, New Jersey (1987).

41. Kevin J. Lang and Barak A. Pearlmutter, ‘‘Oaklisp: An Object-Oriented Scheme with First Class Types’’, *ACM SIGPLAN Notices* **21**(11), pp. 30-37 (November 1986). Proceedings of OOPSLA ’86.

42. Harold Abelson, Gerald Jay Sussman, and Julie Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, Mass. (1985).

43. Pierre Cointe, ‘‘Metaclasses are First Class: The ObjVlisp Model’’, *ACM SIGPLAN Notices*, Orlando, Florida, USA **22**(12), pp. 156-167 (December 1987). Proceedings of OOPSLA ’87.

44. Carl Hewitt, ‘‘Viewing Control Structures as Patterns of Passing Messages’’, pp. 433-465 in *Artificial Intelligence: An MIT Perspective, Volume 2*, ed. P. H. Winston and R. H. Brown, MIT Press, Cambridge, Massachusetts (1979).

45. Henry Lieberman, ‘‘Concurrent Object Oriented Programming in Act 1’’, pp. 9-36 in *Object Oriented Concurrent Programming*, ed. Akinori Yonezawa and Mario Tokoro, MIT Press, Cambridge, Massachusetts (1986).

46. Gul Agha and Carl Hewitt, ‘‘Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming’’, pp. 49-74 in *Research Directions in Object-Oriented Programming*, ed. Bruce Shriver and Peter Wegner, MIT Press, Series in Computer Science, Cambridge, MA (1987).

47. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, ‘‘Mach: A New Kernel Foundation for UNIX Development’’, *USENIX Association Conference Proceedings*, Atlanta, Georgia, pp. 93-113 (Summer 1986).

48. R. van Renesse, H. van Staveren, and A. S. Tanenbaum, ‘‘The Performance of the Amoeba Distributed Operating System’’, *Software - Practice and Experience* **19**(3), pp. 223-234 (March 1989).

49. J. M. Bernabeu-Auban, P. W. Hutto, M. Yousef, A. Khalidi, M. Ahamad, W. F. Appelbe, P. Dasgupta, R. J. LeBlanc, and U. Ramachandran, ‘‘Clouds - A Distributed Object-Based Operating Systemm Architecture and Kernel Implementation’’, *European UNIX Systems User Group Autumn Conference*, Cascais, Portugal, pp. 25-37 (October 1988).

50. A. P. Black, E. D. Lazowska, J. D. Noe, and J. Sanislo, ‘‘The Eden Project: A Final Report’’, 86-11-01, University of Washington, Seattle, Dept. of Computer Science (1986).

51. David Maier and Jacob Stein, ‘‘Development and Implementation of an Object-Oriented DBMS’’, pp. 355-392 in *Research Directions in Object-Oriented Programming*, ed. Bruce Shriver and Peter Wegner, MIT Press, Series in Computer Science, Cambridge, MA (1987).

52. M. Stonebraker and L.A. Rowe, ‘‘The Postgres Papers’’, Memorandum No. UCB/ERL M65/85, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720 (June 87).

53. A. Aho, B. Kernigan, and P. Weinberger, *The AWK Programming Language*, Addison-Wesley Publishing Company (1988).

54. Ralph E. Griswold and Madge T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc, Englewood Clifs, New Jersey (1983).

55. J. Donahue and A. Demers, ‘‘Data Types Are Values’’, *Transactions on Programming Languages and Systems* **7**(3), pp. 426-445, ACM (July 1985).

56. J. Reynolds, ‘‘Towards a Theory of Type Structure’’, in *Colloquium sur la Programmation*, Springer-Verlag, New York (1974).

57. Martin Richards and Colin Whitby-Strevens, ‘‘BCPL — The Language and its Compiler’’, ISBN 0-521-28681-6, Cambridge University Press, Cambridge (1980).

58. Hans-J. Boehm, Alan Demers, and James Donahue, ‘‘Letter to the Editor’’, *ACM SIGPLAN Notices* **21**(1), pp. 17-18 (November 5th, 1985).

59. H. Boehm, A. Demers, and J. Donahue, ‘‘An Informal Description of Russell’’, TR 80-430, Department of Computer Science, Cornell University, Ithaca, New York (October 1980).

60. Robin Milner, ‘‘A Proposal for Standard ML’’, pp. 184-197 in *ACM Symposium on LISP and Functional Programming*, Austin, Texas (1984).

61. K. Jenson and N. Wirth, *Pascal User Manual and Report*, Springer-Verlag (1978).

62. L. Damas and R. Milner, ‘‘Principal Type-Schemes for Functional Programs’’, *Proceedings of the 9th Annual ACM Symposium on the Principles of Programming Languages*, Albuquerque, New Mexico, pp. 207-212 (January 1982).

63. Jon Fairbairn, ‘‘Design And Implementation of A Simple Typed Language Based On The Lambda Calculus’’, Technical Report No. 75, University of Cambridge (May 1985).

64. I. F. Currie, J. M. Foster, and P. W. Core, ‘‘Ten15: An Abstract Machine for Portable Environments’’, *Proceedings of the 1st Eupopean Software Engineering Conference*, Palais des Congrès de Strasbourg, France, pp. 149-160 (9-11 September, 1987).

65. Luca Cardelli, ‘‘A Semantics of Multiple Inheritance’’, pp. 51-67 in *Proceedings of the Symposium on the Semantics of Data Types* (1984).

66. Phillip M. Woodward and Susan G. Bond, ‘‘Guide to ALGOL 68: for users of RS Systems’’, ISBN 0-7131-3490-9, Edward Arnold (Publishers) Ltd., London (1983).

67. Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy, ‘‘Object Structure in the Emerald System’’, *ACM SIGPLAN Notices* **21**(11), pp. 78-86 (November 1986). Proceedings of OOPSLA ’86.

68. A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter, ‘‘Distributed and Abstract Types in Emerald’’, *IEEE Transactions on Software Engineering* **13**(1), pp. 65-76 (January 1987).

69. Luca Cardelli, ‘‘Amber’’, pp. 21-47 in *Combinators and Functional Programming Languages: Proceedings of the 13th Spring School of the LITP*, ed. Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, Val d’Ajol, France (May 1985).

70. Bjarne Stroustrup, ‘‘Multiple Inheritance in C++’’, *Procedings of the EUUG Conference*, Helsinki, pp. 189-207 (May 1987).

71. J. Rees, W. Clinger, and et. al., ‘‘Revised Report on the Algorithmic Language Scheme’’, *ACM SIGPLAN Notices* **21**(12), pp. 37-79 (December 1986).

72. Christopher T. Haynes and Daniel P. Friedman, ‘‘Embedding Continuations in Procedural Objects’’, *ACM Transactions on Programming Languages and Systems* **9**(4), pp. 582-598 (October 1987).

73. John Backus, ‘‘1977 ACM Turing Award Lecture: Can Programming be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs’’, *Communications of the ACM* **21**(8), pp. 613-641 (August 1978).

74. D. C. J. Matthews, ‘‘Poly Manual’’, *ACM SIGPLAN Notices* **20**(9), pp. 52-76 (September 1985).

75. W. R. Cook, ‘‘OOPSLA’87 Inheritance BOF’’, *ACM SIGPLAN Notices*, Orlando, Florida, USA **23**(5), pp. 41-42, Addendum to Proceedings of OOPSLA ‘87 (May 1988).

76. L. V. Mancini, ‘‘A Technique for Subclassing and its Implementation Exploiting Polymorphic Procedures’’, *Software - Practice and Experience* **18**(4), pp. 287-300 (April 1988).

77. Bertrand Meyer, ‘‘Safe and Reusable Programming with Eiffel’’, *Proceedings of the 1st European Software Engineering Conference, ESEC ’87*, Palais des Congrès De Strasbourg, France, pp. 237-245 (9-11 September 1987).

78. David A. Moon, ‘‘Object-Oriented Programming with Flavors’’, *ACM SIGPLAN Notices* **21**(11), pp. 1-8 (November 1986). Proceedings of OOPSLA ’86.

79. David Kranz, Richard Kelsey, Jonathan A. Rees, Paul Hudak, James Philbin, and Norman I. Adams, ‘‘Orbit: An Optimizing Compiler for Scheme’’, pp. 219-233 in *Proceedings of the SIGPLAN ’86 Symposium on Compiler Construction*, ACM (June 1986). Published as SIGPLAN Notices 21(7), July 1986

80. Daniel C. Halbert and Patrick D. O'Brien, ''Using Types and Inheritance in Object-Oriented Languages'', DEC-TR-437, Digital Equipment Corporation, Hudson, MA.

81. Wilf R. LaLonde, Dave A. Thomas, and John R. Pugh, ''An Exemplar Based Smalltalk'', *ACM SIGPLAN Notices* **21**(11), pp. 322-330 (November 1986). Proceedings of OOPSLA '86.

82. Phillip Core, ''Overview of Ten15'', 2nd Draft, Royal Signals and Radar Establishment, Malvern, Worcs (January 1989).

83. R. T. House, ''Alternative Scope Rules for Block-Structured Languages'', *BCS Computer Journal* **29**(3), pp. 253-260 (June 1986).

84. J. C. Mitchell and G. D. Plotkin, ''Abstract Types have Existential Type'', *Proceedings of the 12th Annual Symposium on Principles of Programming Languages*, New Orleans, LA, pp. 37-51, ACM (January 14-19, 1985).

85. William Cook, ''A Proposal for Making Eiffel Type-Safe'', *Computer Journal* **32**(4), pp. 305-311 (April, 1989).

86. J. Welsh, J. Elder, and D. Bustard, *Sequential and Concurrent Program Structures*, Prentice-Hall International (1984). Series in Computer Science

87. Lee R. Nackman, Mark A. Lavin, Russell H. Taylor, Walter C. Dietrich, Jr., and David D. Grossman, ''AML/X: A Programming Language for Design and Manufacturing'', *Proceedings of the Fall Joint Computer Conference*, Dallas, Texas, pp. 145-159, IEEE Computer Society (November 1986).

88. Walter C. Dietrich, Jr., Lee R. Nackman, Christine J. Sundaresan, and Franklin Gracer, ''TGMS: An Object-Oriented System for Programming Geometry'', *Software — Practice and Experience* **19**(10), Also IBM Research report RC 13444 (October 1989).

89. M. A. Wesley, T. Lozano-Perez, L. I. Lieberman, M. A. Lavin, and D. D. Grossman, ''A Geometric Modelling System for Automated Mechanical Assembly'', *IBM Journal of Research and Development* **24**(1), pp. 64-74 (1980).

90. H. G. Baker, Jr., ''List Processing in Real Time on a Serial Computer'', *Communications of the ACM* **21**(4), pp. 280-293 (April 1978).

91. Carl Hewitt and Henry Lieberman, ''A Real Time Garbage Collector Based on the Lifetimes of Objects'', *Communications of the ACM* **26**(6), pp. 419-429 (June 1983).

92. David Ungar, ''Generation Scavenging: A Non-disruptive, High-performance Storage Reclamation Algorithm'', *Proceedings of the ACM SIGPLAN/SIGSOFT Symposium on Practical Software Development Environments — SIGPLAN Notices*, Pittsburg, PA **19**(5), pp. 157-167 (April 1984).

93. E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. M. F. Steffens, ‘‘On-the-Fly Garbage Collection: An Exercise in Cooperation’’, *Communications of the ACM* **21**(11), pp. 966-975 (November 1978).

94. David M. Harland and Bruno Beloff, ‘‘OBJEKT: A Persistent Object Store With an Integrated Garbage Collector’’, *ACM SIGPLAN Notices* **22**(4), pp. 70-79 (April 1987).

95. Jonathon A. Rees, Norman I. Adams, and James R. Meehan, *The T Manual*, Department of Computer Science, Yale University, New Haven, Connecticut (10 January 1984).

96. I. F. Currie and J. M. Foster, *The Varieties of Capabilities in Flex*, Royal Signals and Radar Establishment, Malvern (April 1987).

97. P. L. McCullough, ‘‘Transparent Forwarding: First Steps’’, *ACM SIGPLAN Notices*, Orlando, Florida, USA **22**(12), pp. 331-341, Object-Oriented Programming Systems, Languages & Applications Conference ‘87 (December 1987).

98. The Ten15 group, *Private communication*, Royal Signals and Radar Establishment, Malvern (February 1989).

99. R. R. Fenichel and J. C. Yochelson, ‘‘A LISP Garbage Collector for Virtual-Memory Computer Systems’’, *Communications of the ACM* **12**(11), pp. 611-612 (November 1969).

100. Rodney A. Brooks, ‘‘Trading Data Space for Reduced Time and Code Space in Real Time Garbage Collection on Stock Hardware’’, *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pp. 256-262 (6-8 August 1984).

101. A. W. Appel and D. B. MacQueen, ‘‘A Standard ML Compiler’’, *LNCS*, Portland, Oregon **274**, pp. 301-324, Springer-Verlag (September 1987).

102. J. M. Foster, I. F. Currie, and P. W. Edwards, *Flex: A Working Computer with an Architecture Based on Procedure Values*, Royal Signals and Radar Establishment, Malvern (July 1982).

103. B. W. Lampson, ‘‘Hints for Computer System Design’’, *Proceedings of the Ninth ACM Symposium on Operating System Principles*, Bretton Woods, New Hampshire, pp. 33-48 (October 1983).

104. Martin C. Atkins and Lee R. Nackman, ‘‘Inheritance in AML/X’’, Unpublished internal report, IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, New York (October 1987).

105. Peter Canning, William Cook, Walt Hill, John Mitchell, and Walter Olthoff, ‘‘F-Bounded Quantification for Object-Oriented Programming’’, STL-89-5, Software Technology Laboratory, Hewlett-Packard Laboratories, Palo Alto, CA (March 20, 1989).

106. Jon Bentley, ‘‘Programming Pearls: Little Languages’’, *Communications of the ACM* **29**(8), pp. 711-721 (August 1986).