Emerald: An Object-Based Language for Distributed Programming

by

Norman C. Hutchinson

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1987

University of Washington

Abstract

Emerald:
An Object-Based Language for Distributed Programming

by Norman C. Hutchinson

Chairperson of the Supervisory Committee: Professor Henry M. Levy
Department of Computer Science

Distributed systems have become more common, however constructing distributed applications remains a very difficult task. Numerous operating systems and programming languages have been proposed that attempt to simplify the programming of distributed applications. We present a programming language called Emerald that simplifies distributed programming by extending the concepts of object-based languages to the distributed environment.

Emerald supports a single model of computation: the object. Emerald objects include private entities such as integers and Booleans, as well as shared, distributed entities such as compilers, directories, and entire file systems. Emerald objects may move between machines in the system, but object invocation is location independent. The uniform semantic model used for describing all Emerald objects makes the construction of distributed applications in Emerald much simpler than in systems where the differences in implementation between local and remote entities are visible in the language semantics.

Emerald incorporates a type system that deals only with the specification of objects — ignoring differences in implementation. Thus two different implementations of the same abstraction may be freely mixed in an Emerald program.

Emerald has been implemented. The compiler accepts the responsibility of providing an efficient implementation from object definitions, generating multiple implementations tuned to different usage patterns from the same source code. We discuss these implementation considerations and provide performance data to justify our claim that Emerald can be efficiently implemented.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Distributed computing systems are commonplace, and a number of operating systems and programming languages have been proposed to simplify the construction of distributed applications. Distributed applications typically require the sharing of data between remote entities. Existing systems and languages have either prohibited such sharing, or else provided two levels of support, one for private data and a separate one for shared data. This dissertation proposes (1) that supporting a uniform object model appropriate for both private and shared data is a powerful tool for simplifying the construction of distributed applications, and (2) that such an object model can be implemented efficiently.

The thesis will be proven by presenting a programming language, Emerald, designed to support a single object model, and by showing the positive effect that the model has on other aspects of the language design. We also demonstrate that applications are simple to construct in the language, and that the language can be efficiently implemented.

1.1 Background

We define a distributed system as a collection of autonomous computers connected by a local area network. This definition includes such things as office automation systems and computer networks in university computer science (and other) departments, but specifically excludes computers connected by long-haul networks as well as multiprocessors. We, also assume that the processes making up the system are homogeneous — some thoughts on accommodating heterogeneous systems will be made in Section 8.2.

We use the terms machine, node, computer, and processor interchangeably to refer to the computing elements that make up the system.

Distributed systems as we have defined them are characterized by three common traits:

1. The individual processors that make up the system may fail independently.

2. Network communication is expensive relative to communication within a single processor.

3. Network communication is unreliable. The network typically allows messages to be lost or duplicated in transit, but guarantees (to a high probability) that messages will not be undetectably corrupted in transit.

Since the component processors of the system may fail independently, a distributed system has the potential of providing improved availability to its clients. So long as required data is on a machine that is *up*, a client can proceed with his work even though other components of the system are *down*. However, this potential for added availability exacts a price; the programming of applications on a distributed system is more complicated than the programming of applications on a centralized system in several ways.

First, applications are forced to deal with a partially available collection of processors. At any instant only a subset of the processors in the network are up, and this subset changes dynamically as processors crash and recover. In a centralized system, partial availability is impossible since all components of the system fail and recover at the same time.

Second, distributed applications must deal with network communication which has different semantics from local communication. Locally, entities can communicate through shared memory. At the network level there is no shared memory; the network provides as communication primitives only message sending and receiving. Furthermore, messages accepted for network delivery are not guaranteed to be delivered. Messages can be delayed for arbitrary periods of time, they can be lost and never delivered, or they can be delivered multiple times.

1.2 Review of other work

The construction of distributed applications requires support from the language, the operating system, or both. Historically, dealing with inter-computer communication has been the task of operating systems. Programmers of distributed applications typically programmed in sequential languages and made use of operating system routines to send messages, receive messages, or inquire about the status of the processors that make up the system.

Not too long ago, concurrent programs were written in the same manner, with application programmers writing in sequential languages utilizing operating system routines to manage the creation and destruction of processes as well as mutual exclusion and synchronization between these processes. The advent of concurrent programming languages such as Concurrent Pascal [BH79], CSP [Hoa78], and Mesa [MMS79] greatly simplified the construction of concurrent programs by providing mechanisms for achieving mutual exclusion (through monitors or conditional critical regions) and synchronization (through condition variables, semaphores or synchronous message passing). Recent proposals for distributed programming languages have attempted to achieve a similar simplification in the construction of distributed applications. In reviewing previous efforts to address the issues of distributed computing, we will therefore examine both operating system and programming language approaches.

1.2.1 Operating systems

In general, distributed operating systems have attempted to provide mechanisms for distributed computing that are independent of particular programming languages and therefore universally applicable.

Hydra [WCC$^+$74] was an object-based operating system designed for the C.mmp computer at Carnegie-Mellon University. While C.mmp is a multiprocessor rather than a distributed system, the design of Hydra greatly influenced the designs of later distributed systems. Every Hydra entity is an *object* and is addressed by a *capability* that includes a reference to the object as well as a set of *rights* for performing operations on the object. The system directly supports objects and includes primitives to create new objects and new object types and perform operations on objects addressed by capabilities. Procedures, local name spaces (activation records), and processes are implemented as kernel defined object types.

The StarOS operating system [JJD$^+$79] was implemented at Carnegie-Mellon University in the late 70's as the operating system for the CM* multi-microprocessor computer. It extends the ideas of the Hydra design to more efficiently support many small typed objects named by capabilities.

In addition to its support for passive objects, StarOS also supports cooperating, communicating processes organized as *task forces* which are the active entities in the system; processes are not objects. Process communication and synchronization are performed by accessing shared objects. In contrast to Hydra, where invocations are normally synchronous, a StarOS invocation is normally executed in a newly created process in parallel with its invoker; the invoking process may choose if and when it will await a reply from the invoked function.

The Eden system [AH85, Bla85] was constructed at the University of Washington between 1980 and 1985. Eden extends the concepts of Hydra and StarOS to distributed systems and is an integrated, distributed, object-based operating system. Eden objects are named by capabilities and may contain multiple processes. In addition, objects are mobile; they may migrate from machine to machine in the network. Despite this, invocation of an operation on an object is location-independent.

A programming language for constructing Eden objects was also developed; it is discussed in Section 1.2.2.

While DEMOS [BHM77], the operating system for the CRAY-1 computer developed at the Los Alamos Scientific Laboratory, is not distributed, it introduced ideas found in a large number of later distributed systems. It is primarily a message-passing system, supporting *tasks*, or processes, and one-way communication channels known as *links.*

Communication over links is primarily asynchronous, with a non-blocking *send* primitive and a *receive* primitive that can be either blocking, non-blocking, or interrupt-driven. The kernel also supports a *call* primitive that combines the three operations of link creation, message sending, and reply receipt.

The DEMOS idea of supporting links as the primary mechanism for communication was extended to a distributed environment in the design of the DEMOS/MP operating system [PM83]. DEMOS/MP has all the facilities of the original DEMOS design, allowing users to access the distributed computing system in the same manner as the original centralized system. In addition, DEMOS/MP allows for processes to migrate within the network to perform load sharing.

The V system [CZ83] developed by David Cheriton at Stanford University is an extension of the earlier Thoth single machine operating system [CMMS79] to a distributed environment. Both Thoth and V owe their model of computation to DEMOS. V supports processes that communicate primarily through message passing. V supports only synchronous communication of small fixed size (32 byte) messages. The communication primitives are *send* which sends a message to a specified process and blocks until a reply is received, *receive* which blocks until a message can be received, *reply*, which replies to a message overwriting the original contents of the send buffer, and *forward*, which forwards a previously received message to another process. In addition, V allows processes executing in the same *team* to communicate using shared memory in a completely uncontrolled manner.

A number of later operating systems, including RIG, Accent, Mach, and Amoeba, combine aspects of the architecture of both Hydra and DEMOS. Each of them support process and message passing at the lowest layer, on top of which is constructed an object management system.

Rochester's Intelligent Gateway or RIG [LGFR82] was an early attempt to build a general purpose distributed operating system for a network that included more than one kind of machine. There are two ways to look at the model of computation that the system supported. At the highest level of abstraction, the system consists of a number of resources (or objects) that are managed by servers. These resources are accessible to clients via invocation of operations defined by the servers. Resources are protected from direct manipulation by clients, so the system has much of the flavor of object-based systems like Hydra and StarOS. However, the protection mechanism in RIG is different from that used in object-based systems. In RIG the server gives out handles for resources but maintains the resource data privately. The client can gain access to the resource only by making requests of the server.

At the lowest level, a RIG system consists of processes that communicate via messages. The message communication system provides for synchronous or asynchronous message sending and receipt, as well as an emergency message system for handling high priority messages. Messages are addressed by the pair <process number, port number>. Both the process number and port number are local. Process numbers are relative to the machine the process is on, and port numbers are relative to the process that owns the port. Processes on remote machines are addressed by creating a local *alias* to the remote process number. Once such aliases have been created, interprocess communication is both network and machine transparent. That is, a process need not be concerned about the location of the target process nor the kind of machine that it is executing on to communicate with it. However, the creation of local aliases is the responsibility of the programmer; it is not handled automatically by the system. Some processes in the system are viewed as clients and others as servers, but they are indistinguishable at the lower level.

The Accent operating system [RR81] is a successor of the RIG system. Like RIG, the system provides processes and both synchronous and asynchronous communication. In addition, the operating system supports *ports*, which are protected kernel objects to which messages are addressed. Processes are allowed to manipulate ports only by using process-local port identifiers. This extension of the message system allows for more protection, as ports cannot be fabricated easily. In addition, since ports identifiers can be translated by the kernel when sent in messages, the scheme allows ports to be used to name services independent of the process which implements the service.

The latest version of Accent is named Mach [JR86], and has extended the ideas in Accent to a distributed environment composed of multi-processors. Mach has also achieved a considerable simplification of the notions of both process and message passing over Accent.

The Amoeba system [TvR85] presents the user with a model similar to that of RIG, although the underlying architecture is quite different. The kernel of the operating system provides only synchronous inter-process communication. At a higher level of abstraction, the system consists of a number of servers that manage objects named by capabilities; the objects are accessible only through the invocation of operations.

1.2.2 Programming languages

In addition to operating system efforts to simplify the construction of distributed applications, a number of programming languages have been designed with this goal in mind. Like the distributed operating systems just described, these may be broadly categorized by whether they support processes or objects as their primary computation model.

CSP is Hoare's proposal for structuring distributed applications as groups of communicating sequential processes [Hoa78]. It was originally a means for expression of inter-process communication and non-determinism, and not a full programming language. Processes synchronize and communicate using synchronous, non-buffered message passing (termed input/output). The language has been extended a number of times [BRV84, May83]. These extensions vary in the way communication targets are named, whether processes may be dynamically created, and how communication is typed (if at all).

The Programming Language in the Sky (PLITS) project at the University of Rochester [Fel79] is not a single programming language; it is a methodology through which any *body* language can be transformed into a distributed programming language. PLITS processes (also called modules) communicate by sending messages to other processes. In contrast to other distributed programming languages, messages in PLITS are not typed. A message consists of an arbitrary number of name/value pairs.

The languages NIL [SY83] and LYNX [Sco86] extend the previous languages that supported processes and message passing by making the links over which messages are sent first class values in the language. NIL processes communicate by sending and receiving messages through strongly typed ports rather than by directing them at named processes. LYNX incorporates the link idea of the DEMOS operating system into a programming language without losing the flexibility to dynamically create and bind the ends of links to processes. It adds to DEMOS both secure type checking and the ability to create concurrent threads of control within a process to conveniently manage multiple concurrent conversations.

The languages Distributed Processes (DP), Brinch Hansen's language for the construction of distributed real-time applications [BH78], and Ada, The Department of Defense's standard language for embedded systems [Ada83] also support processes (or tasks). Rather than communicating through message passing, communication in these languages is accomplished by the invocation by one process of operations defined in another. Thus processes in these languages serve a dual purpose. Since a process is an independent thread of control, processes serve as the mechanism for defining concurrent execution. Secondly, because they export operations which may be invoked by other processes, they may be viewed as objects. In fact, Brinch Hansen claims that processes in DP unify the

Concurrent Pascal concepts of process, module, and class [BH78, pp. 940].

Other distributed programming languages provide direct support for only objects. Cook's original StarMod [Coo79] is a derivative of Brinch-Hansen's DP that supports "processor modules", in which a number of server processes can execute concurrently. Operations are exported by these processor modules, and may be serviced by any process within the module, or by a process created specifically to service a request.

The Mesa language [MMS79] was augmented with remote procedure call by Bruce Nelson [Nel81]. RPC provides almost transparent remote procedures to a procedure based language.

The Synchronizing Resources (SR) language implemented at the University of Arizona [And82] generalizes and unifies an number of earlier proposals. Distributed applications are constructed out of *resources*, which may contain multiple processes and may export *operations* which these processes implement. SR provides two mechanisms for invoking operations (*call* and *send*), and two for implementing operations (*proc*'s and *in* statements). The various combinations of these mechanisms provide support for procedure call, rendezvous, process creation, and message passing.

The Argus programming language and system [Lis84] is a very ambitious attempt to define a language in which applications with extreme reliability requirements may be constructed. Argus applications consist of *guardians* which are abstractions of physical machines. Intra-guardian communication is through shared data while inter-guardian communication is by value. Argus achieves its high reliability through a transaction scheme which is automatically administered by the system on remote operations.

The Eden distributed operating system is programmed by means of the Eden Programming Language (EPL) [ABLN85, Bla85]. EPL provides support for synchronous, strongly-typed invocation of Eden objects named by capabilities. The EPL run-time system supports multiple concurrent processes inside each object as well as providing support for the packaging and unpackaging of invocation parameters in messages.

1.3 Motivation for Emerald

Existing distributed programming languages and systems have each supported two different computational models. One model is appropriate for constructing distributed entities that may wish to migrate from machine to machine in the network, communicate with entities on other machines, or be remotely referenced. Such entities are passed by reference when passed as arguments, thus facilitating sharing. The other computational model may be used only to construct entities which are private to a single distributed entity.

The existence of these two models is not an accident. Two very different implementation styles are available in a distributed system. For private entities, constrained to be accessed by only one distributed entity, traditional shared memory approaches provide for an efficient implementation. For distributed entities that are accessible from remote machines, a more general (and therefore, more expensive) implementation style is appropriate.

The languages CSP, PLITS, DP, Ada, NIL and LYNX and the operating systems Accent/Mach, Amoeba, RIG, DEMOS/MP, and V all support processes as their distributed entities. Within a process, traditional programming language data types such as arrays and records are supported. StarMod and RPC support distributed modules whose operations may be invoked remotely, as well as *ordinary* data which is private to a module. SR's resources form the distributed entities for that language; within a resource data is accessed by shared memory. In Argus the distributed entities are *guardians*, while CLU objects as defined by clusters form the private entities. Eden objects are Eden's distributed entities, which consist of Concurrent Euclid modules, monitors, records, and arrays. This information is summarized in Table 1.1.

When two different models of computation exist, the programmer of a distributed application must decide which to use for each entity in his application. Since the semantics of the two models are different, once an object is implemented in one style it must be rewritten for use as the other. For example, if we build a tree out of nodes that are Eden objects, and later need a tree for strictly internal use within some other object we must either design and code a different tree or suffer the inefficiencies of the much more general implementation. In constructing a distributed document editor in Argus the difference in semantics between guardians and CLU objects caused a guardian to be used where a cluster may have been more appropriate [GSW86].

Since these two computational models so closely parallel the physical structure of the systems on which they are implemented, one is tempted to believe that the existence of the two models is natural, and therefore desirable. The same could be said of primary memory and secondary storage in the pre-virtual-memory days. The relative access times were so different that it seemed natural to keep the two concepts separate, and force the programmer to explicitly transfer data between the two. In spite of this, virtual memory has proven very successful.

In contrast to these distributed languages, centralized languages such as Smalltalk [GR83], Alphard [WLS76], and CLU [LAB+79], as well as the multi-processor operating systems Hydra and StarOS, have each supported only a single abstraction mechanism. These languages have demonstrated the utility of a single object model, at least in a centralized environment.

The thesis of this dissertation is that (1) a single object model can be defined that is appropriate for both distributed entities and private ones, and (2) it is possible to implement it so as to take advantage of the physical structure of the distributed system.

1.4 Plan of action

We defend this thesis by presenting a new programming language called Emerald that is based on a single object model and has been implemented with efficiency comparable to existing languages. The basis of the Emerald design is the belief that a single object model is appropriate for constructing distributed applications, and in addition can be made performant [BHJL86, BHJ+87].

Chapter 2 provides an overview of the Emerald programming language. This overview is meant to provide the background for the more detailed discussion in the remainder of the dissertation. Motivation and justification for the design decisions is deferred until later chapters, which discuss the impact of the single object model on other features of the language. Chapter 3 discusses the type system of Emerald, which is rather unique, primarily due to the object model and its support for late binding. Chapter 4 discusses the definition and creation of objects in Emerald. Chapter 5 discusses other features of the language including location dependent operations, support for reliability, and support for concurrency.

An important criticism of a single object model is that it is less efficient than supporting multiple models, each tuned to a particular style of implementation. Chapters 6 and 7 address this issue by discussing our first implementation of Emerald. Chapter 6 discusses how the compiler is able to generate multiple implementations for objects from the same source code. These implementations are tuned to particular usage patterns, and allow the cost of an object to be appropriate to the generality required by it. Chapter 7 presents performance data showing that an efficient implementation of Emerald is in fact possible. The thesis concludes with a summary of its contributions and some ideas for further research.

System Name	Distributed Entity	Private Entity
Accent/Mach Ada Amoeba CSP DEMOS/MP DP LYNX NIL PLITS RIG V	Process	data
Mesa RPC StarMod	Module	data
SR	Resource	data
Argus	Guardian	CLU Cluster
Eden/EPL	Eden Object	CE data
Emerald	object	object

Table 1.1: Models of computation

Chapter 2

Overview of Emerald

Emerald attempts to extend the utility of a single object model to distributed systems. The Emerald object is the only abstraction mechanism in the language, and incorporates the notions of data, procedure, and process. All entities in Emerald are objects. This includes small entities, such as Booleans and integers, as well as large entities, such as directories, compilers, and entire file systems. All objects exists so long as a means is available to refer to them. Each Emerald object consists of:

- A *name*, which uniquely identifies the object within the network.
- A *representation*, which, except in the case of a primitive object, consists of references to other objects.
- A set of *operations*, which define the functions and procedures that the object can execute. Some operations are exported and may be invoked by other objects, while others may be private to the object.
- An optional *process*, which is started after the object is initialized, and executes in parallel with invocations of the object's operations. An object without a process is passive and executes only as a result of invocations, while an object with a process has an active existence and executes independently of other objects.

Each object also has several attributes. An object has a *location* that specifies the node on which that object is currently located. Emerald objects may be defined to be *immutable*. This simplifies sharing in a distributed system, since immutable objects can be freely copied. Immutability is a logical assertion on the part of the programmer rather than a physical property; the system does not attempt to check it.

Emerald supports concurrency both between objects and within an object. Within the network many objects can execute concurrently. Within a single object, several operation invocations can be in progress simultaneously, and these can execute in parallel with the object's internal process. To control access to variables shared by different operations, the shared variables and the operations manipulating them can be defined within a *monitor* [Hoa74, BH79]. Processes synchronize through built-in *condition* objects. An object's process executes outside of the monitor, but can invoke monitored operations should it need access to shared state.

Each object has an optional *initially* section — a parameterless operation that executes exactly once when the object is created and is used to initialize the object state. When the initially operation is complete, the object's process is started and invocations can be accepted.

2.1 Invocation

The only mechanism for communication in Emerald is through *invocation*. An Emerald object may invoke some operation defined in another object, passing arguments to the invocation and receiving results. Assuming that *target* is an object reference, the phrase:

target.operationName[*argument1, argument2*]

means execute the operation named *operationName* on the object currently referenced by *target*, passing *argument1* and *argument2* as arguments. Invocations are synchronous; the process performing the invocation is suspended until the operation is completed (or until the run-time system determines that the operation cannot be completed, see Section 5.3).

An alternative explanation is that the process performing the invocation continues into the invoked object and provides the thread of control that executes the code implementing the operation. All arguments and results of invocations are passed by object reference. That is, references are passed enabling the caller and callee to share the argument and result objects. This same parameter passing semantics is called *call by sharing* in CLU.

2.2 Abstract types

Central to Emerald is the concept of *abstract type*. An abstract type defines a collection of *operation signatures*, that is, operation names and the types of their arguments and results. All identifiers in Emerald are typed: the programmer must declare the abstract type of the objects that an identifier may name. An abstract type is represented by an Emerald object that specifies such a list of signatures. For example, if the variable *MyMailbox* is declared as:

var *MyMailbox : AbstractMailbox*

then any object that is assigned to MyMailbox must implement (at least) the operations defined by AbstractMailbox.

We say that the abstract type of the object being assigned must *conform* to the abstract type of the identifier. Conformity is the basis of type checking in Emerald. Informally, a type S conforms to a type T (written $S \circ\!\!> T$) if:

1. S provides at least the operations of T (S may have more operations).

2. For each operation in T, the corresponding operation in S has the same number of arguments and results.

3. The abstract types of the results of S's operations conform to the abstract types of the results of T's operations.

4. The abstract types of the arguments of T's operations conform to the abstract types

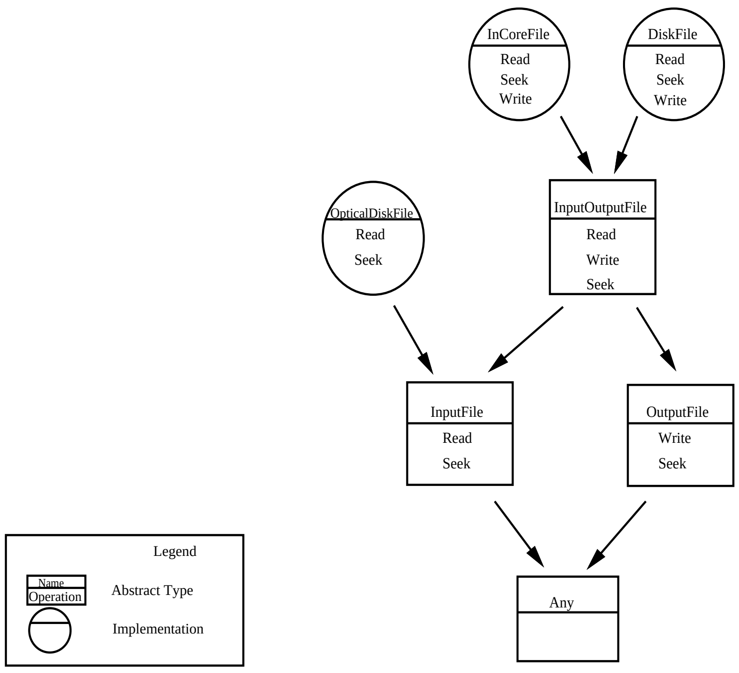

Figure 2.1: Example abstract types and object implementations

of the arguments of S's operations (i.e., arguments must conform in the opposite direction).

Note that conformity is a one-way relationship between abstract types: A $\circ\!>$ B does not imply that B $\circ\!>$ A. In fact, if A $\circ\!>$ B and B $\circ\!>$ A, then A and B are identical types. Abstract types therefore form a partial order, with conformity as the ordering function. This partial order is more fully discussed in Section 3.5.3. Emerald's notion of type conformity is discussed in detail in Chapter 3.

The relationship between abstract types and object implementations is many-to-one in both directions. A single object may conform to many abstract types, and an abstract type may be implemented by many different objects. Figure 2.1 illustrates these relationships. In the figure, A above B means A $\circ\!>$ B.

The object *DiskFile* implements the abstract type *InputOutputFile*, the abstract types *InputFile* and *OutputFile* (which require only a subset of the *InputOutputFile* operations), and also the abstract type **Any** (which requires no operations at all). The abstract type *InputOutputFile* illustrates that an abstract type may have several implementations, perhaps tuned to different usage patterns. Temporary files may be implemented in primary memory (using *InCoreFile* objects) to provide fast access while giving up permanence in the face of crashes. On the other hand, permanent files implemented using *DiskFile* would continue to exist across crashes.

Since Emerald objects may conform to more than one abstract type, it may be appropriate to change one's *view* of a particular object at run-time. This change may either be a *widening*, which corresponds to a move up in the abstract type partial order, or a *narrowing* which corresponds to a move down. Narrowing requires no run-time check of its validity, since any object conforming to some type in the partial order also conforms to all types that it is greater than (with respect to ∘>) . Widening on the other hand requires that the system check that the given object in fact does support the operations required by the new type.

An example of where such view changes are required is in the implementation of a directory system. Suppose we define the abstract type Directory as follows:

```
const Directory == type Directory
    operation Add[name : String, thing : Any]
    operation Lookup[name : String] → [thing : Any]
    operation Delete[name : String]
end Directory
```

Suppose further that we have a variable declared as

```
var f : InputOutputFile
```

that currently names a file object. If we wish to insert this file into a directory *d*, we may execute the invocation:

```
d.Add["myfile", f]
```

Since the second argument to *Add* on directories has type **Any**, we narrow the type of *f*: *InputOutputFile* to **Any**. Now suppose we want to get the same object back out of the directory *d*. We would like to execute the assignment:

f ← *d.Lookup*[*"myfile"*]

However, the type of the result of *Lookup* is **Any**, and **Any** does not conform to *InputOutputFile*, the type of *f*. Therefore, the preceding statement is not type-correct, and is rejected by the compiler. On the other hand, we know that the object that will be returned by executing *Lookup* on *d* with the argument *"myfile"* is in fact an *InputOutputFile*, and so we insert an explicit change of view.

f ← **view** *d.Lookup*[*"myfile"*] **as** *InputOutputFile*

The compiler cannot guarantee that this widening will be legal at run-time; a run-time check is generated at this point.

2.3 Object creation

In most object-based systems, new objects are created by an operation on a *class* (in Smalltalk or Simula terms) or *type object* (in Hydra or StarOS terms). This class object defines the structure and behavior of all of its *instances*. In addition, the class object responds to *new* invocations to make new instances.

In contrast, an Emerald object is created by executing an *object constructor*. An object constructor is an Emerald expression (bracketed by **object** <name> and **end** <name>) that defines the representation, the operations, and the process of an object. For example, suppose the Emerald program in Figure 2.2 is executed; it results in the creation of a single object. If we wished to create more *oneEntryDirectories* we would embed the object constructor of Figure 2.2 in a context in which it could be repeatedly executed, such as the body of a loop or operation. This is illustrated in Figure 2.3. Execution of this example creates the single object specified by the outermost object constructor. That object exports an operation called *Empty*; invoking the *Empty* operation executes the inner

```
const myDirectory : Directory == object oneEntryDirectory
    export Add, Lookup, Delete
    monitor
        var name : String ← nil
        var An : Any ← nil
        operation Add[n : String, o : Any]
            name ← n
            An ← o
        end Add
        function Lookup[n : String] → [o : Any]
            if n = name then
                o ← An
            else
                o ← nil
            end if
        end Lookup
        operation Delete[n : String]
            if n = name then
                name ← nil
                An ← nil
            end if
        end Delete
    end monitor
end oneEntryDirectory
```

Figure 2.2: A oneEntryDirectory object

object constructor, creating a new object that conforms to the abstract type *Directory*. The code generated when compiling an object constructor is called the *concrete type* of the objects created by execution of the constructor and serves to define the structure of these objects as well as provide the implementation for the operations defined on them. Conceptually, each object so created possesses its own copy of the code for *Add*, *Lookup*, and *Delete*. In practice, there will be at most a single shared copy of the concrete type on each machine.

2.4 Supporting multiple implementations

The most important goal of the Emerald design is the support of a uniform object model. The semantics of all objects, whether large or small, local or distributed, must be consis-

```
const myDirectoryCreator == immutable object oneEntryDirectoryCreator
    export Empty
    operation Empty → [result : Directory]
        result ← object oneEntryDirectory
            export Add, Lookup, Delete
            monitor
                var name : String ← nil
                var An : Any ← nil
                operation Store[n : String, o : Any]
                    name ← n
                    An ← o
                end Store
                function Lookup[n : String] → [o : Any]
                    if n = name then
                        o ← An
                    else
                        o ← nil
                    end if
                end Lookup
                operation Delete[n : String]
                    if n = name then
                        name ← nil
                        An ← nil
                    end if
                end Delete
            end monitor
        end oneEntryDirectory
        end Empty
    end oneEntryDirectoryCreator
```

Figure 2.3: A oneEntryDirectory creator

tent. This uniformity should hold both for the programmer who builds objects and types, and for the application that invokes them. On the other hand, for objects to be useful, they must be efficiently implemented.

In Emerald, all objects are coded using the single object definition mechanism we have just illustrated. At compile time, the Emerald compiler chooses among several implementation styles for the object, picking one that is appropriate to the object's use. Three different implementation styles are used; each makes a different tradeoff between representation efficiency, invocation overhead, and generality.

- *Global objects* are those that can be moved within the network and can be invoked by other objects not known at compile time (in other words, references to them can be exported). These objects are heap allocated by the Emerald kernel and are referenced indirectly through a descriptor. An invocation may require a remote procedure call.

- *Local objects* are local to another object (i.e., a reference to them is never exported from that object). They are heap allocated by compiled code. These objects never move independently of their enclosing object, and are referenced with a pointer to their data area. An invocation may be implemented by a local procedure call or by inline code.

- *Direct objects* are local objects except that their data area is allocated directly in the representation of the enclosing object. They are used mainly for built-in types, structures of built-in types, records, and other simple objects whose organization can be deduced at compile time.

Thus, Emerald is similar to the programming languages and operating systems surveyed in Section 1.2 in that there are several different implementation styles with varying performance characteristics. However, unlike these languages, the implementation differences are hidden from the programmer. The compiler chooses the best implementation based on compile time information. In many cases, the compiler can also determine the concrete type of objects and can use this information for further optimizations. If the compiler knows only the abstract type then it must assume the most general object invocation mechanism.

2.5 Distribution

Emerald is designed for the construction of distributed applications. As previously stated, we believe that objects are an excellent way of structuring such programs because they provide the units of processing and distribution. This belief has been confirmed by our experience with distributed applications in Eden [AH84, AH85, ABBW84, Bla85].

When constructing distributed applications for Eden, we noticed that distributed applications fall into two distinct classes. Some applications, such as replicated nameservers and distributed databases, have only come into existence because of distributed environments. The distributed nature of the system is important to the specification of the function that these applications are to perform. To construct such applications, it must be possible to control the locations of the objects that make up the application. For example, it must be possible to ensure that two replicas of an important resource are placed so that a single failure cannot make them both inaccessible. To facilitate the construction of these applications Emerald provides primitives to control the placement and movement of objects.

Other distributed applications are really displaced centralized applications such as mail systems and compilers. Their construction in a distributed environment is merely complicated by distribution. To assist in the construction of these applications, the manipulation and invocation of operations on objects in Emerald, as it was in Eden, is location independent. An object need not concern itself with the location of any other object that it uses.

As a distributed programming language, Emerald is useful for the construction of both classes of distributed application: those that are born to distribution, as well as those that have had distribution thrust upon them.

2.6 Summary

We have briefly described and illustrated the programming language Emerald. Its most notable feature is the single object model which may be used to construct all objects, ranging from local data abstractions including records and arrays to entire distributed applications including multiple concurrent processes such as file systems or compilers.

Emerald can perhaps be most easily understood by enumerating the features that it shares with existing languages, and those that are unique to it. It incorporates the single object model of Smalltalk, the syntax of an Algol-like language and the distribution related features of Eden (including mobile objects) in an efficient, compiled language. Its novel features include support for abstract types, multiple compiler-generated implementations from the same source code for different situations, and object constructors for object definition and creation. With this background, the following three chapters discuss in detail the important Emerald design decisions; the next chapter discusses types.

Chapter 3

Types

This chapter describes the type system of Emerald and, more importantly, discusses the factors that affected its design.

3.1 What are types?

The role of types in programming languages has received a great deal of attention in the literature. The discussion has lately focused on polymorphism, type inference, and whether *Type* (the type of all types) is itself a type. We will discuss these issues later. We wish to first answer a much more fundamental question: what are programming language types?

The development of typed systems can be best understood by examining some untyped ones. There are a number of examples of untyped universes:

- Sets in mathematics
- Bit strings in computer memory
- S-expressions in lisp

Each of these universes is untyped, or, more correctly, each of these universes has only one type, therefore all values in the universe have the same type. Consider, for example, the universe of bit-strings in computer memory. In most computer architectures, memory is not typed. That is, integer values, machine instructions, multi-linked data structures and matrices of real numbers are all represented as bit strings. The interpretation of one bit string as a sequence of instructions for the processor to execute and another bit string as a matrix of real numbers is a matter of convention. It helps us to understand what is going on to classify these bit strings by their intended use; we call some bit strings *programs* and others *data*. We make these distinctions to organize our own universes, and because performing arbitrary operations on bit strings without regard for their types may be meaningless. For example, it is usually meaningless to add floating point data to machine instructions. On the other hand, there is nothing fundamental in the machine that prevents us from using data improperly, for example, executing real matrices as programs or manipulating programs as integers.

Type systems are a natural outgrowth of our informal classification of things. Informally, a type in a programming language encapsulates the notion of a collection of entities with similar attributes and operations. For example, we think of the integers as a type with operations like multiplication, addition, and division.

3.2 The purpose of programming language types

The previous section leads us to the conclusion that types are an outgrowth of our efforts to classify the things that we deal with. That is, types help us to group similar objects together and concentrate on the features that they have in common.

Let us look at the stated purposes of types in programming languages. According to Mark Manasse [Car86], the fundamental problem addressed by a type theory is to ensure that programs have meaning. Donahue and Demers [DD85] claim that the purpose of a programming language type system is to prevent the misinterpretation of values — not to ensure that a meaning exists, but to make that meaning independent of representation discussions. According to Cardelli and Wegner [CW85]:

A major purpose of type systems is to avoid embarrassing questions about representations, and to forbid situations in which these questions might come up. A type may be viewed as a set of clothes (or a suit of armor) that protects an underlying untyped representation from arbitrary or unintended use.

In fact, Donahue and Demers state that a programming language is strongly-typed exactly when it prevents this misinterpretation of values [DD85].

In addition to protection from misinterpretation, type systems serve other roles in programming languages. One thing that we expect from a typed programming language is notification from the system when we have committed a *type error* in programming. Such *type checking* provides early feedback to the programmer that he has committed a programming error.

Let us compare a brief program fragment written in CLU (a typed language) to one written in Smalltalk (an untyped one). We expect a CLU implementation to report that

```
c : char
c := 'a'
    ⋮
c := c + 372
```

is not correct when we attempt to compile the program. On the other hand, a Smalltalk implementation presented with

```
| c |
c ← 'a'
    ⋮
c ← c + 372
```

is unable to report at compile time that the + operator will fail. Such errors in Smalltalk can be detected only at run-time.

Even in those cases where type checking is not completely done at compile time, a type system allows run-time messages to more closely pinpoint the cause of the trouble.

In Smalltalk, a language without declared types, type errors committed by the programmer eventually get caught as "Message not understood" errors when an attempt is made to invoke an unimplemented operation on an object. Usually, the root of the problem is not that the object should have the operation defined for it, but rather the object is not of the expected type at all.

Another benefit claimed for programming language type systems is increased performance. In most cases, compile time type checks permit more efficient storage allocation and the complete elimination of run-time type checking. In addition, in many cases compile time type information allows us to generate more efficient code. Consider the implementation of the + operator in Smalltalk and CLU. In both of these languages, + is overloaded. That is, it can be applied to operands of a number of different types. In Smalltalk, due to the lack of type information in the program text, the implementation of the + operator must determine the types of its arguments at run-time and act accordingly. In CLU, the types of the arguments can be determined statically by the translator, enabling the translator to select the appropriate operator and compile in-line code for the + operation.

In a large number of programming languages, the type system is used to convey information about the implementation of the thing described. Thus the definition of a tree-node type

```
type Node = record
    var data : Integer
    var left, right : ↑Node
end record
```

typically defines an implementation as much as it provides a description of the properties possessed by values of the type.

This close coupling of implementation with description is also apparent in the definition of the compatibility rules between types in these languages. Consider the packed and non-packed variants of records in Pascal.

```
type t1 = packed record
    var a : char
    var b : integer
    var c : char
end record

type t2 = record
    var a : char
    var b : integer
    var c : char
end record
```

These two types are not compatible, even though there is no difference in the operations that can be performed on them. The non-compatibility stems from the fact that the two types are expected to be implemented differently. Even in a language such as CLU in which the user can define his own data types, two clusters that define data types with the same operations but different implementations are not compatible.

A final purpose that types typically serve in programming languages is the creation of new entities. In fact, in most programming languages there is no way to cause the creation of an object without first creating its type. Examples of this are creators in CLU clusters and set creators in Concurrent Euclid.

3.3 What should types do?

We have seen that programming language types serve (at least) six purposes:

1. representation independence

2. early error detection

3. more meaningful error reporting

4. improved performance
5. definition of the implementation of values
6. instance creation

Interestingly, only those purposes having to do with error checking and reporting are at all related to the reason that type systems were invented in the first place: to assist us in classifying the objects that we manipulate. To provide such assistance, types should concern themselves with the attributes of the objects that they represent. Defining the interface to objects is clearly the role of types. For example, when you know that some entity has type *Directory*, you know that it implements operations Add, Lookup, and Delete with particular arguments and results.

Emerald separates the other purposes that types serve in programming languages from their role in classifying objects. The major purpose traditionally served by types, ensuring representation independence, is served in Emerald by objects themselves. In object-based languages, only an object may have access to its own representation. An attempt to manipulate an object in an unintended manner results in an illegal invocation of some unimplemented operation. In Smalltalk, for example, it is impossible to apply a floating-point addition operation to integer values because those integer objects only implement the integer addition operation. The object model ensures representation independence, thereby freeing the type system from this responsibility. The remainder of this chapter discusses the development of the Emerald type system, which addresses the issue of defining object interfaces and explains how Emerald provides more support for classifying objects than traditional type systems. Chapter 4 discusses the other roles typically played by types: defining object implementations and object creation. Chapter 6 discusses the implementation issue of improved performance.

3.4 Requirements for Emerald's type system

To be useful for the construction of distributed, system-level applications, Emerald must provide:

- a single model of objects appropriate for objects at all levels of the system
- distribution
- system-level application support, which involves the addition of newly defined and created entities to existing systems
- an efficient implementation

Several of these requirements placed constraints on the type system of Emerald.

First, the type system can only be concerned with the *abstract* nature of the entities being described. This is true for two reasons.

1. To support the addition of newly defined objects to an executing system, the type system must not distinguish between two objects based on their implementations. To see this more clearly, consider the problem of adding a new kind of file to an existing file system. All the existing programs that manipulated files must be able to manipulate these new files, assuming only that this new implementation of the file type meets the specification of that type. It is clear from our Eden experience that this kind of flexibility in the type system is important for the kind of applications that we want to support. For old code to invoke newly created objects it must be possible to have these new objects implement existing types, i.e., the new object must be able to conform to the old type.

2. We mentioned previously that the Emerald compiler chooses an implementation for each object based on the attributes of the object and the way that it is used. These multiple compiler-generated implementations from the same source code clearly must

all have the same type. Therefore, that type must be concerned only with the interface to the object, not its implementation.

Second, our desire for efficiency requires us to do as much type checking as possible at compile type.

Third, we require the ability to delay type checking until run-time. It must be possible for a reference to an object to be both widened and narrowed in type at run-time. Again, this requirement is a result of our application domain. System-level applications often wish to delay type checking until run-time. An example is the implementation and use of a hierarchic directory system. We wish to be able to put objects into the directory system without regard to their types, and then later retrieve them and ensure that they are of the proper type for later processing. This implies that type information about objects must be available for run-time inspection.

Fourth, the type system must support polymorphism. Typically, programming languages allow operations to be parameterized by data values, but not by types. That is, a procedure that creates a stack in Pascal may be parameterized by an integer representing the maximum size that the stack may grow, but not by the type of the elements that will be pushed onto the stack. Polymorphic languages allow constructs to be parameterized with types. We wish to retain the flexibility of dynamically typed languages such as Smalltalk and EPL (in which capabilities are dynamically typed) within the framework of a statically typed language. These languages have the ability to abstract the qualities of a stack object away from the qualities of the objects that are to be pushed onto it. We need to provide similar expressive power.

To meet this set of requirements, the Emerald type system has the following major features:

- Types define the interface to objects, but provide no implementation information.

- Types are themselves objects. Therefore, types exist at run-time allowing run-time type checking to be performed. In addition, passing types as parameters to support polymorphism is simplified.

- All identifiers are typed statically, and all assignments and operation invocations are type checked at compile time.
- To implement polymorphism, the type system concerns itself not only with the types of types that are passed as parameters, but also with their values.

We will discuss each of these attributes of the type system in the following sections.

3.5 Abstract types

The Emerald type system must allow two objects with differing implementations to have the same type. These two implementations may either be user-defined (a new kind of file added to the file system) or compiler-produced (two representations generated from the same source code).

A number of existing languages including Alphard [WLS76], Modula [Wir77], Euclid [LHL+77], CLU [LSAS77, LAB+79], Gypsy [GCKW79], and Ada [Ada83] claim to support abstract data types. In these languages, however, each object has exactly one type. This tight binding of types to objects is too restrictive for our applications. Our type system must allow the construction of types, instances of which can be used without knowledge of their internal representation.

We define an *abstract type* to be a collection of operation signatures, where an operation signature includes the name of the operation, and the names and types of its arguments and results.

3.5.1 Informal definition of Emerald's type system

In Emerald, all identifiers are typed abstractly, i.e., the programmer declares the *abstract* type of the objects that an identifier may name. Such a declaration captures his knowledge of the set of invocations to which those objects should respond. The only exception to this rule is that the type of constants may be omitted. If omitted, the type is inferred by the compiler.

The notion of type *conformity* is central to Emerald. The legality of an assignment is based on the conformity of the type of the assigned expression and the abstract type declared by the programmer for the identifier. This conformity will always be checked at compile time. Conformity was introduced in Section 2.2. Roughly, a type P conforms to another type Q if P provides at least the operations of Q. (P may also provide additional operations.) Moreover, the types of the results of P's operations must conform to the types of the results of the corresponding operations of Q. Finally, the types of the arguments of the corresponding operations must conform *in the opposite direction*, i.e., the arguments of Q's operations must conform to those of P's operations.

To illustrate the need for the parameter matching rules, consider the following examples. **Any** is the abstract type containing no operations, thus every type conforms to it.

```
type AnyPusher
   operation Push[Any]
end AnyPusher

type IntegerPusher
   operation Push[Integer]
end IntegerPusher
```

These rather useless types define "bottomless pits" into which integers and arbitrary objects can be pushed. Intuitively, one would expect *AnyPusher* to conform to *IntegerPusher*, because an implementation of *AnyPusher* can be used wherever an *IntegerPusher* is required. The rules bear this out; the two types are identical except for the argument types of *Push*, and these conform in the opposite direction, i.e., **Integer** conforms to **Any**. Now consider:

```
type AnyPopper
   operation Pop → [Any]
end AnyPopper
```

```
type IntegerPopper
    operation Pop → [Integer]
end IntegerPopper
```

Here *IntegerPopper* conforms to *AnyPopper*, because the results of *Pop* conform in the *same* direction. Finally, observe that:

```
type AnyStack
    operation Pop → [Any]
    operation Push[Any]
end AnyStack

type IntegerStack
    operation Pop → [Integer]
    operation Push[Integer]
end IntegerStack
```

are incomparable; they do not conform in either direction. The reason for this should be obvious; users of an *IntegerStack* object expect its *Pop* operation to return an **Integer**, so an *AnyStack* clearly won't do. Users of an *AnyStack* expect to apply its *Push* operation to arbitrary objects; the *Push* of *IntegerStack* can be applied only to an **Integer**.

Note that Emerald's notion of type conformity differs from inheritance in Smalltalk. In Smalltalk, a subclass does not necessarily conform to its superclass; for example, it may override some of the operations of the superclass so that they expect different classes of argument. Moreover, one class may conform to another without a subclass relationship existing between them. What a subclass and its superclass *do* have in common is part of their representation and some of their methods. In short, inheritance is a relationship between implementations, while conformity is a relationship between interfaces.

3.5.2 Formal definition of Emerald's type system

The above explanation of conformity in Emerald was not well-founded; the conformity of two types depended on the conformity of the types of the arguments and results of the operations defined by the types. This section will present a formal definition of Emerald's type system including conformity, as well as an algorithm for checking conformity.

Let F, I, T be disjoint sets, F being the set of operation names, I being the set of identifier names, and T being the set of type names. Further, let AbstractType $\in T$, an_j and $rn_k \in I$, and let a_j and $r_k \in T \cup I$. A *signature*, s, is either:

- the distinguished null signature Λ, of undefined arity, or
- a pair <functional, parameters> where:
  - functional(s) is a Boolean value
  - parameters(s) is an expression of the form

    $$<an_1, a_1> \times \cdots \times <an_n, a_n> \rightarrow <rn_1, r_1> \times \cdots \times <rn_m, r_m>$$

    with the following two restrictions which ensure that identifiers used as types are bound by some previous argument in the same signature:

    1. if some $a_j \in I$ then

       $$\exists l \text{ s.t. } l < j \text{ and } an_l = a_j \text{ and } a_l = \text{AbstractType}$$

    2. if some $r_k \in I$ then

       $$\exists l \text{ s.t. } an_l = r_k \text{ and } a_l = \text{AbstractType}$$

    S then has *arity* $<n, m>$.

For the moment we will not be considering the argument and result names an_j and rn_k. They are included in preparation for the discussion on polymorphism in Section 3.8.3.

Let S be the set of all signatures. A *type declaration* of $t \in T$ is a pair <immutable, operations> where:

- immutable(t) is a Boolean value, and
- operations(t) is a total function from F to S.

Obviously, in actually declaring the operations of a type, one need only specify the operations which have non-null signatures. For example, the type IntegerStack of the previous section can defined as:

$$\begin{array}{lcl} \text{immutable(IntegerStack)} & \equiv & \text{false} \\ \text{operations(IntegerStack)}(x) & \equiv & \begin{cases} \text{<false, Integer} \rightarrow\text{>} & \text{if } x = \text{push} \\ \text{<false,} \rightarrow \text{Integer>} & \text{if } x = \text{pop} \\ \quad\Lambda & \text{otherwise} \end{cases} \end{array}$$

Suppose $\mathcal{T}$ is a binary relation on T, i.e., $\mathcal{T} \subset T \times T$. Intuitively, $\mathcal{T}$ is a set of pairs which we hope are true assertions about conformity, i.e., $<t,\ u> \in \mathcal{T} \Rightarrow t$ conforms to u. We haven't defined conformity yet, however, so the intuition can't be formalized. Now suppose that s, s' are elements of S. Then $\mathcal{T}$ induces a relation $\mathcal{S}$ on S as follows. $<s,\ s'> \in \mathcal{S}$ if either $s' = \Lambda$ or all three of the following hold:

1. functional(s') $\Rightarrow$ functional(s)

2. arity(s) = arity(s')

3. writing

$$\begin{aligned} \text{parameters}(s) &= <an_1,\ a_1> \times \cdots \times <an_1,\ a_n> \rightarrow <rn_1,\ r_1> \times \cdots \times <rn_m,\ r_m> \\ \text{parameters}(s') &= <an'_1,\ a'_1> \times \cdots \times <an'_n,\ a'_n> \rightarrow <rn'_1,\ r'_1> \times \cdots \times <rn'_m,\ r'_m> \end{aligned}$$

   we have

$$\begin{aligned} <a'_j,\ a_j> &\in \mathcal{T}, \qquad \text{for } j = 1, 2, \ldots, n, \text{ and} \\ <r_k,\ r'_k> &\in \mathcal{T}, \qquad \text{for } k = 1, 2, \ldots, m. \end{aligned}$$

Informally, corresponding pairs of results must be in $\mathcal{T}$ and corresponding pairs of arguments must be in $\mathcal{T}^{-1}$. To illustrate this, consider the *Pop* operations on the *IntegerPopper* and *AnyPopper* types from the previous section. We have:

$$\begin{aligned} s &= \text{<false,} \rightarrow \textbf{Integer}\text{>} \\ s' &= \text{<false,} \rightarrow \textbf{Any}\text{>} \end{aligned}$$

For $<s,\ s'>$ to be in $\mathcal{S}$:

1. false $\Rightarrow$ false

2. $<0, 1> = <0, 1>$

3. <**Integer**, **Any**> $\in \mathcal{T}$.

All three of these conditions are true (we will see later how to show that <**Integer**, **Any**> $\in \mathcal{T}$), and therefore $<s, s'>$ is in $\mathcal{S}$.

Now, what is to distinguish an arbitrary $\mathcal{T}$ from our desired conformity relation? Exactly the requirements that the immutabilities match and that the corresponding pairs of signatures are in $\mathcal{S}$. Formally, we say that $\mathcal{T}$ is *valid* if, for all type names t and u, and all operation names $f \in F$, $<t, u> \in \mathcal{T}$ implies both the following conditions:

1. immutable(u) $\Rightarrow$ immutable(t)

2. <operations(t)(f), operations(u)(f)> $\in \mathcal{S}$

We may now define conformity. A type t conforms to a type u if there exists some valid relation containing $<t, u>$. We write t conforms to u as $t \circ\!\!> u$.

Lemma:

The union of two valid relations is valid.

Proof:

Follows immediately from the definitions.

Since valid relations are closed under union, we may safely combine separate systems of declarations. If two systems of declarations are separately valid, then their union is also valid.

Now we can define a *decision procedure* which will check whether

$$t \circ\!\!> u$$

is true. Starting with $\mathcal{T} = \{ <t, u> \}$, we will build two relations $\mathcal{T}$ and $\mathcal{S}$ recursively. $\mathcal{T}$ will be a valid relation on T, and $\mathcal{S}$ will be the relation on S induced by $\mathcal{T}$. Whenever we insert $<a, b>$ into $\mathcal{T}$, we must also insert $<$operations$(a)(f)$, operations$(b)(f)>$ into $\mathcal{S}$ for all f such that operations$(a)(f) \neq \Lambda$. This ensures that $\mathcal{T}$ remains valid. Additionally, whenever we insert $<s, s'>$ into $\mathcal{S}$, we insert all the appropriate $<a'_j, a_j>$'s and $<r_k, r'_k>$'s into $\mathcal{T}$ so that $\mathcal{S}$ is indeed the derived relation for $\mathcal{T}$. We *fail* in attempting to insert a pair $<t, u>$ into $\mathcal{T}$ if and only if immutable$(u) \not\Rightarrow$ immutable(t). We *fail* in attempting to insert a pair $<s_1, s_2>$ into $\mathcal{S}$ if and only if the arities of s_1 and s_2 mismatch, or functional(s_2) $\not\Rightarrow$ functional(s_1), or $s_1 = \Lambda$ when $s_2 \neq \Lambda$. If we succeed, we will have constructed a valid relation containing $<t, u>$, thereby proving that $t \circ\!\!> u$. In fact, we will have constructed the smallest relation containing $<t, u>$. On the other hand, we only inserted necessary elements into the relations $\mathcal{T}$ and $\mathcal{S}$, so if the procedure *fails*, then t does not conform to u.

Let us apply this decision procedure to check the conformity of *IntegerStack* and *AnyStack*. To insert <*IntegerStack*, *AnyStack*> into $\mathcal{T}$, immutable(*AnyStack*) must imply immutable(*IntegerStack*) (which they do since both are false), and in addition we must insert into $\mathcal{S}$ the following two pairs of signatures:

<operations(*IntegerStack*)(*Pop*), operations(*AnyStack*)(*Pop*)>

<operations(*IntegerStack*)(*Push*), operations(*AnyStack*)(*Push*)>

Looking at the definition of *IntegerStack* and *AnyStack*, these pairs of signatures are:

<<false, $\rightarrow$ **Integer**>, <false, $\rightarrow$ **Any**>>

<<false, **Integer**$\rightarrow$>, <false, **Any**$\rightarrow$>>

The arities and functional components of these two pairs of signatures correspond, therefore we must only insert the pairs <**Integer**, **Any**> (from Pop) and <**Any**, **Integer**> (from Push) into $\mathcal{T}$. The insertion of <**Integer**, **Any**> causes no difficulties. We attempt to insert into $\mathcal{S}$ all those pairs of operation signatures for which operations(**Any**)$(f) \neq \Lambda$, but operations(**Any**)$(f) = \Lambda$ for all $f \in F$. Note that this gives formal justification to our earlier statement that all types conform to

Any. In attempting to insert the pair <**Any**, **Integer**> into $\mathcal{T}$, we must insert <operations(**Any**)(+), operations(**Integer**)(+)> (among others) into $\mathcal{S}$. We fail in doing this since operations(**Any**)(+) $= \Lambda$ where operations(**Integer**)(+) $\neq \Lambda$. We therefore conclude that, because of the existence of the *Push* operation, *IntegerStack* does not conform to *AnyStack*. This coincides with our intuitive result in Section 3.5.1.

Note that since the union of valid relations is valid, there is no need to start with empty relations $\mathcal{T}$ and $\mathcal{S}$; any valid relation $\mathcal{T}$ on types and its induced relation $\mathcal{S}$ on signatures may be used as a starting point. In actually implementing this procedure, the relations $\mathcal{T}$ and $\mathcal{S}$ may be retained after conformity checking, thus eliminating the need to compute them again.

3.5.3 Types form a lattice

Our definition of conformity implies that T, the set of types is a partial order under $\circ\!>$. This information was graphically depicted in Figure 2.1. The definition of $\circ\!>$ makes it simple to conclude that T has a least element: the non-immutable type with no operations. This type is the bottom of the partial order, and is called **Any** since every type (and therefore every object) conforms to it.

For T to be a lattice, the join ($\sqcup$ or least upper bound) and meet ($\sqcap$ or greatest lower bound) between each pair of elements must exist. Since T is a partial order with a least element ($\perp$ or the type Any), all the meets exist. For the joins to exist, it must be possible to define a type that conforms to any two arbitrary types. As Emerald's type system has been defined so far, it does not allow this construction for two arbitrary types. In particular, if both types define the same operation, but the arity of the signatures is different, it is not possible to define a type that conforms to both. As an example, a type that conforms to both:

```
type T1
   operation o[Any]
end T1
type T2
   operation o[Any, Any]
end T2
```

must define an operation o that takes both one and two arguments, clearly an impossible task. We may make T into a lattice by the addition of a single additional element $\top$. In contrast to $\bot$, which is definable in Emerald as the type Any:

```
type Any
   % no operations
end Any
```

$\top$ is not definable in Emerald, since it must implement every operation with every possible combination of argument and result types. It must therefore be explicitly added to the set of types, and the definition of conformity given above must be modified to state that $<\top, u> \in \mathcal{T}$ for all u in T. $\top$ is the predefined Emerald type **None**, so named because no object (other than the object **nil**, to which all variables are initialized) can implement it.

We may look at the type lattice in terms of information content. A type t conforms to a type u if t provides more information (about the objects that conform to it) than does u. $\bot$ contains no information, thus every type conforms to it. $\top$ contains too much information (in fact, contradictory information) thus no other type can conform to it.

3.5.4 Discussion

The reader may notice that our definition of type conformity relies heavily on the names chosen for operations. There are two disadvantages to this:

1. Two types may not conform when they "really" should if the operation names are not identical. Suppose that someone defines an object that is directory-like except that instead of using the name *Add* for the operation that adds something to a directory she chose the name *Insert*. Our emphasis on the operation names in conformity checking causes this new type to not conform to Directory.

2. Two types may accidentally conform because they have operations with the same names and parameter types, even though the semantics of the operations are very different.

These two problems could be resolved by considering the semantics of the operations in addition to their signatures. We have not yet pursued this idea, but discuss it as an avenue for further research in Section 8.2.

3.6 Types are objects

In most languages, types are not first-class values. It is illegal to pass a type as a parameter or to invoke operations on types. The programming language Russell [DD79, DD85] was an experiment in making types first-class values. Types could be passed as parameters to functions, computed and returned from functions and assigned to variables. Abstract types in Emerald are likewise first-class citizens.

Abstract type objects obey a particular invocation protocol: they export a function (without arguments) called *getSignature* that returns a **Signature** object. **Signature** is a pre-defined abstract type. In other words, an abstract type is an object that conforms to the following abstract type:

```
immutable type AbstractType
   function getSignature → [Signature]
end AbstractType
```

Typically, abstract types are created using *type constructors*. An example of a type constructor is:

```
type Directory
   operation Add[name : String, thing : Any]
   operation Lookup[name : String] → [thing : Any]
   operation Delete[name : String]
end Directory
```

This constructor is executable, and when executed causes the creation of an object that conforms to **AbstractType**. The execution of the *getSignature* operation on the resulting object returns another object: an abstract type with three operations: *Add*, *Lookup*, and *Delete*. Type constructors are, however, not the only mechanism for creating abstract types. This is discussed further in Section 4.2.

3.7 Static typing

To perform static type checking the compiler must be able to determine the type of every identifier and the result type of every expression. On the other hand, arbitrary objects can be abstract types as discussed above. Syntactically, therefore, types are arbitrary expressions. For example, a variable declaration (without initialization) has the form:

'var' <identifier> ':' <expression>

Since assigning a type to the identifier declared in this way requires knowledge of the value of the expression, the compiler must evaluate all expressions appearing in type positions. Expressions that may be evaluated by the compiler are called *manifest*. Type constructors are manifest if all expressions appearing in type positions within them are manifest. Invocations of operations on objects are manifest only if all four of the following conditions are true:

1. the target is immutable
2. the operation is a function
3. all the arguments are manifest
4. the body of the operation is sufficiently simple

The first three conditions guarantee that the answer is independent of when the invocation is performed. This allows us to execute it early — at compile time. The final condition frees the compiler from performing arbitrary computations at compile time to evaluate manifest expressions. Currently, because the compiler is not part of the Emerald environment, only operation bodies that consist of the return of a manifest value are considered manifest.

While all assignments and invocations are type checked statically, Emerald incorporates a mechanism for performing run-time type checks. This mechanism is the *view* expression, which changes the abstract type through which an object is viewed. It has the form:

'view' <expression> 'as' <typeExpression>

The abstract type of the expression is the type given by typeExpression (which must be manifest). A run-time check is generated by the compiler if it cannot determine statically that the expression is guaranteed to conform to the type. The view expression is the only method for widening the type of a reference, and only where view expressions are used are run-time type checks performed. View expressions that narrow the type of a reference are redundant because conformity implies implicit narrowing. A redundant view expression generates no run-time check.

3.8 Polymorphism

In attempting to define an Emerald type system that incorporates polymorphism, we went through two stages.

3.8.1 A first try

Our first attempt at incorporating polymorphism into Emerald naively assumed that conformity and the fact that types are objects would be enough. That is, we expected that we could define a polymorphic stack as in Figure 3.1. The *where* clause provides a convenient place to introduce new constants; *stackType* is declared as a constant with the indicated value (an abstract type). We may then use *Stack* as:

const *stackOfInteger* == *Stack.of*[**Integer**]

For this example, everything is fine. **Integer** conforms to **AbstractType**, therefore

Stack.of[**Integer**]

```
const Stack == immutable object aStackCreator
    export of
    function of[eType : AbstractType] → [result : stackType]
        where
            stackType == type stackType
                operation Push[eType]
                operation Pop → [eType]
                function Top → [eType]
                function Empty → [Boolean]
            end stackType
        end where
        result ← object aStack
            % representation declarations
            operation Push[anElement : eType]
                ⋮
            end Push
            ⋮
        end aStack
    end of
end aStackCreator
```

Figure 3.1: A polymorphic stack

is type correct. Confidently, we then proceed to define a more interesting polymorphic object: a sorted collection. This object is supposed to maintain a list of objects so that they may be traversed in increasing order. Figure 3.2 contains our first attempt. The

```
const Sortable == immutable type Sortable
    function <[Sortable] → [Boolean]
end Sortable
const SortedCollection == immutable object aSortedCollectionCreator
    export of
    function of[eType : AbstractType] → [result : collectionType]
        where
            collectionType == type collectionType
                operation Add[eType]
                operation getElement[Integer] → [eType]
                function Size → [Integer]
            end collectionType
            eType ◇> Sortable
        end where
        result ← object aCollection
            % representation declarations
            operation Add[anElement : eType]
                ⋮
            end Add
            ⋮
        end aCollection
    end of
end aStackCreator
```

Figure 3.2: A polymorphic sorted collection

expression:

eType ◇> *Sortable*

in the where clause indicates that we wish the actual argument to the *of* operation to be an abstract type that conforms to *Sortable*. That is, it must be immutable and have a < function that orders its values. We use this object as:

const *IntegerCollection* == *SortedCollection.of*[**Integer**]

In type checking this expression, we check whether the type of **Integer** conforms to **AbstractType**, which it does since **Integer** is an abstract type (exports the required *getSignature* function). Then, because of the where clause, we need to check the *value* of **Integer** against the value of *Sortable.* Unfortunately, **Integer** does not conform to *Sortable.* For **Integer** to conform to *Sortable*, the arguments of the < operation must conform in the opposite direction; i.e., *Sortable* must conform to **Integer**. Since **Integer** has (among others) the operation + which *Sortable* does not have, *Sortable* does not conform to **Integer**, and therefore **Integer** does not conform to *Sortable.*

We therefore conclude that our definition of conforms must be incorrect, since integers should be perfect candidates for insertion into sorted collections. After a number of unsuccessful attempts at defining conformity so that **Integer** will conform to *Sortable*, we next try to prove that it cannot be done. This turns out to be very simple. If **Integer** conformed to *Sortable*, then **Character** should also conform to *Sortable*, since the < operation on characters has the same signature as that on integers (except for the changed type name). We may then write the following program fragment:

```
var s1, s2 : Sortable
s1 ← 1000000
s2 ← 'a'
assert s1 < s2
```

Assuming that both **Integer** and **Character** conform to *Sortable*, the type system has no choice but to conclude that this program fragment is type correct. **Integer** conforms to *Sortable* so the assignment to *s1* is type correct (through an implicit narrowing). A similar argument implies that the assignment to *s2* is type correct. The comparison in the assert statement is of two *Sortables*, and so is type correct. Unfortunately, that comparison makes no sense. There are two < operations that we could attempt to use: the one on integers or the one on characters. Neither of these however is able to compare characters to integers. We therefore need to make this program illegal, so we conclude that it is not possible to allow **Integer** to conform to *Sortable.*

3.8.2 A second try

After our first attempt to support polymorphism with just conformity and types as objects, we concluded that conformity by itself was not enough. What we needed was a way to make **Integer** conform to *Sortable*, but only sometimes. Such a mechanism exists in the form of *type variables* [CW85, Car86]. A type variable is very much like a type, except that the rules for conformity checking are modified slightly. In particular, when attempting to conform a type t to a type variable T one first identifies the local names for the types, and then performs the normal conformity checking. With this definition of conformity between types and type variables, and defining constrained formal parameters to be type variables rather than type constants, we may implement polymorphism.

Looking again at the definition of *SortedCollection* in Figure 3.2, no syntactic changes are required to make it correct. The constraint

eType *>* *Sortable*

in the where clause causes *eType* to become a type variable. When we execute

SortableCollection.of[**Integer**]

we must check that the type constant **Integer** conforms to the type variable *eType.* First we make their local names the same by substituting **Integer** for *Sortable* throughout the definition of *Sortable.* We are then checking whether **Integer**:

```
const Integer == immutable type Integer
   function +[Integer] → [Integer]
   ⋮
   function <[Integer] → [Boolean]
end Integer
```

conforms to the modified definition of *eType*:

```
const eType == immutable type Integer
   function <[Integer] → [Boolean]
end Integer
```

which it clearly does.

3.8.3 Formal definition of polymorphism

To include polymorphism in our formal definition of Emerald's type system, we need to make the following additions. The basic definitions and conformity checking algorithm remain the same. Previously, all assignments were checked using the conformity algorithm including the implicit assignments of arguments to formal parameters in operation invocations. To handle polymorphism, we need a new rule for type checking operation invocations.

First we need to introduce some syntax to handle the substitution of names necessary to deal with type variables. Let t, u_i, and v_i be type names. We define $t[u_1/v_1, \ldots, u_n/v_n]$ to be the type formed by substituting in t the name u_i for each free occurrence of the name v_i. For example, with the definitions of **Integer** and *Sortable* as previously given,

$$\mathit{Sortable}[\mathbf{Integer}/\mathit{Sortable}]$$

is the type:

```
immutable type Integer
   function <[Integer] → [Boolean]
end Integer
```

We now present the type checking rule for operation invocation. An operation invocation has the form:

$$e.o[e_1, \ldots, e_n]$$

Let the type of e be t, and the types of e_i be t_i for $i = 1, 2, \ldots, n$. This invocation is type correct if and only if:

1. operations$(t)(o) \neq \Lambda$, and

2. writing operations$(t)(o)$ as:

$$<\!an_1,\, a_1\!> \times \cdots \times <\!an_1,\, a_n\!> \rightarrow <\!rn_1,\, r_1\!> \times \cdots \times <\!rn_m,\, r_m\!>$$

   then for all $i = 1, \ldots, n$:

(a) $t_i \circ\!\!> a_i$, and

(b) if $a_i =$ AbstractType then $e_i \circ\!\!> an_i[e_i/an_i]$.

3.9 Comparison

We can compare Emerald's type system with that in a number of existing and proposed languages. The languages Russell [DD79] and the typed λ-calculus of Cardelli [Car86] both treat types as first class citizens as we do. First-class types allow the simple expression of polymorphism using the parameterization scheme already in the languages. In addition to first-class types, Cardelli's language defines *Type* (the type of all types) as a type itself, just as our notion of **AbstractType** (the type of all types is also a type). However, because these languages are value-based rather than object-based their type systems have the additional burden of providing the representation independence that we discussed in Section 3.2. Therefore the type of an identifier determines not only the abstraction, but also the implementation, of the values that may be assigned to it.

Our type system is perhaps closest to that of Owl [SCW85]. The Owl type system also concentrates on the specification of objects rather than their implementation, and its definition of subclass compatibility is very similar to Emerald's notion of conformity. However, types in Owl are not objects, and Type is not a type. This forced the designers to use a second parameterization mechanism for the creation of polymorphic types.

3.10 Summary

The purpose of Emerald's type system is to assist the programmer in classifying the objects used in an application. Other purposes traditionally served by a programming language type system are addressed by other features of Emerald. In addition, Emerald's type system is constrained by the intended application domain of the language in three areas:

1. All type checking, except where explicitly requested by the user is to be done at compile time.

2. It must be possible to delay type checking until run-time by explicit programmer request.

3. Polymorphism is supported. Types may be passed to operations, returned as the results of operations, and manipulated in arbitrary ways at run-time subject only to the constraint that all expressions in type positions must be manifest.

Since Emerald types are concerned only with the specification and not the implementation of objects, Emerald's type system better supports the programmer's classification of his objects; Objects that may serve the same purpose may have the same type independent of their implementation. Emerald's type system serves only to help the programmer classify the objects used in an application, and detect errors of incorrect object usage. Other purposes traditionally served by programming language types — object definition and creation — are discussed in the next chapter.

Chapter 4

Objects

Emerald is designed around a single uniform model of object. This chapter discusses that model, and the rather unique mechanism used in Emerald for object definition and creation.

4.1 Object definition and creation

In Chapter 3 we discussed one of the roles that types serve in existing programming languages: the specification of the interface to objects. We emphasized that the Emerald type system is not concerned with the implementation of the objects whose interfaces it describes. We now discuss the Emerald mechanism for defining and creating objects.

4.1.1 A bit of history

Object-based programming languages and systems have traditionally been based on the concept of a *class* or *type object*. In Simula, each object is an instance of a class. To create an object, the programmer first defines the class, which is a template for object construction, and then uses a primitive language construct, the *new* expression, to create an instance of the template. This same idea is found in Smalltalk. The behavior of individual objects is defined by their class; changes to the state of the class affect the behavior of the instance. Similarly, CLU objects are created by invoking creation operations on clusters, which define the behavior of all their instances. On the operating system side, Hydra, StarOS, and Eden all require the creation of a type object which defines the behavior of its instances before any instances can be created.

Class based mechanisms for object definition are based on the idea of collecting the common features of objects and defining them in one place. There are two places where object attributes can be defined: in the class, where they are shared among all the instances, or in the instances where they are private. To take a concrete example, consider defining geometric points in 2-space. All points require operations to move themselves and calculate their distances from other points. In addition, every point performs these operations in the same manner. Definitions of these operations can be reasonably placed in the class. On the other hand, the location of each point may be different, so that information should be contained in the instances.

This class-based approach to the definition of objects has become so prevalent that often it seems to be the only mechanism available. However, this is not the case.

In Sketchpad [Sut63] objects are created by incrementally defining their components and attributes. Alternately, objects may be created by copying existing objects. These copied objects may then be specialized as necessary. The SW-2 system [LH85] also defines and creates objects without reference to classes, although they do use the term *class* in describing how operations on objects may be shared (or inherited). Rather than concentrating on classes, these languages concentrate on the objects themselves.

In defining a distributed programming language, we had a number of concerns about using classes to define and cause the creation of objects:

1. The operations on classes that create new objects typically have access to the "class variables". To implement this in a distributed environment implied that either object creation could only be done on the machine where the class object resided, or that the class object must be replicated. Neither alternative appealed to us.

2. The behavior of an object is determined not by the behavior of its class, but rather by the data of its class; this data is subject to alteration. For example, in Smalltalk,

the + operation on small integers is typically defined to be the + operation defined by the hardware. However, this is not guaranteed. It is possible (in fact, easy) to redefine the behavior of the + operation on all the small integers in the system by modifying the data in the Integer class object. This possibility has three major drawbacks in a distributed system.

First, it is not possible to discover anything about the way that objects are used by static analysis. This is because the operations of the object itself, and the objects that use it, may be modified dynamically. This implies that all binding of operation names to code must be done dynamically, or at least that any static binding must be able to be re-bound when the class data changes. This has serious performance implications, particularly in a distributed environment.

Second, as a distributed programming language, Emerald is intended to be used by multiple users simultaneously. The security aspects of any of these users redefining addition on all the integers in the system, including those of other users, are frightening.

Finally, Emerald is intended for the construction of distributed applications, not for their rapid-prototyping. The ability to quickly redefine the behavior of existing objects is not so important in this environment. In fact, since the construction of distributed applications is more difficult than sequential ones, the language should provide help to increase confidence that a given solution is correct. Once confidence in the correctness of a particular object has been obtained, the language should not allow modifications to it that may introduce errors.

4.1.2 Object constructors

For the reasons outlined in the previous section, the definition and creation of Emerald objects is not based on the class notion. Rather, the definition and creation of an object in Emerald is done with the *object constructor* introduced in Chapter 2. An object constructor defines the representation and operations of a single object as well as the active behavior of the object. When executed, an object constructor causes the creation of a single object.

What do we lose?

Object constructors lose some of the flexibility that class-based object creation provides. In particular, once an object has been created it is not possible to modify its behavior. This is not simply a restriction that we have placed on object construction, but is inherent in the use of object constructors, or more correctly, in the abandonment of classes. Since objects do not rely on any other "class" object for the definition of their behaviour, there is no way to discuss the notion of changing the "class", thereby affecting the behaviour of the "instances". As argued above, this flexibility is not so important — in fact, not even desirable — in a distributed environment.

What do we gain?

Object constructors have a number of advantages over classes. First, object constructors are cleaner. Figure 4.1 demonstrates the relationships between a directory object, and its classes and metaclasses in Smalltalk. To create a single directory, its class must first be created and initialized appropriately. Since Directory Class is also an object, it must also have a class. This is Directory Metaclass, which is an instance of Metaclass itself. But Metaclass is also an object, and therefore must have a class. At this point the Smalltalk hierarchy loops; Metaclass is its own class. In addition, the Directory Class object is the only instance of its class. Directory Metaclass exists solely to contain the definition of Directory Class. In contrast, Figure 4.2 shows what is necessary when we use object constructors to define a directory creator and one "instance". This simplification results from the ease in which 1-of objects may be defined in Emerald. The directory creator object requires no class object for its existence. In fact, there isn't even a class/instance relationship between the directory creator and the created directory object.

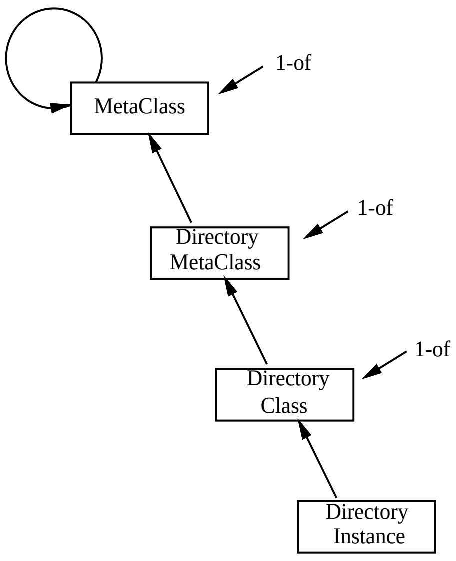

Figure 4.1: Smalltalk instance/class/metaclass structure

Second, object constructors may be nested. Traditional class/instance structures may be simulated by using a two-level nesting of object constructors. Section 2.3 illustrates the definition of a directory creator object that creates new directories in response to invocation of its *Empty* operation.

Object constructors are not limited to two-level nesting; they may be nested to as many levels as the programmer requires. In conjunction with the ability to pass abstract types as parameters, this leads to a uniform syntax for the construction of polymorphic object creators. For example, consider the built-in object **Array**. **Array** exports an *of* operation that expects an abstract type argument, as in:

Array.*of*[**Integer**]

The result of this invocation is an object that exports an operation *Create* of zero arguments. When *Create* is invoked, as in

Array.*of*[**Integer**].*Create*

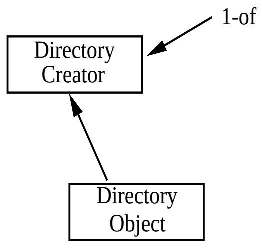

Figure 4.2: Emerald object/creator structure

the result is an array object, i.e., an object that exports operations like *setElement*, *getElement*, *lowerbound*, and *upperbound*.

In a similar way, one could extend the *oneEntryDirectoryCreator* of Figure 2.3 on page 21 to define a typed *oneEntryDirectory* creator creator that is parameterized by the type of the directory entry as shown in Figure 4.3.

As a final advantage, once an object is created, it may not have its behavior modified. This guarantees the integrity of individual objects. Once confidence has been obtained in an object definition and objects have been created, it is not possible to modify these objects. In addition, it is possible to analyze object constructors statically to perform optimizations based on their attributes and the manner in which they manipulate component objects. This important performance optimization is discussed in Chapter 6.

4.2 Objects as types

In Section 3.6, we introduced type constructors as one method for constructing abstract types. Abstract types are not limited to objects created using type constructors. Abstract types include any objects conforming to the following type:

```
immutable type AbstractType
   function getSignature → [Signature]
end AbstractType
```

```
const myTypedDirectoryCreatorCreator == immutable object typedDCreatorCreator
   export of
   function of[ElementType : AbstractType] → [result : DirectoryCreatorType]
         where
            TypedDirectory == type TypedDirectory
               operation Add[String, ElementType]
               operation Lookup[String] → [ElementType]
               function Delete[String]
            end TypedDirectory
            DirectoryCreatorType == type T
               operation Empty → [result : TypedDirectory]
            end T
         end where
      result ← object typedDirectoryCreator
         export empty
         operation Empty → [result : TypedDirectory]
            result ← object oneEntryDirectory
                           .
                           .
                           .
            end oneEntryDirectory
         end Empty
      end typedDirectoryCreator
   end of
end typedDCreatorCreator
```

Figure 4.3: A typed directory creator creator

For example, if we add the following function definition to Figure 2.3,

```
function getSignature → [result : Signature]
   result ← Directory
end getSignature
```

we may use *oneEntryDirectoryCreator* as an abstract type. We may now write

```
var aDirectory: myDirectoryCreator
aDirectory ← myDirectoryCreator.Empty
```

rather than

```
var aDirectory : Directory
aDirectory ← myDirectoryCreator.Empty
```

Given the dual role of *myDirectoryCreator*, we see that it may have been appropriate to give it a less descriptive name.

A better example may be seen by examining how we may augment the definition of our *typedDirectoryCreatorCreator* in Figure 4.3 to take advantage of the ability to use arbitrary objects as abstract types. Suppose we would like to create a directory into which we may insert only *MailBoxes*. We may do so by declaring:

```
const mailBoxDirectory == myTypedDirectoryCreatorCreator.of[MailBox]
```

Since *mailBoxDirectory* is a constant we may allow the compiler to infer its type, thus saving us from doing it. But now suppose we actually require a variable that should reference *mailBoxDirectories*. Emerald requires that the type of variables be explicitly provided, therefore we are forced to declare the type:

```
const MailBoxDirectoryType == type MailBoxDirectoryType
    operation Add[String, MailBox]
    operation Lookup[String] → [MailBox]
    function Delete[String]
end MailBoxDirectoryType
```

This allows us to declare our variable and use it in the following manner:

```
var mailBoxDirectory : MailBoxDirectoryType
⋮
mailBoxDirectory ← myTypedDirectoryCreatorCreator.of[MailBox].Empty
```

MailBoxDirectory is exactly the type we would obtain by substituting *MailBox* for *ElementType* in the declaration of *TypedDirectory* in the where clause of the operation *of*. In fact, in determining the type of the result of

```
myTypedDirectoryCreatorCreator.of[MailBox].Empty
```

the type system must perform exactly that substitution. We may take advantage of the work that the type system performs by adding a *getSignature* operation to the object returned by the *of* invocation on *myTypedDirectoryCreatorCreator*. The result of doing this (and changing the name as suggested above) is shown in Figure 4.4. We may now declare and initialize *mailBoxDirectory* as follows:

```
const TypedDirectory == immutable object TypedDirectory
  export of
  function of[ElementType : AbstractType] → [result : DirectoryCreatorType]
      where
        TypedDirectory == type TypedDirectory
          operation Add[String, ElementType]
          operation Lookup[String] → [ElementType]
          function Delete[String]
        end TypedDirectory
        DirectoryCreatorType == immutable type T
          function getSignature → [Signature]
          operation Empty → [result : TypedDirectory]
        end T
      end where
    result ← object typedDirectoryCreator
      export getSignature, Empty
      function getSignature → [theType : Signature]
        theType ← DirectoryCreatorType
      end getSignature
      operation Empty → [result : TypedDirectory]
        result ← object oneEntryDirectory
                  .
                  .
                  .
        end oneEntryDirectory
      end Empty
    end typedDirectoryCreator
  end of
end TypedDirectory
```

Figure 4.4: TypedDirectory with *getSignature*

```
var mailBoxDirectory : TypedDirectory.of[MailBox]
⋮
mailBoxDirectory ← TypedDirectory.of[MailBox].Empty
```

We previously stated that the abstract type of every identifier in Emerald must be manifest. The expression *TypedDirectory.of*[*MailBox*] is manifest since the target (*TypedDirectory*) is immutable, the operation (*of*) is a function, the argument (*MailBox*) is immutable, and the body of the operation is sufficiently simple. The expression can therefore be evaluated by the compiler.

Similarly, the primitive object **Array** has been defined in such a manner that the object returned by the *of* operation may be used as an abstract type. This allows us to write

```
var a: Array.of[Integer]
a ← Array.of[Integer].create
```

Chapter 5

Other Features of Emerald

There are several other areas of the Emerald design in which either our single object model or our distributed target environment or both have affected our design choices. This chapter examines these features of Emerald in an effort to more fully understand the impact that supporting a single object model in a distributed environment has on other aspects of the language design.

5.1 Location dependent operations

Emerald is designed for the construction of distributed applications. As previously stated, we believe that objects are an excellent way of structuring such programs because they provide the units of processing and distribution. This belief has been confirmed by our experience with distributed applications in both Eden [AH84, AH85, ABBW84, Bla85] and Emerald.

The tendency of many distributed systems is to hide distribution from the programmer. For example, in Xerox RPC [BN84], remote procedure calls were added to Cedar Mesa. In so far as it was possible, remote procedure calls were designed to be semantically identical to local procedure calls. This is obviously a desirable property and is what makes RPC so attractive; programs can be written and debugged on a single node using local procedures and then easily distributed.

Emerald supports the same notion with object invocation. All objects are manipulated through invocation, and all invocations are location independent; it is the responsibility of the run-time system to locate and transfer control to the target object. Remote invocation achieves the same benefits as remote procedure call.

Some distributed systems, recognizing the utility of location-independent operation invocation (or message passing) have proposed that all location dependent decisions should be made by the system [All83]. They attempt to present the programmer with the model of a centralized system, hiding from him the fact that it is implemented on distributed hardware. In fact, Tanenbaum and van Renesse [TvR85] state:

> A *distributed* operating system is one that looks to its users like an ordinary centralized operating system but runs on multiple, independent central processing units (CPUs). The key concept here is *transparency.* In other words, the use of multiple processors should be invisible (transparent) to the user.

While it is crucial that invocation be location independent, or that distribution be transparent with respect to invocation, it is not necessary that an object's location be invisible. Many applications may choose to ignore distribution, but others may wish to benefit from location dependence. For example, a replication manager may wish to distribute object replicas on different nodes, or two applications may wish to be co-located during periods of high communication. Applications that are concerned with distribution may wish to discover and modify objects' locations, but they still benefit from location-independent invocation.

We can gain some insight into the proper role of location-independence by looking again at virtual memory. A virtual memory system provides referential transparency — access to memory words is independent of their current placement in the physical memory hierarchy. This is a desirable property. Some virtual memory systems completely hide the physical memory hierarchy from the user, while others have recognized that there are times when a programmer can exercise control over it to his advantage. An example is the provision for virtual memory control through the *vadvise* and *mmap* primitives in UNIX^{TM}. In fact Mach [Ras86], the latest in a series of virtual memory operating systems that includes RIG and Accent, has allowed the user more control over virtual memory management that any of its predecessors. Of course, referential transparency is not sacrificed, and system defaults mean that a programmer not desiring to deal with virtual memory management is not required to.

For these reasons, the Emerald language includes a small number of location primitives. Basic to these primitives are node objects, which are the logical location entities in the system, and are abstractions of physical machines. An object can:

- *Locate* an object, i.e., determine on what node it resides.
- *Move* an object to another location.
- *Fix* an object at a particular node, which may involve moving it there first.
- *Unfix* an object, i.e., make it movable following a *fix*.
- *Refix* an object, i.e., atomically unfix and then move and fix an object in a new place.

In all cases, location is specified through a reference to a target object; the location thus described is the node on which the target currently exists.

5.2 Call by move

The choice of parameter passing semantics is crucial to both remote procedure call and object invocation. In an object-based system, the obvious choice is call-by-object-reference. Since the value of a variable is a reference to an object, it is that reference (the object name) that is passed in an invocation. This is the same semantics as in CLU (where it is called *call by sharing*) and Smalltalk. In a distributed system, this presents a potentially serious performance problem; any invocation by a remotely invoked object of its parameters is likely to cause another remote invocation. For this reason, systems such as Argus have required that parameters to remote calls be passed by value, not by reference [HL82].

Because Emerald objects are mobile, it may be possible to avoid many remote references by moving parameter objects to the site of the callee. Whether or not this is worthwhile depends on the size of the parameter object, the number of active invocations, and the number of invocations to be issued by the called object. We expect that parameter objects will be moved in two cases. First, based on compile-time information, the Emerald compiler may decide to move an object along with an invocation. For example, small immutable objects may be copied cheaply and are obvious candidates. Second, the programmer may decide that an object should be moved based on knowledge about the application. To make this possible, Emerald provides a parameter passing mode that we call *call-by-move.* A call-by-move parameter is passed by reference, as is any other parameter, but at the time of the call it is relocated to the destination site. Following the call it may be specified to either return to the point of call or remain on the destination site.

Call-by-move is a convenience and a performance optimization. The move could be done explicitly with the *move* primitive, but that would require more explicit code and would not allow packaging of parameter objects in the same message as the invocation. While call-by-move co-locates the parameter with the target object, it increases the cost of the call and may cause extra remote references from the call's initiator.

One goal of the Emerald design is to provide a framework within which object mobility may be studied. The results of our investigations into inexpensive object mobility and call-by-move are reported in [Jul87].

5.3 Reliability and availability

Reliability and availability are two closely related problems that significantly complicate the construction of distributed applications. Reliability is defined to be 1 minus the probability of lost data due to hardware or software failures. Availability is defined to be 1 minus the probability that access to a particular piece of data will be denied due to failures.

A number of languages and systems have addressed the issues of reliability and availability in distributed computer systems. These include the language Argus [Lis84] and the operating system Clouds [All83], each of which provide atomic transactions at the lowest layer in the system, and the ISIS system [Bir85] which supports replication by means of a family of broadcast protocols with increasingly strong ordering constraints.

Neither reliability nor availability were goals of the Emerald design. Therefore, Emerald provides very primitive features to application programmers interested in highly reliable or available applications.

Reliability

The *checkpoint* primitive of Emerald allows a collection of objects on a single machine to atomically save their state. The semantics of checkpoint is similar to that in Eden [Bla85]; the primary difference is that where in Eden checkpoint was specified procedurally (the programmer explicitly *wrote* all the interesting data to a checkpoint *file*) in Emerald the data to checkpoint is specified declaratively (with *attached* declarations). When a node recovers after a failure, all checkpointed objects will be restored by the kernel to the state of their most recent checkpoint. After the state of all objects has been restored, the *recovery* code of the objects is executed, which allows programmer defined recovery actions to occur.

Availability

Two Emerald features provide primitive support for the construction of available software systems. First, the status of the nodes that make up the Emerald system is available through operations on Node objects. This provides information a "replication manager" could use in deciding where replicas should be placed. Second, unavailability handlers provide a mechanism for detecting when invoked objects are unavailable due to failures. When an invocation or location dependent operation is attempted on an object that is not available due to a node failure, the invocation is aborted and the unavailable handler of the enclosing block is executed if it exists.

While these features are very low level, they form a sufficient basis for the construction of higher-level transaction and replication schemes [Pu86].

5.4 Protection

There are two protection related issues. The first is the possibility that some malfunctioning process or object will corrupt the environment in ways that affect properly functioning objects. Operating systems typically provide protection from corruption of this kind through the use of address spaces. In Emerald, rather than adopt this "heavy-weight" solution, we have adopted the Concurrent Pascal philosophy [BH77], which states that new program pieces added on top of old ones must not be able to make the latter fail. Even though all Emerald objects on a node share an address-space, the compiler and run-time system provide the initialization and run-time checks necessary to guarantee that no Emerald programmer can corrupt memory.

The second issue concerns the protection of resources from unauthorized access. A number of object-based operating systems have been capability based [Lev84]. Capabilities are a protection scheme that moves the responsibility for protection from the system to the application programmer. Each object is allowed (even expected) to apply private interpretation to a set of *rights* bits in the capabilities used to access it.

In the presence of objects implementing more than one abstraction, which our abstract type system encourages, the assignment of meaning to the rights in a capability can be difficult. The interpretation that an object will make of the rights bits in the capabilities used to address it is part of its interface, and therefore must be determined by the abstract type. However, the number of rights bits in each capability is limited, usually severely (Eden capabilities have 16 rights bits). An object attempting to implement two abstractions that assign different meanings to the same bit is unable to separate the two.

Section 8.2 discusses one extension of the abstract type system to the area of protection that we are currently investigating.

5.5 Concurrency

With the Emerald focus on a single uniform object model, the question arises: what about processes? Distributed applications must be able to create multiple threads of control: how is this to be done? Our solution is the same as that in StarOS, Argus, and Eden — processes are contained in objects and cannot be directly referenced. Processes communicate and synchronize through shared objects.

Each object has an optional process which is started after the object is initialized upon its creation. Within a single object, multiple operation invocations can be in progress simultaneously, and these can execute in parallel with the object's internal process. To control access to variables shared by different operations, the shared variables and the operations manipulating them can be defined within a *monitor* [Hoa74, BH79]. Processes synchronize through built-in *condition* objects. An object's process executes outside of the monitor, but can invoke monitored operations should it need access to shared state.

5.6 Summary

The last three chapters have discussed the features of Emerald that make it appropriate for the construction of distributed applications. These features include its novel type system, the way that objects are constructed, and its distribution, reliability and concurrency related features. The following chapters discuss the other claim that we made for Emerald: that it can be efficiently implemented in a distributed environment.

We discuss the approach that we have taken to providing an efficient implementation in Chapter 6, and provide performance measurements and discussion in Chapter 7.

Chapter 6

The Cost of Abstraction

We stated previously that the major goal of Emerald was to design a distributed programming language that incorporates a single, uniform object model *and* can be implemented efficiently. In attempting to meet this goal, we are forced to address a fundamental tradeoff between abstraction and efficiency. Traditionally, the cost of a language construct is directly related to its expressive power: low-level abstractions can be implemented at low cost, constructs providing more powerful abstractions have a higher cost.

A common approach to this tradeoff is to limit the expressive power of languages by providing only constructs that have an obvious and efficient implementation. This is the motivation for the condition variable as proposed by Hoare [Hoa74], who admits that:

> The synchronization facility which is easiest to use is probably the conditional wait:
>
> *wait(B);*
>
> where B is a general Boolean expression, but this may be too inefficient for general use in operating systems ...

The condition variable is primitive so that it can be implemented efficiently. This same argument explains the existence of two models of computation in each of the distributed programming languages and systems discussed in Chapter 1. The cost of the language constructs for local objects is low — appropriate for the restricted generality that these objects provide. The language constructs for distributed objects are more expensive due to their increased generality and functionality.

An alternative to limiting the power of the supported abstractions is used in the programming languages NIL [SH84] and SETL [SSS81]. Both of these languages support an abstract data model. NIL provides a *relation* primitive type. Each relation contains a variable number of rows, each row being a user-defined n-tuple of values. This relation type is very expressive, and subsumes arrays, linked lists, sets, queues, stacks, etc. The programming language SETL provides sets, tuples, and maps of arbitrary element types as primitive data types.

In each of these languages, the programmer is encouraged to use these abstract data types for constructing his application. The default implementation of these abstractions is not efficient, but these systems provide either manual, semi-automatic, or automatic means of choosing more efficient implementations of the abstract data types used in a program.

In Emerald, we have recognized the expressive power of a single uniform model of computation and have therefore provided an abstract object model and an abstract type system. Objects may be referenced in a uniform manner independent of their location even though they may move at arbitrary times. The type system captures the interface to objects but conveys no implementation information.

As in NIL and SETL, the responsibility for providing an efficient implementation for these abstract entities rests with the compiler. Emerald objects can be implemented in a general way that preserves the full generality of the abstraction. Therefore, as in NIL and SETL, a correct implementation of a program can be easily generated by using this most general implementation for all of the objects created or manipulated by the program. The criteria on which the compiler bases its decision of which implementation should be used varies between the languages. In NIL and SETL the selection of an implementation for a particular object is determined by the semantics of its data type. For example, the selection may be based on the ratio of modification and inspection operations performed on the object, the domain and range types of a relation, and whether a relation is one-to-one. In Emerald, the expressive power is not in the semantics of particular data types but rather in the semantics of objects themselves. The selection of implementation for an object is based on what is known about the object and how it is manipulated.

The abstraction power of Emerald comes from two sources:

1. The single object model which unifies private, local objects and shared, global objects.

2. The abstract type system, which unifies all objects implementing an abstract interface.

These features simplify the construction of applications by allowing the programmer to define each object only once. A single directory description can be used both for network-wide file system members and for a private compiler symbol table. While the object model and type system are very general, no application requires this full generality for every object that it uses. Objects are often used in restricted ways. If a compiler can detect that the full power of an abstraction is not required, it can provide a more efficient implementation.

6.1 Getting rid of abstract types

As is discussed in Chapter 3, Emerald supports two notions of type. Abstract types capture the interface to objects but not their implementation; these are declared for every identifier in Emerald and form the basis for type checking. Objects are created using object constructors; the code generated by the compiler for an object constructor forms the concrete type of the objects created by executing it. In general, only the abstract type of object references and invocation targets are known to the compiler. Optimizations are possible when the concrete types of object references and invocation targets are known at compile time. When the concrete type of an object reference is determined statically, the compiler knows that no run-time searching will ever be required in response to an invocation. Therefore, the data structures that support this run-time search (such as operation vectors [BHJ+87] or caches [CPL83]) need not be allocated or maintained for this reference. When the concrete type of an invocation target is statically determined, the run-time search can be eliminated and a simple jump to a compiler-determined code address substituted in its place.

It is impossible to know the concrete type of every object reference. In fact, one motivation for supporting abstract types was to allow the addition of newly defined objects to an existing system. Clearly, "old" objects are unable to know the concrete type of "new" objects that were defined after them. However, not every object reference requires this generality. Consider the Directory example in Figure 6.1. Assuming that *aode.empty* always returns an object with the same concrete type, we can determine at compile time the concrete type of the constant *state* and take advantage of this when generating code for its invocations.

Some *primitive* abstract types are constrained to have only one implementation or concrete type — an implementation provided by the system. The reasons for this restriction are twofold:

1. As is discussed in Section 3.5.4, an Emerald abstract type captures the interface to objects, but does not define the semantics of these objects. The correct functioning of some language constructs depends on the semantics of objects implementing a particular abstract type. For example, the types **Boolean**, **Condition**, **Node**, **Signature**, and **Time** are intimately related to if statements, monitors, object location, type checking, and the real-time related operations of the language. The only way that these language constructs can guarantee the semantics of the objects that they manipulate is to force the use of a particular implementation. Therefore,

```
const DirectoryCreator == immutable object DirectoryCreator
    export empty
    operation empty → [aNewDirectory : Directory]
        aNewDirectory ← object aDirectory
            export Lookup, Add, Delete
            const DirectoryElement ==
                record DirectoryElement
                    var name : String
                    var obj : Any
                end DirectoryElement
            const aode == Array.of[DirectoryElement]
            monitor
                const state == aode.empty
                function Lookup[name : String] → [o : Any]
                    var de : DirectoryElement
                    var i : Integer
                    i ← state.lowerbound
                    loop
                        exit when i > state.upperbound
                        de ← state.getElement[i]
                        if de.getName = name then
                            o ← de.getObj
                            exit
                        end if
                        i ← i + 1
                    end loop
                end Lookup
                operation Add[name : String, o : Any]
                    ⋮
                end Add
                operation Delete[name : String]
                    ⋮
                end Delete
            end monitor
        end aDirectory
    end empty
end DirectoryCreator
```

Figure 6.1: Directory

these types may not be reimplemented.

2. To provide an efficient base set of types from which others may be constructed, the type may be constrained to be primitive. The types **Character**, **Integer**, **Real**, **String**, and **Vector** are examples.

6.1.1 Determining concrete types

The optimizations that we may perform when concrete types are determined at compile time include more space-efficient storage of object references for identifiers and improved code for invocations. Therefore, our algorithm for concrete type determination must be able to deduce the concrete types of object identifiers (constants and variables) and expressions used as invocation targets. Clearly, the concrete type of a variable or constant identifier is determined by the concrete types of the expressions assigned to it. Therefore, determining concrete types involves figuring out the concrete types of expressions and propagating this information to identifiers to which they are assigned.

There are two kinds of expression in Emerald. First are primitive expressions such as built in operators (== which decides if two object references refer to the same object and **locate** which finds the current location of an object), execution of object constructors, and the various forms of literals. These expressions are implemented by the system, and the concrete type of the object returned is therefore known.

The second kind of expression is an operation invocation. Determining the concrete type of the result of an invocation requires that the compiler examine the code that implements the operation. This implies that the concrete type of the invocation target must be known, otherwise the code that will execute in response to an invocation request cannot be known. In addition, the concrete type of the result returned by that operation must also be known. There are two situations where the concrete type of this result may be known.

In the simplest case, the invocation could always return an object of the same concrete

```
object aRecord
    export getThing, setThing
    var thing : Any
    operation setThing[theThing : Any]
        thing ← theThing
    end setThing
    function getThing → [theThing : Any]
        theThing ← thing
    end getThing
end aRecord
```

Figure 6.2: A record-like object

type. An example is the invocation of the *empty* operation on the *DirectoryCreator* of Figure 6.1. The body of this operation simply returns a reference to a newly created directory object. In this case, the concrete type of the expression is simply the concrete type of the returned object.

Second, the concrete type of an invocation result may depend on the concrete types of the arguments to this or some other operation on the object. As an example of this case, consider the declaration of a record-like object in Figure 6.2. The concrete type of the result of *getThing* operations on *aRecord* depends on the concrete types of the arguments to the *setThing* operations. Our currently implemented algorithm does not attempt to detect the concrete types of such invocation results. We defer until Section 6.3 the discussion of a better algorithm which could.

6.1.2 The concrete type determination algorithm

The previous discussion demonstrates how the concrete type of identifiers and expressions in Emerald depends on the concrete types of other expressions. The basic notion of the concrete type determination algorithm is therefore to construct a directed graph whose vertices represent identifiers (variables and constants), and expressions. An edge from a vertex a to a vertex b indicates that the concrete type of a depends on that of b. Specifically, we traverse the program parse tree to build a directed graph $G = \{V, E\}$, where each edge $(a, b) \in E$ indicates that the concrete type of a depends on that of b. A vertex is created in V for each identifier definition node (constant and variable), invocation node, and expression node in the original program tree. Each vertex may be marked with one of three marks:

Undefined

The concrete type of this vertex has not yet been considered.

Unknown

The concrete type of this vertex cannot be determined by the algorithm.

Known

The concrete type of the vertex is known, and is marked in the vertex.

Depending on the kind of node encountered, add edges to the graph G:

Assignment statement ($i \leftarrow e$):

Constant declaration (const $i == e$):

Variable declaration with initialization (var $i : t \leftarrow e$):

All of these are assignments of e to i, therefore we add an edge from i to e.

Invocation ($t.opname[a_1, \ldots, a_n]$):

Add an edge from the vertex representing the invocation to the vertex representing the target of the invocation (t).

To illustrate the graph creation process, consider the example Emerald program in Figure 6.3. This program results in the construction of the graph in Figure 6.4. Each assignment statement results in the addition of a single edge to the graph.

The heart of the analysis phase is an algorithm that detects the strongly-connected components of a directed graph. A strongly-connected component of a directed graph $G = \{V, E\}$ is a maximal set of vertices $V' \subseteq V$ such that for every pair of vertices $<u, v>$

```
**var** $x$, $y$, $z$ : **Any**
$x \leftarrow$ **object** *object1* $\cdots$ **end** *object1*
$y \leftarrow$ **object** *object2* $\cdots$ **end** *object2*
$z \leftarrow x$
$x \leftarrow z$
$y \leftarrow x$
```

Figure 6.3: Concrete type determination example

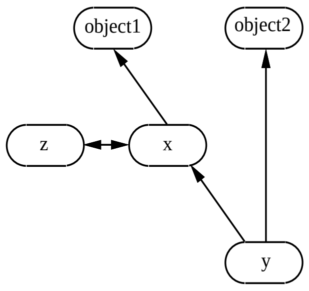

Figure 6.4: Concrete type determination example — dependency graph

$\in V'$, there exists a path in G from u to v and from v to u. In other words, there is a path from every vertex in a strongly-connected component to every other vertex in the strongly-connected component. In terms of detecting concrete types, such a strongly-connected component is a set of expressions whose concrete types depend on each other, and therefore must all be the same.

We propagate information backwards along the edges of the graph using an algorithm [AHU74, pp. 189-195] that finds the strongly-connected components of a graph. This algorithm finds strongly-connected components of G in "leaf to root" order. "Leaf to root" order implies that, if an edge (a, b) exists in G, then a depends on b, and therefore the component containing b will be presented either before the one containing a (if a and b are not in the same strongly-connected component) or at the same time as a (if they are in the same strongly-connected component). From the example in Figure 6.4, the strongly-connected components are:

- *object1*
- *object2*
- x, z
- y

This is also a valid ordering, since the prerequisites of each vertex are presented before any vertex that depends on it. Other valid orderings are also possible.

As each strongly-connected component is produced, we determine the concrete type of each vertex in the set depending on the kind of node that vertex represents. As the vertices representing concrete types of the prerequisites of a vertex are considered, a mark of undefined implies that the prerequisite vertex is in the same strongly-connected component as this one, and therefore will have the same concrete type. We therefore look at the vertices in each set as it is discovered by the strongly-connected components algorithm and attempt to find a concrete type that can be assigned to all vertices in the set. For each vertex we compute a concrete type based on the prerequisites of that vertex. If any prerequisite of a vertex is marked unknown, then the result concrete type is unknown. Otherwise:

Identifier node (e.g., x, y, z):

If the concrete types of the prerequisites are all the same or undefined, then the result concrete type is that concrete type. Otherwise, the result concrete type is unknown.

Expression node (e.g., locate e, $e == e$):

These expressions return primitive types (**Node** or **Boolean**) as results and so we know the result concrete type.

Literal node (e.g., *"a string", 2,* **object** ... **end):**

Literal nodes are leaves of the graph, and therefore the result concrete type is obvious.

Result of an invocation node (e.g., $t.opname[a_1, \ldots, a_n]$**):**

If the concrete type of the invocation target is unknown, then the concrete type of the result is also unknown. Otherwise, if executing the invocation always returns the same concrete type, C, then C is the result concrete type. Otherwise, the result concrete type is unknown.

If the result concrete types of every vertex in the set are the same, then mark each vertex in the set as known with that concrete type. Otherwise, mark each vertex in the set as unknown.

Looking at the example, the concrete types of *object1* and *object2* are trivially computed (let us call them *ct1* and *ct2*), and these expression nodes marked as known with those two concrete types. When looking at the set of vertices $\{x, z\}$, we find that these identifiers only depend on each other and the object constructor *object1*, therefore both these vertices may be marked known with concrete type *ct1*. When looking at y, we find that it depends on *object2* (with concrete type *ct2*) and x (with concrete type *ct1*). We therefore conclude that we do not know the concrete type of y, and mark it unknown.

The execution time of this algorithm is linear in the size of the program.

- A single pass is made over the parse tree to construct the graph.
- The number of vertices in the graph is limited by the number of constants, variables, expressions, and invocations in the original program.
- The number of edges in the graph is similarly limited by the number of statements and expressions in the program, since each programming language construct contributes no more than 1 edge.

- The worst case (and average case) running time of the algorithm to propagate information along the edges of the graph is $O(Max(|V|, |E|))$ [AHU74, pp. 189-195].

6.2 Making objects local

Important optimizations are also possible when the Emerald compiler can determine that objects are local to some containing object. In general, the implementation of Emerald objects must allow them to be moved at arbitrary times, and code generated to invoke an object must take into consideration the possibility that the object will not be on the same node as the invoker. When it is known at compile time that an object will always be local, we may perform the following optimizations:

- We may use a location *dependent* pointer to it (a real address), rather than a location independent reference. This saves both space for the pointer and access time when performing invocations on the object.

- We may generate code for invocations that assumes that the object is resident on this machine; no check of residency must be done before proceeding with an invocation.

- We may compile operations on the object in-line. This is not possible for arbitrary objects since the object may move during the execution of the operation, and arbitrary primitive operations (e.g., instance variable access and assignment) cannot be performed remotely. In order to perform this optimization, we must also know the concrete type of the object.

Since objects that are declared by the programmer to be immutable may be freely copied when references are sent across machine boundaries, we know that immutable objects will always be resident. Therefore, we may perform the same optimizations on them as we do for objects that we know are local to some containing object.

For an example, consider the directory implementation in Figure 6.1 on page 73. This directory has a constant *state* which is initialized to be an empty array of *DirectoryEle-*

ments. In principle, nothing prevents the *state* array from being moved to some other node in the network or exported as the result of some invocation on the directory. However, in this example, the array *state* is completely local to its containing directory — no references are exported, moreover, it isn't moved. Since the *state* object is not remotely accessed and does not move independently of its containing object, the compiler is free to provide a more efficient implementation — one that *cannot* be remotely accessed or moved. Note that we are not restricting what the programmer may do with his objects: we are taking advantage of what the programmer chooses not to do with them.

6.2.1 Determining locality

The determination of local objects is similar to the determination of concrete types described in Section 6.1.1. Again, we are interested in the locality of object references for identifiers (allowing us to create more space-efficient objects), and the locality of invocation targets (to generate improved code for invocations). The concrete type of an expression is not affected by operations performed on the object it denotes, but may be determined solely from the concrete types of its sub-expressions. Object locality on the other hand is affected by actions performed on it (such as moving it) or identifiers to which it may be assigned. In other words, while concrete type information flows only one way on assignments (the concrete type of the expression affects the concrete type of the identifier to which it is assigned), locality information flows both ways (moves of the identifier or expression affect the locality of both). In addition, before it can known that an object is local, the compiler must know its concrete type. Any object may move itself, give out references to itself as the results of invocations, and in other ways prevent the compiler from using a local implementation for it. When we use a local implementation for an object, we must know that it does none of these things. Therefore, we attempt to detect the locality of objects only after concrete types have been determined.

In detecting which Emerald objects are local, we therefore must find objects that are created and manipulated purely locally, and whose concrete types do not do things that make them non-local. Again the most complicated task is determining the locality of invocation results. There are two cases to consider.

In the simple case, the invocation returns a reference to a newly created object. The locality of this object depends on its concrete type (what it does to itself) and how the object is manipulated. No additional dependencies are required.

The more difficult case corresponds to the more complicated case for determining concrete types, when the locality of an invocation result depends on the locality of the arguments to this or some other invocation on the object. Consider again the record-like object of Figure 6.2. The locality of the result of a *getThing* invocation depends on the locality of the arguments to the *putThing* invocations executed on this object. Our implemented algorithm does not detect such potentially local invocation results. Section 6.3 discusses this further.

6.2.2 The local object determination algorithm

An algorithm similar to the one used to determine the concrete type of object references detects whether objects are local to their containing objects. In constructing the graph, almost every edge (a, b) added to the graph in the previous algorithm is accompanied, when detecting locals, by the reverse edge (b, a). The marks on the vertices are also slightly different:

Undefined

The locality of this vertex has not yet been considered.

Nonlocal

This vertex represents a non-local object.

Local

The vertex represents a local object.

In addition, the actions performed in each step, dependent on the kind of parse tree node that the vertex represents, are different. Specifically:

Assignment statement ($i \leftarrow e$):

Constant declaration (const $i == e$):

Variable declaration with initialization (var $i : t \leftarrow e$):

All of these include an assignment of e to i, therefore add edges from i to e and from e to i.

Result of an invocation ($t.opname[a_1, \ldots, a_n]$):

If the concrete type of the target of the invocation is known, and the invocation causes the creation and return of a new object then do nothing. Otherwise, mark the vertex nonlocal.

Move, fix, unfix, refix statement:

Since performing one of these operations on an object forces us to use the most general implementation, mark the vertex representing the operand expression nonlocal.

The algorithm of Section 6.1.2 is used to detect the strongly-connected components of the dependency graph. The vertices that form a strongly-connected component may be marked local only if all three of the following conditions hold:

1. All prerequisite vertices are marked local.

2. The concrete type of the vertices is known. Since each edge added in determining concrete types is also included when detecting locals, the concrete types of all prerequisites will be the same if it is known.

3. That concrete type neither moves (or fixes or refixes) itself nor returns a reference to itself from any invocation.

If these conditions hold, then the vertices in the strongly-connected component are marked local; otherwise, they are nonlocal.

6.3 A better algorithm

The algorithms just presented for determining the concrete types and locality of objects do not attempt to discover anything about the results of any but the most trivial invocations. As mentioned in Sections 6.1.1 and 6.2.1, there are cases when the *attributes* (concrete type or locality) of an invocation result depend on the attributes of the arguments to other invocations on the object. Consider again the record-like object from Figure 6.2.

```
object aRecord
    export getThing, setThing
    var thing : Any
    operation setThing[theThing : Any]
        thing ← theThing
    end setThing
    function getThing → [theThing : Any]
        theThing ← thing
    end getThing
end aRecord
```

The attributes of the result of *getThing* operations on *aRecord* depend on the attributes of the arguments to the *setThing* invocations performed on it. In fact, to deduce something about the attributes of the result of a *getThing* operation, the compiler must have available information about the attributes of the argument to every *putThing* invocation executed on this object. This implies that the compiler must be able to find all invocations on this object. This is only possible when the scope of references to the object is limited, or in other words, when the object is used only locally. The locality determination algorithm just presented finds exactly these references, and can therefore be used to provide this information. Unfortunately, we have seen that determining locality also depends on concrete type determination. These two algorithms must be applied together to determine the concrete type of the results of invocations that depend on the arguments to other invocations on the object. There are two possible ways to combine these algorithms.

First, we could apply the concrete type and locality determination algorithms iteratively, stopping when no progress is made. It is easy to construct artificial situations where the concrete type of a depends on the locality of b which depends on the locality of a which depends on the concrete type of a. Such circular dependencies cannot be resolved by iterative application of these algorithms. On the other hand, our preliminary experience shows that such situations do not often occur in real programs.

Second, we could build a graph that contained vertices representing both the concrete types and locality of program constructs and use the strongly-connected components algorithm to discover an evaluation order. This combined graph is much larger than the two separate graphs. Since the attributes of an invocation result may depend on the attributes of any argument to the invocation, we must add edges from every invocation node to every argument. If we knew the concrete type of the target when constructing the graph however, we would only add edges to arguments that affect the attributes of the result. This greatly expands the size of the dependency graph, and the size of the strongly-connected components. In addition, the analysis of each strongly connected component when discovered in the graph is more complicated. We can have situations where the concrete types and locality of a number of identifiers are all mutually dependent. Much more complicated algorithms must be developed to sort out such situations.

Statistics on the success of our currently implemented algorithm, an algorithm that could determine the attributes of invocation results as hinted to above, and a perfect algorithm are presented in Section 7.3. These statistics show that our current simple algorithm performs almost as well in practical situations as the better algorithms just outlined.

6.4 Discussion

In contrast to languages that have restricted their expressive power to constructs that have obvious efficient implementations, or have two or more user-visible constructs with different power and implementations, Emerald has a very general object model and an abstract type system. The responsibility for implementing objects efficiently (relative to the generality required by each object) rests with the compiler. We have discussed the algorithms used by the compiler to detect situations where the full generality of the object model and type system are not required, and outlined alternative approaches that lead to better algorithms.

Chapter 7

Performance

Two major goals drove the design of Emerald. The first was that Emerald should support a uniform object model suitable for the construction of both local and distributed objects with identical semantics. We were unwilling to compromise this goal for any reason. Second, we intended that the performance of objects should be appropriate to the uses to which they are put. This chapter discusses the performance goals of Emerald and provides measurements to substantiate our claim that these performance goals have been met. In addition, we discuss the primary factors affecting the performance of Emerald objects and compare Emerald performance to that of similar systems.

7.1 Performance goals

In designing Emerald, we anticipated three categories of objects:

Primitive objects

Primitive objects include characters, integers, reals, and Booleans. Since these objects are small and immutable they can be freely copied rather than shared and accessed through pointers. Therefore, we expect that they should be allocated directly in the data areas of objects that reference them and manipulated by in-line code invoking hardware operations. They should be as efficient as primitive objects (values of primitive types) in any other programming language.

Local objects

Some objects are used in ways that make them local to some containing object. Such objects should have a minimal amount of storage overhead and operations invoked on them should have performance comparable to the cost of a procedure call in a traditional programming language.

Global objects

When the full generality of objects is used, we are willing to pay more in both storage overhead and time for operation invocation. However, even though objects in this category have the potential to be on other machines requiring network communication for operation invocation, we expect that a significant fraction of invocations will be on objects that are currently co-located with their invoker. For this reason, we have two goals for global object invocation:

- When the target of an invocation is on the same machine as its invoker (resident) the invocation time should approach procedure call time.
- When an invocation target is on a different machine from its invoker (non-resident) invocation involves the run-time kernel and network communication. In such cases, the invocation time will be orders of magnitude greater than invocation times for invocations on the same machine. Our goal is to perform these remote invocations in time not much worse than the network overhead required to send and receive the messages. The performance of remote invocations is determined by the communications hardware in addition to the operating system upon which Emerald is implemented as well as the structure of Emerald's run-time kernel. The performance of global invocations is not reported here, but can be found in Eric Jul's dissertation [Jul87].

The Emerald object model was designed with these three categories of implementation in mind. We expect to achieve the performance of direct code for primitive objects, procedure calls for local objects, and remote procedure calls for non-resident global objects. Our expectation of near procedure call performance for resident global objects was ambitious. Typically, potentially remote objects are managed exclusively by the run-time kernel of a distributed system. Therefore, invocations of such objects typically involve significantly more machinery, and therefore more expense, than do invocations on objects known to be local.

7.2 Emerald performance

The current prototype of Emerald is implemented on a local network of DEC^{TM} MicroVAX^{TM} II workstations running the ULTRIX^{TM} operating system. The only impact that using ULTRIX as a prototyping environment has on these performance measures relates to stack bounds checking. All Emerald objects on a machine execute in a single address space. Translated into ULTRIX terms, an Emerald node is an ULTRIX process. Within this address space, the Emerald kernel manages multiple Emerald processes, each requiring its own stack. Since ULTRIX does not allow page-by-page memory management control, each Emerald invocation includes an explicit *stack check* to detect when a process has exhausted its current stack allocation.

Table 7.1 shows the performance of a number of primitive operations on the MicroVax II. These provide a basis against which the performance of Emerald operations may be measured. Table 7.2 shows the time taken by invocations in Emerald.

Primitive Operation	Time (microseconds)
integer addition	0.4
real addition	3.4
procedure call/return	13.4
procedure call/return with stack check	16.4

Table 7.1: MicroVax II primitive operation times

Emerald Operation	Example	Time (microseconds)
primitive integer invocation	i ← i + 23	0.4
primitive real invocation	x ← x + 23.0	3.4
local invocation	localobject.nop	16.6
resident global invocation (known concrete type)	globalobject.nop	19.4
resident global invocation (unknown concrete type)	globalobject.nop	23.1

Table 7.2: Timings of Emerald invocations

7.2.1 Discussion

We feel that these performance measures are very good. In fact, except for micro-optimizations, better performance cannot be realized given our hardware. Invocations on direct objects are compiled into native code for the machine. Invocations on local objects are compiled into a procedure call sequence with efficiency comparable to that of the procedure call instructions on the machine. The most surprising statistic is the performance of resident global objects. The overhead for invocation of Emerald objects implemented in the most general manner but currently on the same machine as the invoker is less than 50% of the procedure call time. In other distributed systems, invocation of an object on the same machine requires milliseconds (Clouds [Spa86]), or tens of milliseconds (Eden [ABLN85], ISIS [BJRA85]). This high cost of invocations on potentially remote objects stems from 2 factors:

1. Resolving the reference used to perform the invocation. Global objects are usually referenced with location-independent names or capabilities. These must be resolved before checking the location of the invoked object.

2. Crossing a protection boundary. Each object is a separate protection domain; only code defined by the object itself is allowed to manipulate its representation. When

> an invocation is performed, the access rights must be checked, and the representation must become available for direct access. A significant fraction of the cost of a resident invocation in Clouds is related to performing this mapping [Spa86].

Since the Emerald compiler provides the protection necessary to ensure that the representation of object cannot be corrupted, all Emerald objects on a machine may safely share the same address space. Emerald also optimizes for the local case, since it is expected to be most common. Location dependent references (real pointers to descriptors) are used, even for global objects, allowing the compiled code access to information concerning the object. Invocation is done by building an activation record on the caller's stack, checking to see if the target object is resident, and if so, executing the invocation without any kernel calls. If the residency check fails, the kernel is called to transmit the activation record to the target machine and perform the invocation there.

7.3 Concrete type and locality determination

Chapter 6 discusses the optimizations performed by the Emerald compiler based on compile-time determination of the concrete type and locality of objects. The performance figures in Table 7.2 indicate that when the concrete type of a reference is known at compile time that a 16 percent reduction in invocation time results. When an object is known to be local, the reduction in invocation time increases to 28 percent. In addition, when an object is known to be local, the compiler is allowed to do procedure integration, or expand the body of the invoked operation in place of the invocation sequence. This has the potential of removing the invocation overhead completely, at the expense (usually) of expanded object code size. Invocations of global objects may not be expanded inline since they have the potential to move at any time. These improvements indicate that detection of situations where either the concrete type or the locality of a reference can be determined at compile time is important to the performance of the Emerald system.

7.3.1 Success of our algorithm

The fraction of references for which concrete type or locality information can be determined at compile time varies with the application. We have analyzed the performance of our algorithm (as currently implemented in the compiler), on a collection of object definitions comprising the Emerald mail system.

Table 7.3 summarizes the results of our analysis. It presents the static count of invocations generated by type for three detection algorithms when executed on the mail system itself (not including the user interface objects). These three algorithms are:

- A perfect algorithm — one that could detect every situation where an object is used in restricted ways.
- No detection — the only optimizations are those made available because of the primitive types which may have only one implementation.
- The algorithm currently implemented in the compiler.

The column labeled Δ shows the difference between the perfect algorithm and the others.

	Perfect algorithm		No detection		Implemented algorithm	
Invocation Type	Number	Δ	Number	Δ	Number	Δ
total	595	-	595	-	595	-
inlined	451	-	412	-39	451	-
local	59	-	0	-59	59	-
self	7	-	7	-	7	-
immutable	14	-	14	-	14	-
global	64	-	162	+98	64	-
known concrete type	557	-	464	-93	543	-14
unknown concrete type	38	-	131	+93	52	+14

Table 7.3: Performance of three detection algorithms — mail system

Without concrete type and local object detection, 98 invocations that could be opti-

	Perfect algorithm		No detection		Implemented algorithm	
Invocation Type	Number	Δ	Number	Δ	Number	Δ
total	345	-	345	-	345	-
inlined	211	-	211	-	211	-
local	0	-	0	-	0	-
self	24	-	24	-	24	-
immutable	0	-	0	-	0	-
global	110	-	110	-	110	-
known concrete type	321	-	243	-78	321	-
unknown concrete type	24	-	102	+78	24	-

Table 7.4: Performance of three detection algorithms — user interface

mized to either inlined operations or local procedure calls remain with the most general implementation. The currently implemented algorithm detects all 98 of these potential optimizations. The only things left undetected by the current implementation are the 14 invocations on immutable objects. A perfect algorithm could detect the concrete types of these invocations. The better algorithms described in Section 6.3 could also determine these 14 concrete types. The remaining 64 global invocations (38 of which are to objects where the concrete type are not known) are actually required by the application. The semantics of the mail system requires the generality of these remaining invocations to allow for multiple implementations of the mailbox, mailmessage, and directory abstractions.

Table 7.4 presents these same statistics for the objects comprising the mail system user-interface. The user interface to the mail system is heavily biased by its input/output related invocations on the standard input and output streams. Invocations on these streams are global, since the streams are fixed on particular machines (the one where the display is), and the interface objects may be moved arbitrarily. In this example, the implemented algorithm performs as well as a perfect algorithm would.

The overall static and dynamic behavior of the implemented algorithm on the mail system can be seen in Table 7.5. The dynamic measures are the result of sending and later reading ten short mail messages.

	Static		Dynamic	
Invocation Type	Number	%	Number	%
total	940	100.0	4429	100.0
inlined	662	70.4	2995	67.6
local	59	6.3	306	6.9
self	31	3.3	277	6.3
immutable	14	1.5	75	1.7
global	174	18.5	776	17.5
known concrete type	864	91.9	4121	93.0
unknown concrete type	76	8.1	308	7.0

Table 7.5: Overall frequency of invocations by type

Both statically and dynamically, over 2/3 of invocations are primitive and generate machine instructions. Of the remaining 1/3, an additional 11-15% (local, self, and immutable invocations in the table) are implemented as procedure calls, since it is known at compile time that the target will be on the same machine as the invoker. Only 18% of all invocations require an invocation sequence capable of dealing with objects that may be resident on other machines.

The high frequency of input/output related invocations in the mail system user interface objects skews this information somewhat. Table 7.6 presents this same information removing the invocations on the standard input and output streams. If we discount the effect of the Input/Output related operations of the mail system, we find that only 11.2% of invocations statically or 6.5% of invocations dynamically are on global objects. We may also see that only approximately 8% of invocations actually use the flexibility of the abstract type system.

7.3.2 Discussion

While our currently implemented algorithm for the detection of concrete type and locality information is successful in determining the attributes of only very simple invocation results, they perform very well in actual situations, capturing almost all of the concrete

	Static		Dynamic	
Invocation Type	Number	%	Number	%
total	863	100.0	3907	100.0
inlined	662	76.7	2995	76.7
local	59	6.8	306	7.8
self	31	3.6	277	7.1
immutable	14	1.6	75	1.9
global	97	11.2	254	6.5
known concrete type	787	91.9	3599	92.1
unknown concrete type	76	8.1	308	7.9

Table 7.6: Frequency of invocations by type — discounting input/output

type and locality information that can be found in the mail system application. Their current major flaw lies in their inability to propagate concrete type and locality information through "record-like" objects — records, vectors, and arrays. The more sophisticated algorithms outlined in Section 6.3 could perform this propagation, at the expense of increased compilation cost.

7.4 Summary

A major criticism of object-based programming languages and systems is that they are inefficient. Our purpose in discussing the performance of the current implementation of Emerald is not to brag about shaving off a microsecond here or there. We have, rather, wished to demonstrate that the use of an object-based language — even in a distributed computing environment — is not an inherent source of inefficiency. While the Emerald model of mobile objects viewed through loose-fitting abstract types is very general, it can be implemented efficiently; our current implementation is a proof by construction. Much can be done to allow the luxury of a general model, while only paying for that generality actually used in an application.

Chapter 8

Conclusion

We have addressed the problem of constructing application programs for execution in a distributed environment. In a distributed environment, there are two natural implementation styles available. The more expensive style is appropriate for entities that are to be remotely accessed or moved. Private data within such a remotely accessible entity may use a much simpler, more efficient implementation style. The performance difference between the two styles can be as high as a factor of 1000; an operation that can be done in microseconds on a local object may require milliseconds to perform on a remote object. A number of programming languages and operating systems have also addressed the problem of constructing distributed applications. These languages and systems have reflected the semantic and performance differences between local and remote computation in the models of computation that they support. Each of them provide two levels of support: one whose semantics and performance are appropriate for the construction of private entities, the other appropriate for distributed entities that may be shared and remotely accessed.

The programming language Emerald provides a single model of computation, the object, which is appropriate for constructing objects at all levels of a distributed system. Primitive objects such as integers and characters, local data objects such as records and arrays, and distributed objects such as files, directories, and compilers are defined in a uniform way. As in existing systems, there are multiple implementation styles that may be used for each object; the task of providing an implementation whose cost is appropriate to the generality required by individual objects is given to the compiler.

8.1 Contributions

This dissertation has presented a number of significant contributions. First, we have demonstrated that a single object model can be defined that is appropriate for the construction of every object in a distributed system. A language that incorporates such a single object model has been designed and implemented.

Second, we have designed and implemented a type system that completely separates specification from implementation, treats types as first class objects, and provides polymorphism in a simple and straightforward manner. To our knowledge, no type system incorporating all of these features has previously been defined or implemented.

Third, we have recognized two instances of the general principle that very abstract models can be implemented at no cost when not actually used by an application. The definition of the Emerald object model implies that in the most general case nothing about the implementation or location of a target object may be known when compiling code for invocations. This generality of the object model and type system provides the flexibility necessary for the construction of some distributed applications (such as the directory system). On the other hand, in most instances, objects do not require the full generality of either the object model or the type system, and the compiler can generate much improved code by detecting these situations.

These three contributions are not entirely independent. Our original purpose in designing Emerald was to investigate the possibility of a language where a single, uniform model was used for the construction of both local and distributed objects. The evolution of the type system and the compiler techniques for detecting situations where the full power of an abstraction is not required followed from that goal. In fact, the type system could not divorce itself from concerns about implementation unless there was some other feature (the object model) that provided representation independence. In addition, the compiler techniques to detect the use of limited abstractions would not be appropriate unless a single, general abstraction was available. On the other hand, these contributions are also important independently. A type system that allows multiple implementations of an abstraction to co-exist may be useful in an environment other than distributed objects. Compiler techniques to detect situations where only limited generality is used in a program have application to more than just object-based programming.

8.2 Further research

The design and implementation of the Emerald programming language has demonstrated the validity of the thesis that a uniform object model can be efficiently implemented in a distributed environment. This work has also brought to our attention areas where further work is required.

The cost of abstraction

The Emerald programming language is based on the philosophy of providing a simple high level abstraction, and relying on the system for its efficient implementation. A similar philosophy can be seen in the design of the SETL and NIL programming languages. Two questions remain to be answered:

- This principle has previously been applied to programming languages (with the advent of high level languages to replace machine code), and operating systems memory hierarchies (with virtual memory replacing explicit overlays). In both these cases, and in our case as well, the preliminary response to these proposals is that the cost would be exorbitant. Can this model be applied to other aspects of programming language or operating system design?

- We have taken this principle to its limit. There is no mechanism in Emerald by which a programmer my relate to the compiler his knowledge (or belief) that a certain object is used in restricted ways and thus should be a candidate for an optimized implementation. We have previously criticized two other instances where designers have gone two far in favor of abstraction: the design of distributed systems that completely hide distribution from the programmer, and the design of virtual memory systems that completely hide the memory hierarchy. How should information that a programmer knows about the nature of his application be communicated to the compiler? When is it appropriate to allow programmer involvement in the implementation of an abstraction, and when is it not?

The type system

We noted in Section 3.5.4 that the Emerald type system relies heavily on the names chosen for operations, with the result that types could either conform accidentally when they should not or not conform when they should based solely on the names chosen for operations rather than on the semantics of these operations. We suspect that these problems could both be resolved, and that Emerald could be more suitable for the construction of verified software, if a type definition included semantic information as well as syntactic information. If a type definition was a formal specification of the objects that it described then the conformity algorithm would be able to compare two types based on their formal specification rather than the syntax of their operation signatures. We suspect that such a type system would provide a basis in which object definitions would be routinely formally proven to match their specifications.

Protection

One interesting feature of the Emerald implementation is the tight coupling of the compiler and operating system. The Emerald compiler takes responsibility for tasks that traditionally belong to the operating system (such as protection from faulty programs), and leaves to the operating system tasks that traditionally are the compiler's (such as the handling of illegal dereferences of **nil** and performing remote invocations). This principle may be extended to compiler provided security as is mentioned in Section 5.4.

Object-based operating systems have often used capabilities to provide security. Capabilities are protected object references containing access rights; the invocation of an operation on an object requires a particular access right in the capability used to name the object. In the face of the multiple inheritance that Emerald's abstract type system encourages, the assignment of abstract access rights to concrete bits in the capabilities is very difficult, probably impossible [Bla85].

However, abstract types as defined in Emerald have a number of similarities with capabilities [Lev84]. Invoking an operation on an object referenced by an Emerald constant or variable requires the abstract type of the reference to include the requested operation. As defined in Chapter 3, the widening of Emerald abstract types (to include more operations) is restricted only by the operations that the object actually implements. If it were possible to prohibit the widening of particular references to objects, the security of capabilities could be provided by the type system. We are currently investigating this possibility.

Accommodating heterogeneity

We have assumed throughout the design and implementation of Emerald a very homogeneous system: the processors in the system must all be identical. Since such homogeneous environments rarely occur in practice, it seems natural to extend Emerald to execute in a heterogeneous system.

The accommodation of multiple machine types could be handled (conceptually) quite simply. The data area of each object is fully described by run-time descriptors in order to support garbage collection and translation when objects are moved. If this information was expanded slightly (to mark data that does not require attention when moving between homogeneous processors, but which may need to change representation when moving between heterogeneous processors), it could be used to change representation when moving an object from a machine of one type to one of another type. Also, when object descriptions are compiled, machine code must be generated for each machine type in the system; this could also be done as needed rather than all at once.

Exploiting parallel hardware

Emerald was designed for the construction of application programs to execute in a loosely coupled network of uniprocessors. The ever-decreasing cost of processors that led to the feasibility of such distributed systems is now leading to very affordable multiprocessors. We are now investigating whether Emerald is an appropriate language for the programming of such multiprocessor machines.

Use of the language

The design of the Emerald programming language relies heavily on experience gained with the design, implementation, and use of the Eden distributed operating system and the Eden Programming Language. While the object model of Emerald is very similar to that of Eden, Emerald and Eden are different, and the appropriateness of Emerald as a language for the construction of distributed applications can only be measured after applications have been constructed using it. Therefore, while preliminary experience in the use of the language by its authors and a few other interested bystanders has been favorable, its use by a wider spectrum of programmers for a wider variety of applications is necessary before we can substantiate our claim that Emerald simplifies the construction of distributed applications.

8.3 Summary

The construction of application programs to execute on distributed hardware is a difficult task. The Emerald programming language simplifies this task by providing a single abstraction model — the object — that may be used to define both local and distributed entities. Since all objects are defined using the same mechanism and have the same semantics, an application can be developed using local objects and then distributed without changes to the definitions of the objects used.

Although all objects have the same semantics and are defined using the same object model, it is not the case that they all have the same implementation. The Emerald compiler detects situations where the full generality of the object or type system abstraction is not required for a particular object, and provides an improved implemenation. In this way, one can have a very general model, but only pay for the generality actually used in an application.

Bibliography

[ABBW84] Guy T. Almes, Andrew P. Black, Carl Bunje, and Douglas Wiebe. Edmas: A locally distributed mail system. In *Proceedings of the Seventh International Conference on Software Engineering*, Orlando, Florida, March 1984.

[ABLN85] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, January 1985.

[Ada83] *Reference Manual for the Ada Programming Language*, January 1983. ANSI/MIL-STD-1815A.

[AH84] Guy Almes and Cara Holman. Edmas: An object oriented locally distributed mail system. Technical Report 84-08-03, Department of Computer Science, University of Washington, Seattle, Washington, December 1984.

[AH85] Guy Almes and Cara Holman. The eden shared calendar system. Technical Report 85-05-02, Department of Computer Science, University of Washington, Seattle, Washington, June 1985.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffery D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[All83] James E. Allchin. *An Architecture for Reliable Decentralized Systems.* PhD thesis, Georgia Institute of Technology, Atlanta Georgia, September 1983. Also: Technical Report GIT-ICS-83/23, Georgia Institue of Technology.

[And82] G. R. Andrews. The distributed programming language SR — mechanisms, design and implementation. *Software — Practise and Experience*, 12(8):719–754, August 82.

[BH77] Per Brinch Hansen. *The Architecture of Concurrent Programming.* Prentice Hall Series in Automatic Computation. Prentice Hall, 1977.

[BH78] Per Brinch Hansen. Distributed processes: A concurrent programming concept. *Communications of the ACM*, 21(11):934–941, November 1978.

[BH79] Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(5):50–56, May 1979.

[BHJ$^+$87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1), January 1987. Also Technical Report 86-02-04, Department of Computer Science, University of Washington.

[BHJL86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86. ACM, October 1986. Also Technical Report 86-04-03, Department of Computer Science, University of Washington, revised June 1986; Published in SIGPLAN Notices, vol. 21, no. 11, November 1986.

[BHM77] Forest Baskett, John H. Howard, and John T. Montague. Task communication in DEMOS. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, pages 23–31. ACM, November 77.

[Bir85] Kenneth P. Birman. Replication and fault-tolerance in the ISIS system. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 79–86. ACM, December 1985.

[BJRA85] K.P. Birman, T.A. Joseph, T. Raeuchle, and A.E. Abbadi. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, SE-11(6):502–508, June 1985.

[Bla85] Andrew P. Black. Supporting distributed applications: Experience with Eden. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 181–93. ACM, December 1985.

[BN84] Andrew D. Birrell and Bruce J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BRV84] F. Baiardi, L. Ricci, and M. Vanneschi. Static checking of interprocess communication in ECSP. In *Proceedings of the 1984 Symposium on Compiler Construction*, pages 290–299. ACM SIGPLAN, June 1984. In ACM SIGPLAN Notices 19:6.

[Car86] Luca Cardelli. A polymorphic *lambda*-calculus with type:type. Technical Report 10, Digital System Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, May 1986.

[CMMS79] David R. Cheriton, Michael A. Malcolm, Lawrence S. Melen, and Gary R. Sager. Thoth, a portable real-time operating system. *Communications of the ACM*, 22(2):105–115, February 1979.

[Coo79] Robert P. Cook. ∗Mod — a language for distributed programming. In *Proceedings of the First International Conference on Distributed Computing Systems*, pages 233–241. IEEE, October 1979.

[CPL83] Thomas J. Conroy and Eduardo Peligri-Llopart. As assessment of method-lookup caches for Smalltalk-80 implementations. In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, chapter 13, pages 239–247. Addison-Wesley, 1983.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[CZ83] David R. Cheriton and Willy Zwaenepoel. The Distributed V kernel and its performance for diskless workstations. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 129–140. ACM, October 1983.

[DD79] A. Demers and J. Donahue. Revised report on russell. Technical Report TR 79-389, Department of Computer Science, Cornell University, September 1979.

[DD85] James Donahue and Alan Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, July 1985.

[Fel79] Jerome A. Feldman. High level programming for distributed computing. *Communications of the ACM*, 22(6):353–368, June 1979.

[GCKW79] D. I. Good, R. M. Cohen, and J. Keeton-Williams. Princples of proving concurrent programs in gypsy. In *Proceedings of the Sixth Symposium on Principles of Programming Languages*, pages 42–52. ACM, January 1979.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley Publishing Company, 1983.

[GSW86] Irene Greif, Robert Seliger, and William Weihl. Atomic data abstractions in a distributed collaborative editing system. In *Proceedings of the Thirteenth Symposium on Principles of Programming Languages.* ACM, January 1986.

[HL82] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[JJD$^+$79] A. K. Jones, R. J. Chansler Jr., I. Durham, K. Schwans, and S. R. Vegdahl. StarOS, a multiprocessor operating system for the support of task forces. In *Proceedings of the Seventh ACM Symposium on Operating System Principles*, pages 117–127. ACM, December 1979.

[JR86] Michael B. Jones and Richard F. Rashid. Mach and matchmaker: Kernel and language support for object-oriented distributed systems. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 67–77. ACM, September 1986.

[Jul87] Eric Jul. *Object Mobility in Emerald.* PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington, 1987. In preparation.

[LAB$^+$79] Barbara Liskov, Russ Atkinson, Toby Bloom, Eliot Moss, Craig Schaffert, Bob Scheifler, and Alan Snyder. CLU reference manual. Technical Report MIT/LCS/TR-225, Massachusetts Institute of Technology, Laboratory for Computer Science, October 1979.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems.* Digital Press, Bedford, MA, 1984.

[LGFR82] Keith A. Lantz, Klaus D. Gradischnig, Jerome A. Feldman, and Richard F. Rashid. Rochester's intelligent gateway. *IEEE Computer*, 15(10):54–68, October 1982.

[LH85] Mark R. Laff and Brent Hailpern. Sw 2 - an object-based programming environment. In *SIGPLAN 85 Symposium on Language Issues in Programming Environments*, pages 1–11. ACM, July 1985.

[LHL$^+$77] B. Lampson, J. Horning, R. London, J. Mitchell, and G. Popek. Report on the programming language EUCLID. *SIGPLAN Notices*, 112(2), 1977.

[Lis84] Barbara Liskov. Overview of the argus language and system. Programming Methodology Group Memo 40, M.I.T. Laboratory for Computer Science, February 1984.

[LSAS77] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[May83] D. May. OCCAM. *ACM SIGPLAN Notices*, 18(4):69–79, April 1983.

[MMS79] James G. Mitchell, William Maybury, and Richard Sweet. Mesa language manual. Technical Report CSL-79-3, Xerox Palo Alto Research Center, April 1979.

[Nel81] Bruce Jay Nelson. Remote procedure call. Technical Report CSL-81-9, Xerox Palo Alto Research Center, May 1981.

[PM83] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 110–119. ACM, October 1983.

[Pu86] Calton Pu. *Replication and Nested Transactions in the Eden Distributed System.* PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington, August 1986. Also Technical Report 86-08-02, Department of Computer Science, University of Washington.

[Ras86] Richard F. Rashid. From RIG to Accent to Mach: The evolution of a network operating system. Computer Science Department, Carnegie-Mellon University, May 1986.

[RR81] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating systems kernel. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pages 64–75. ACM, October 1981.

[Sco86] Michael L. Scott. The interface between distributed operating system and high-level programming language. In *Proceedings of the 1986 International Conference on Paralled Processing*, St. Charles, IL, August 1986.

[SCW85] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. Owl reference manual. Technical report, Eastern Research Lab, Digital Equipment Corporation, Hudson, Massachusetts, February 1985.

[SH84] Robert Strom and Nagui Halim. A new programming methodology for long-lived software systems. *IBM Journal of Reserach and Development*, 28(1):52–59, January 1984.

[Spa86] Eugene H. Spafford. *Kernel Structures for a Distributed Operating System.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, May 1986. Also Georgia Institute of Technology Technical Report GIT-ICS-86/16.

[SSS81] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. An automatic technique for selection of data representations in SETL programs. *ACM Transactions on Programming Languages and Systems*, 3(2):126–143, April 1981.

[Sut63] Ivan E. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference*, pages 329–346, Detroit, Michigan, May 1963.

[SY83] Robert E. Strom and Shaula Yemini. NIL: An integrated language and system for distributed programming. In *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, pages 73–82, San Francisco, CA, June 1983. ACM. Also SIGPLAN Notices, 18:6, June 1983.

[TvR85] Andrew S. Tanenbaum and Robbert van Renesse. Distributed operating systems. *Computing Surveys*, 17(4):419–470, December 1985.

[WCC$^+$74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

[Wir77] Niklaus Wirth. Modula: A language for modular multiprogramming. *Software — Practice and Experience*, 7:3–35, 1977.

[WLS76] William A. Wulf, Ralph L. London, and Mary Shaw. An introduction to the construction and verification of Alphard programs. *IEEE Transactions on Software Engineering*, SE-2(4):253–264, December 1976.