Managing Permanent Objects

Nathaniel Mishkin

YALEU/DCS/RR-338
November, 1984

Copyright © Nathaniel Mishkin, 1984

Contents

1 Introduction 1

1.1 An example 1
1.2 Traditional approaches 2
1.3 Distributed computation 4
1.4 Applications 4
1.5 Outline of the rest of the thesis 5

2 Problems 7

2.1 Permanent data 7
2.1.1 Integrity and atomicity 8
2.1.2 Abstraction 8
2.1.3 Storage control 9
2.1.4 Sharing and concurrency 10
2.1.5 Security 12
2.1.6 Reliability 12
2.1.7 Performance 13
2.2 Reference 13
2.2.1 A first cut 13
2.2.2 Dividing the world 14
2.2.3 Dividing the world is not free 16
2.2.4 Reusability of references 16
2.3 Types and Code 17
2.4 Previous Work 18
2.4.1 Capability systems 18
2.4.2 Hydra 18
2.4.3 IBM System 38 18
2.4.4 Intel iAPX 432 19
2.4.5 Smalltalk 19
2.4.6 Eden 19
2.4.7 Object-oriented machines 19
2.4.8 APL 20
2.4.9 POMS 20
2.5 The Smalltalk - Hydra spectrum 20
2.5.1 Object size 20
2.5.2 Number of objects 21
2.5.3 Sharing of objects 21
2.5.4 Cost of dereferencing 21
2.5.5 Language integration 22
2.5.6 Summary 22
2.6 Message passing instead of object moving 22
2.7 Summary and approach of this work 23

3 Implementation **25**
3.1 The object model . . . 25
3.2 The Environment . . . 26
3.2.1 The T programming language . . . 26
3.2.2 The Apollo DOMAIN computing environment . . . 28
3.3 Introduction to the implementation issues . . . 29
3.3.1 OM within T . . . 29
3.3.2 Reference . . . 30
3.3.3 The structure of data structures . . . 31
3.3.4 Local and non-local references . . . 31
3.3.5 RPointers within T . . . 33
3.3.6 Object code . . . 33
3.4 Heaps . . . 34
3.4.1 Heaps in plain T . . . 34
3.4.2 OM Heaps . . . 34
3.4.3 Naming OM heaps . . . 35
3.4.4 Active heaps . . . 36
3.4.5 OM heaps in T . . . 37
3.4.6 Summary of heap features . . . 37
3.5 Intra-heap references . . . 38
3.5.1 Active objects . . . 38
3.5.2 An example: OM pairs . . . 39
3.5.3 Arguments to OM procedures . . . 40
3.6 OM types: Introduction . . . 40
3.6.1 Non-extends . . . 41
3.6.2 Extends and type identifiers . . . 41
3.7 Inter-heap references . . . 42
3.7.1 Non-local references and garbage collection . . . 42
3.7.2 LPointers in detail . . . 44
3.7.3 Making LPointers . . . 45
3.7.4 Dereferencing LPointers . . . 46
3.7.5 Comparison with Bishop's ORSLA . . . 47
3.8 Concurrent access to heaps . . . 48
3.8.1 Controlling concurrency . . . 48
3.8.2 Garbage collection . . . 49
3.9 Heap structure in detail . . . 50
3.10 OM Types: More details . . . 52
3.10.1 Getting code into T . . . 52
3.10.2 User-defined types . . . 53
3.10.3 OM operation dispatch . . . 56
3.10.4 Type redefinition . . . 56

4 Programmer interface **59**
4.1 Simple syntactic tools . . . 59
4.2 Programming with two kinds of references . . . 60
4.2.1 The dynamic approach . . . 61
4.2.2 The static approach . . . 62
4.3 A pre-processor . . . 64
4.4 The mixed object environment . . . 66
4.5 Finding the first reference . . . 67
4.5.1 File systems . . . 67
4.5.2 File systems as a model for OM naming . . . 67
4.5.3 A general naming strategy . . . 68
4.5.4 Naming in the current implementation . . . 68

4.6 Sample applications 69
4.6.1 OM/UMail 69
4.6.2 Naming server 70
4.7 A more ambitious scheme 72
4.7.1 Active References 72
4.7.2 A smart compiler 73
4.7.3 Active references and heap activation 75
4.7.4 Object allocation 75
4.7.5 Benefits and costs 76

5 Conclusion **77**
5.1 Reviewing the problems and their solutions in OM 77
5.1.1 Integrity and atomicity 77
5.1.2 Abstraction 77
5.1.3 Storage control 77
5.1.4 Sharing and concurrency 77
5.1.5 Security 78
5.1.6 Reliability 78
5.1.7 Performance 78
5.1.8 Reference 78
5.2 Design Philosophy 78

List of Figures

2.1 Multiple table mapped address dereferencing 15

3.1 T reference format 27
3.2 Three objects in two heaps 32
3.3 RPointer representation 33
3.4 Sketch of the garbage collector 43
3.5 An inter-heap reference 46
3.6 Heap format 51
3.7 Index element format 52

4.1 An LPointer to two pairs 62

Acknowledgements

I would like to thank my advisor, Mike Fischer for his insights and assistance. He helped to define my task and show me the path from the kernel of a thesis to its present, completed form. I would also like to thank the other members of my committee, Paul Hudak and Alan Perlis. Paul's positive attitude about my being able to finish this work was a great help. Alan's contribution to my understanding of computer science goes far beyond his assistance in this thesis. His seminars and our conversations in general exposed me to the broad range of issues in the field.

Much of my six years at Yale were spent systems hacking; i.e. building large, practical, useful (and used) software systems – mail systems, text editors, networks, etc. – largely for the fun of it. During this time I had the wonderful experience of working with Steve Wood, John Ellis, and Bob Nix. Though our personalities are quite different, we became good friends and worked amazingly well as a team – our compulsiveness and shared programming ethic bound us together. I learned at least as much of what I consider to be the important parts of computer science from and with them as I did from all other sources combined. Steve deserves special mention for being the mentor (if only for a short time) in systems hacking for the rest of us.

As relates to this thesis, I am particularly grateful to John Ellis for his endless comments and queries. The ideas embodied in section 4.7 are a direct result of several conversations with John. Bob Nix was a source of inspiration in that he was the first of the four of us to finish his dissertation and, as a result, shamed me into finishing mine. (Udi Shapiro deserves the credit for shaming Bob into finishing his. Hopefully I can shame others.)

John O'Donnell, in addition to creating a top-notch computing facility and helping me in this work, was (and is) a good friend and made my time at Yale more pleasant. Mary-Claire van Leunen enlivened my experiences and gave me a small inkling that writing prose is something I know almost nothing about. Judy Martel, Eric Key, Jonathan Rees, Norman Adams, Margot Flowers, and Mike Dyer all made my time at Yale very enjoyable.

I am indebted to Jonathan Rees and Norman Adams, creators and developers of the T programming language, upon which the work described in this dissertation is built.

I doubt that I would be where I am today if back in 1971 Mrs. L. Yelman, my high school math teacher, hadn't had the foresight to teach computer science and lobby for computing resources at South Side High School. It was BASIC and Teletype model 33s, but it was all I needed to get me going.

Finally, the constant support, patience, and love of my wife, Judy, has been an endless source of amazement and joy. She probably didn't think it would take me quite this long to finish, but she didn't complain as the years stretched on. I dedicate this work to her.

Chapter 1

Introduction

Programmers are often confronted with the problem of writing programs that need to manipulate (create, access, modify, delete) permanent objects (data structures). By "permanent objects" we mean objects that live longer than one invocation of a program. These objects must be stored in the computer's file system.

Generally the capabilities of file systems and the tools for manipulating file systems are primitive. File systems present only the simplest of data types (e.g. one-dimensional array of characters). More complex data structures can be built on top of these simple data types, but the implementation time is significant. As a result, a programmer is not inclined to think that he is dealing with a permanent object when he really is. He simply views his programs as reading a file, constructing some transient data structures in main memory, reading or modifying those data structures, and possibly rewriting the file that was read as the first step. The programmer is not encouraged to view the disk file merely as a data structure in another guise. Often the format of the output of the program is designed to be useful for human readers of that output in spite of the fact that the only person who is likely to read it is the programmer himself while debugging his programs.

In this thesis we will be concerned with the issues of creating, modifying and administering permanent objects in T [44,46], a dialect of Scheme [52], which is in turn a dialect of Lisp [39]. The goal of our work is to blur the distinction between permanent and non-permanent objects; i.e. to make the writing of programs that manipulate permanent objects nearly as easy as the writing of programs that manipulate non-permanent objects. We will describe the design and implementation of a programming system that allows permanent objects to be accessed using primitives that are analogous to the primitives used to access non-permanent objects. The system we will describe has been built and used for non-trivial applications.

The work described in this thesis differs from previous work in permanent objects in that it supports a potentially very large set of objects of both small and large size, and it allows these objects to be accessed by different users and application programs.

1.1 An example

Consider a user's electronic mail box. Within a program that manipulates a mail box there is a mail box data structure that might consist of a linked list of mail message objects. Each mail message might consist of: a string containing the text of the message; some boolean flags indicating, for example, whether the message has been read by the user; a pointer into the text of the message where the message headers begin; and a pointer into the text of the message where the message body begins.

The mail box is of interest to at least two programs: a mail user interface program that lets a user

read and modify the contents of his mail box; and a mail delivery program that adds new, incoming mail to the mail box. These programs may be invoked multiple times to manipulate the mail box. The mail box exists independently of the programs that access it.

A typical implementation strategy taken by a programmer (e.g. as in OZ [19], a mail user interface for the DECSYSTEM-20) who does not view a mail box as a permanent object is this: the mail user interface reads in a text file that contains all the user's messages. The program breaks the file into individual messages. Depending on the conventions of the mail system, the messages may be separated by some sequence of characters that are guaranteed not to appear in the text of a message; or each message may be preceded by a text string of digits which when interpreted as an integer specifies the length in bytes of the message that follows. Once broken up into individual messages, the program allocates objects to hold the messages and links the messages together to form the entire mail box data structure as described above. Perhaps the first line of each message contains a string of ones and zeros indicating the values of the various message flags. This string will have to be parsed into boolean values and stored in the appropriate slots in the message data structure.

The user interface program manipulates the mail box object in response to user commands. When the user exits the program, the program re-writes the text file to reflect the new state of the mail box. This procedure consists of traversing the mail box data structure and writing its contents in the format expected by all programs that manipulate the mail box file.

The most serious problem with this approach is the cost of parsing the file on program startup and formatting the file on program termination, especially as the size of the mail box increases. Our goal is to demonstrate that, given the right tools, the programmer *can* think of something like a mail box as a permanent object and that as a result, programs that manipulate the object can be simpler to write and more efficient in execution.

1.2 Traditional approaches

There are at least two traditional approaches for dealing with permanent objects. For studying both these approaches, it will be convenient to think of the objects as having two representations: internal and external. The internal representation is the format of the data structure when it resides in main memory; the external representation is the format of the data structure when it resides in stable storage (e.g. disk files in a conventional file system).

The first approach consists of writing an ad hoc set of subroutines that convert from the internal representation to the external representation and set of routines that do the reverse conversion. This approach is marginally better than the one taken to solve the mail box problem above.

The second approach is to interface the application program that needs to use permanent data structures with an existing "database manager." We use the term "database manager" in a very general way to mean a set of programs or subroutines that have been designed to store and retrieve data from a file system.

In cases where the internal representation is simple (e.g. a character string or a vector of integers), the temptation is great for a programmer to use the ad hoc solution. He says: "I don't want to get involved with the complexity of such-and-such database system. I'll just write my strings/numbers/etc. out to a simple text file." Unfortunately, this seductive reasoning results in a program that is not only not as fast as it might be (due to the representation conversions), but one that is also hard to modify and hard to extend. Having implemented one ad hoc solution, the programmer is unlikely to want to implement another one (or modify the existing one) in order to accomodate increased functionality. As a result, the functionality does not get implemented.

What are the arguments in favor of using an existing database manager? A clear advantage is that much of the programmer's work is already done for him by the database manager. The programmer need not be concerned with the details of the file system. Most sophisticated database systems offer

some degree of reliability in the face of hardware failure. Database managers take care of storage allocation.

Unfortunately, interfacing to a database manager may introduce some problems. The program interface to the database manager forms an "embedded language." That is, the set of calls by which the application program accesses data maintained by the database manager is a language of its own. This language is built on top of the language in which the calls to the database manager are written. As a result, the programmer is no longer programming using solely the primitives of the base programming language. In fact, the primitives of the base programming language may not even be applicable to the application's data, which now resides in the world of the database manager.

Thus, taking the database approach to solving the permanent object problem obliges the programmer to work in two languages: the base language and the database embedded language. Often this complexity is great enough to dissuade the programmer of a medium-size application from using the database manager.

Creating and using embedded systems is not always bad. In most large programming projects one ends up constructing and using some sort of embedded language. Some languages support such embedding better than others (e.g. Lisp systems generally have a powerful macro facility). Even in languages that do not allow modification to their syntax, the subroutines that the programmer defines for use by himself, but especially for use by other programmers working on the same project, define the semantics of a language. When a programming project adopts a set of conventions and interfaces that make up the specification of an embedded language, the comprehensibility of the overall project increases; functionality can be expressed in terms of the embedded language instead of in terms of the base language.

There is a key difference between embeddings such as the ones that go on all the time and the embedding of a large database system. In the former case, the programmers in the project design the embedded system themselves, to their own specifications. In the latter case, the embedded language is typically not under the control of the project that uses the database. As a result, the programmer may be forced to use an embedded language that is not at all appropriate to his application.

A significant limitation of both the ad hoc and the database approach to storing permanent data is that they are unable to deal with pointers. By "pointer" we mean the traditional programming language construct that allows indirect reference to data. Since pointers are convenient tools for the programmer, it is undesirable that they should be unavailable when storing permanent objects.

The limitations of the traditional approaches outlined above become clear when dealing with even simple data structures. For example, Lisp has a primitive procedure called *map* that applies a procedure to a linked list of objects. Lists are easy to create and *map*, and other procedures provide a clean and convenient mechanism for accessing the data in the list. Use of lists in Lisp programs is pervasive; use of lists in external representations is unusual.

If the linked list is maintained within the database manager, two problems can arise. The first problem is that the database manager might not export references to the middle of a linked list. That is, the database manager might export references to individual data items, but not to data structures that it views as being internal to the database system. As a result, there is no reference that can be passed as the procedural argument to *map*.

A second problem that can arise is that even if the application can obtain a reference, the list that is constructed and maintained by the database manager might not be manipulatable by *map* (and the elements of the list by the procedural argument to *map*) because of differences between the representation maintained by the database manager and the representation expected by the Lisp system. The cost of this representation conversion is unacceptably high. In the permanent object system we built, representation conversion is not necessary.

The goal of the work described in this thesis is to develop a system for managing permanent objects

that is more general than the ad hoc methods but less cumbersome than the methods that require interfacing to a database system.

1.3 Distributed computation

A secondary goal of this thesis is to demonstrate how a system for supporting permanent objects can aid in the development of distributed applications. By "distributed application" we mean an application, parts of which run at different times on different processors connected by a high speed local network. Recently, much research has been concerned with the problem of being able to take advantage of the newly available small processors (e.g. the Motorola 68000) configured in a network in order to make applications run faster or more reliably. Much of this research has addressed concurrency problems: if there are multiple processes running on multiple processors accessing the same data (or replicated copies of the data), how do you coordinate their activity to insure the integrity of the data?

Before one can address the issues in controlling concurrent access to data, it is first necessary to consider the problems in simply accessing the data. The issue of making the data available to the multiple processes has been discussed elsewhere, but not to the level of detail necessary to illuminate the hard problems that arise in a real implementation.

Though distributed computing is not the main topic of our work, we designed our permanent object system and built our implementation in a way that does not preclude the later introduction of sophisticated concurrency control mechanisms. Our current implementation has rather course-grained concurrency control. However, even this level of control is useful for distributed applications where concurrency is low – i.e. where conflicting requests for access to data occur infrequently. For example, the applications in our mail system example – the mail user interface and the mail deliverer – are examples of applications that might be distributed among a number of processors. In this case the expected degree of concurrency is low, and simple concurrency control techniques (e.g. waiting for a file to become unlocked) are sufficient to solve the problem.

1.4 Applications

A permanent object system has many potential applications in addition to the mail system example given above. We list some applications that deal with permanent data, describe existing implementations, and descibe how a general permanent object system could be used in an implementation.

- Compiler auxiliary files.

The T compiler produces a *support file* that contains all the macro and constant definitions in the module being compiled. The support file can be referenced in other files so that when those files are compiled, the information from the support file can be used to produce more efficient code. Presently in T, support files are text files containing printed T expressions. The compiler must read and parse the entire support file when it is referenced from the file being compiled. Using a permanent object system, the data structures describing the macro and constants definitions could be permanent objects and accessed more quickly. We could take this path further and replace source text files themselves with permanent objects describing the program source.

- Text formatter database.

The Scribe document preparation system [47] uses a set of database files describing output devices, document formats, and bibliographies. These files contain text string Scribe commands. When a reference is made to a particular device, document format, or bibliography from a document being formatted, Scribe must linearly scan one or more of the document text files. This scan can be very expensive, especially in the case of large bibliographies. Using a permanent object system, the database could be represented as a set of permanent objects and accessed more efficiently.

- Registry of users.

The Unix [13] system for registering users is a text file containing one line for each user. Each line contains (among other things) a user ID, password, and full name. Any applications that need the information must read and parse the text file. Mechanisms to control concurrent access to the registry would be useful but are non-existent, hence exposing the system to data corruption. The file has a rigid format and the presence of programs that rely on the format makes it difficult to extend the registry to hold new kinds of information about users. The rigidity is partially a result of the ad hoc way the data is stored and accessed. Using a permanent object system, each user could be represented as a permanent object and the entire registry as a permanent collection of those objects. The objects could be designed to allow both extensibility of information about users and concurrency down to the individual user level.

- On-line help systems.

The on-line help system used on the DECSYSTEM-20 at the Yale Department of Computer Science consists of a text file (called the *index file*) that contains a list of indices (words) and help file names. Users query the system using an index and the system responds by offering to display the contents of the help files associated with that index. Whenever the index file is modified (by a help system administrator), a binary file must be produced (by running a special program that converts the index text file text to an index binary file). The format of the text file is designed to simplify the administrator's job. The format of the binary file is designed to make the help system programs run efficiently. Using a permanent object system, there would be no need to have two representations (text and binary) of the index file. The index could be a permanent object that could be accessed both by the help system programs in response to users' queries and by help system administrators to change the contents of the index.

All of these applications involve permanent, structured data that must be changed in a controlled way. Many existing implementations of such applications use inefficient techniques (such as those that require unnecessary parsing and formatting of data) or ungeneral mechanisms designed for a single application. An efficient, general, and simple object management system will improve the performance of such applications and encourage programmers to write more such useful applications.

1.5 Outline of the rest of the thesis

The focus of this thesis is a system we call *OM*, a system for managing permanent objects. We designed and implemented a running version of OM. We also designed and implemented two sample applications systems that run using the facilities provided by OM.

Chapter 2 covers the problems associated with building a system that meets the goals described above. Chapter 3 discusses the implementation of OM. Chapter 4 discusses how programmers write application programs using OM; in this chapter we also describe the sample application programs we built. Chapter 5 summarizes OM and discusses how well it solves the problems raised in chapter 2.

Chapter 2

Problems

In this chapter, we outline some of the problems faced by a system that maintains permanent objects. Our basic model for the computing environment in which permanent objects are maintained is traditional: we assume a CPU with a fast main memory of limited size and a larger, slower disk memory. Data is transferred back and forth between disk and main memory in relatively large units (compared to the smallest units the CPU can deal with) and at a relatively slow rate (compared to the rate at which the CPU can access main memory).

2.1 Permanent data

Many of the problems that arise from wanting to preserve objects result from the fact that since objects can be manipulated only within main memory and since main memory can not hold all the permanent objects, there needs to be a controlled, reliable mechanism for moving data in and out of main memory. The experience gained in designing virtual memory and database systems is relevant to the understanding and the solving of these problems. A permanent object system of the sort we've outlined can use techniques from both virtual memory systems and database systems. Virtual memory systems provide a model of how to refer to objects that are "not really there". Database systems offer examples of how to deal with the permanence issues.

We will discuss the following topics in permanent data:

- Integrity and atomicity
- Abstraction
- Storage control
- Sharing and concurrency
- Security
- Reliability
- Performance

In our discussion of these problems, we will be giving each problem only a short characterization. The orientation will be very practical since we are interested in how they relate to the system we have actually built. In designing this system we have tried to be practical so that the the system could be actually built. Later, we will discuss how our system addresses these problems.

2.1.1 Integrity and atomicity

By ***integrity*** we mean the functionality that insures that the permanently preserved data is not corrupted. What are the major potential sources for such corruption?

The most obvious source of corruption is a machine crash. (In addition to actual machine crashes, abnormal termination of individual processes or failure of pieces of hardware (e.g. disk or network communication hardware) can cause problems similar to a crash.) Some of the permanent data may have been in the main memory of the crashed machine. If the main memory copy of the data contained changes that were not yet reflected in the copy of the data maintained in stable storage, then applications that use the data could be in trouble.

For example, suppose some large data structure is being modified when a crash occurs and also suppose that only part of the modified structure has been rewritten to stable storage before the crash. Assume that parts of the data structure contain related information – e.g. a string of characters and an integer indicating the length of the string. Suppose the part of the data structure containing the integer length got written to stable storage but the part containing the characters of the string did not. Then an application program that accesses that string might access too few or too many characters. (In the latter case it would presumably see "garbage".)

Another source of corruption is program error. In the course of application program debugging (or later when some unforeseen bug arises in production use of the application) the application might present some logically inconsistent pieces of data for permanent storage. The problem here is in defining what "logically consistent" means. If the permanent data storage system is to reject certain pieces of input then the consistency rules must be specified and be part of the system. Unfortunately, the specification of the data consistency rules may be non-trivial (and a task in which the programmer may be unwilling to engage). In addition, if the data storage system is to be relatively simple, modular, and efficient, it may not be easy for it to maintain the set of rules for a large set of applications.

The traditional approach to maintaining integrity of permanent data is to use techniques which guarantee the ***atomicity*** of a set of changes to data. Atomicity is a property that implies that if any of a set of changes are made (i.e. made to the permanent copy of the data in stable storage), they all are made. If for some reason the system fails in the middle of a set of changes, the system guarantees that it appears that none of the changes have been made. There are various implementation techniques that can be employed to assure atomicity when requested. As will become apparent later, these techniques are not easily applicable to the system we design. This is a limitation of our current system, but since our goal is to gain experience with a real permanent object maintainance system, we are willing to tolerate the potential for loss of integrity for experimental purposes.

2.1.2 Abstraction

It should be the goal of any data storage system to provide some level of abstraction. For our purposes, an abstraction is a mechanism that does two things:

- It translates logical references to data into physical references to the data itself.
- It hides details of the representation of the data (e.g. how many bits are allocated to what) from the programmer.

By logical reference, we mean the name of a field in a structure or a key into a table mapping logical references into physical references. "Physical reference" is a relative term. What we really mean is "less logical reference". That is, in a system that presents layers of abstraction, only the bottom layer can be considered to be addressed by physical references (e.g. physical main memory address or disk block address). Each software layer above the bottom layer uses references that are logical with respect to references used in the layer below. If layer A is below layer B, a major function

of layer A is to translate layer B's logical references into layer A's less logical (i.e. more physical) references.

For example, at one layer, a reference might be a person's last name represented as a string; this reference might be passed to a lower layer that is supposed to display information about the referenced person; this lower layer in turn maintains a table mapping strings into integer object identifiers. This layer might deal with references to objects other than people; i.e. it is a layer with multiple immediate higher layers that all call the lower layer with strings as references. A still lower layer maps the integer identifier into some disk address that indicates where the information about the object resides. This layer too might have multiple immediate higher layers.

The overall problem of choosing the form of references and designing the translation mechanism is critical in any data storage system; our approaches will be discussed later.

Another role of abstraction is to hide the representation and implementation of a data structure in one part of a system from another part of the same system. The purpose of this sort of abstraction is to (hopefully) allow changes to the representation or implementation to be made without having to scour the entire system for places where a programmer has "cheated" by employing some piece of information about the data structure which, by the "official" specification of the interface with which he is supposed to work, he is not entitled to employ.

For example, suppose some module of a system chooses to implement sets as linked lists; this module exports subroutines that manipulate sets, but it does not "reveal" that sets are actually lists. If the client of the module always uses the subroutines provided by the set module, the client is unaffected if the set module is changed to represent sets as bit vectors. If, however, the client *does* rely on the fact that sets are implemented as linked lists, he violates the set abstraction and hence when the implementation of that abstraction changes, the client breaks.

2.1.3 Storage control

A system that maintains data permanently must deal with the issue of controlling the allocation of storage occupied by the data. The system must be able to allocate blocks of storage of varying sizes and it must be able to know when storage occupied by data has become "free" – available for allocation to another piece of data.

The literature is full of techniques for allocating storage. (Knuth's work [30] is a standard reference for these techniques.) Some techniques require that data storage be explicity freed by the application that owns the data occupying the storage. An alternative technique is *garbage collection*. Garbage collection is a process that separates the space of objects into garbage and non-garbage. An object is garbage if there is no way to obtain a reference to the object; otherwise the object is non-garbage. The literature contains many descriptions of garbage collection techniques. (Cohen's survey [15] contains an excellent summary of these techniques.)

The main advantages of using a storage control policy that relies on garbage collection are:

- Allocation can typically be done very quickly.
- There is no *dangling reference* problem.

In a garbage collection based storage system, storage can be allocated out of a monolithic heap (i.e. a storage pool with no internal structure). The state of the storage pool consists of an index (called a *heap pointer*) into the heap. The allocation procedure consists simply of advancing the heap pointer by the number of storage units requested by the caller and then returning the old heap pointer to the caller. Such a procedure can be implemented in a few machine instructions and hence can be open-coded, avoiding the cost of a procedure call to the allocation procedure.

In storage systems that are not based on garbage collection and hence rely on the explicit freeing of storage, the dangling reference problem can arise. A dangling reference is a reference from one data structure to another where the reference is to a piece of storage that has been previously explicitly

freed. This is a problem since the freed storage may be reallocated to some new data structure and the dangling reference would then refer to something other than it is supposed to. In a garbage collection based system, since there is no explicit free operation, there is no way for a reference to be dangling. The garbage collection procedure is defined in such a way that any reference to a object ensures that the data object will not be freed.

However, there is a serious drawback to depending on garbage collection. While it at first appears that allocation is cheap, to be fair one has to the factor in the cost of the garbage collection. Such a factoring produces a more accurate cost of the allocation operation. Also, in traditional garbage collectors, while the garbage collector is running, no other part of the program can run. If garbage collection takes a long time and it occurs frequently enough, this time can be intolerable. However, recent work in incremental and parallel garbage collecting strategies lessen some of the pain garbage collection causes [11,18,1,26,27].

2.1.4 Sharing and concurrency

By the ability to share objects we mean that nothing about an object restricts it to being used by one user, or one application program, or one process.

When we say a set of processes run concurrently, we mean that all the processes are active and runnable over some period of time. By concurrent access to objects we mean access to objects by concurrent processes. We will use the term *concurrency* to mean the measure of concurrent access to objects. The degree of concurrency is determined by how many processes are competing for access to a set of objects over how long a period of time. We say there is a high degree of concurrency if a large number of processes want access to a similar set of objects over a short period of time. We say there is a low degree of concurrency if a small number of processes want access to a similar set of objects over a long period of time.

A system that supports sharing need not necessarily support a high degree of concurrency. Enabling concurrency does require that the problems of sharing have been solved.

Let us first consider the problems related to sharing per se. The main problem here is that all information about an object must be accessible from a reference to the object. No information about the object can be encoded in procedures that are known only to some user or application program. Also, the format of references to objects (section 2.2 discusses the issue of reference format in detail) must not rely on a particular user's or application's context.

For concurrent sharing, let us first consider multiple processes sharing a single main memory. We assume for reasons of correctness and efficiency that the system should allow just one copy of a particular object in main memory no matter how many processes are sharing that object. The implementation of sharing depends on the lower level memory architecture of the underlying operating system. On operating systems that do not support virtual memory, the implementation is easy: translate identical references from different processes to the same object into the same address in physical memory where the object has been read.

Operating systems with virtual memory support come in two varieties: (1) ones in which all processes run in the same virtual address space (which is larger than the amount of physical memory on the machine); (2) ones in which each process runs in its own separate virtual address space. In case (1) the implementation of sharing is the same as in a system without virtual memory.

In order to be able to implement the sharing of objects in case (2), the system must support primitives that allow the manipulation of the process page map. That is, it must be possible to arrange the page map of two processes so that references to some set of virtual addresses in one process produce the same values as references to a possibly different set of addresses in another process. Given these primitives, the system can arrange that there is one copy of the object in main memory and that all references to it from all processes point to the single copy of the object.

Given the ability to manage processes state and main memory as described above, concurrent read

access to objects presents no particular problems. The real problems of sharing arise when either (1) one or more processes want to be able to *modify*, not just *read* the data; or (2) multiple processes wishing to read or modify the data do not share a common main memory. The problem raised by case (1) is mainly one of semantics. The problem raised by case (2) is in addition one of implementation efficiency.

Assuming one is willing to accept sometimes unpredictable behavior, there is nothing preventing the implementation of shared objects in a single main memory with one or more writers being the same as the implementations of the read-only case described above. Changes can be made by any process that has a reference to the object and those changes will be visible to other processes with a reference to the object. For some limited set of applications, this laissez faire approach is acceptable. For example, suppose the shared data consists of an integer that needs to be incremented when a particular event occurs in any one of a number of processes. If the machine has an atomic increment-memory instruction, then this implementation will work fine.

In general however, if there are to be writers coexisting with other writers or readers, there must be synchronization in order to allow the predictable and correct modification of data structures. For an example of how lack of synchronization can lead to problems, consider the following classic update problem: suppose one process is traversing a list of objects representing a list of employees and modifying the salary field of each employee based on some formula (say to account for inflation); suppose also that at the same time another process is modifying a single employee's salary to account for a raise because the employee has been promoted. The two processes might clash in the following way: suppose both processes (being unconstrained by any synchronization mechanism) fetch the salary field for the employee being promoted. The first process computes the new salary and stores is back into the person objects; the second process nearly simultaneously computes the raise and stores that new salary back. Instead of the employee ending up with an increase in salary due to both inflation and promotion, he gets only one increase (ignoring this sort of interaction, there is the issue of which increase computation should be done first, but that is not a synchronization concern).

One obvious way to deal with this sort of concurrency problem is to make all requests for modifications go through a single process (often called a *monitor*) which is the only one that can actually modify the object. This sort of solution has two problems. First, it limits concurrency – all modification requests are forced to line up and be executed serially. This problem can be ameliorated by having multiple modifier processes each of which is responsible for a disjoint set of objects. Unless you have one process per object,[1] concurrency may still be limited. Another problem is that the cost of modifications goes up tremendously; it is now much more expensive to modify than read data. Some database systems are constructed in this fashion, and in fact both writes *and* reads go through an intermediate process. Since we are designing a system that is supposed to make accessing and modifying permanent objects as similar as possible to accessing and modifying transient objects, we consider it unacceptable to have such an intermediate process.

An alternative traditional technique for dealing with concurrency is to use *locks* (such as semaphores). (There are many language constructs in existence and proposed for dealing with synchronization, but they all ultimately rely on locks.) A lock controls what set of processes have what kind of concurrent access to a piece of data. The lock can be specified to allow multiple readers and no writers, or one reader/writer and no other readers, or multiple readers and writers (the unconstrained case above). Since there is both time and space overhead to each lock, a single lock may control more than one piece of data in order to reduce the overhead. The *locking granularity* says how small a set of objects need to be locked in reality in order to lock just one object. A system with small granularity is one with the potential for high concurrency – since the number of objects locked with one lock is low, the chances that other processes can work on other, unlocked data is high. Conversely, large granularity can potentially limit concurrency. Thus, lock granularity is traded off against potential concurrency.

Now let us consider the issue of concurrent sharing when the multiple processes do not share a

[1] Hewitt [25] proposes such a system, but it is not clear how practical it is.

common main memory. Note that in the case of shared main memory, the thing that enabled sharing to be implemented easily is that the same processor using a single memory can implement a memory reference relatively efficiently. While on virtual memory systems the cost of implementing a memory reference is somewhat higher than on non-virtual memory systems, the cost is still tolerably low. Trying to extend the "virtuality" of memory to non-shared physical memory is not likely to result in acceptable performance. That is, one can imagine making the memory reference operation work over a network of computers each with its own private memory. However, real implementations of systems with such a facility have never been entirely successful. At best, the programmer has been forced to be aware that some memory references (i.e. ones to local memory) are cheap, and others (i.e. ones to another computer's memory) are considerably more expensive.[2]

For the practical purposes of building an implementation on conventional machines, we chose to disallow concurrent sharing from multiple processes using disjoint main memories. This is a limitation, but not one that is impossible to live with because one can often divide a problem so that the processes that need to access data concurrently can share main memory with each other. Also, even when processes must run in disjoint memories, it is often possible to partition a data structure so that parts that have no inter-dependencies can be manipulated in separate memories.

2.1.5 Security

For many applications it is important that a permanent data storage system provide security mechanisms. That is, it should provide a way of allowing some users to have one kind of access and other users to have another kind of access. There are two issues to be addressed in this area: (1) What is the granularity of the specification of the class of users? (2) What is the granularity of the data to which a single security specification applies?

The issue of the granularity of the specification of the class of user basically comes down to this: how many bits of specification do you want to allocate to identifying users? Ideally the specification should allow different access to be specified for each distinguishable user. If there are a lot of users, this will require a lot of bits. If this specification has to be duplicated for each object to be protected, then this form of specification is unacceptable. If however multiple objects that are to be protected identically can share the same protection specification, we are less likely to worry about the length of the specification. The space of possible protections is large, but in practice the number of different protections used is relatively small compared with the total number of objects being protected. The situation is further helped if users can be characterized as being members of a class (say, systems programmers) rather than individuals. Then the protection applicable to an entire class of users can be expressed simply by referring to the class instead of to each individual user.

Intertwined with the issue of how protection is specified is the issue of how small a set of objects can be protected by a single specification. Even if we use a scheme in which different objects can share the same specification, we still need a way to express which specification we want. If we use, say, a 32 bit integer to identify which specification we want, it is unlikely that we would want to protect sets of objects as small as or approaching 32 bits in length since if we did, the storage overhead of the specification would be as large or nearly as large as the data itself. For practical purposes, it is usually acceptable to allow the size of the set of data to be protected to be relatively large.

2.1.6 Reliability

Reliability is a measure of how long a system runs without failure. Researchers in the field have made many suggestions about how programming projects can produce more reliable systems. It is not clear what the practical implications of this research are however. For our purposes, we will have to rely on our intuitions about reliability. For instance, we know that a system that spreads one set of logically related objects over multiple disk drives is prone to reliability problems – as

[2] C.mmp [54] is an example of a system with this property.

the number of mutually dependent pieces in a system increases, the chances that the failure of any individual piece affecting the reliability of the system increases.

2.1.7 Performance

Performance is a measure of how fast a system runs and how much space it uses. If programmers are to use permanent objects the way they use objects in a traditional programming environment in which objects live only in main memory, the performance of routines that create and access permanent objects must be similar to the performance of analogous routines in the traditional environment. It is easy to let the cost of the operations in the permanent environment creep up. By doing so, the permanent object system begins to look like a traditional file system as programmers recognize the performance problems and use I/O techniques (like buffering) to improve performance.

2.2 Reference

Given the desire to maintain objects permanently, one needs a way to refer to those objects. The object reference can be thought of as the object's name. There are a number of questions that arise in designing a reference mechanism:

- What is the form of the reference?
- What is the mechanism and cost of dereferencing (i.e. the procedure that obtains a piece of an object given the object's reference)?
- How many layers of reference does the system provide?
- How does the underlying hardware affect the choice of reference?
- What is the programmer's and the user's view of the reference?

The first thing to note is that an object reference is ultimately a string of bits. In this section we will discuss the issues associated with choosing the format of that string and the mechanisms for dereferencing given the bit string.

2.2.1 A first cut

Permanent objects' permanent home is on stable storage – a disk for instance. A natural first approach to the problem of choosing the reference form is to say that an object reference is simply the disk address at which the object begins. Suppose a disk address is simply an integer offset that indicates how far from the beginning of the disk the object being referenced is. Dereferencing then simply consists of reading the appropriate number of bytes from the disk into main memory where the object can be manipulated by the CPU. Let us refer to this as the *pure address* strategy.

What are the problems with the pure address approach? One problem is that since it is reasonable to assume that objects will tend to be larger than the interval between disk addresses, if our object references are disk addresses, then we are wasting bits. This is because even when the disk is is full of objects there will be disk addresses that don't correspond to the starting position of some object. Logically, these unused addresses represent bit patterns that could in principle be used as object references. We are not proposing that this scheme be modified to use those addresses, only that their existence implies that we are not getting full mileage out of the bits we have allocated to the task of making up references. Thus, if we have an N bit references, we are typically going to be able to make somewhat less that 2^N object references. Ideally, we would like to be able to get exactly 2^N references.

Another problem with the pure address approach is that it makes it difficult to move objects around. Objects might move around for several reasons: (1) garbage collection, (2) storage compaction

(without garbage collection), and (3) "logical reasons". By logical reasons we mean reasons that are not real requirements of the system. For example, suppose the system consists of many disks attached to many computers. Extend the notion of disk address so that the disks are arranged in some order and each disk is assigned a subrange of the entire disk address space. An example of a logical change is a user's request to move his set of objects to a disk that is attached to his computer instead of one attached to another computer. If we use the disk address scheme, then moving an object requires that all references in other objects to the object being moved must be updated to refer to the new address. In general, this is equivalent to garbage collection – the entire object space may have to be swept to find all the references.

A refinement on the pure address approach that solves the above problems is to have a table that maps references onto disk addresses. The reference assigned to an object is a key into the mapping table. The problem of unused bit patterns goes away because the reference can be any one of the bit patterns possible; the table is responsible for translating all valid bit patterns (i.e. patterns that have been assigned by the mapping mechanism) into disk addresses. The problem of moving objects also goes away. An object's moving is transparent to the holder of a reference because the only change that needs to be made is to the mapping table slot where the actual disk address appears. Let us call this the *mapped address* approach.

Let's look at this mapping mechanism in more detail. The obvious implementation is to have a vector whose length is the total number of objects (and by extension, references) we wish to allow. Dereferencing then simply consists of indexing into the vector at the position indicated by the reference and returning the disk address found at that slot. This vector must be placed at some known place on this disk. While simple, this approach obliges us to maintain a potentially large table many of whose slots may be unused if all the possible references are not being used at any given time. Each dereference requires that we read the disk potentially twice: once to read the disk address from the vector and once to read the data located at that disk address.

We want dereferencing to be fast – dereferencing is in the *inner loop* of all processing of permanent data. Slowing down dereferencing slows down everything. The mapping mechanism must be fast. As an optimization we can keep a copy of the mapping data structure in main memory. This saves us one of the disk accesses. Unfortunately, having the table in main memory makes us feel even worse about the table's size.

We could use a more sophisticated mapping mechanism like hash tables. One decides how big to make a hash table based on the expected number of keys (i.e. references) one needs to map into values (i.e. disk addresses). Thus, we can reduce the size of the table. However, the cost of looking something up in a hash table is considerably greater than the direct lookup that is done in a vector.

Any kind of mapping scheme that requires large parts of the mapping data structure to reside in main memory has two major problems. The first problem is one of reliability. We are keeping a data structure that is critical to maintaining the consistency of the complete system in volatile storage; if the system crashes, we're in big trouble. To reduce the potential for disaster, we can periodically copy the mapping data structure back to disk. Nevertheless, the risk remains.

A second problem with keeping the mapping data structure in main memory has to do with concurrency. Multiple processors that do not share a common main memory do not have equal access to the mapping data structure. It is likely that any mechanism that tries to simulate equal access will have serious performance problems; one processor will run quickly while the others run slowly.

2.2.2 Dividing the world

The cause of both the storage overhead and concurrency problems noted above is ultimately that the mapping mechanism is flat and unpartitioned. If we could break it up into smaller pieces then (1) the amount of mapping data structure that needed to be resident at any time would be reduced, and (2) multiple processors could run concurrently as long as they stayed in separate areas of the map.

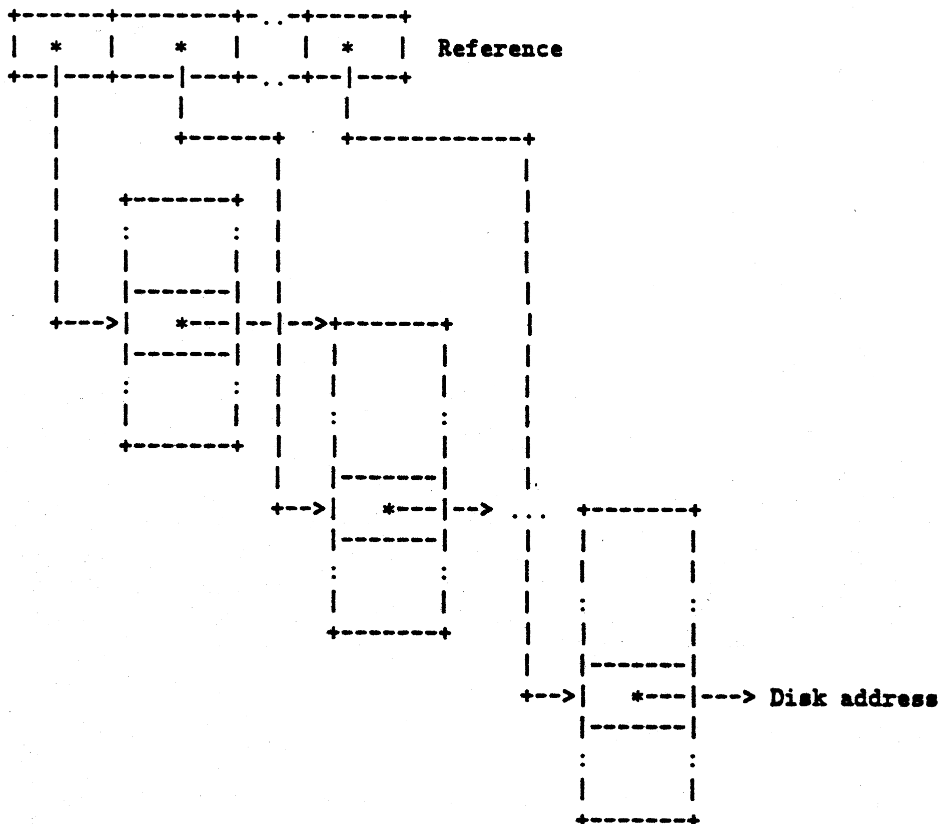


Figure 2.1: Multiple table mapped address dereferencing

The traditional approach for breaking up a mapping mechanism is to divide the reference bit string into multiple parts. This is a technique that is often applied in virtual memory systems. Each substring of bits is a key into a table. All but the last substring are keys into tables that map bit strings into table identifiers. The last table maps a key into a disk address. The first table is at some "well known" location. Let us call this scheme the ***multiple table mapped address*** approach.

Dereferencing in this scheme consists of breaking up the reference into the separate bit strings and then starting with the well known table, looking up each substring in successive tables (the location of each table is the result of the previous lookup) until the last substring is used. The last substring, instead of being an index into a table of table identifiers, is an index into a table of disk addresses. At any given time, only one of the tables has to be in main memory. In practice, references are broken up into just two or three pieces.

A problem with the multiple table approach is that even if all the tables happen to already be in main memory, we have to make as many memory references as there are tables in the course of just one full dereference operation. In virtual memory systems, this problem is partly helped by introducing special hardware that stores the last few references that were dereferenced along with the identifier of the final table used for each reference. Any future reference whose upper substrings match an entry in the special hardware table can skip the process of looking through all the tables and simply use the result saved in the special hardware table. This process is sometimes called ***translation lookaside*** (or ***translation caching***).

Let us now consider the issue of storage allocation in our simple disk address based object system. How is the disk space managed so that allocation is fast? If we want to rely on garbage collection

to reclaim free space, we can use the allocation mechanism described earlier – simply have a heap pointer that indicates the boundary between used and unused disk space. Unfortunately, now we have introduced a bottleneck analogous to the one introduced by our first simple mapping mechanism. The problem now is allocation instead of dereferencing and the bottleneck is the heap pointer (or in general whatever data structures are associated with the allocation process) instead of the mapping table. All requests for storage have to go through the allocation data structure.

Just as we broke up the mapping mechanism, we will now break up the allocation mechanism. The straightforward way to do this is to divide the entire storage pool into pieces and associate separate allocation data structures with each piece. Let us call a piece of the entire storage pool a ***heap***. In fact, it will turn out to be convenient if we break up the storage pool along the same lines as the broken up mapping mechanism. That is, the last table in the set of mapping tables will contain disk addresses that are in just one of the areas of the disk (i.e. storage pool).

An advantage of the heap approach is that now instead of storing full, presumbly long, disk addresses in the table, we can store just the offsets from the beginning of the heap on the disk; one entry in the table contains the base address (a full disk address) of the heap covered by the table. In addition to being small, another advantage of offsets is that they are position independent. That is, if necessary, we can move a heap (say to another disk) without having to change anything except the base address. Another advantage that we will go into detail on later is that if a few more changes are made to the strategy, it will be possible to do partial garbage collections, i.e. garbage collection of a heap rather than the entire storage pool. This means that one of the onerous aspects of garbage collections – the long time to do garbage collection – can be somewhat ameliorated.

2.2.3 Dividing the world is not free

Note that as a result of the divisions in the reference and allocation structures, we have introduced the problem that there will be some set of references that will not be used. How does this happen? Without loss of generality, assume that the reference is divided into just two parts. The first part is conceptually a reference to a heap; the second part is a reference to a particular object within the heap. The maximum number of objects in a heap is fixed by the size of the second part of the reference. We expect that the assignment of objects to heaps will not be random with respect to the meaning of the objects – that for reasons that will be elaborated on later, programs and users will place logically related objects in the same heap.

Assuming this model of the use of heaps, it is possible that some heaps will contain more objects than others. As a result, there will be heaps for which the second part of the reference is larger than it needs to be. Unfortunately, in our reference scheme, the sizes of the parts of the reference are fixed. (Through the use of clever encoding techniques it is possible to have a reference scheme in which the size of the pieces of the reference can vary "by need"; we consider such techniques too expensive for our purposes.) Thus, each lightly populated heap will result in a number of references that are not used (and are not logically usable). Clearly, we need to pick the size of the pieces of the reference to minimize this problem. But in doing the division, since we are using direct lookup and not hashing, we are obliged to pick sizes of the parts of the reference that allows for the maximum – not expected – number of objects per heap. Since we can assume that most heaps will not be completely full, we will have unused references. This is the price we pay for introducing partitioning.

2.2.4 Reusability of references

While we didn't explicitly state it, in both the pure address and the mapped address approach, we have assumed that references can be reused. That is, if an object is deleted (i.e. discovered to be unreferenced after garbage collection), we can reuse the reference to refer to some newly created object. In the pure address strategy, this simply means that we can put some new object in a place where some old object lived and that the disk address of that place (the reference to the old object)

now becomes a reference to the new object. In the mapped address strategy, it means that we can reuse the slot in the mapping data structure that held the translation between the reference and the old object's disk address to hold the translation between between the reference and some new object's disk address; we return the reference to the old object to the allocator of the new object.

An alternate approach to reusing references is (obviously) to not reuse references. Each time a new object is made a new reference (one that has never referred to any object) is made up. The approach is called the *unique identifier* (or simply *UID*) approach.

The first question that arises in the UID approach is "how do you generate UIDs?" One traditional approach is to use a clock; a clock is a continuous source of unique numbers. The second question is "how many objects will ever be created?" This question needs to be answered in order to decide how many bits long the reference should be. Note that in the case of non-UID systems, the size of the reference is determined by how many objects can exists *at any instant*, not how many will ever exist. In either case, experience tells us that we should overestimate. Choosing too small a reference is something to avoid because running out of references is a fixed barrier; when it happens, your system falls apart. Choosing too large a reference has the cost that you can waste space (and hardware) allocating bits that you never use. There is no simple answer to the problem. One thing is for sure though – a UID system needs more bits for a reference than does a non-UID system.

An advantage to the UID approach is that objects can be explicitly deleted (i.e. freed) without having to worry about dangling references. To be more precise, dangling references are still a problem, but they are a problem that will be detected. As noted earlier, in a non-UID system, explicit freeing leaves open the possibility that the reference will be reassigned to a new object and that dangling references (i.e. references to the old object that are now references to the new object) will be dereferenced producing meaningless results. In a UID system however, when the object is freed, the reference is marked as being invalid, and dereferencing it will cause an error that can be detected by the storage system.

The disadvantage of a UID system is that the cost of dereferencing is high. The data structure that maps the UID into a disk address will have to be complicated (e.g. a hash table). There is no natural way to divide the reference as was done above. (One can imagine dividing the reference and having multiple mapping tables, but doing so would not produce the desired benefits.)

2.3 Types and Code

By *code* we mean the programming language procedures that implement the abstractions discussed in section 2.1.2. It must be possible to get from a reference to an object to the code that implements abstractions on the object. We call the characteristic of an object that determines what code should be used to implement abstractions on the object, the object's *type*.

In a traditional programming language like Pascal, it is not necessary for the representation of an object to contain an indication of the type of the object. This is because all variables have types and an object is the value of a variable or the value of some field of an aggregrate whose type is known. In T (like all Lisps), variables do not have types. Thus, the type of an object must be explicitly associated with the object itself. The obvious mechanism for doing this is to allocate some space in the object to hold a *type identifier*. Optimizations of this scheme will be discussed in section 3.2.1.

Given that type IDs are kept in objects, we need a mechanism that takes a type ID and returns a procedure that takes some operation that is to be performed on an object and implements the operation on the object. This procedure is the handler mentioned in section 3.1. Ideally, code in our world of permanent objects would be a permanent object itself. Thus the result of the mechanism just described could simply be a reference to a handler object. In fact, if code can be represented as a permanent object, the type ID could simply be a reference to the handler object.

One reason for wanting the type ID to be something other than a reference to a code object is that we can assume that there will probably be more objects than types of objects. As a result, the

number of bits needed to hold a reference is larger than the number of bits needed to hold a type. Since every object will have a type ID embedded in it, we would like to minimize the size of the type ID. Another reason for having a layer of indirection between type IDs and code objects is that it permits a level of abstraction. Types can be thought of independent from their implementations. Implementations can be changed without having to modify all the objects that contain the type ID.

If we introduce a layer of indirection between type IDs and code, we must keep a global table that maps type IDs onto code objects. In order to avoid making this table a bottleneck in the system, each process would presumably keep a local cache of the map. (Note: this solution works only as long as the handler corresponding to a given type ID never changes.)

Another issue about types and code that needs to be addressed is how to deal with type redefinition. Suppose we create a type, create some objects of that type, and then want to modify the behavior of objects of that type (i.e. how those objects respond to operations). Do we want to modify the behavior of existing objects of that type, or only objects created after the type is modified? Also, what if the type definition wants a different number of slots assigned to objects of that type? There are cases when one wants old objects to "see" the new type definition – for instance when one is fixing a bug in some method. There are cases when one wants them not to see the new definition. In this case, one might be inclined to call the change an introduction of a new type, not a redefinition of an existing type. But this would be hiding the relationship between objects of the old type and objects of the new type. Suppose a bug is fixed in a method – one would want the bug fixed in both the old and the new handler (type definition).

2.4 Previous Work

Many other researchers have worked on systems that tried to solve some of the problems discussed in this chapter. We will briefly discuss some of that work.

2.4.1 Capability systems

The term *capability system* [20] is usually applied to a system that is specially designed to keep track of references to objects. Levy's masters thesis [35] contains an excellent summary of these systems. In capability systems, access to data is controlled by the fact that a process can refer only to objects for which it has capabilities. A *capability* is essentially a high-level machine address. The only way to obtain a field of an object is with a machine instruction (or kernel call on machines that do not have capability-based hardware) that takes a capability and an offset into the object. Unlike other systems it is not possible for unprivileged processes to create capabilities from other data types. Part of creating a process is the assigning an initial set of capabilities to the process. The process can then pass those capabilities onto processes that it invokes.

2.4.2 Hydra

The Hydra operating system for the C.mmp multiprocessor [54,16] has been an influential model for researchers interested in capability systems. The underlying hardware (which consists of PDP-11s) is not capability-oriented. However, Hydra supports capability-based references to objects. This functionality is supplied by machine instructions that trap to the kernel which then authenticates the reference and does the requested operation.

2.4.3 IBM System 38

While they have long had an attraction to researchers capability systems have not become common in the real world. The IBM System 38 is one of the few commercial systems based on the capability

model. The System 38 hardware is capability-oriented. In spite of the fact that is has an object-oriented model – the system presents a *one-level object store* which eliminates the distinction between objects in main memory and objects stored on the disk – the System 38 does not provide anything other than a traditional programming environment (COBOL and RPG-II).

2.4.4 Intel iAPX 432

Intel's iAPX 432 microprocessor and associated operating system, iMAX 432 [29,41] is a recent commercial entry into the world of capability systems. The 432 system is also object-oriented, but unlike the System 38, the 432 makes apparent to the programmer the distinction between active and passive objects. Passive objects are referred to using 80 bit UIDs. Active objects are referred to using 24 bit 432 access descriptors.

The 432 is not in widespread use and the status of the iMAX project is unclear.

The recent trend in computer architecture design has been toward machines with a considerably simpler model [43]. The firmware that supports capability systems is extensive. As a result, it is hard to debug and hard to optimize.

2.4.5 Smalltalk

Smalltalk [31,22] is a language, operating system, and programming environment. The only successful Smalltalk implementations have been on microcoded personal workstations.[3]

Smalltalk is the canonical object-oriented environment. The Smalltalk language introduced many of the concepts and much of the terminology of object-oriented programming.

2.4.6 Eden

The Eden project [32,2,3,4] is a project attempting to build a distributed computing environment around object-oriented principles. Eden objects are relatively expensive and heavy-weight and hence are used to represent a collection of data. In Eden, objects are active entities. When an operation is applied to an object, a process corresponding to the object (not to the invoker of the object) is activated to run the object's method for the operation. Part of the Eden project is the development of a programming language, EPL, based on Concurrent Euclid. The purpose of EPL is to allow Eden objects to be coded conveniently. Using EPL, active Eden objects can have multiple threads of control.

Originally, the Eden project expected to run on the Intel 432 microprocessor. However, the present Eden prototype is running on multiple VAXes connected via a local Ethernet.

2.4.7 Object-oriented machines

There have been several proposals from MIT for "object-oriented machines". The machines bear a resemblance to capability machines in that the hardware is specifically designed for keeping track of references. None of the proposed machines have been built.

Bishop [14] describes ORSLA, a system with a very large linear, paged address space. All processes run within the same address space. Object references are virtual addresses, not UIDs. The address size is proposed to be somewhere between 40 and 50 bits (the minimal addressable unit is a 64 bit word). As an optimization, a reference contains some object size and type information in addition

[3] The recent implementation of Smalltalk on the SMI SUN 68000-based workstation [17] apparently approaches the performance of the better microcoded implementation; it is not clear if this implementation will succeed in making Smalltalk more widely used for large applications.

to the virtual address of the object. Thus, the size of an ORSLA reference is between 58 and 81 bits.

Bishop's main idea is a scheme for partitioning the address space into *areas* and allowing areas to be garbage collected independently. The area scheme depends on inter-area references going through *inter-area links* so that the garbage collector can determine the root set for the collection of a single area. The proposed hardware would make inter-area links transparent to the programmer.

Luniewski [38] describes AESOP, an object based personal computer. AESOP incorporates some of the ideas of ORSLA. In addition, Luniewski investigated some of the programming language issues involved with working on the proposed architecture. He adopted the CLU language model [36].

Snyder [49] describes another object-oriented system based on CLU. His thesis discusses some of the lower level hardware issues associated with such a system. Also, Snyder proposed the use of reference counts instead of garbage collection to allow storage to be reclaimed.

2.4.8 APL

The APL workspace [21] is one of the earliest examples of a mechanism that supports permanent structured objects. Early APLs provided only a mechanism for *copying* objects from one user's workspace into another. Modern APL systems provide mechanisms for also sharing values among workspaces.

2.4.9 POMS

The Persistent Object Management System (POMS) [7,8,9,10,42] is a project that has extended ALGOL to deal with permanent objects. The underlying permanent storage mechanism is the Chunk Management System (CMS). CMS provides a database-like interface for POMS. On first reference to a permanent object POMS requests the image of the object from CMS; POMS deals with a *copy* of the object and the changes made by the program using POMS is not made permanent until the program commits the changes at which point the image of the object is copied back into CMS.

2.5 The Smalltalk - Hydra spectrum

In looking at the various systems that have adopted the object-oriented model, one can see a range of concerns to be addressed. Smalltalk and Hydra are at opposite ends of several spectra:

	Smalltalk	Hydra
object size	*small*	*large*
number of objects	*large*	*very large*
cost of dereference	*small*	*large*
language integration	*good*	*bad*
objects sharable?	*no*	*yes*

2.5.1 Object size

All object-oriented systems are designed to support well a particular range of object sizes. Ideally, a system should support a range of sizes from just a few bytes to thousands and millions of bytes. In practice, it is difficult to support such a range. One finds that a system discourages the use of small objects by introducing a fairly large storage overhead per object. For instance, if the system imposes a 16 byte overhead per object, it is unlikely that programmers will create many objects of

16 bytes or less – programmers will tend to combine several logically related small objects into one larger object to minimize the overhead. This obscuring of logical objects reduces the usefulness of the system. In fact, if the per-object penalty is large enough, one tends to view objects the same way one views files in a traditional operating system.

Smalltalk is oriented toward dealing with small objects – every piece of data in Smalltalk is an object; the Smalltalk implementors' experience has shown that average object size is only about 20 bytes [22]. The per-object overhead is 8 bytes (4 bytes in the object and 4 bytes in the object table). The largest object is 128K bytes (large, but probably not large enough for all applications).

Hydra is oriented toward dealing with somewhat larger objects than Smalltalk. The per-object overhead for an active object is 56 bytes; the per-object overhead for a passive object is 32 bytes. Almes [1] points out that these overheads can in principle be reduced to 32 and 16 bytes respectively.

2.5.2 Number of objects

Another design aspect of object-oriented systems is the number of objects that can exist at the same time. Traditional Smalltalk implementations use 16 bit object references and hence can support 32K objects. The reason this number isn't 64K is because Smalltalk implementations typically encode integers in the range $-2^{15} .. +2^{15}-1$ in the object reference itself; one of the bits in the reference is taken to mean "I am a small integer, not a real reference".

Some more recent Smalltalk implementations have used 32 bit references [12], but it is not clear that they are designed so that they can actually support 2^{32} objects. LOOM [28,51] is an experimental system for extending the Smalltalk object space by introducing a secondary object memory; the Smalltalk interpreter automatically moves objects between primary and secondary memory. References to objects in secondary memory are 32 bits long.

As opposed to Smalltalk, Hydra was designed to support a large user community that would work on C.mmp. As a result, Hydra was designed to support a larger number of objects than Smalltalk. Hydra uses a 64 bit object reference which is composed of a 60 bit field which contains the value of a 1 microsecond clock at the time of the object's creation, and a 4 bit processor ID. The increased per-object storage overhead of Hydra as compared to Smalltalk is in part due to the larger reference size.

2.5.3 Sharing of objects

Another area in which Hydra comes out ahead of Smalltalk is in the area of sharing. Again, since Hydra was designed to be a multi-user environment, it needed to support the sharing of objects among users. In Smalltalk, each user works in his own object space and there is no (attractive) mechanism for sharing Smalltalk objects among different Smalltalk users.

2.5.4 Cost of dereferencing

In Smalltalk, all objects are entered in an *object table* (*OT*). A reference to a Smalltalk object is an index into the OT. The OT entry for an object contains the memory address of the object, a reference count, and other miscellaneous information. Obtaining a field of an object requires a memory reference to the OT in addition to the memory reference to obtain the field itself. The size of the OT is fixed and the entire OT must be in main memory. For Smalltalks with 16 bit references, each entry in the OT is 32 bits long. Thus the total size of the OT is 128K bytes.

In Hydra, an object can be either passive or active. An object is activated automatically when a field of the object is request. Hydra uses UIDs as object references and hashing as part of the dereference mechanism. A data structure called the *active GST* keeps track of all active objects (i.e. objects in main memory). A data structure called the *passive GST* keeps track of all passive

objects (i.e. objects on disk). Obtaining a field of an object requires a hashed lookup in the active GST; if the object is found there then the main memory address of the object is extracted from the active GST entry and used to pick of the field of the object. If the object is not in the active GST, the object is activated (which requires reference to the passive GST) and then the procedure proceeds as it does for an active object. This process of obtaining a piece of a Hydra object is handled by operating system code and is initiated by a user process by executing a kernel call – a special machine instruction that is trapped by the Hydra operating system.

Hydra's reference mechanism is clearly more expensive than Smalltalk's. The kernel call in Hydra can be used to copy out large pieces of an object into a process's local memory; evidentally this feature is used to minimize the number of kernel calls necessary to obtain an object's state. The expense of dereferencing encourages programmers to make large objects whose contents can be retrieved with one kernel call.

2.5.5 Language integration

From our point of view, the most serious deficiency of Hydra is the evident lack of an environment for programmer's to design and build systems based on object-oriented principles. From the descriptions of Hydra, it is not at all clear how one actually programs on it. Smalltalk, on the other hand, is the ultimate in object-oriented programming environments. The language and the environment are complete integrated. Tools are provided for inspecting the object space.

2.5.6 Summary

The point of our Smalltalk/Hydra comparison is not to show that one or the other is better. Rather, the point is to show how two systems which are both "object-oriented" can turn out so differently as a result of different goals. Smalltalk's implementors were interested in making a single-user programming environment to exploit the concepts of object-oriented programming. Hydra's implementors were interested in making a multi-user, reliable, multi-processor operating system based on object-oriented principles.

In our system we have tried to find a mid-point in the spectrum of possibilities that characterize the differences between Hydra and Smalltalk. It would be fair to say however, that we started at the Smalltalk end of the spectrum and tried to generalize to a system that has some of the properties of Hydra. The Eden project is an example of a project that started at the Hydra end of the spectrum and attempted to support the programming ease and efficiency of Smalltalk.

2.6 Message passing instead of object moving

An essentially different line of research that is concerned with sharing of data concerns the support for *message passing*[4] in a programming system. This research has proposed the introduction of programming language primitives that send data to and receive data from other processes. In the systems discussed above, data is manipulated simply by dereferencing a pointer to the data. Multiple processes can access the data; there is no explicit moving of data among the processes wanting to access the data. This sort of access to data seems natural and does not require novel programming language constructs. However, access to the data is unconstrained – synchronization is not part of the model. The message passing approach can be viewed as an attempt to allow the synchronization of processes's access to data.

[4]N.B. In Smalltalk and other languages with similar goals, this term is often used to mean something like "generic procedure call" but this is *not* the sense we intend here.

Extensions to CLU have been proposed to allow message passing. [23,24,50]. More recently, the Argus project [37] has introduced the notion of *guardian* as the repository for shared data; communication with guardians is implemented via the lower-level message passing mechanism.

2.7 Summary and approach of this work

The framework in which we have designed and built our system for maintaining permanent objects includes the following assumptions:

- The system runs on conventional hardware.
- The system runs within a conventional operating system.
- Application programs that use the system are written in an extended conventional language.

All our assumptions, but especially these three, result from our desire to build a system in which we could experiment with *programming* in a permanent object system. Requiring that we build hardware or operating system software or design a new programming language would have increased the scope of the project beyond our ability. Given the alternatives of a less than ideal system with which we could actually experiment or a perfect system that would at best be only partially implemented, we chose the less than ideal system. In addition, from a purely experimental point of view, we wished to demonstrate that the implementation of these concepts does not absolutely require sophisticated new languages, hardware, or operating system software.

- The entire space of objects can be naturally divided into subspaces (heaps) of objects.

That we assume that the space of objects can be naturally divided means that there will be some set of applications for which our system will not be useful. For instance, if the object space consists of a large highly connected graph of objects of the same type, there may be no natural way to divide that space. Note however, that if the undividable space is small enough so that the application's objects can fit within the largest possible heap, the application can use our system.

- The system does *not* provide complete transparency for the application programmer.

A system that provides complete transparency does not require that the programmer know the pattern of inter-heap references, or what kinds of objects reside in what heaps, or in what heap the next object should be allocated. In our system, the programmer *does* have to know these things. We hope that experience with using a system like ours can help in designing a practical system in which complete transparency *is* possible.

- The system does *not* provide high reliability in the face of hardware or communications failure.

This assumption is related in part to the first two assumptions. Given that we were unwilling to build hardware or operating systems, it is difficult to improve the reliability of our system beyond the level provided by conventional hardware and software. The gross reliability of our system is as good as the conventional system on which it is built. This level is good enough for people who use the conventional system, so it is reasonable to believe that it will suffice at least for our initial implementation. In the long term, higher reliability is probably required since in a system of the sort we built, the loss of a very small amount of data can potentially lead to disasterous results.

- The system does *not* provide mechanisms for a high degree of concurrency.

We are interested in supporting the sharing of objects by multiple processes. Secondarily, we are interested in allowing as much concurrency as is possible using conventional techniques (locking, busy waiting). We believe that our system supports the solution to problems that have a low degree of concurrency. The system does not support concurrency among processes running in separate physical memories.

- The system supports only fairly coarse protection.

If a system is to provide access to permanent objects that is nearly as fast as access to transient objects, it seems that it must rely on special hardware to allow protection down to the level of individual objects.

Chapter 3

Implementation

In this chapter we will discuss the design and implementation of OM, our system for supporting permanent objects. We will be concentrating on the lowest levels of the system.

3.1 The object model

The model of data that we will use in this thesis is typically called *object-oriented.* This model has been popularized by Smalltalk. Since the term has different meanings to different people, we will briefly describe what it means to us.

The entities in object-oriented system are (not surprisingly) objects. An object is a piece of contiguous storage. Atomic objects have pre-defined storage layout. Non-atomic objects are divided into equal-sized slots; each slot contains a reference to some object. Integers and strings are examples of atomic objects. A vector is an example of a non-atomic object.

An important concept in the object-oriented view is the notion of reference. Objects do not contain objects, they contain references to objects. Thus, two different objects can refer to the same object. Two references are said to be *identical* if they refer to the very same object. Two objects are said to be *equivalent* if there is no way to tell them apart. That is, any procedure applied to one object yields the same result as the same procedure applied to the other object. Two references can be non-identical yet refer to equivalent objects.

Objects can be mutable or not. An object is mutable if the storage occupied by the object can be modified. Integers are immutable objects. Strings and vectors can be mutable. A mutation to an object is sometimes called a *side-effect.*

Computation occurs by invoking *operations* on objects. (An operation is the same as a Smalltalk message.) When an operation is invoked we say the object *responds* to the operation by executing some code. We call the code that implements the response a *method.* We call a collection of methods a *handler.* The *type* of an object is defined by its handler. This is an operational view of types. Operations are generic; i.e. they can be applied to any object. However, an object does not necessary handle every operation. An error occurs if an operation is applied to an object that does not handle that operation. The entire process from operation invocation to method execution is called *operation dispatch.*

The object model just described is essentially that of T. Much of the terminology we use is T's. One reason for using this model is that it is T's model and our system will be running within T and used by programmer's familiar with T's model. Another reason we use this model is that it is simple – objects can be accessed in a uniform way.

3.2 The Environment

The environment in which we implemented OM consists of the T programming language and the Apollo DOMAIN computing environment. When we began the project, we were fully aware of the fact that by trying to work within an existing environment, we would have to compromise on what functionality we would be able to support. OM does not provide the complete transparency and ease of use that many unbuilt systems have proposed.

A clear advantage of working with existing tools is that we were able to more quickly address the issues in which we were interested: What is it like to program a large system where all data is stored as permanent objects? Can such a system be made efficient? Another advantage in not being language designers is that our end product is not a system that is unfamiliar to a ready user community – a community already familiar with T is more likely to use a language that is much like T than they are a totally new language. Finally, it is unproven that a permanent object system actually requires a special purpose language, hardware, or operating system. We wanted to see how sophisticated a system could be built within a relatively traditional environment.

3.2.1 The T programming language

T is a dialect of Scheme, which in turn is a dialect of Lisp. Scheme differs from Lisp mainly in the fact that variables are consistently lexically scoped. In this respect, Scheme is more like traditional Algol-like languages than are traditional Lisp implementations. The latter support dynamic scoping; i.e. the value of a variable is determined by the contents of the control stack, not the lexical position of the variable.

Scheme supports procedures as "first-class objects". That is, procedures are legitimate objects (like strings, vectors, and pairs[1] that can be bound to variables and passed as arguments to other procedures. Procedure objects are created by the LAMBDA special form[2]. Procedure objects are also known as *closures* because when a LAMBDA form is executed, it returns a procedure object that is *closed over* the lexical environment of the LAMBDA form. That is, when the procedure object is called and the body of the procedure is executed, references to variables that are free with respect to the LAMBDA form but that are in the lexical scope of the LAMBDA form yield the values the variables had *at the time the procedure object was created.*

T is essentially a practical realization of Scheme. Before T, there were no practical Scheme implementations in widespread use.

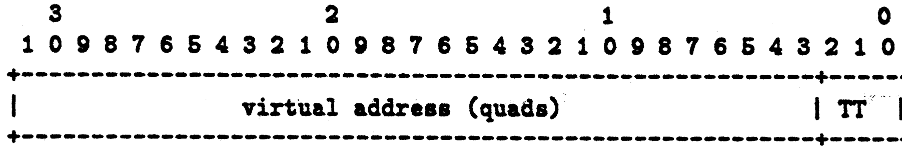
Like many Lisps, T runs in an interactive environment. This environment contains a T interpreter that allows the debugging and incremental redefinition of procedures. The T compiler takes source files and produces object modules that can be loaded into the T interactive environment for execution.

T, like all Lisps, is a language of reference. That is, the values of variables are references to objects, not objects per se. Different variables can refer to the same object. Objects can contain references to other objects. Procedures return references to objects. In general, objects are allocated in heap storage. T uses a copying garbage collector to reclaim storage occupied by objects that are no longer reachable.

T references are 32 bits long. The low 3 bits of the reference are used as a *type code*. The type code is used to determine the type of the object being referred to. For instance, if the type code is 5, then the object at the address specified by the reference is an adjacent pair of references – an 8 byte T pair. The high 29 bits of the reference is a virtual address in quads (8 byte chunks), not bytes. Figure 3.1 shows the format of a T reference. Note that a quad address left justified in a 32

[1] Lisp's traditional *cons* cell is called a "pair" in T.

[2] *Special form* is the traditional Lisp term for syntax in the language that is used to denote something other than a call on a procedure.




```
TT = T type code
1 quad = 8 bytes
```

Figure 3.1: T reference format

bit word is a byte address (i.e. if the low 3 bits of a reference are masked to zeros, a valid machine address results).

Let us briefly examine how this sort of type code scheme works. In T, choosing a granularity[3] of 8, the low three bits of the machine address of an object are always zero. Hence in a T reference these three bits can be used to store the type code; this requires that before using a T reference as a machine address, the low three bits must be cleared. If the type code of a reference is known (or can be assumed), then the clearing of the type code can typically be done in the same instruction that fetches a field of the object: the displacement field of the instruction is simply decremented to account for the increment that the type code will cause. In T, as in most Lisps, the machine code produced for primitive procedures such as CAR assumes that its argument is a pair and hence it can assume the type code is a particular value.

Given fixed word and type code field sizes, the total number of unique references is also fixed. As the minimum object size is decreased, the total number of usable references decreases (assuming some objects are larger than the minimum object size). Thus, in effect, as the granularity decreases, the total number of objects that can exist at a time decreases. As the minumum object size is increased, if there are a number of objects that are logically smaller than the minimum object size, the total amount of wasted space increases.

T uses a granularity of 8 because Lisps traditionally make heavy use of objects that contain exactly two references.

Since T needs to support more than 8 types of objects, one of the 8 possible type codes is used to mean "the type of the object is encoded in the first cell (4 byte quantity) of the object". This type code is called the *extend* type code. The first cell of an extend-type object is called the object's *template pointer.* Objects represented in this way are called *extends.*

In principle, all type information could be encoded in templates and no type code in the reference would be needed. However, there are two reasons for type codes. First, they allow certain objects to be represented without the extra storage of a template pointer; e.g. without type codes in references, *cons* cells would have to be 3 cells long instead of 2. The second reason for putting the type code in the reference is to speed up the process of determining whether a reference is to an object of one of the frequently used types. For example, given a reference to an object, one can determine whether the object is a pair simply by looking at the reference – the contents of the object itself need not be examined.

T supports the style of object-oriented programming described in section 3.1. Recall that the first step in operation dispatch is to get from a reference to an object to the handler associated with the object. The way this is implemented in T is as follows (we make some minor simplifications): if the reference's type code is not extend, then the handler is obtained from a fixed vector of handler procedures; the vector is indexed by type code. If the reference's type code *is* extend, then object's template pointer is taken to be the reference to the object's handler procedure. Once the handler

[3] By granularity we mean the smallest unit of allocation – i.e how small objects can be.

is obtained, T calls it, passing the operation being invoked as an argument; the handler returns the method associated with the operation. T then calls the method.

T application programmers are not aware of the machinery of operation dispatch described above. The OBJECT special form allows programmers to allocate objects and specify how the objects are to respond to operations. Handlers are part of the T implementation and are not visible to programmers. Operation invocations are syntactically identical to procedure calls. In T source code, procedure calls are expressed as a list whose head is an expression that yields a procedure object and whose tail is a list of arguments to the procedure. The only difference between this and the syntax of operation invocation is that the head must yield an operation object. The first argument to the invocation is the object to which the operation is applied. Operations are defined using DEFINE-OPERATION which has a syntax similar to DEFINE, the procedure definition special form. The body (code) of the DEFINE-OPERATION is called the *default method* – the code that is to be executed in case the operation is applied to an object that does not handle the operation. If the body is empty, then when the operation is applied to an object that does not handle the operation, an error is signalled.

As an space optimization, certain objects are not represented in heap storage. These objects are said to be represented *immediately*. Immediate objects are represented within a reference. References with certain type codes are taken to be immediate objects. For example, if a reference has type code 0, then the high 29 bits of the reference are taken to be an integer in the range $-2^{28}..+2^{28}-1$; T calls such integers *Fixnums*. Immediate representations are important because one wants to minimize the allocation of heap storage that will become garbage quickly. For instance, if Fixnums were not represented immediately, then the + procedure would have to allocate space in the heap to hold its result. If this result was not saved, but only passed to another procedure, as in (* 2 (+ 3 4)), then the result of + becomes garbage, resulting in the heap's filling up quickly.

3.2.2 The Apollo DOMAIN computing environment

The computing environment in which we developed OM is the Apollo DOMAIN [5,6,33,34]. DOMAIN is an integrated environment of high performance personal nodes[4] attached by a high speed (12M bit/sec) local ring network. The present Apollo hardware uses a Motorola MC68000 or MC68010 microprocessor [40]. The 68000 instruction set is traditional and memory is byte addressed. A node typically has from about 1M to 2M bytes of private main memory; it is not possible to share main memory among multiple nodes. Each user node has a high resolution bitmap display; Apollo makes server nodes that do not have displays but which can be accessed from other nodes on the ring. The DOMAIN software supports multiple processes on a single node; each process runs in its own virtual address space. We discuss below those feature of the DOMAIN system that are relevant to our work (we make some minor simplifications for ease of presentation).

The DOMAIN virtual memory architecture presents a virtual address space that is in principle 2^{24} (16M) bytes long; part of that space is reserved by the operating system and the amount available to user code is about 8M bytes (it is expected that later Apollo hardware will support a larger virtual address space as true 32 bit microprocessors become available).

The process virtual address space is divided into 1K byte pages. For a page to be usable it must be *mapped* to a disk file. By page's being "mapped" we mean that it corresponds to a page in a disk file. A memory reference by a machine instruction to a virtual address in the mapped page yields a piece of the disk file page. Depositing a value into a virtual address modifies the contents of the file. The pager is responsible for optimizing updates of main and disk memories.

Parts of the address space are made usable by issuing a *map* system call. The call takes a file identifier (discussed below), an offset into the file, a length to be mapped, and some locking information (discussed below). The call returns the virtual address at which the file is mapped. In general, the process has no control in selecting to which part of the address space a file is mapped. Execution

[4]Apollo uses the term *node* instead of *workstation* and so shall we.

of machine instructions that refer to parts of the virtual address space that are not mapped result in hardware exceptions. The smallest amount of virtual address space that can be mapped is 32K bytes; the amount mapped is always rounded up to the nearest 32K byte quantity. The *unmap* system is used to remove association between the address space and some file.

Multiple processes running on the same node can concurrently map the same part of the same file. In general, the files may be mapped to different places in each process's virtual address space. Both processes see any changes made by the other. Multiple processes running on different nodes can map the same file for read access only. While the lowest levels of Aegis allow multiple processes on different nodes to map the same file for write access, the results of modifications to the file are undefined. This sort of access to files is not officially supported by Apollo.

It is possible to map a segment of a file where the segment is longer than the current length of the file. As references are made to parts of the address space that correspond to parts of the file that do not exist, disk space is allocated and associated with the appropriate part of the file. Disk space is not allocated unless and until the reference is made.

The DOMAIN operating system, Aegis, does not present any traditional I/O system calls like *read* or *write*. The only I/O is done by the pager. User I/O is provided via a user-state subroutine library. This library is implemented using the mapping primitives.

One aspect of the DOMAIN system that makes it unique among commercial workstations is that any file on any disk attached to any node in the local network can be transparently accessed by any process on any node. By "transparent" we mean that the accessing process does not need to consider whether or not the file is on the disk attached to the node on which the process is running.

Files are identified by a 64-bit unique identifier (UID). File UIDs are unique across all Apollo nodes. (The UID has the creating node's hardware node number embedded in it.) The *home node* of a file is the node whose disk contains the file. The *map* primitive takes the UID of a file to map. The file referred to by the UID can be local or remote (i.e. on the same node as the process executing the *map* call or not).

The call to *map* results in no disk I/O. Pages of the mapped file are page faulted on demand from the home node of the file. The first reference to a mapped page will cause a page fault. At that point the pager either reads the file from the local disk, or sends a *page-in* request to the home node of the mapped file. In the latter case, the pager must figure out what the file's home node is based on the file's UID. To do this, presently the DOMAIN system allocates 20 bits of the UID to be the node ID of the home node of the file. This means that a file can not be moved between nodes (a file can be copied between nodes and the original copy can be deleted, but the copy will have a new UID which contains the node number of the node to which the file was copied).

Aegis provides a set of system calls for naming files. These calls allow users to specify text *path names* of files (path names are like Unix file names [48]). The purpose of the naming system is to translate path names into UIDs. The naming system contains directories (which are represented as files) that translate path names into UIDs. No information about files per se (e.g. file length, location of file on disk) is stored in the naming system. The naming system could logically be implemented outside of Aegis (modulo a few details).

Aegis also provides a simple file locking mechanism. For our purposes, it suffices to say that one can control how many processes have write access to a file at the same time. This control is exercised at the time a file is mapped.

3.3 Introduction to the implementation issues

3.3.1 OM within T

How should the permanent object system be related to T? We see two different approaches to this

question. The first approach is to think of T as the implementation vehicle for the system. In this approach the programmer is lifted up from T and works consistently in a permanent object world presented by the system. The T programming language might be modified in some ways to better handle the system's facilities and concepts.

The second approach to setting the relationship between T and the permanent object system is to think of the system as a set of utility procedures that are available to the T programmer. The user of OM still programs in T; he calls system procedures procedures to copy T objects into the permanent object space and back. The language modifications are only those that can be implemented with T's syntax modification tools (macros). The programmer has to be aware of when he's dealing with a permanent object and when he is dealing with a transient object.

The second approach is clearly less desirable, but it is much easier to implement. Another advantage of the second approach is that it does not require one to be a language designer. We believe that one should decide what a language that is designed to deal with permanent objects should look like only after one sees the ways in which standard languages are inadequate. In OM, we adopted the second approach. The result was acceptable, but less than ideal in ways we will summarize later.

OM is written entirely in T. However, OM was written with a detailed understanding of how T is implemented. We present the OM implementation with respect to the T language and operating environment. The details we discuss are in general not apparent to the programmer who wants to use OM. When we use a phrase like "To T, *feature* is ..." or "In the OM implementation, *feature* is ..." we are describing how some aspect of OM is implemented, not how it appears to the programmer who uses OM.

3.3.2 Reference

Our first concern is the form of an *OM reference* – a reference to a permanent object. Just as all T (i.e. non-OM) procedures take T references to T objects as arguments, OM procedures take OM references to OM objects as arguments. Note that to T, OM references are objects of some user-defined data type. But to the programmer using OM, OM references are (not surprisingly) references to OM objects.

To create OM references within T we could use the standard T mechanism for introducing new types of objects. Unfortunately, this mechanism is expensive – all objects of user-defined types are represented in the heap. It is unacceptable for OM references to be represented in the heap. If they were, all procedures that return OM references would need to allocate storage simply to return the OM reference. (We are not talking about allocating space for the OM object itself.)

Fortunately, one of T's 8 type codes is unused by T. With virtually no modifications to the implementation of T, we can use this type code for OM references. Using this type code, OM references can be represented immediately. Whatever format we choose for the reference, it must fit in the upper 29 bits of a T reference.

Are 29 bits enough for an OM reference? If we were able to use all 2^{29} references, it might be. However, we intend to use the divided mapped address reference design discussed in section 2.2.2. As noted in section 2.2.3 this means that we expect that we can not use all the possible references. Suppose we divide the reference roughly in half – say 14 bits of heap identifier and 15 bits of within-heap reference. This allows 16K heap identifiers and 32K references per heap, assuming we maintain a 32K entry table that translates the in-heap reference to an actual byte offset within the heap.

Let us consider a modification to the multiple table mapped address scheme. Instead of treating the least significant part of the reference as an index into a table of byte offsets, it can be the byte offset itself. The advantage of this scheme is that we save one table lookup (memory reference) for each dereference. If we are dividing the reference into just two pieces, this savings is significant – we have just one table lookup instead of two. The disadvantage is that we partially re-introduced the problems associated with the pure address strategy: the inability to easily move objects and the

underuse of all possible reference bit strings. However, lacking translation lookaside hardware, we are willing to pay this price. As we will show, these problems can be reduced somewhat.

If we use this modified version of the multiple table mapped address scheme, 14 bits of in-heap reference does not look so attractive: the maximum heap size would be just 16K addressing units (presumably bytes) – clearly not large enough. If we expand the in-heap reference, we must reduce the heap identifier part of the reference, thus reducing the total number of heaps. If we want to have heap identifiers be unique for all time (to allow heaps to be manually deleted), this limitation is unacceptable.

We conclude that given the properties we want of our object system, 29 bits is not enough for an object reference. Before pursuing a remedy to this problem, let us first consider some of the properties of data structures.

3.3.3 The structure of data structures

Data structures are directed graphs of objects. In practice, the graphs representing data structures are not arbitrary. One sees trees, lists, vectors, DAGs, etc., and connections between graphs of these types to form larger graphs. As a result, often a large data structure has natural points of division. For example, if a data structure is a list of trees, then the graph is naturally partitionable at the connection points between the trees. Note that this is a *static* property of a data structure.

Graphs of data structures may also be partitioned based on their *dynamic* properties. For instance, some vertices may be examined more frequently than others. There may be locality of reference among the vertices; i.e. a graph might be partitioned into subgraphs whose vertices are accessed around the same time.

In building a permanent object storage system, one can ignore the partitionability of data structures. That is, if the system provides references that allow an object to refer to any other object then it can certainly implement any data structure. However, such a system is overly general. In general it is not necessary for an object to contain a reference to any other object in the world; it need only refer to some smaller world of objects.

Providing the general functionality is expensive: in a system of reference, like Lisp, the size of objects other than atoms[5] is proportional to the size of the reference. Thus, we want to make the size of a reference as small as possible since doing so will reduce the amount of space required to represent an object. Of course, if we make the size of a reference too small, we make it impossible to refer to an adequately large number of objects.

3.3.4 Local and non-local references

How do we take advantage of the locality of reference among objects in a data structure while still allowing references among arbitrary sets of objects? Our solution is to allow two kinds of references: local and non-local. Local references are used to refer to "nearby" and logically related objects: objects in the same heap as the source of the reference. Non-local references can be used to refer to any object. Local references are smaller than non-local references. There are two dereference mechanisms, one for local references and one for non-local references.

There is a problem with having objects connected by different kinds of references: when a program is traversing a data structure, following references, it needs to know what kinds of reference the object currently being examined contains so that (1) it can extract the appropriate number of bits from the object, and (2) it can apply the appropriate dereference mechanism. Recall that our original model of the implementation of an object (see section 3.1) is that an object is a vector of *equal-sized* slots containing references to other objects. In a straightforward implementation, if there are different size references, the slots of an object have to be variable size and the object has to have a descriptor

[5]The Lisp term for objects that do not contain references to other objects.

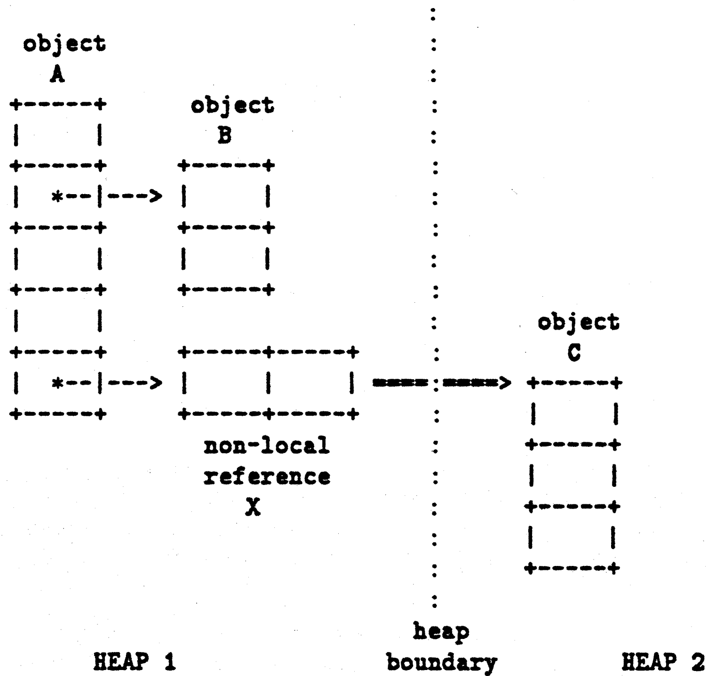


Figure 3.2: Three objects in two heaps

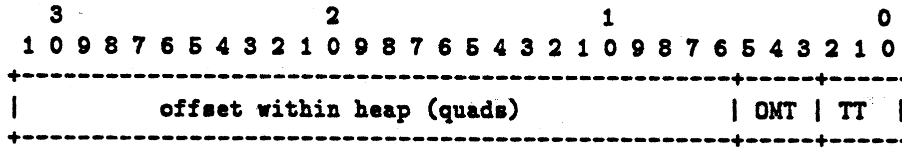
of some sort that allows procedures that want to extract slots from object to tell where each slot begins and which kind of reference it contains. This implementation would increase the cost of accessing slots in objects by an unacceptable amount.

A slightly different implementation approach that supports two kinds of references has the non-local references stored outside the object. An object has fixed size slots, but in addition to being able to contain a reference to a local object, a slot can contain a reference to a non-local reference. Any object slot that needs to contain a non-local reference instead contains a reference to a non-local reference. This reference to a non-local reference can be simply a local reference. Such a local reference can be distinguished from a local reference to a local object by reserving a bit for just that purpose. This bit can be either in the local reference or in the storage pointed to by the local reference. We will discuss this in detail later.

In summary, all objects that reside in the same heap can refer among themselves using local references. The slots in an object are the size of a local reference. For an object in one heap to refer to an object in another heap, it must go through an intermediate non-local reference. We assume that inter-heap references are infrequent. Another way of saying this is that objects that are part of one data structure or partition of a data structure are in a single heap.

Figure 3.2 diagrams three objects in two heaps. Object A (which has 5 slots) contains a reference to object B (which has 2 slots). Both objects A and B are in heap 1. Object A also contains a reference to object C (which has 3 slots). Note that object C is in heap 2. Thus, for object A to refer to object C, there must be a non-local-reference (labelled X in the diagram).

The nice property of this approach is that it is cheap in terms of both time (i.e. time to follow a reference) and space (i.e. space occupied by a reference) to connect two objects that are in the same heap. Thus, we are optimizing the kind of activity we expect to occur most frequently: local traversal and local reference. By local traversal we mean the following of references between objects within the same partition; programs tend to localize their traversal to a partition of a data structure (this is similar to the locality of reference argument in virtual memory systems). By local reference



- TT = 3 = T type code for RPointer type
- OMT = OM type code

Figure 3.3: RPointer representation

we mean reference between two objects within the same partition; an object within a partition most frequently needs to contain a reference to another object within the same partition.

An important property of local references is that they are meaningful only in the context of some heap. That is, in order to know what a local reference refers to, one has to know with what heap the local reference is associated. The simple and obvious rule here is that a local reference is always associated with the heap from which the local reference was itself extracted. Local references are not created out of thin air. All local references come from inside of objects that reside in some heap or are returned by a primitive that creates new objects. In the latter case, the heap is known because it was supplied by the caller to the creation primitive. In the former case whoever did the extraction must have known what heap he was extracting from and can associate the extracted local reference with that same heap. The only question is how one gets the first reference from the first heap. We will address this question later.

We use the term *RPointer* to mean "local reference". RPointers are byte offsets from the base of the heap in which the object being referred to resides. (The "R" in "RPointer" comes from the fact that RPointers are *R*elative to the base of a heap.) We use T's spare type code to indicate an object of the T type *OM RPointer*. RPointers are represented immediately in the 29 upper bits of the T reference.

We can now re-address the issue of the size of reference. A 29 bit RPointer allows heaps up to more than 500M bytes; objects can be up to this length. This certainly seems like enough for the near future.

3.3.5 RPointers within T

It is important to understand that to T there is nothing special about RPointers – they are simply 29 bit objects. OM mimics T's implementation of types: the low 3 bits of the RPointer form a type code which indicates the type of the OM object referred to by the RPointer. The meanings of the OM type codes (i.e. what type code means what type) is different from the meanings of the T type codes.

Figure 3.3 shows the format of an RPointer within a T reference.

3.3.6 Object code

In the present implementation of OM, object code can not reside in OM heaps. All code is loaded into the transient heap. The reason for this limitation is that the nature of an object module produced by the T compiler requires that when it is loaded into a process, portions of the module must have process virtual address written into the representation of the module in memory. We can not allow such process-dependent information in OM heaps.

The structure of object code is complex and intertwined with the T compiler. When compiling a module the T compiler produces an object module that contains a pure, position-independent code section and an impure data section. The pure code refers through the data section to get at values of variables in other modules. (Since T doesn't have special "function cells" but rather uses the normal variable binding mechanism to store procedure values, the looking up of values of variables in other modules is common.) When a module gets loaded, the part of the data section that is used this way by the code gets filled in to contain references to all the non-local variables that are referred to by the code in the module. Hence, the data section becomes impure.

3.4 Heaps

3.4.1 Heaps in plain T

T allocates objects using a simple heap allocation system. At startup it allocates two large pieces of the process virtual address space. We call these pieces the *transients heaps.* Since Aegis does not support the traditional concept of swap space (i.e. pieces of the disk that are dedicated to backing process pages that are not part of a disk file) T obtains these pieces of address space by mapping two temporary files into the process virtual address space. These files are deleted when T exits.

Only one transient heap is active at a time. A heap pointer held in a hardware register is initialized to hold the virtual address of the beginning of the active transient heap. When a procedure wants some storage to hold an object, it simply increases the heap pointer by the amount of storage it wants (rounded up to the nearest multiple of 8 bytes) and uses the old value of the heap pointer as the reference to the new object. When the heap pointer reaches the end of the active transient heap (i.e. when there is not enough room in the active transient heap to allocate an object) a *GC flip* occurs: the inactive heap becomes the active heap and a copying garbage collector is invoked to copy all the reachable objects from the previous active heap into the current active heap. The set of reachable objects is determined by recursively following all references from the *root set* of objects known a priori by the garbage collector and all references from variables on the program execution stack.

3.4.2 OM Heaps

Since OM runs within an existing operating system that has its own ideas about using the disk, we have to work within the operating system's filesystem. This is not too much of a problem – we can simply embed our system within a single large file. However, if we expect to work in a multi-user, multi-application environment it probably makes more sense if we use one file per heap. This allows individual users or applications to use normal file system primitives to copy, delete, backup, protect, and if necessary examine the contents of heaps he controls. If all the objects resided in one large file our system would have to duplicate these tools. We can consider each heap file as a separate disk and the system can function along the lines discussed earlier about a multi-disk system.

OM's basic extension to T is the introduction of support for multiple simultaneously active heaps. OM provides primitives for creating objects in these heaps. These primitives take an argument that identifies the heap in which the object is to be created. OM also provides primitives for accessing slots within objects. These primitives take both an argument that identifies the heap in which the object resides and an RPointer argument which identifies the particular object within the heap.

An OM heap is mapped into the process virtual address space when objects in the heap need to be referenced. The process of mapping is not very cheap; it requires no disk I/O until a reference is made, but the *map* call *is* a system call (requiring a context switch) and the manipulation of the memory translation hardware by Aegis is expensive compared with the cost of doing a single memory reference. However, we assume that once a heap is mapped that many references to objects within the heap will be made. We believe that the way to measure the performance of a system

such as ours is to measure the average cost of a reference. If the number of references per mapping operation is high, than the average cost of a reference is not substantially affected by the cost of the mapping operation.

OM heaps are position independent. An OM heap is a collection of OM objects that refer to each other using RPointers. Recall that RPointers are offsets from the base of the heap. This allows heaps to be placed at any position in the virtual address space without having to relocate the contents of the heap.

There are two reasons for wanting to avoid relocation to account for the position that a heap is mapped at. First, the cost of activating a heap (i.e. the steps required before the objects of interest in a heap can be examined) would be intolerably high. Worse yet, the cost would be proportional to the number of objects in the heap, not the number of objects one needs to examine ("on demand" relocation seems overly complex). The second problem is that if the heap's contents are relocated, then the heap can not be used simultaneously by multiple processes running on the same node. This is because the processes can not guarantee that the heap would be mapped into the same part of the virtual address space for all processes wanting to access the heap.

3.4.3 Naming OM heaps

The primitives that activate and deactivate heaps must have a way of referring to heaps. As we said earlier, heaps are stored in DOMAIN files, one heap per file. There are three possible ways of naming heaps:

1. Use DOMAIN path names (variable length strings).
2. Use DOMAIN file UIDs (64 bit integers).
3. Make up our own naming scheme.

Approach (1) is the obvious approach – users are already accustomed to dealing with DOMAIN path names. The DOMAIN naming system allows files to be organized hierarchically; related heaps could have similar names. The drawback to using path names is that they are long and not of fixed length; this increases the overhead required for manipulating them. As we will see later, heap names need to be embedded inside OM data structures and will be manipulated fairly frequently. Also, using path names means that the heap activation time includes the time it takes to turn a path name into a file UID.

Approach (2) solves the overhead problems and saves the pathname to UID conversion. However, UIDs are elements of a flat name space. The DOMAIN user interface is designed to deal with path names, not UIDs. For a prototype system such as the one we built, we want to make it convenient to deal with failures using tools in the surrounding environment. Using UIDs would have made this difficult.

We adopted approach (3). Our naming scheme uses 29 bit heap unique identifiers called *HIDs*. Being fixed length and small, HIDs are easy to manipulate. HIDs are assigned by OM which keeps a global word that holds the number of the next HID to assign. We assume that heaps are created relatively infrequently so that having a single global word won't be a serious bottleneck.

OM maintains a permanent global table translating HIDs into DOMAIN path names. When a HID is presented to the heap activation primitive, the HID is translated into a DOMAIN path name which is in turn translated into a UID of a file which is then mapped. (The translation table also can translate file names into HIDs; this feature is useful for debugging.) There is no reason that the HID translation table couldn't convert HIDs into UIDs except for the prototyping problems mentioned above. If OM were to be made into a production system, we *would* have the table contain a HID to UID translation.

The global table is itself represented as a OM object – a permanent hash table. The HID and path name of the heap holding the table object are known a priori. This heap is called the *HID heap*. The HID heap is also the place where the "next HID to use" counter is kept.

The HID-to-path-name translation table is a potential bottleneck. One way in which we reduce this problem is by having processes that are using OM keep a cache of translations that have already been requested. (The cache can implemented as a hash table kept in each process's address space.) Once a HID has been translated in one process, future translations of the same HID do not need to consult the global table. This is possible since HIDs are unique and never reassigned to refer to some other heap.

Another way to avoid the bottleneck of a global HID translation table is to have multiple tables. This can be implemented by dividing the HID into pieces in a way analogous to the scheme for dividing object references. In our prototype system the OM user can specify the path name of the HID heap so he can run his own private world of heaps and permanent objects; he can thus reduce the number of processes contending for access to the HID heap. In the prototype system each isolated application area – i.e. a set of application programs that do not need to refer to objects in another set of application programs – has its own HID heap. This is *not* a restriction of the current system; rather it is a suggested mode of operation that seems prudent while aspects of the system are still under development.

3.4.4 Active heaps

When a heap is activated, the heap can be characterized as a virtual address in a process and a length (i.e. the amount of space the heap occupies in the address space). We refer to the starting virtual address of an active heap as an *RHeapB*. The RHeapB of an active heap is all that is needed to dereference RPointers into the heap: the RHeapB is added to the RPointer to form a virtual address of a particular object.

It turns out that it is necessary to associate some additional information with an active heap. An object of a type called *RHeap* holds all the information associated with an active heap. RHeap objects contain:

- The RHeapB of the heap.
- The number of bytes mapped.
- The HID of the heap.
- The activation count of the heap for this process.

The heap activation primitive returns an RHeap record. All the OM primitive procedures for manipulating OM objects take an RHeap argument.

The purpose of the RHeapB field is clear. The reason for the byte count field is that the DOMAIN unmap primitive requires the length to unmap – there is no way to tell the Aegis to unmap as much as was mapped. Storing the HID of the active heap allows quick conversions from a reference for an active heap to the HID of the heap.

The idea behind the activation count is to optimize multiple invocations of the activate primitive on the same HID. Such multiple invocations do *not* result in the heap being mapped multiple times. (Aegis allows this, but it is clearly a waste of address space.) Rather all activations of a heap other than the first activation simply increment the activation count and return the previously allocated RHeap object for that heap. A table translating HIDs into RHeaps allows this; only one RHeap object is ever created in a process for a single heap. This table is part of the process context – it resides in the transient heap. Deactivating a heap decrements the heap activation count. When the activation count drops to zero, the heap is unmapped.

Encapsulating RHeapBs in RHeaps allows a heap to be moved around the process virtual address space simply by changing the RHeapB slot of the RHeap associated with the heap. This sort of motion is necessary because the heap is mapped for a particular size, and when it needs to grow beyond the size for which it was mapped, it must be unmapped and then remapped, and there is no guarantee that the heap will be mapped into the same spot.

One price for embedding RHeapBs in RHeaps is that it adds one extra memory reference (and probably one extra machine instruction) to the dereference procedure: given an RPointer and an RHeap, the RHeapB must first be extracted from the RHeap before an object's virtual address can be calculated. The cost of the memory reference is not a great concern since the DOMAIN hardware has a memory cache; we can safely assume that for multiple dereferences into a single heap, the RHeapB of the that heap will be in the cache.

The issue of managing the process virtual address space is one which we have not pursued extensively. We rely on the Aegis mapping primitive to determine where to map heaps. Its allocation strategy appears to be adequate for our purposes. One optimization we could easily make is to not actually unmap a heap just because its activation count has dropped to zero. We could keep it mapped until the address space became full and some heap that isn't already mapped is activated. At that point we could unmap the inactive heaps using some LRU strategy. This optimization will sometimes eliminate the cost of the mapping and unmapping operations. One reason we haven't adopted the optimization is because (1) it hasn't proved necessary, and (2) it raises some concurrency problems: if a process on one node wants to activate a heap that is inactive but still mapped into a process on another node, it will be unable to do so.

Another possibly useful address space management technique is to unmap heaps that are still active, but which do not appear to be being accessed. Such a heap could be unmapped and the RHeapB slot of the Rheap for the heap could be modified so that references through it would cause an addressing error. The error could be trapped by OM and the heap re-mapped. This technique would be useful if a process needed to have multiple large heaps simultaneously active. This is especially true in the present DOMAIN 24 bit addressing environment. With an address space of 4G byte (32 bit address), it is not clearly as important.

3.4.5 OM heaps in T

OM uses some knowledge about the internals of T in order to make OM heaps accessible to T procedures. To T, RHeapBs appear to be T extends; i.e. references to RHeapBs are extend-type references. With the T type field masked to zeros, an RHeapB reference is the starting address of a mapped heap. Note that this address is outside of the transient heap and hence, from T's perspective, the RHeapB reference is invalid. (For a reference to be valid to T, when viewed as an address, the reference must be to a part of the address space where the current transient heap is mapped.) Fortunately, the invalidity doesn't matter. The only potential serious source of problem might be the T garbage collector. However, when the garbage collector encounters an apparently invalid pointer, it simply copies the pointer to the new transient heap and does not follow it.

Another small problem is that T expects to see a template pointer in the first cell of an extend. Clearly it is not possible to embed a T template pointer into an OM heap – it would violate the process context and position independence properties of the heap. But not filling in the template pointer slot causes no problems unless an operation is applied to the RHeapB reference; in no other case does T refer to the template pointer slot.

Since OM heaps appear to T to be extends, a cell from an OM heap can be accessed using `EXTEND-ELT`, the T primitive for accessing a cell in an extend. The lowest level OM procedures use `EXTEND-ELT`.

3.4.6 Summary of heap features

Having described the implementation properties of heaps, let us review why the heap approach makes sense.

Heaps are position independent. Since heaps never contain any machine addresses, heaps can be mapped into any part of a process's virtual address space without any relocation being required.

Relocation is undesirable because it is time-consuming and makes it impossible to share the heap between multiple processes.

Heaps take advantage of clustering properties in configurations of objects (i.e. data structures).

Heaps take advantage of DOMAIN paging facilities. Only those disk pages of heaps that contain objects that are actually referenced are transferred from the disk into main memory. This transfer is the responsibility of the Aegis paging system.

3.5 Intra-heap references

OM extends T with a set of procedures that take RPointers and RHeaps as arguments. These procedures fall into two general categories: *allocators* and *accessors*.

An allocator creates a new OM object. An OM object is a contiguous piece of an OM heap. The slots of the object can contain immediate values or references to other OM objects. An allocator takes at least two arguments – the heap in which the object is to be allocated, and the type of the new object. An allocator may take additional arguments which specify things like the initial values of parts of the object. In terms of the OM implementation, an allocator takes at least one RHeap argument and returns an RPointer. In terms of the OM interface that the programmer sees, the allocator returns an OM object.

An accessor retrieves or modifies a slot in an OM object. In terms of the OM implementation, an accessor takes at least one RHeap argument and one RPointer argument and returns an RPointer. In terms of the OM programmer interface, an accessor takes an OM object and returns an OM object.

An RPointer and an RHeap argument taken together form one logical argument that refers to one object. Thus, for simplicity we will sometimes say that such procedures take "RPointer/RHeap arguments". Also, we say that some pair of variables `R/H` refer to an object if `R` is a variable whose value is an RPointer to an object in a heap that is referred to by `H`. Since we are describing the OM implementation, we tend to say that a procedure takes an RPointer/RHeap and returns an RPointer. However it is important to note that RPointers are an artifact of the OM implementation. The programmer who uses OM thinks of the procedure as taking or returning an OM object.

3.5.1 Active objects

Note that accessors and allocators manipulate only objects in active heaps. We call such objects *active objects*. There are no primitives that take an RPointer and, say, a HID to identify a particular object. Access to an object in this way would be very inefficient. Each access would have to insure that the heap referred to by the HID is active. If it is active, the associated RHeap would have to be located; if it is not active, the heap would have to be activated. But would the heap be deactivated after the access is complete? Clearly activating and deactivating around each access is too expensive. The set of active heaps might be treated like pages in a virtual memory system. Heaps would be activated and deactivated based on some usage pattern.

One might argue that we have brought this expense on ourselves. That is, by introducing the notion of heaps we have also introduced the inefficiency of having to activate and deactivate heaps. However, in any system that deals with disk storage these problems will arise. In the ideal world accessing the disk would be as fast as accessing main memory and the disk could be treated as an enormous flat address space. Access to an object would be implemented as a direct fetch of the object's representation from the disk. In the real world, data must be transferred from the disk to main memory in large chunks if access is to be efficient. Viewed in this way, heap activation is simply the preparation for bulk disk data transfer. No system can avoid this kind of preparation.

In short, we feel that the complexity of managing heap activation is better left to a higher level

of the system. The higher levels, having a notion about the logical behavior of a program, will be able to better guess when a heap should be activated and deactivated. At the low level of primitive accessors and allocators, activation is explicit and only active heaps can be manipulated.

3.5.2 An example: OM pairs

Let us consider an OM pair (also known as a "*cons* cell"). OM pairs are 8 bytes long – enough for 2 slots. The procedure `!CONS` creates a new OM pair and initializes the pair's slots. `!CONS` takes three arguments: the RHeap of the heap in which the pair is to be allocated, the initial contents of the first slot (called the *car*), and the initial contents of the second slot (called the *cdr*).

Let us look at what `!CONS` must do. First, it must allocate space from the heap. For every builtin OM type, there is a procedure that allocates an object of that type and does nothing to the contents of the object. For OM pairs, this procedure is called `!PAIR-ALLOC`:

```
(DEFINE (!CONS P1 P2 HEAP)
  (LET ((RP (!PAIR-ALLOC HEAP)))
    (SET (!PAIR-CAR RP HEAP) P1)
    (SET (!PAIR-CDR RP HEAP) P2)
    RP))
```

`!CONS` calls `!PAIR-ALLOC` and then uses the OM pair accessors to initialize the contents of the the pair. `!PAIR-ALLOC` uses one of a set of low-level procedures that manipulate heap contents directly and are not accessible to the user of OM. One of these procedures is called `RHEAP-ALLOC`.

```
(DEFINE (!PAIR-ALLOC HEAP)
  (MAKE-RPOINTER (RHEAP-ALLOC HEAP 2) %%!PAIR-TAG))
```

`RHEAP-ALLOC` takes an RHeap argument and a number of cells to allocate and returns an integer offset into the heap. `MAKE-RPOINTER` is a primitive that creates an RPointer (immediate) object from an integer offset and an integer value for the RPointer tag field.

Before allocating space, `RHEAP-ALLOC` must insure that there is room in the heap. Two questions that must be answered before allocation can happen:

1. Can the size of the heap be extended without extending past the amount for which the heap is currently mapped?
2. If the answer to (1) is no, should the heap be extended or garbage collected?

Every OM heap has a heap pointer at a fixed, known location within the heap. The heap pointer is used just like the the T transient heap pointer. When a heap is activated, it is mapped for its current size. (Determining the current length of a heap does not require mapping the first page of the heap for the sole purpose of extracting the heap length field from the heap. This is because the length can be obtained from the file length maintained by Aegis.) As we said earlier, the actual amount of address space mapped is the next highest multiple of 32K bytes. Thus, the heap pointer can typically advance some before the heap needs to be remapped for a larger size.

The process of remapping for larger sizes continues as the heap expands until the heap grows to a specified size. This size is called the *heap max* which, like the heap pointer, is at a fixed, known location within the heap. The heap max is a settable parameter for a heap. When the size of a heap reaches the heap max for that heap, the garbage collector is invoked to reduce the size of the heap (we will discuss garbage collection later). It is up to the application programmer to divide his data in such a way that heap sizes do not grow in an unbounded way (i.e. that when a heap reaches its heap max that it is not because the heap is full of *non*-garbage).

To describe `RHEAP-ALLOC`'s behavior concretely: it compares the the heap pointer plus the allocation request to the length for which the heap is mapped. If there is room, the heap pointer is simply

incremented. If there is not room but the heap's size is less than the heap max, the heap is remapped for a larger size. (While remapping is expensive relative to the cost of incrementing the heap pointer, remapping happens infrequently compared to the number of times the heap pointer is incremented.) If the heap max is reached, the garbage collector is invoked.

!PAIR-CAR and !PAIR-CDR are the primitive accessors for OM pairs; they access a pair's *car* and *cdr* slots, respectively. In the context of the SET special form, these accessors modify the contents of a pair. Let us consider !PAIR-CDR in detail (!PAIR-CAR works analogously). !PAIR-CDR takes an RPointer and RHeap argument and calls RPOINTER-EXAMINE.

```
(DEFINE (!PAIR-CDR P HEAP)
  (RPOINTER-EXAMINE P HEAP 1))
```

The RPOINTER-... procedures are part of the OM implementation and are not available to application programmers. All OM objects are accessed using these procedures. RPOINTER-EXAMINE takes an RPointer, an RHeap, and a cell index, computes the total offset from the base of the heap, and calls RHEAP-EXAMINE.

```
(DEFINE (RPOINTER-EXAMINE RP H I)
    (RHEAP-EXAMINE H (+ I (RPOINTER-CADDRESS RP))))
```

RHEAP-EXAMINE is a procedure that takes an RHeapB and an integer cell index and returns the contents of the specified cell of the heap. RPOINTER-CADDRESS extracts the cell number part of an RPointer. RHEAP-EXAMINE is just another name for EXTEND-ELT, the T procedure for accessing an element of an extend (recall that heaps look like extends to T).

All the procedures mentioned in the preceding paragraph are *integrable* [6] so that there is no procedure call overhead. OM contains no explicit machine language instructions. It relies solely on T primitives and the T compiler.

The T compiler compiles CAR into 2 68000 instructions (3.8 μsec on a 10mHz 68000). The T compiler compiles !PAIR-CAR into 14 instructions (14.4 μsec). A T compiler that was somewhat smarter, but still had no built-in knowledge about OM procedures could reduce that to 7 instructions (11.4 μsec), 3 of which were simply shifts on registers (i.e. had no memory operand). There is an ongoing effort by the implementors of T to produce a new T compiler that can produce substantially better code than the current T compiler [45], and we expect that the new compiler will be able to produce the 7 instruction version.

3.5.3 Arguments to OM procedures

OM's procedures for manipulating OM objects are modelled after T's procedures for manipulating T objects. The major difference between OM's and T's procedures is that OM procedures take one additional argument for each argument that refers to an object in a heap. This extra argument is an RHeap. Some OM procedures take several RPointer arguments and only one RHeap argument. These procedures assume that all the arguments refer to objects in a single heap – the one specified by the RHeap argument. Some OM procedures take one RHeap argument for each RPointer argument.

The fact that OM procedures require these extra arguments make them somewhat inconvenient for the application programmer. We will pursue this issue in the next chapter.

3.6 OM types: Introduction

As in the T type system, some OM types are identified with type codes and others with the extend mechanism. The following types have reserved type codes:

[6] T's term for procedures whose bodies are substituted inline at the call position.

- Pair
- String
- Text
- Null
- Extend
- Type ID
- Non-local reference

One type code is presently unused.

3.6.1 Non-extends

We have already discussed pairs. One additional item about pairs is that they are often chained together by their *cdrs* to form a *list* of pairs.

Character strings are implemented in two parts. An object of the text type is a fixed length vector of characters with a length at the front. An object of the string type is a reference to an object of the text type plus an index and a length which select a portion of the text object.

The null type is a set containing exactly one object – *null*. *Null's* major function is to mark the end of a list: the *cdr* of the last pair in a list of pairs contains *null*.

In addition to these types, all T objects that are represented immediately (e.g. fixnums and characters) are valid OM objects. T objects that are *not* represented immediately can not be OM objects because their representation is part of the transient heap.

3.6.2 Extends and type identifiers

The extend type is not really a type at all but a flag that tells OM that the type of the object being referred to (called the *extend*) is determined by the contents of the first slot of the extend. In T, this slot contains a reference to the handler, the object code object that implements operations on objects containing that reference. In both T and OM extends are used to represent all objects of user-defined type. Since OM can not store object code in heaps, we need some way of indirectly referring to an object's handler. Even if we could store object code in heaps, we might still want this level of indirection.

We have already explained why it is difficult to include object code in OM heaps and why we have decided that all object code resides in the transient heap. However, it is not possible to refer to an object in the transient heap from an object in an OM heap. The contents of the transient heap are specific to a single process. If we were to put a reference to a transient heap object into an OM heap, the OM heap would not be free of dependencies upon a particular process. Thus, we can not make the first slot of an extend pointer to object code that resides in the transient heap. Since extends reside inside heaps and the object code that supports extends reside outside heaps, it is necessary to have a mechanism for finding something outside a heap from something inside a heap. This mechanism must rely on some data structure that is not tied to a process's context.

OM has objects of type *type identifier* for identifying the type of an object without reference to an object in the transient heap. Type IDs are represented immediately in the upper 26 bits of RPointers. Type IDs are simply integers in the range $[0..2^{26}-1]$. Each type ID identifies some type – ultimately some piece of code that implements operations on objects of the type. Unlike T's template pointers (which can be considered a type ID of sorts since template pointers define how objects respond to operations), type IDs are: (1) not direct pointers to object code, and (2) *are* presented to the application programmer. A type ID is an indirection mechanism that allows the

specification of an extend's type to be separate from the object code that implements operation on the object. Type IDs are stored in the first cell of OM extends.

Extend types come in two varieties: primitive and user-defined. Primitive extend types are extend types about which OM has built-in knowledge. Vectors are examples of primitive extends (i.e. objects of some primitive extend type). The essential property of primitive extend types is that their type ID is fixed and known by OM. The handlers for primitive extends are built in to OM. We will discuss user-defined types later.

3.7 Inter-heap references

The previous two sections have dealt with the issues of objects within a single heap. This section deals with the mechanisms that allow objects to refer across heaps.

3.7.1 Non-local references and garbage collection

An OM object can be completely identified by identifying the heap in which the object resides and the particular object within the heap. As discussed in section 3.4.3, heaps are named with heap identifiers – HIDs. Given a HID, we can identify an object within the heap named by that HID with an RPointer. Thus, it seems that a HID, RPointer pair can be the non-local reference discussed in section 3.3.4.

However, this scheme is not adequate since it makes the independent garbage collection of heaps impossible. Independent garbage collection requires that it is possible to identify all the objects that are referred to by other objects. In general, non-local references to an object appear outside the heap that contains the object. Thus, given the present scheme, in order to garbage collect a single heap, all heaps must be examined to see if they contain non-local references to objects in the heap being garbage collected. Scanning all the heaps to find references into the heap being garbage collected would be nearly as expensive as garbage collecting all the heaps and as a result we could not consider the garbage collector as capable of garbage collecting heaps independently.

To garbage collect heaps independently it is not necessary to know *where* the non-local references to objects in the heap being garbage collected are, only *that* such non-local references exist and to what they refer. At garbage collection time, knowing that the non-local references exist need not require finding all the non-local references as long as every time such a reference is formed, that fact is recorded some place easily accessible to the garbage collector. That is, that when a non-local reference is formed, an entry is made in a special part of the heap containing the object being referred to. We call this part of the heap the *heap index.*

The heap index is a vector of RPointers to all the objects inside the heap that are referred to by non-local references outside the heap. The size of a heap's index is fixed at the time the heap is created. We call the process of adding an RPointer to the heap index *exporting.* A reference count is associated with each RPointer in the index. The reference count indicates how many non-local references are using that element in the index. If the reference count for an element is zero then the element is considered to be free – it can be used the next time an RPointer needs to be exported.

Garbage collecting a heap consists simply of following all the references leading from objects in the heap index. All objects found by this procedure are copied into a new heap. Once all the objects are copied, the old heap can be deleted. Note that the entry in the HID heap (translating HIDs to DOMAIN file names or file UIDs) must be updated to reflect the fact that the heap is backed by a new file. We will discuss more of the details of garbage collection in the next section.

The heap index allows the non-garbage in a heap to be identified. However, a problem still remains: in general, after garbage collection the offsets of the non-garbage objects have changed. Thus any non-local references in other heaps will be wrong. To solve this problem in the present non-local reference scheme requires that the garbage collector can find and fix all the non-local references to

```
Gc( Heap );
begin
    NewHeap := MakeHeap();

    for I := 1 to SizeOHeapIndex( NewHeap ) do
        SetHeapIndexSlot( NewHeap,
                          I,
                          GcOne( HeapIndexSlot( Heap, I ), NewHeap )
                          );

    DeleteHeap( NewHeap );
    end;

GcOne( Obj, Heap );
begin
    if Atomic( Obj ) then
        NewObj := CopyAtom( Obj, Heap )
    else begin
        NewObj := MakeObj( SizeOfObject( Obj ), Heap );

        for I := 1 to SizeOfObject( Obj ) do

            SetObjSlot( NewObj, I, GcOne( ObjSlot( Obj, i ) );
        end;

    return NewObj;
    end;
```

Figure 3.4: Sketch of the garbage collector

the heap being garbage collected. However, if we modify the format of non-local references, we can avoid the problem.

Let us change the format of non-local references to contain a HID and a ***heap index offset***, rather than a HID and an RPointer. A heap index offset is an integer that identifies a particular element of a heap index. We call these non-local references *LPointers* (the "L" is for "long"). As a part of garbage collection, the index is copied from the old heap to the new heap, all the elements of the index being modified to contain the new positions of objects referenced from the index. Since LPointers refer to objects indirectly through the heap index, and because the garbage collector has insured that the elements of the index refer to the same objects they did before garbage collection, the LPointers do not need to be modified.

Figure 3.4 contains a sketch of the garbage collector. The garbage collector performs a tree walk of all the objects in the heap. The heap index is the root of the tree. When an atom (leaf) is reached, its contents are simply copied into the new heap. For an internal node, a node is created in the new heap. The new node's slots are filled with the values of recursively applying the garbage collector to all the old node's slots.

Note that for the purposes of the above sketch, *LPointers are atoms*. That is, the garbage collector tree walk does not follow LPointers to objects in other heaps. The point of our scheme is to allow heaps to be garbage collected independently, not to garbage collect all heaps at once.

How is the heap index maintained? So far all we've said is that when a LPointer is formed, an entry is made in the heap index; the entry contains an RPointer to the object which the LPointer is

to identify. What happens when the object that refers to the LPointer becomes garbage? At that point, the LPointer becomes garbage. When all the LPointers that name a particular heap index element become garbage, then the object referred to by the RPointer in the heap index elements becomes garbage too. Garbage collection as we've described it doesn't do anything about garbage LPointers and there is no mechanism for deallocating elements of the heap index.

Our goal is to free elements of the heap index when all the LPointers that are using an element becomes garbage. To do this we modify the garbage collector so that after all the non-garbage has been copied from the old heap to the new heap, all the garbage LPointers in the old heap are examined. For each garbage LPointer, the reference count of the index element of the heap referred to by the LPointer is decremented by one. When the count reaches zero, the space occupied by the object that is no longer referred to by any LPointers is not reclaimed – the space is reclaimed only when the heap containing that object is itself garbage collected. At that time since the object is no longer referred to from the index, the object will not be copied into the new heap and the space is thus reclaimed (assuming the object that is not referenced from the index is also not referenced from some non-garbage object in the heap).

To be able to traverse all the garbage LPointers at the end of garbage collection, it must be possible to find all the LPointers in a heap. This can be achieved by maintaining a linked list of LPointers whose root is at some fixed place in the heap. Traditional garbage collection techniques require one to be able to determine whether an object has been copied out already. Thus, at the end of garbage collection, this list can be traversed and any LPointers that have not been copied to the new heap are garbage and the procedure described above can be applied to them.

Note that the above scheme does not handle circular references across heaps. For example, if object A in heap 1 contains a reference to an LPointer to object B in heap 2, and object B contains a reference to an LPointer to object A, then even if there are no other references to objects A and B, then the space occupied by A and B will never be reclaimed by the garbage collector. In general, only by garbage collecting a set of heaps at once can the circularly linked garbage objects in that set of heaps be found and removed.

3.7.2 LPointers in detail

LPointers must be large enough to contain a HID and an offset into a heap index. Ideally, LPointers would be represented as T immediate values the way RPointers are. Unfortunately, T does not have any more spare type codes. However, it is worth examining the packing of LPointers into T references since in the long run T's reference format might change to allow more immediate types.

Are 32 bits enough to hold a HID and an offset into a heap index? First we need to decide whether HIDs are to be unique for all time. Unique HIDs allow HIDs to be explicitly deleted. If HIDs are unique, reference from LPointers to the contents of a deleted heap can be detected because we are guaranteed that the HID will not have been reassigned to another heap. While we argued against using UIDs for object references because of performance problems, since the frequency at which HIDs have to be "dereferenced" is less than the frequency at which object references have to be dereferenced, we choose to use UIDs for HIDs because of explicit deletion capability.

Having decided to use unique HIDs, we must be fairly generous in allocating bits for HIDs. It is not unusual for a moderate size timesharing system to have more than 32K files (requiring 15 bits) *at a single instant.* Over the lifetime of a system, the total number of files created can be presumed to be much larger. The ideal way to generate UIDs is to allocate them consecutively as they are needed. However, this requires access to a central piece of data that holds the next UID to assign. To avoid this centralization, UID generation schemes typically embed a processor ID in the UID and let each processor pick its own local part of the UID. Since the size of the processor ID is fixed and determined by how many processors are expected to ever exist, this generally increases the number of bits that must be allocated to the whole UID. Also, since it is desirable that UIDs are in fact reliably unique, UID generation schemes typically use a monotonically increasing hardware

clock as part of the UID. Since the resolution of the clock must be small enough to allow two UIDs generated back-to-back to be unique, the number of bits assigned to the clock-based part of the UID is typically large. DOMAIN file UIDs are generated using essentially the scheme described above and are 64 bits in length.

Even if we were fairly miserly in our allocation of bits to HIDs, it seems unlikely that we could be miserly enough so as to be able to pack both a HID and a heap index offset into less than 32 bits.

In choosing the format of LPointers, once we decide that LPointers can not be made to fit within a normal sized (i.e. local) reference, our options are less constrained: the format of LPointers can be chosen to be what seems logically correct, not simply what can be packed into a small place. However, this freedom has a price. T (and OM) procedures pass and return fixed-size references; there is no provision for passing and returing aggregates (objects with non-immediate representations). Thus, all aggregates must be allocated in the heap. Heap allocation is not free – the more heap allocated objects there are, the more expensive garbage collection becomes.

We chose LPointers to be 2 cells (8 bytes) long. The first cell contains a HID and the second contains the heap index offset. Since the heap allocation granularity is 8 bytes, it would not have made sense to have a more compact LPointer. There can no doubt that one cell is sufficient to hold the index offset. Given our model of the use of heaps – that data structures are partitioned so that most of the references are between objects in the same heap – 2^{32} incoming references is certainly sufficient.

That 4 bytes are sufficient to hold a HID is more open to question. It is certainly enough given OM's present scheme for generating HIDs – consecutively and based on a central count held in the HID heap – but we do not expect that this scheme would be used in a production version of OM because of the problems discussed above. Other systems, like the DOMAIN, that use UIDs generally are more liberal in their allocation of bits to UIDs. OM's design does not preclude the use of larger HIDs. In the current implementation of OM, as an aid to debugging, both the index offset and the HID are represented as T Fixnums, thus reducing the number of incoming LPointers and the number of heaps to 2^{28}. There is no reason why these values could not be full 32-bit integers.

LPointers are a type of OM object. They can be manipulated by OM procedures that are available to the OM programmer. Note that this makes LPointers different from RPointers, which are an artifact of the OM *implementation* and in principle are of no more business to the OM programmer than are addresses to the T programmer.

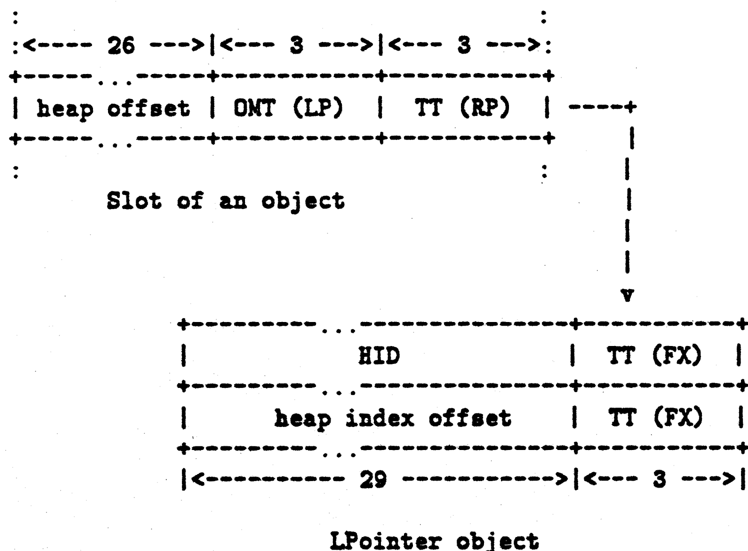
Figure 3.5 diagrams a slot of an object that contains a reference to an object in another heap.

3.7.3 Making LPointers

LPointers are made with the `!EXPORT-RPOINTER` procedure. This procedure takes an RPointer/-RHeap to specify some object to be exported. It also takes another RHeap argument to specify in what heap the LPointer is to be allocated. The procedure returns (an RPointer to) a newly allocated LPointer.

```
(DEFINE (!EXPORT-RPOINTER RP HEAP TO-HEAP)
  (LET ((ELT (RHEAP-ALLOC-INDEX-ELT RP HEAP)))
    (IF (NOT ELT)
        (ERROR "can't allocate index element"))
    (MAKE-!LPOINTER TO-HEAP (RHEAP-HID HEAP) ELT)))
```

`!EXPORT-RPOINTER` uses the RHeap primitive `RHEAP-ALLOC-INDEX-ELT` to allocate and initialize a slot in a heap index. `RHEAP-ALLOC-INDEX-ELT` returns the integer offset of the slot in the index. `MAKE-!LPOINTER` allocates an LPointer in the heap specified by the first argument and initializes the two slots of the LPointer to the second and third arguments respectively. The newly created LPointer is added to the `TO-HEAP`'s list of LPointers contained within `TO-HEAP`. Note that we take advantage of the fact that we store the HID in the RHeap structure (`RHEAP-HID` extracts the HID field from the RHeap structure).



TT (RP) = 3 = T type code for RPointer

TT (FX) = 0 = T type code for Fixnum

OMT (LP) = 1 = OM type code for LPointer

Figure 3.5: An inter-heap reference

In the current version of OM, RHEAP-ALLOC-INDEX-ELT is not terribly smart. It simply scans the heap index looking for an element whose reference count is zero. The process of finding a free index element could certainly be optimized. For example, we could link together all the free entries.

The procedure !EXPORT-RPOINTER-WITH-EXISTING-INDEX-ELT is similar to !EXPORT-RPOINTER except that it requires the RPointer passed to it *already be present in the heap index.* If the RPointer is found in the index, the appropriate reference count is increased by one and the index offset is used in the newly created LPointer. If the RPointer is not found in the index, the procedure behaves just like !EXPORT-RPOINTER. The idea behind !EXPORT-RPOINTER-WITH-EXISTING-INDEX-ELT is that it is desirable that multiple LPointers to the same object share the same index element. That way the size of the index can be minimized. If an application program knows that an object it is exporting is not already in the index, it can use !EXPORT-RPOINTER which does not require the index to be scanned (assuming the optimized version of RHEAP-ALLOC-INDEX-ELT. Otherwise it must use !EXPORT-RPOINTER-WITH-EXISTING-INDEX-ELT

Recall that in our initial discussion of non-local reference in section 3.3.4 we pointed out that it would be necessary to have a bit to distinguish local references to local objects from local references to non-local references. It should now be clear that this bit becomes available simply by virtue of our type tag scheme. One of the RPointer type codes is used to indicate a reference to an LPointer.

3.7.4 Dereferencing LPointers

OM's primitive procedures manipulate active objects. The procedures take one or more RPointer/RHeap arguments to indicate what objects are to be manipulated. LPointers can refer to any object, active or not. Thus, in general, given an LPointer to an object, it is first necessary activate the object. This conversion results in an RPointer/RHeap that refers to the now-active object and

can be used to manipulate the object.

The procedure `!LPOINTER-CONTENTS` takes an RPointer/RHeap to an LPointer (recall that LPointers are themselves OM objects) and returns an RPointer to the object referred to by the LPointer. Note that the returned RPointer can be interpreted only in the context of the heap identified in the LPointer.

`LPOINTER-CONTENTS` is defined as:

```
(DEFINE (!LPOINTER-CONTENTS LP HEAP)
     (RHEAP-INDEX-ELT-VALUE (HID->RHEAP (!LPOINTER-HID LP HEAP))
                            (!LPOINTER-INDEX LP HEAP)))
```

Let us examine it in some detail. `!LPOINTER-INDEX` and `!LPOINTER-HID` are the accessors for LPointer objects. `!LPOINTER-HID` returns the HID field of an LPointer; `!LPOINTER-INDEX` returns the index offset field of an LPointer.

`HID->RHEAP` takes a HID and, if the heap named by the HID is active, returns the RHeap for the active heap; if the named heap is not active, the procedure returns *false*. Note that before calling `!LPOINTER-CONTENTS` the heap referenced by the LPointer argument to `!LPOINTER-CONTENTS` must have been activated; e.g. by executing:

```
(ACTIVATE-HEAP (!LPOINTER-HID LP HEAP))
```

`RHEAP-INDEX-ELT-VALUE` returns the RPointer at the specified offset into the specified active heap's index.

Suppose a variable contains (an RPointer to) an LPointer to a pair. The following procedure returns the *cdr* of the pair:

```
(DEFINE (!PAIR-CAR-VIA-LPOINTER LP HEAP)
    (!PAIR-CDR (!LPOINTER-CONTENTS LP HEAP)
              (ACTIVATE-HEAP (!LPOINTER-HID LP HEAP))))
```

3.7.5 Comparison with Bishop's ORSLA

Bishop's thesis [14] describes ORSLA, a system that is in some ways similar to ours. ORSLA depends on special hardware; neither the hardware or software was actually built. ORSLA has *areas* which correspond to OM heaps. ORSLA has only one kind of reference. However, to enable the independent garbage collection of areas, all references between areas go through *inter-area links* (IALs). IALs are special objects understood by the hardware. The hardware makes a reference to an IAL appear to be to the object to which the IAL refers. IALs are similar to OM's LPointers except that IALs contain actual object references, not something like LPointer's offset into a table of object references.

Each area has two distinguished lists: a list of all IALs inside the area, and a list of all IALs outside the area that refer to objects inside the area. The first list contains *outgoing* IALs and the second list contains *incoming* IALs (these terms are with respect to a particular area). Since every IAL is both inside some area and pointing into some other area, every IAL is on two lists. The root of the ORSLA garbage collection of an area is the list of incoming IALs.

Since ORSLA has a single form of reference, it is conceivable that IALs could be placed in the area of the object to which the IAL refers instead of the area of the object that contains the reference to the IAL. However, as Bishop notes, this would make it impossible to garbage collect areas independently since when the IAL moved as a result of its being in a heap that was being garbage collected, the reference to the IAL from the object in the other heap could not be fixed.

Note that in OM, the analog of an IAL is the combination of an LPointer *and* an element of a heap index. That is, in a sense we have a two-piece IAL, half of which is in the source of the non-local

reference and half of which is in the target of the non-local reference. Only the latter half is relevent to the garbage collector. That this piece of information is in the heap being garbage collected, rather than in some other heap, is important. It means that the locality of reference of the garbage collector is improved – it doesn't have to touch all the heaps in which non-local references to the heap being garbage collected reside. In the ORSLA garbage collector, the roots of the garbage collector are spread throughout many heaps, all of which have to be touched.

OM does have a locality of reference problem though: at the end of the garbage collection, if there are garbage LPointers, the indexes of the various heaps referred to by the LPointers will have to be modified. Thus, the degree of non-locality of reference in OM garbage collection is proportional to the number of garbage outbound non-local references. The degree of non-locality of reference in ORSLA garbage collection is proportional to the number of non-garbage inbound non-local references. Which system's garbage collector has the better behavior (i.e. minimizes the amount of non-locality of reference) can be determined only experimentally. Note that OM's garbage collection procedure is amenable to techniques for increasing locality. For example, the heap indexes might be stored separately from the heaps themselves. Multiple indexes might be packed together to increase the locality of reference.

3.8 Concurrent access to heaps

If we want the data structures stored in heaps to be accessible by multiple processes running concurrently, we need to examine what techniques need to be used to assure the integrity of the data.

In this section we will consider the case of multiple processes running within a single physical main memory (i.e. on a single DOMAIN node) trying to concurrently access a heap. OM does not allow a single heap to be accessed by multiple processes that are not sharing a single physical main memory. This is because the OM implementation uses the Aegis file mapping primitives and these primitives do not support that sort of concurrent access.

3.8.1 Controlling concurrency

The correct manipulation of certain parts of a heap requires that a single process have exclusive access to the heap while the manipulation is happening. Advancing the heap pointer is an example of such a manipulation. The allocation mechanism must be able to get the current value of the heap pointer and then increment it atomically. Similarly, the heap index must be accessed in a way that insures that two processes do not obtain the same index element as a result of exporting an RPointer. These concurrency problems are not limited to the parts of the heap that are examined and modified by only the OM implementation. In general, application programs that can run concurrently on the same heap need to control access to objects in the heap.

Aegis has two mechanisms for controlling concurrent access to data: file locking and eventcounts.

File locking allows a process to map a file in a way that restricts the way other processes can map the file. For example, a process can map a file for read/write access and lock the file so that other processes can have read but not write access to the file. File locking provides fairly coarsely grained control of concurrency. The lock is set when the file is mapped; the success of the mapping operation is determined by what locks are already set at the time the operation is executed. Thus, using the locking mechanism requires the process to re-map the heap file before and after each operation, or set of operations that need to be atomic. Aegis does not have a mechanism for automatically blocking a process that attempts to map a file in a way that is not allowed by the existing locks. Thus, the process would have to "busy wait", re-trying the map operation periodically. Clearly, this overhead would be unacceptably high for operations like advancing the heap pointer.

However, file locking *is* appropriate when the lock will be held for a relatively long period of time and the expected concurrency is low. For example, in a mail system, there is a possibility for concurrency in the accesses made by the program that adds to mail to a mail box and the program that reads the mail box. Each program can map and lock the heap for the duration of some logical operation. For the former it would be during the operations that constitute adding the new message; for the latter it would be during an operation like displaying the headers of all the messages in a mail box. These durations are long compared to the duration of the operation of advancing the heap pointer. If there is contention for the heap, it is acceptable for the programs to busy wait, trying to map every second or so. The user won't notice and the program won't be wasting CPU cycles too much.

Eventcounts allow processes to synchronize at a finer level and with less overhead than with file locking. Eventcounts are equivalent in power with semaphores. All processes accessing the same part of a heap must agree to obey the semaphore associated with that part of the heap. In the case of application related data, the semaphore can be referenced from the object whose contents are to be accessed concurrently. The Aegis eventcount primitives allow a process to block until the eventcount indicates that the process has exclusive access to the object.

The problem with eventcounts is that they introduce overhead. The overhead is in the cost of the check of the eventcount before the data can be accessed. (This check is a system call to Aegis.) In cases where it is appropriate, the file locking approach has less overhead because no checks need to be made before each access.

A special case that we expect OM needs to deal with is concurrency on *only* the OM-internal parts of the heap (e.g. the heap pointer and index) and not on an application's object inside the heap. Since the access patterns to these internal data structures are well known, it is reasonable that concurrency control be implemented using the hardware "test-and-set" instruction and busy waiting since we know that the process will never have to wait too long. Ideally, the busy wait loop should include a call to the operating system suggesting that it select another process to run[7]. As opposed the the eventcount approach, in this approach the operating system call happens only if the resource is locked. Thus, with a resource that is almost always unlocked (e.g. the heap pointer), the test-and-set approach is much cheaper than the eventcount approach.

3.8.2 Garbage collection

The OM heap garbage collection procedure we've described is correct only if no processes are manipulating a heap when the garbage collector begins processing that heap. If the garbage collector can be invoked asynchronously (e.g. in the middle of an object allocation primitive) then it is possible that the only reference (RPointer) to an object is in a variable on a process's execution stack (or in a register). Since the garbage collector traces objects only from the heap index, an object referred to from only the stack will be discarded. Also, in general, RPointers on the stack to objects that are not discarded will be incorrect after the garbage collection because the garbage collector may have moved the objects.

Traditional garbage collectors solve the problem of references from the stack by putting those references in the root set at the start of the collection. Unfortunately, we can not easily use this technique because it is not possible to tell what heap an RPointer on the stack refers to. Without knowing the heap associated with these RPointer, the garbage collector can not trace through the RPointers on the stack.

There is no easy solution to this problem. The current implementation of OM simply does not allow the garbage collector to be invoked asynchronously. This restriction is severe but does not make the current implementation unusable. Not being able to garbage collect asynchronously is a problem only if applications are creating garbage rapidly. If garbage is not being created rapidly, the rate at which the heap needs to be garbage collected is low. If each run of an application does not create a lot of garbage, it is a reasonable restriction that the garbage collector can be invoked

[7] Unfortunately Aegis does not supply the necessary functionality to do this, but it would not be difficult to add.

only between (and not during) runs of the application. Our view is that OM heaps are used for archival storage, not intermediate results that quickly become garbage. Such intermediate results should be allocated in the transient heap.

If we want to support asynchonous invocation of the garbage collector, we must make it possible for the collector to determine the heap associated with every RPointer on the stack. For each RPointer in a stack frame (resulting from one procedure activation) there must be an RHeap that is associated with that RPointer and that RHeap must be in the same stack frame as the RPointer. This could fail to be the case only if some procedure took an RPointer argument but no RHeap argument. But no such procedures exist because such procedures could not do any useful operation. Since a single frame can contain many RPointers and RHeaps, the problem for the garbage collector is to pair up the RPointers with the RHeaps.

With sufficient knowledge about the way the compiler lays out stack frames and by requiring every RPointer argument to be followed by an RHeap argument (or by adding some declarative syntax that achieves the same effect) it would be possible to write a garbage collector which could deduce the RPointer/RHeap pairings on the stack and hence be able to trace references to OM objects from the stack.

3.9 Heap structure in detail

Figure 3.6 shows the actual format of a heap. The part above the dashed line represents the transient heap. `H` is some variable whose value is (a reference to) an RHeap structure which describes some active heap. The last slot of the RHeap structure is an RHeapB which to T appears to be a pointer to an extend that is outside the transient heap. The section of the figure below the dashed line is a part of the same process's address space into which some heap is mapped.

Note that the RHeapB from the RHeap is actually a pointer to the fourth cell of the heap. This is because T's convention for extend references is that the reference points to the first data cell of the extend – i.e. the slot following the T template pointer. To T, heaps appears as *vector-type extends*. A vector-type extend is an extend that has a length cell before the template pointer. Vector-type extends are used to implement Lisp's traditional vector of references. Vector-type extends are also used to implement *byte vectors* and *bit vectors*. During the debugging of OM, we were able to set the template pointer slot of the heap to point to the byte vector template in the transient heap. This enabled us to use the T standard byte vector primitives for examining the heap.

The heap has two major sections: the header and the data sections. The header contains:

Heap pointer: The cell number (i.e. offset from the base of the heap) of the first free cell in the heap.

Max heap pointer: The maximum value the heap pointer should be allowed to reach. When `RHEAP-ALLOC` notices that the heap pointer has reached this value, the garbage collector is invoked.

Head of LPointer list: The head of the list of LPointers contained within this heap.

Size of index: Maximum number of elements in the heap index.

Index elements: Vector of RPointers and reference counts.

Data cells: Section in which OM objects are allocated.

The heap pointer is initialized to the cell number of the first data cell. `RHEAP-ALLOC` uses and increments the heap pointer. Note that the offset part of an RPointer is the offset from the base of the heap, ***not*** the offset from the beginning of the data cell section. If the offset were from the beginning of the data cell section then the RPointer dereference procedure would need to contain an additional addition operation to account for the size of the heap header. Since this size is a function of the heap index size, which is not constant for all heaps, the dereference procedure would have to get the heap index length from the heap, adding another memory reference to the procedure.

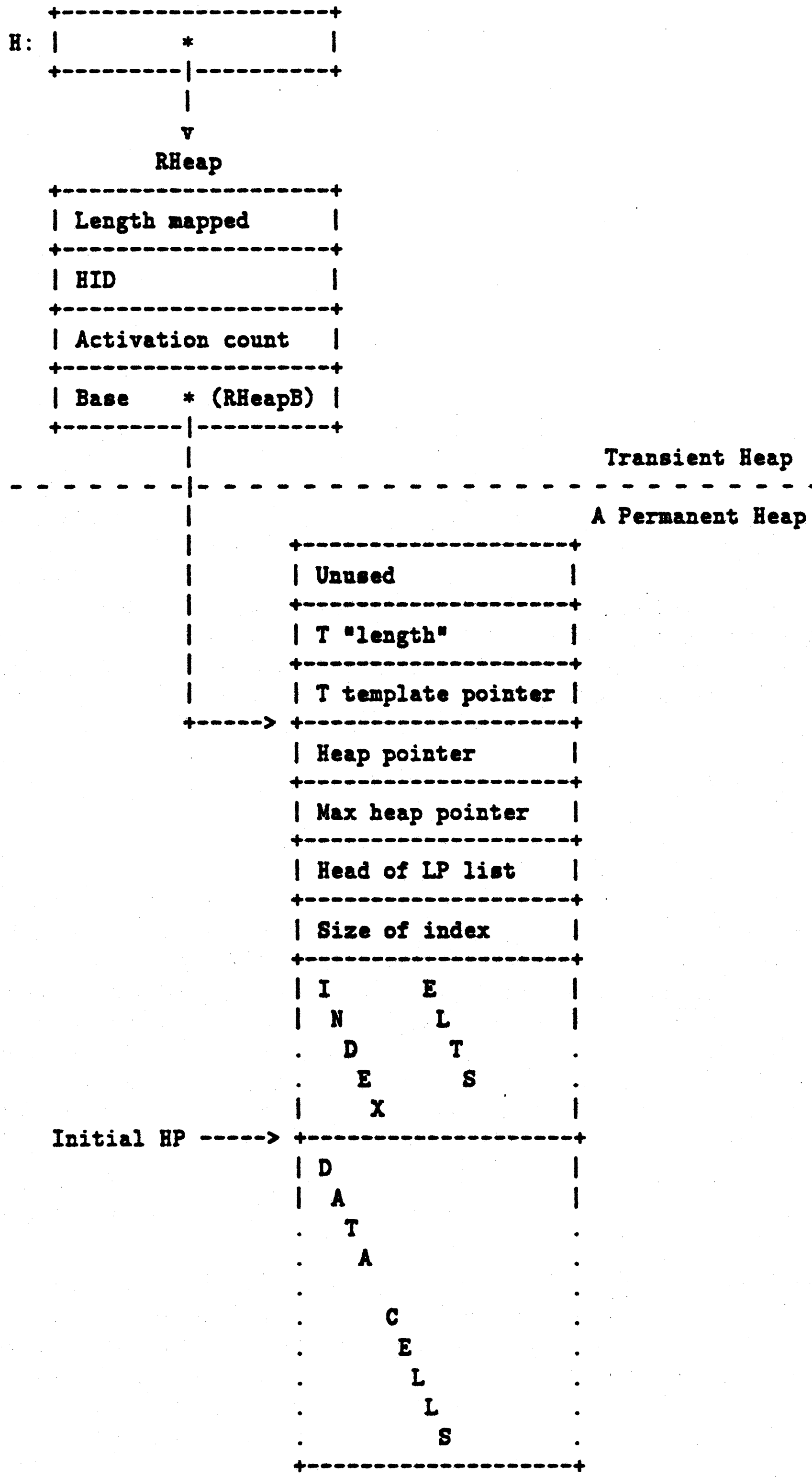


Figure 3.6: Heap format

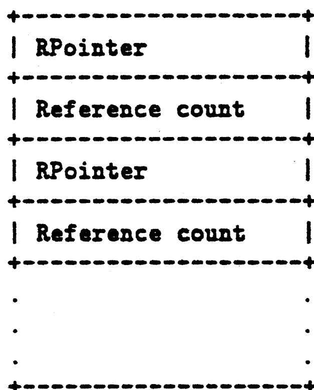


Figure 3.7: Index element format

The size of the heap index is fixed at the time a heap is created. An alternative approach would be to allocate the index within the data cell section and to maintain a pointer from the heap header to the current heap index. With this approach, when the index fills up, a new copy could be allocated and the pointer from the header could be adjusted. This approach is slightly more complicated and introduces yet another layer of indirection that must be followed at LPointer dereference time. For these reasons, the current OM implementation simply uses a fixed length vector in the heap header. The elements of the vector alternate between RPointers and reference counts as shown in figure 3.7.

3.10 OM Types: More details

3.10.1 Getting code into T

Before discussing the issue of user-defined types in OM, we must briefly examine the environment in which we expect programmers to work. We are not attempting to build a single-language, integrated program editing, debugging, and production-use environment like Smalltalk. (Such an environment would be nice to have, but is outside the scope of this work.) Programmers will write their programs using a conventional text editor and have another context consisting of a T interactive system augmented by OM. The text editor may be embedded within the same process as the T system or may be in a separate process but in either case, the maintenance of the programmer's code is outside the scope of T and OM.

T source code in text files must be compiled before it can be incorporated into a T environment. By "incorporation" we mean a process that makes user procedures and definitions available within a T environment.

T has two compilers: the *standard compiler*, which produces tree-oriented intermediate code that can be executed by an interpreter that is present in the T environment, and *TC*, which produces native machine instructions (that can be executed by the real processor). TC is much slower than the standard compiler. However, the compiled code produced by TC executes much more quickly than the compiled code produced by the standard compiler. TC produces its result into a file (called an *object file*) of machine instructions which can then be read into the T environment. The standard compiler dispenses with the object file and produces the intermediate code directly into the T environment. It is not possible to save the output of this compiler[8], but it runs so fast that it

[8] Note that this is a good example of a situation in which a permanent object system would be very useful – the compiled code could be saved as a permanent object

is acceptable to have the programmer's source code compiled each time it needs to be incorporated into a T environment.[9]

The T interactive environment communicates with the user via a "read, compile, interpret, print" loop that reads a T source string, compiles it into intermediate code, interpretively executes the intermediate code, and prints the results and then repeats the cycle. The "compile, interpret" step is sometimes called *evaluation* and the loop is called the "read, eval, print loop" (or *REPL* for short).

The `LOAD` procedure takes a file name argument and incorporates the contents of the file. If the file is an object file, the binary loader is invoked. Otherwise, the contents of the file is incorporated by applying the REPL to the file.

3.10.2 User-defined types

User-defined extend types are created using the `DEFINE-!OBJECT-TYPE` special form. This form defines a type and an associated set of methods for objects of that type. The syntax and behavior of `DEFINE-!OBJECT-TYPE` is related to T's `OBJECT` form, so we will examine the latter first.

The `OBJECT` form is both declarative and procedural. It declares a set of handled operations and associated methods, and allocates an object that responds to the declared set of operations in the specified way. The syntax of `OBJECT` is:

```
(OBJECT call-part method-part)
```

The `call-part` can be ignored for our purposes. The `method-part` is a list of method clause. The syntax of a method clause is:

```
(method-head method-body)
```

Where a `method-head` looks like:

```
(operation arg1 ... argn)
```

`operation` is an expression (typically just a variable) whose value is an operation. The `argi` are the arguments to the operation. Within the `method-body` – the code that implements the method – the `argi` are bound to the values in the operation invocation. The first argument is always the object to which the operation is being applied; this argument is called the *self argument*. If the method wants to apply another operation to the object, it applies the operation to the value of the self argument.

Execution of an `OBJECT` special form yields (a reference to) a new object. The new object is closed over the lexical environment in which the `OBJECT` form appears. Method bodies can contain references to variables that are lexically apparent from but defined outside the `OBJECT` form. When a handled operation is applied to the result of the `OBJECT` special form, the appropriate method is selected from the object's `method-part` and is executed; references to closed-over variables in the method yield the values those variables had at the time the object was created.

How does the behavior of `OBJECT` map onto our model of objects as a vector of slots containing references to other objects? The `OBJECT` special form does not say anything about slots. Note however, the implementation of the "closing over" procedure requires that space be allocated to hold the values of closed-over variables at the time the closure is created. This space, plus a reference to an object that contains the methods, *is* the object. Thus, the closed-over variables *are* the slots in the object.

Consider the following piece of code:

---

[9] Most Lisp systems call something like the standard compiler a *reader*, and something like TC a *compiler*. In fact, in T, most users are not aware that there is a standard compiler that is converting their source code into intermediate code; they think that T is simply interpreting their source code.

```
(DEFINE FOO
        (LAMBDA (X Y Z)
          (OBJECT NIL
                  ((ONE-OP SELF N)
                   (+ N X Y))
                  ((ANOTHER-OP SELF N)
                   (CAR Z)))))
```

This code assigns a procedure of three arguments to the variable FOO. The procedure returns an object that handles two operations called ONE-OP and ANOTHER-OP. The object is closed over the variables X, Y, and Z which are the arguments to the procedure.

Note that each execution of the OBJECT form yields a new, distinct object:

```
(SET A (FOO 1 2 '(THIS IS A LIST))
(SET B (FOO 10 20 '(ALPHA BETA GAMMA))

(ONE-OP A 5) => (+ 5 X Y) => (+ 5 1 2) => 8
(ONE-OP B 5) => (+ 5 X Y) => (+ 5 10 20) => 35
```

The representation of the object that is the value of A is something like:

```
A ---> +---------+
       |    *----|----> Object code for ONE-OP and ANOTHER-OP
       +---------+
     X:|    *----|----> 1
       +---------+
     Y:|    *----|----> 2
       +---------+
     Z:|    *----|----> (THIS IS A LIST)
       +---------+
```

The traditional term (from Smalltalk) for variables that are available to the method clauses is *instance variables*. Instance variables are the names of the slots of an object. The values of instance variables are what make one instance of an object created by the OBJECT special form different from another instance of an object created *by the same* OBJECT form.

Smalltalk and Lisp Machine Lisp [53] support object-oriented programming facilities similar to T's. One way in which their facilities differ from T's is that in Smalltalk and LM Lisp there are separate primitives for declaring types of object and creating an object of a particular type. Also, in the declarative form the instance variables are declared explicitly and are not determined by the context surrounding the declaration. The number of instance variables that are declared determines the size of objects.

Taking after Smalltalk and LM Lisp, OM has a declarative mechanism for introducing new object types. The reason we adopted this approach is that we feel that it is required in a permanent object system. The goal of T's object-oriented support is to allow object types to be unnamed and implicitly created; T object type definitions are dependent on context (i.e. the context surrounding the OBJECT form). Our goals are different.

As a programmer debugs procedures, he edits, compiles, and re-incorporates all or parts of files. He may destroy his T process and start a new one and incorporate his procedures into it. In T, the incorporation (not the execution) of a procedure that contains an OBJECT form constitutes the definition of a new type. We do not believe that this is the appropriate way to introduce new types into a permanent object system.

Creating a type in a permanent object system is a serious thing: the system is obliged to retain all the information related to the type for as long as objects of that type exist. In our system, since we

can not store object code in heaps, this retention means the writing of an external file that describes the type (more on this later). Thus, it seems undesirable that types are created essentially as a side-effect like in T.

It must be possible to modify the methods that make up a type. This means it must be possible to refer to a type – that the type have a name. In T, it is not possible to incorporate a revision of an existing OBJECT form. In fact, there is no way to refer to an existing OBJECT form: it is buried within an opaque compiled object. In OM, types have text names and a programmer can get all the information about type simply by knowing the type's name.

The essence of the problem of user-defined types in our system is that code that implements types must be treated differently from ordinary user code. OM need not and does not keep track of all user procedures that are incorporated into a running T/OM environment. But OM *must* keep track of code and other information that applies to type definitions, regardless of whether those definitions apply to types that are being used in any active T/OM environment.

DEFINE-!OBJECT-TYPE is the OM special form for introducing new types. The syntax of DEFINE-!OBJECT-TYPE is:

```
(DEFINE-!OBJECT-TYPE type-name
                     options
                     instance-variables
                     method-clauses)
```

type-name is the name of the new type. **options** is a list containing certain options about whether the instance variables are accessible outside the **method-clauses**. **instance-variables** are the names of the slots of the object. **method-clauses** is similar to the method clauses of OBJECT.

Operations applicable to OM objects are created using !DEFINE-OPERATION which is analogous to T's DEFINE-OPERATION.

All information about OM types is stored in a special heap called the *type heap*. OM has special knowledge about this heap in much the same way that it does about the HID heap discussed earlier. The type heap contains several things:

- The next type ID to assign.
- A table translating type names to type IDs.
- A table translating type IDs into type names.
- A table translating type IDs into type source file names.
- A table translating type IDs into lengths.

We will explain how this information is maintained by explaining the behavior of DEFINE-!OBJECT-TYPE. The execution of a DEFINE-!OBJECT-TYPE form causes a new OM object type to be created. A new type ID is generated by reference to the type heap. A slightly modified version of the DEFINE-!OBJECT-TYPE form is written to a new file (called a *type source file*) whose name is entered into the table translating type IDs into type source file names in the type heap. This file is owned by the OM system, not the user. The name and ID of the type is entered into the type-name-to-type ID translation table and the type-ID-to-type-name translation table. The type ID and type length (number of slots) is entered into the type-ID-to-length translation table.

When the DEFINE-!OBJECT-TYPE form is compiled, the method clauses are *not* compiled. When the result of compiling the DEFINE-!OBJECT-TYPE form is executed, it is manipulating method clause source code, *not* object code. Thus, incorporating a source file containing a DEFINE-!OBJECT-TYPE form does not result in the compilation of method clauses. This aspect of OM types will become clearer as we describe operation dispatch in OM.

3.10.3 OM operation dispatch

Operation dispatch is the process of invoking an object's method in response to an operation being applied to the object. In OM, operation dispatch happens when an OM operation is applied to an OM object. The operation invocation is syntactically identical to a procedure call, except that the head of the form must evaluate to an operation object instead of a procedure object. The first two arguments to the operation must specify the object to which the operation is to be applied. These two arguments must be an RPointer and an RHeap.

Operation dispatch begins by extracting the type ID from the first slot of the object to which the operation is being applied. This type ID is looked up in a per-process table (residing in the transient heap) that translates type IDs into *active types*. An active type is one whose handler has been incorporated into the transient heap. If the type ID is found in the table, the associated handler is invoked. The handler is simply a procedure that compares the operation object being invoked against all the operation objects listed in the the `DEFINE-!OBJECT-TYPE` for the type ID. If the operation is handled by the type, the associated method is invoked. Otherwise, if the operation has a default method, it is applied. Otherwise, an error is raised since the operation can not be handled.

If the type ID is not found in the per-process active type table, the operation dispatch mechanism translates the type ID into a type source file name by referring to the type heap. The type source file is then compiled by the standard compiler, incorporated into the T/OM environment, and a handler is constructed. If a version of the type source file that has been compiled by TC exists, that compiled version will be incorporated instead of invoking the standard compiler. The type ID and handler are entered into the active type table and operation dispatch proceeds as described above.

3.10.4 Type redefinition

In any permanent object system, suppose a programmer has defined a type and then creates some objects of that type. Now suppose that the programmer wants to modify the type. Does he want to modify the behavior of existing objects of that type or does he want only objects created after the change to have their behavior based on the modified type and to have old objects retain their old behavior? If the former, what sorts of changes to a type are compatible with existing objects? If the latter, in what sense, if any, are the unchanged and changed types the same type?

There are cases where type definitions need to be modified without creating a new type. Fixing bugs is one example: if a change to a type definition is the fixing of a bug in the definition, old objects will probably want their behavior modified to the new, less buggy behavior.

There are cases where the changing of a type definition must be treated carefully. For example, suppose the new definition specifies a larger number of instance variables. If the new definition is applied to old objects, an error will occur when the slot that doesn't exist in old objects is referenced. One might be tempted to say that the new definition with a larger number of instance variables is creating a new type. This attitude is not entirely adequate though. The old type and new type might have much in common. By forcing them to be different types, we are causing whatever similarity the two types have to be lost. For example, methods in the new type that don't refer to the new instance variables might be identical to methods in the old type. If a bug is found in such a method, the fix should be applied to both the old and new type.

Conventional databases have had to deal with problems similar to those described above. The traditional solution is to force the user to dump his data and then reload it using the new type (schema). This is essentially a result of the fact that database systems typically use highly compact and optimized data structures to represent data. Such representations are not easy to change dynamically.

We do not yet know how to solve the problem of type redefinition. The Smalltalk and LM Lisp object type systems essentially do not deal with the problem in full generality. The underlying structure

of our system allows both existing types to be modified and new types to be created. Presently `DEFINE-!OBJECT-TYPE` always creates a new type (i.e. type ID). Once created, an object's behavior is not changed by subsequent executions of `DEFINE-!OBJECT-TYPE`. However, since this form specifies the type name, it would be trivial to make it optionally modify the behavior of an existing type ID to which the type name translates. All that is required is that instead of adding an entry to the tables in the type heap that take a type ID as a key, that those tables be updated to reflect the new definition.

To deal with the case where a new type needs to be generated (e.g. when the number of instance variables has changed) we would like to consider the new type to be a new *generation* of an existing type. For some purposes different generations of the same type will be considered different types, but for other purposes they might be considered the same type. For example, the two types would be considered different by the operation dispatch mechanism. However, if an object type definition editor were to be included as part of T/OM, the two types might be considered to be same for the purpose of method modification.

Chapter 4

Programmer interface

The previous chapter dealt with the low-level implementation issues in OM. We now address the issues related to how programmers actually use OM. The major topics of this section are the syntactic tools the programmer uses and semantic issues the programmer must deal with. At the end of this chapter we describe two sample uses of OM.

4.1 Simple syntactic tools

T, like most Lisps, has a mechanism for modifying the syntax of the language. This mechanism is called a *macro*. OM defines some macros to make programming using OM more convenient and less prone to error.

WITH-ACTIVE-HEAP is a macro that controls heap activation. The underlying activation control primitives, ACTIVATE-HEAP and DEACTIVATE-HEAP are inconvenient and if not used correctly can lead to heaps not being properly deactivated. For example, in:

```
(DEFINE (FOO HID)
  (LET ((HEAP (ACTIVATE-HEAP HID)))
    ...
    (DEACTIVATE-HEAP HID)))
```

if an error occurs within the "...", and the stack is unwound to top-level, DEACTIVATE-HEAP will not be called, and the heap will be left active. To avoid this potential problem, the procedure should be written:

```
(DEFINE (FOO HID)
  (UNWIND-PROTECT
    (LET ((HEAP (ACTIVATE-HEAP HID)))
      ...
      )
    (DEACTIVATE-HEAP HID)))
```

UNWIND-PROTECT is a T special form that insures that its second form (the call to DEACTIVATE-HEAP in this case) will be executed.

By using WITH-ACTIVE-HEAP, the above can be simplified to:

```
(DEFINE (FOO HID)
  (WITH-ACTIVE-HEAP HEAP HID
      ...
      ))
```

Which expands into a definition which is identical to the UNWIND-PROTECT version above.

!WITH-LPOINTER is a more sophisticated macro that controls the activation of heaps based on LPointers. Recall that objects referred to by LPointers can not be examined until the LPointer is converted to an RPointer/RHeap that refers to an object in an active heap. !WITH-LPOINTER simplifies the writing of code that does the conversion. For example, consider a procedure that takes an RPointer/RHeap to an LPointer:

```
(DEFINE (FOO LP LP-HEAP)
  (!WITH-LPOINTER ((V LP LP-HEAP))
    (!PAIR-CDR V!R V!H)))
```

The first part (called the *specification*) of the !WITH-LPOINTER form specifies the LPointers that will be used within the second part (called the *body*) of the !WITH-LPOINTER form. The LPointer specification is a list of triples (the example above has only one triple). The first element of the triple is a pseudo-variable that will be described shortly. The second and third elements of the triple are a reference (RPointer/RHeap) to the LPointer being used.

Just before the body of the !WITH-LPOINTER is executed, all the heaps named by the LPointers in the specification are activated. After the body is executed, all these heaps are deactivated. (The macro uses ACTIVATE-HEAP and DEACTIVATE-HEAP so dynamically nested !WITH-LPOINTERs actually simply manipulate the heap activation count.)

The pseudo-variables are used to refer to the RPointer/RHeap pairs that result from converting the LPointer reference into a reference to an active object. Within the body of the !WITH-LPOINTER, two variables are introduced; one is bound to an RPointer that refers to the object referred to by the LPointer and the other is bound to the RHeap that results from activating the heap referred to by the LPointer. The names of these variable are constructed from the name of the pseudo-variable. For pseudo-variable *var*, the variable *var*!R can be used to refer to the RPointer, and the variable *var*!H can be used to refer to the RHeap.

Also, every occurence of the pseudo-variable itself is replaced by *two* variables that are bound to the RPointer/RHeap that refers to the object referred to by the original LPointer. Thus, the example above could be rewritten:

```
(DEFINE (FOO LP LP-HEAP)
  (!WITH-LPOINTER ((V LP LP-HEAP))
    (!PAIR-CDR V)))
```

4.2 Programming with two kinds of references

The previous chapter described the primitives for derefencing RPointer and LPointers. However, it did not address the question of how a program is to know which dereference mechanism should be applied to a particular reference. Should the decision about how the reference should be derefenced be made dynamically or statically? For example, given the expression:

```
(!PAIR-CAR R H)
```

should !PAIR-CAR (statically) assume that R/H refers to an OM pair, or should it (dynamically) see if R/H refers to an LPointer that needs to be dereferenced to reach the pair?

Another issue related to having two kinds of references is the kind of reference returned as the value of a procedure. Given the nature of our implementation environment, a procedure always actually returns an RPointer. But is it an RPointer to the object being returned, or is it an RPointer to an LPointer to the object being returned?

4.2.1 The dynamic approach

The dynamic approach requires that the primitive that extracts an RPointer from an object look at the type of the RPointer. If the type tag indicates that the type is LPointer, the primitive could then invoke the dereference mechanism on the LPointer. Accessors that retrieve a slot in an object have to check to see if the type of their argument is LPointer. Recall that all such accessors call RPOINTER-EXAMINE to get the contents of a slot. We could rewrite RPOINTER-EXAMINE to be:

```
(DEFINE (RPOINTER-EXAMINE RP HEAP I)
  (COND ((LPOINTER? RP)
         (!WITH-LPOINTER ((P RP HEAP))  ;*** Deref. LPointer
           (RPOINTER-EXAMINE P!R P!H I)))
        (T
         (RHEAP-EXAMINE HEAP (+ I (RPOINTER-CADDRESS RP))))))
```

This generality comes only at the price of increasing the cost of the dereference mechanism: every time an RPointer is extracted from an object, the RPointer must be examined to see if it refers to an LPointer.[1]

The dynamic approach also requires that accessors that modify a slot in an object have to check to see if the reference being stored is to an object in another heap. We could rewrite RPOINTER-DEPOSIT (the procedure used by all accessors that modify slots in objects) to be:

```
(DEFINE (RPOINTER-DEPOSIT RP1 HEAP1 I RP2 HEAP2)
    (COND ((NOT (= HEAP1 HEAP2))
           (RPOINTER-DEPOSIT
                RP1 HEAP1
                I
                (!EXPORT-RPOINTER RP2 HEAP2 HEAP1) HEAP1))
          (T
           (RHEAP-DEPOSIT HEAP1 (+ I (RPOINTER-CADDRESS RP)) OBJ)))))
```

In addition to the cost in time, there is a cost due to increased code size. RPOINTER-EXAMINE is expanded in line. The addition of the LPOINTER? test will increase the size of the expansion. To save space, the code to dereference the LPointer can be left out of the in line expansion; only the test and a call to a procedure to do the LPointer dereference will be included. (In the case where the RPointer points to an LPointer, the cost of an extra procedure call is not significant since the LPointer dereference is expensive anyway.) However even with the LPointer dereference moved to a subroutine, the size of the compiled RPOINTER-EXAMINE will increase by about 1/3 (recall from section 3.5.2 that the original sequence is about 8 instructions; the LPointer test and subroutine call will be at least 3 instructions). The size of the expanded RPOINTER-DEPOSIT will increase also.

Besides the time and space efficiency problems with the dynamic approach, there is a logical problem: the RPointer returned after automatically dereferencing an LPointer (in RPOINTER-EXAMINE) will be to an object in heap different from the object from which contained the RPointer to the LPointer. The returned RPointer is useless to the procedure that called the accessor since the procedure does not have a handle on the heap that contains the object the returned RPointer refers to. One obvious way to get around this problem is to make RPOINTER-EXAMINE return an LPointer in case it has dynamically dereferenced an LPointer:

```
(DEFINE (RPOINTER-EXAMINE RP HEAP I)
    (COND ((LPOINTER? RP)
           (!WITH-LPOINTER ((P RP HEAP2))
```

[1] If we had the option of building hardware we would argue that this test could be performed in parallel with the RHEAP-EXAMINE; but we're not so we won't.

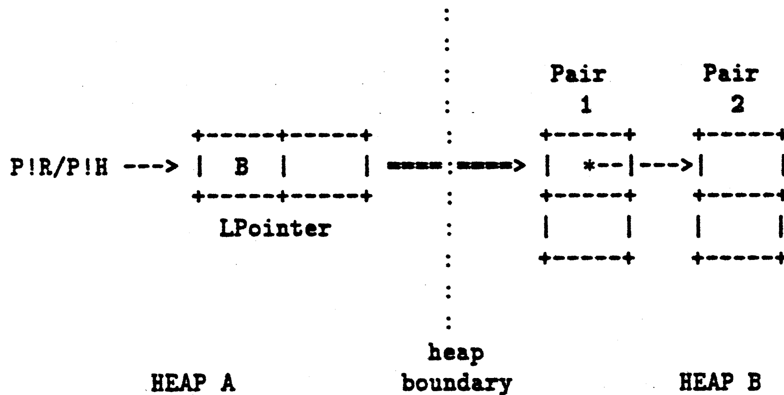


Figure 4.1: An LPointer to two pairs

```
      (!EXPORT-RPOINTER (RPOINTER-EXAMINE P! I) HEAP)))
    (T
     (RHEAP-EXAMINE HEAP (+ I (RPOINTER-CADDRESS RP))))))
```

However, this solution is unsatisfactory. Assume `P!R/P!H` refers to an LPointer in heap A that refers to a pair in heap B and assume that the *car* of that pair is also a pair. Figure 4.1 shows how the pairs are arranged. To retrieve the *cdr* of the second pair, using the dynamic approach, we could write:

```
(LET ((X (!PAIR-CDR (!PAIR-CAR P!R P!H) P!H)))
  ...)
```

Since `P!R/P!H` refers to an LPointer, the LPointer will be dynamically dereferenced by `!PAIR-CAR`. The value returned by `!PAIR-CAR` will be a newly allocated LPointer (in heap A) to the object referred to from the *car* of the first pair. When `!PAIR-CDR` is applied to the LPointer returned by `!PAIR-CAR`, the LPointer will be dynamically dereferenced.

Simply to follow this *car-cdr* chain, we allocated a LPointer and did an LPointer dereference. The LPointer becomes garbage as soon as the `!PAIR-CDR` is executed.

4.2.2 The static approach

Instead of automatically dereferencing and creating LPointers, we can leave it up to the programmer to specify where LPointers are and where LPointers need to be created as part of the programming process (i.e. statically). The static approach is predicated on the fact that the structure of an application's objects – i.e. which objects are in which heaps and where the inter-heap references are – is fixed. OM is a system designed to deal with applications whose data structures are fixed in this way.

To use the static approach, the programmer must adopt a certain style of programming. The goal of the style is to minimize (and hopefully reduce to zero) the amount of storage (especially garbage) that is allocated by procedures that do not create logically new objects. That is, we don't want procedures to allocate storage simply to return results that in principle do not require storage to be allocated. In particular, we want to avoid allocating LPointers when it is not necessary to do so.

If the structure of an application's data is fixed, the need for the generality of the dynamic approach is reduced. For example, it is not necessary for accessors to dynamically check to see if a reference is through an LPointer if it is possible to statically assert that the reference is never through an

LPointer. In case the programmer can't assume where the LPointers are, he can insert the check for LPointers himself (or simply introduce a layer of procedures that do the check and dispatch accordingly). In this case, the system is no more or less efficient than the dynamic approach. In all other cases however, the dereference mechanism is cheaper.

Another aspect of the static approach is that LPointers are explicitly created. Note however that LPointers will not need to be created in all the cases in which the dynamic approach would have created them. For example, using the static approach, following the *car-cdr* chain described above would be written as:

```
(!WITH-LPOINTER ((Q P!R P!H))
  (LET ((X (!PAIR-CDR (!PAIR-CAR Q!R Q!H) Q!H)))
    ...))
```

Note that we are assuming the original RPOINTER-EXAMINE – the one that does *not* automatically dereference and create LPointers.

Within the body of the !WITH-LPOINTER, Q!R/Q!H refers to the first pair in heap B. The !PAIR-CAR returns an RPointer to the object referenced by the first pair's *car* – the second pair in heap B. The !PAIR-CDR returns an RPointer to the object referenced by the second pair's *cdr*. Note that we can use Q!H as the second argument to !PAIR-CDR because we know that the second pair is in the same heap as the first pair (which is identified by Q!R/Q!H).

Unlike in the dynamic approach, the above expression does not cause a gratuitous LPointer to be created and then dereferenced. The general case of which the expression is an example is the successive application of procedures to an object:

```
(F1 (F2 ... (Fn P!R P!H) ... P!H) P!H)
```

where P!R/P!H is a reference to an LPointer and the return values of the Fi are objects in the same heap as the object referred to by that LPointer. In the dynamic approach, since P!R/P!H refers to an LPointer, an LPointer will be allocated for each intermediate object, and this LPointer will be dereferenced right away by the next procedure application. The static approach avoids this cost by making the programmer explicitly specify (via !WITH-LPOINTER) that a piece of code should run "within a particular heap" and that intermediate results should not have an LPointer allocated to refer to them.

In the static approach, since LPointers are never automatically allocated, it is also up to the programmer to explicitly specify calls to !EXPORT-RPOINTER. Which procedures allocate and return LPointers is a convention determined and followed by the programmer. His procedures fall into one of two classes: those that work within a single heap and return RPointers to their results, and those that span heaps (by dereferencing LPointers) and return LPointers to their results. Procedures in the latter class will have the form:

```
(DEFINE (G Q!R Q!H)
  (!WITH-LPOINTER ((P Q!R Q!H))
    (!EXPORT-RPOINTER (F1 (F2 ... (Fn P!R P!H) ... P!H) P!H)
                      P!H
                      Q!H)))
```

Procedures like G take an LPointer to some object, dereference the LPointer, apply a set of procedures to objects within the same heap as the object referred to by the LPointer, and then return an LPointer (in the same heap as the original LPointer) to the return value of G.

4.3 A pre-processor

The fact that an OM procedure that takes a reference to an OM object takes two arguments to pass the reference is a nuisance to the programmer. The two arguments logically identify a single object. Normally a programmer uses one argument to identify a single object.

The pre-processing approach takes advantage of the fact that there is a great degree of regularity in the way RPointers and RHeaps are passed among procedures. Note that all OM procedures that take an RPointer and RHeap return an RPointer that refers to an object that is the same heap as the RPointer argument. With some small syntactic modifications to T, the programmer can be relieved of the chore of specifying both the RPointer and RHeap argument. A pre-processor can automatically turn the programmer's one argument version of the code into the two argument version that the OM primitives expect.

The basic idea of the syntactic modification is that the programmer will declare all variables that hold a reference to an OM object. For example:

```
(DEFINE (P (OMVAR A) B (OMVAR C))
  (IF (Z?)
      (+ (Q A) B)
      C))
```

This defines a procedure P that takes three arguments, the first and last of which are references to OM objects. A pre-processor takes the definition and transforms it into the two-argument style:

```
(DEFINE (P A!R A!H B C!R C!H)
  (IF (Z?)
      (+ (Q A!R A!) B)
      C!R))
```

var!R and *var*!H are substituted for all occurences of *var*. However, if *var* appears in return position, just *var*!R is substituted for *var*.

The pre-processor is not general yet. The transformation above relies on the fact that Q returns an integer, not an OM object, and that the result of Q is being passed to a procedure that takes integers, not OM objects. What if Q returned an OM object (i.e. an RPointer) and instead of + receiving the result, the procedure being called expects an OM object as its first argument? That is:

```
(DEFINE (P (OMVAR A) B (OMVAR C))
  (IF (Z?)
      (R (Q A) B)
      C))

(DEFINE (R (OMVAR X) Y)
  (IF Y
      X
      Y))
```

Note that R is really a procedure of three arguments: the X argument gets expanded into two arguments by the pre-processor. Thus, when P calls R it needs to supply the RHeap argument that goes with the RPointer returned by Q.

In general, a call form A that:

1. Invokes a procedure that returns an RPointer, and
2. Appears in the argument position of some other call form B that takes an OM object in that position,

must have an RHeap inserted after call form A:

```
(R ... (Q ...) * ...)
-       -     -      -
|       |     |      |
|       +--A--+      |
|                    |
+---------B----------+
```

The * marks the point of insertion.

For the pre-processor to do this insertion, it must know something about the procedure being invoked. In particular, for a procedure Q, it must know the RHeap that is to be associated with the RPointer that Q returns. Fortunately, this is generally a static property of the procedure. The RHeap of the returned RPointer is the same as the RHeap of one of the objects passed to the procedure. Thus, we can augment the definition of Q with a declaration of what argument's RHeap is the RHeap of the returned RPointer:

```
(DEFINE (Q (OMVAR M)) (RETURN-RHEAP M)
  ...
  )
```

This says that Q returns an OM object identified by the RPointer returned by Q and the RHeap associated with Q's first argument, M. This is enough information so that the pre-processor can transform the definition of P into:

```
(DEFINE (P A!R A!H B C!R C!H)
  (IF (Z?)
      (R (Q A!R A!H) A!H B)
      C!R))
```

The A!RHEAP in the call to R is inserted based on the fact that the definition of Q says that the RHeap of the result of Q is the same as the RHeap of Q's first argument.

If the RETURN-RHEAP clause is omitted, the pre-processor assumes that the procedure returns a non-OM object (e.g. an integer).

While it appears that the pre-processor can automatically generate RHeap arguments for many cases, the programmer is still responsible for knowing when a data structure crosses a heap boundary. Doesn't the programmer have to mention an RHeap explicitly at this point? The answer is "no" because of the pre-processor in combination with the !WITH-LPOINTER macro enables the programmer to forget about the RHeap argument even in this case.

Consider the following simple example of a procedure that deals with data in multiple heaps. Suppose a procedure P is passed a list of LPointers. Each LPointer is a reference to a vector of integers in another heap. Suppose we want P to sum up all the integers in all the heap. We could write P as follows:

```
(DEFINE (P (OMVAR L))
  (COND ((!NULL? L)
         0)
        (ELSE
         (+ (!WITH-LPOINTER ((VEC (!PAIR-CAR L)))
              (LOOP (INITIAL (SUM 0))
                    (INCR I FROM 0 TO (- (!VECTOR-LENGTH VEC) 1))
                    (DO (SET SUM (+ SUM (!VECTOR-ELT VEC I))))
                    (RESULT SUM)))
            (P (!PAIR-CDR L))))))
```

Note:

- `!NULL?` is a primitive procedure that takes an RPointer/RHeap and returns *true* if the RPointer is to the null object.
- `!PAIR-CDR` is a primitive procedure that takes an RPointer/RHeap to an OM pair and returns the *cdr* of the pair. `!PAIR-CDR` is declared to the pre-processor to return an RPointer that is in the same heap as the argument to `!PAIR-CDR`. Thus, in the recursive call to `P` inside the definition of `P`, the RHeap associated with `L` (i.e. the second real argument to `P`) will be inserted after the call to `!PAIR-CDR`.
- `!VECTOR-LENGTH` is a primitive procedure that takes an RPointer/RHeap to an OM vector and returns an integer.
- `!VECTOR-ELT` is a primitive procedure that takes an RPointer/RHeap to an OM vector and an integer offset into the vector, and returns the RPointer at the specified offset. `!VECTOR-ELT` is declared to the pre-processor to return an RPointer that is in the same heap as the first argument to `!VECTOR-ELT`. However, in this example since the call to `!VECTOR-ELT` appears inside a call to a non-OM procedure (i.e. `+`), the RHeap is not inserted.

Note that the one clause in the specification part of the `!WITH-LPOINTER` has just two elements: the pseudo-variable `VEC` and the expression `(!PAIR-CAR L)`. Since like `!PAIR-CDR`, `!PAIR-CAR` is declared to the pre-processor to return an object in the same heap as its argument, the specification clause will be filled out to be the full triple, the last element being the RHeap that was passed to `P`.

While we have not actually implemented the pre-processor described above, we do not believe that the implementation would be all that difficult. The main inconvenience to the progammer introduced by the pre-processor is one that is found in any system of declarations: declaration must precede reference. Lisp systems are typically more flexible, allowing references to procedure that have not yet been defined. However, this flexibility is possible only when "compiling" the reference does not require any information that appears in the definition. The pre-processor *does* require such information, and hence the definition must precede the reference. We believe that this is not too onerous a task for the programmer.

4.4 The mixed object environment

OM runs within a T environment. Programs that use OM can create normal T objects (in the transient heap) and OM objects (in a permanent heap).

OM provides primitives for copying objects between the transient and a permanent heap, and between permanent heaps. These primitives are not general structure traversers. That is, they do not take a reference to an object of arbitrary type and copy that object and all objects reachable from that object into another heap. In general, with a large graph of objects (data structures), finer control is required; when copying a data structure, objects will need to be allocated in different heaps. No simple, single copying primitive could handle all possibilities of where objects are to be allocated. Thus, OM provides primitives that copy atoms (including LPointers) between heaps. More sophisticated copying procedures can be built out of the primitives.

Being able to allocate objects in the transient heap and then later copy them into a permanent heap can be useful. This is because it allows one to write procedures that allocate new objects without regard to what heap the objects should be allocated in. This may be a convenient programming style for certain applications. In such applications, at a certain level of abstraction all procedures that allocate new objects always do so in the transient heap; at the next higher level of abstraction, the objects are copied into the appropriate permanent heap.

Another advantage of being able to copy transient objects into a permanent heap is that it allows allocation to be a bit more reckless. In programs that allocate objects but in which it is not statically

possible to know which of the allocated objects will be permanent, if all the object are allocated in a permanent heap, some would be garbage. These garbage objects are costly in terms of garbage collection time (a program that generates a lot of garbage causes the garbage collector to be invoked more frequently). If, however, the objects are always allocated in the transient heap and then the ones that are to be permanent are copied into a permanent heap *and* the amount of garbage is not too great, the garbage is "free". A program that creates a certain amount of garbage in the transient heap can do so with no time penalty if it doesn't allocate so much that the garbage collector is invoked on the transient heap before the process exits. Since the transient heap *is* transient, all its contents are by definition garbage when the process exits; garbage collection on that heap is implemented simply by deleting the entire heap. Thus, no garbage collection time penalty (other than the time required to delete the heap file) is incurred.

There is some clumsiness that results from writing program that deal with both OM objects and T objects. In the current implementation of OM, there is no easy way to avoid this. In another implementation of OM we expect we would simply dispense with T objects altogether and have a unique OM transient heap associated with each process. This heap would be like any other OM heap except that the maximum size of its heap index would be zero – i.e. there could be no references from other OM heaps into this heap. Thus, when the process exits, the heap can be deleted. This strategy would eliminate the clumsiness of dealing with the T transient heap without sacrificing the advantages associated with that heap as described above.

4.5 Finding the first reference

In order to manipulate an object, a program must have a variable whose value is a reference to the object. But when a program starts, the values of all its variables are undefined. How does a program go from having no references to having some references?

Programs do not operate in a vacuum. Programs are started because people want them started. People give arguments to programs. If the programs are to manipulate permanent state, the arguments must indirectly identify objects (if they did not name objects, the program could not conceivably manipulate state). However, these identifications are not OM references, but something more high level – something that is meaningful to a person, e.g. the string name of a "mailbox" or a number. The problem is to transform the kinds of arguments people give into references to objects.

There are two general questions involved here. First, what are the set of objects that are known a priori by the system? Second, what are the mechanisms for finding other objects from the known objects?

4.5.1 File systems

Traditional computer file systems provide a model for dealing with problem of finding objects given only a small set of known objects and some logical identification of the desired object. The objects in a traditional file system are directories and files. The known object is typically a "root directory" that is in some known place on the disk. The file system has a mechanism for finding a file given the string name of a file and the root directory.

Filesystem directories are a simple mechanism for converting high level references into lower level references. However, they have the two main properties in which we are interested. First, they have a piece of information that is known a priori. Second, they contain system maintained functions and data structures that convert high level references into lower level references.

4.5.2 File systems as a model for OM naming

One strategy for giving high level names to OM objects is to implement our own hierarchical naming

system. We define an object that is known a priori by OM – a *directory object* – that maps string names onto LPointers. A directory object is any object that responds to the *DirectoryLookup* operation (that takes a string) by returning an LPointer. The result of the lookup could be a reference to yet another directory object, or to a leaf in the directory tree. In this way, an object can be completely named with a list of strings. Looking up an object given a list of strings simply requires traversing the tree of directory objects starting at the root directory object and returning the LPointer that was the result of the last lookup.

Note that the above system is more flexible than a tradition filesystem naming system. A directory object is free to implement *DirectoryLookup* in any way it chooses. The obvious approach would be for the object to simply maintain a hash table mapping strings onto LPointers. However, it could do more sophisticated things. The object might treat certain strings in a special way. For example, we could make a directory object that when presented with a person's name yielded a person's mailbox object. However, this same object when presented with the string "MyMailBox" would yield the mailbox object associated with the person that owns the process executing the operation.

4.5.3 A general naming strategy

Note that this hierarchical naming system need not be the only way to support high level names. Different applications can implement different systems. OM does not commit applications to a particular naming system. All that OM itself must supply is a top level to all the naming systems – i.e. a single directory that maps naming system names onto *naming system objects*: an entry point into a data structure that can be used by procedures that want to translate high level names to object references. The hierarchical naming system that takes a list of strings and produces an LPointer is simply one naming system in the top level. This naming system can be entered in the top level under some well known name (e.g. "TreeNames"); the value of this entry is the root directory object for the naming tree.

4.5.4 Naming in the current implementation

The current implementation of OM does not include the general top level naming system name table described above. Since OM is running on top a conventional file system that has a hierarchical naming system, we took advantage of that naming system. The DOMAIN naming system lets us name heaps. However, we still need to identify a particular object in the heap.

In early versions of OM we allowed procedures to treat the heap index as a record with named fields. The names were artifacts of the source code and were *not* stored in the heap itself. This system is analogous to records in Algol-like languages: a program refers to a field of record by name, but when the program is compiled, the names disappear and the field is identified simply by its offset from the beginning of the record. In OM, elements of the index could be given symbolic names; these names could be used in conjunction with an RHeap to obtain an element of the index. The symbolically named elements of the index were excluded from the pool of index elements that are assigned as a result of `EXPORT-RPOINTER`. Using this record-like scheme a first reference could be obtained simply by activating a heap (using its path name) and referring to one of the symbolically declared heap index elements.

The problem with the scheme as we implemented it was that there was no way to be sure that a symbolic name was not being used to retrieve and element from the index of a heap that was not of the right "type" (i.e. that the particular elements of the heap index were not reserved for references by the particular set of symbolic names). The problem is analogous to one that would arise if in Pascal a field name from any record type could be used after a name of a variable whose type was any record type. E.g. in:

```
type r1 = record
        a:  integer;
```

```
            b:  integer;
            end;

type r2 = record
            x:  char;
            y:  integer;
            end;

var
    v1: r1;
    v2: r2;

begin
    v1.y := 0;
    end
```

the reference to `v1.y` is invalid. But in the system we implemented in OM, this sort of illegal reference would go undetected. The cause of the problem is that heaps are not typed. However, if we associated a type code with each set of symbolic index element names and we stored this type code in all heaps for which we wanted to allow index elements to be referred to symbolically, then references through symbolic element names could be dynamically checked to see if they were being applied to the right type of heap.

Instead of implementing this typing system, we abandoned the record-like approach to the heap index. OM already has a type system and there is no point in introducing another one.

The current OM naming systems consists of a facility that allows the programmer to identify one *distinguished object* per heap. The distinguished object mechanism is a way of specifying and obtaining a known object within a heap. A heap's distinguished object can be obtained simply by having the heap's HID. The OM primitive `DISTINGUISHED-REFERENCE` takes an RHeap (gotten by activating a heap) and returns an RPointer to the heap's distinguished object. When used in the context of the `SET` special form, `DISTINGUISHED-REFERENCE` can be used to set a heap's distinguished object. In this context, the program must supply an RPointer to the primitive.

Before the distinguished object mechanism can be used it is necessary to get the HID of some heap. In the current naming system implementation, HIDs can be obtained using the OM primitive `FILE-NAME-HID`. This primitive takes a DOMAIN path name and produces the HID of the heap that has that path name. Using `DISTINGUISHED-REFERENCE` and `FILE-NAME-HID` it is possible to get a reference to a known object. Thus, DOMAIN path names are the logical names of the known OM object.

4.6 Sample applications

To see how usable the design and implementation of OM is, we built two sample applications that use OM. These applications are representative of the kinds of applications OM is designed to handle.

4.6.1 OM/UMail

UMail is a display-oriented electronic mail user interface program that runs on the DOMAIN system. UMail lets users send messages and receive and store messages in mail boxes. UMail does not use OM; OM/UMail does. In UMail mail boxes are stored in simple text files. When UMail starts, it reads and parses the text file into an internal data structure. When UMail exits, it rewrites the text file if the contents of the internal representation of the mail box changed. The cost of the parse and rewrite steps is barely tolerable for moderately large (50-100 message) mail boxes. Electronic

bulletin boards, a sub-class of mail boxes, are general larger and using UMail to examine them is virtually impossible. This was one of the reasons that made us to want to make an OM version of UMail.

In UMail, the internal representation of a mail box is a "mail box object". This object handles certain operations; e.g *SelectMsg*, *AddMsg*, *DeleteMsg*, *ExpungeDeletedMsgs*. The local state of a mail box consists of a list of "message objects". A message object handles the operations: *InitMsg* and *Print*. The local state of a message object includes: the text of the message, pointers to various interesting headers in the text; internal times representing the date the message was sent and delivered; and flags (e.g. "message to be deleted").

The changes necessary to turn UMail into OM/UMail were relatively straightforward, if tedious (lacking the pre-processor). The conversion went far enough to demonstrate that we could maintain the OM versions of the mail box and message objects. An entire mail box object, with all the messages it references, is kept in a single heap. One heap contains exactly one mail box. The first reference is obtained by constructing the DOMAIN path name of the heap file from the logical (abstract) name of the mail box (e.g. a bulletin board or user name), activating the heap, and following the heap's distinguished reference, which refers to the mail box object.

OM/UMail did not replace UMail as the production mail user interface. We stopped working on OM/UMail as it became apparent that we could learn more about how well OM works from designing and implementing an application from scratch, rather than converting an existing application.

4.6.2 Naming server

The second sample application to use OM is a naming database manager (NDBM). We use the terms "database" and "database manager" in a very general sense – as terms that mean "a collection of permanent, structured data" and "a set of programs that manipulate that data".

The motivation for the NDBM project was to replace the DBM available on a DECSYSTEM-20 in the Yale Computer Science Department. The data held in the DEC-20 database includes:

- Personal information. E.g. people's home address and phone number, user IDs, electronic mailing addresses.
- Host (computer) information. E.g. host nicknames, network addresses.
- Mailing list information. Members, maintainers and descriptive information about electronic mailing lists.

The DEC-20 DBM is written in Lisp. The permanent, external representation of the database is a single, large text file containing the printed representation of a single, large Lisp list. When the DBM starts, it reads and parses the file into a Lisp list, the internal representation of the database. The time to read and write the database is very long. The data is not simultaneously sharable among several processes. Access to the data is by a network server process that handles one transaction at at time from other processes. The user interface to the database manager is one of these other processes.

The DEC-20 DBM uses the relational model. However, the generality of the relational approach was never exploited. One reason for this is that the generality was not needed. Another reason is because the DBM implementation is not very sophisticated, and the time to execute the relational operations is quite high.

The implementation of NDBM is in no way based on the DEC-20 DBM. However, the NDBM is designed to hold the same data as the DEC-20 DBM.

In designing the NDBM we viewed the task as a permanent data structure problem, rather than as a traditional database design problem. The database is relatively small (several hundred people, several hundred hosts, a hundred mailing lists) and we were not interested in applying sophisticated database technology.

In thinking about the problem of storing the kinds of data we needed to store, we developed a way of thinking about stucturing data in general, rather than structuring the particular data at hand. As a result, the NDBM is a more a DBM framework than an actual DBM. It is a framework in the sense that it defines a set of operations and their semantics, but does not supply the implementation of the operations. It does not specify any properties of the data to be stored in the database. An *instance of a framework* is a set of objects that behave in the way specified by the framework.

The framework defines two sets of types of objects: *item* and *descriptor*. A type is in the set of item types if it responds to the operations defined on item types. Item types are analogous to record types in a conventional database. Item objects – i.e. objects whose type is an item type – are like records in a traditional database. The local state of an item object contains information about the entity being described by the object.

For example, an instance of the framework might have a type called *Person* which is in the set of item types. Each person in the instance is represented by a single object whose type is *Person*. A *Person* object presumably contains strings containing a person's home address, phone number, etc.

Item objects can also contain references to other item objects. E.g. a *MailingList* object can have a list of *Person* and *MailingList* objects.

A type is in the set of descriptor types if it responds to the operations defined on descriptor types. Descriptor types are used to create and organize item objects. Every descriptor type has exactly one associated item type. Descriptor objects – i.e. objects whose type is a descriptor type – are like database schemas in a traditional database. The local state of a descriptor object presumably contains data structures that allow individual items to be stored and retrieved. We say that a descriptor object *covers* a set of item objects. A descriptor object *covers* an item object if it is possible to obtain a reference to the item object by applying the lookup operation to the descriptor object.

Descriptor types must handle operations like:

ItemType: Return the item type associated with the descriptor type.

NewItem: Create and return a new item; add the item to the descriptor index (lookup table).

LookupItem: Given a key (e.g. a string), return the item object associated with that key.

WalkItems: Apply a procedure (passed as an argument) to all the items that the descriptor object covers.

Show: Produces a printed representation of all the items the descriptor object covers.

Item types must handle operations like:

DescriptorType: Return the descriptor type associated with the item type.

Show: Produce a printed representation of the item object's contents.

In any instance of a framework, both item and descriptor types are free to handle additional operations. For item types, it is expected that they will handle all sorts of operations peculiar to the instance. E.g. an instance containing the *Person* item type described above would presumably handle an operation to retrieve a *Person* object's home address string.

The framework imposes a convention on how objects in an instance should be spread out across heaps. The convention is that there is exactly one heap per item type in the instance. All the objects of the same item type reside in a single heap. The descriptor object that covers the item object resides in the same heap. There is one additional heap, called the *master heap*, that contains only one object: a vector of all the descriptor objects in the instance. The master heap's distinguished reference points to this vector.

We implemented an instance of the framework that is designed to hold the kinds of data in the DEC-20 database. We then moved virtually all the contents of the DEC-20 database into the instance of the framework. The instance has eight type: four descriptor types and four item types.

The four item types are: *Person*, *Host*, *MailingList*, and *UserID*; the four descriptor types are: *PersonDesc*, *HostDesc*, *MailingListDesc*, and *UserIDDesc* The descriptor types support translation between string keys (e.g. a person's name) and references to item objects by using hash tables that are part of the local state of the descriptor objects. Descriptor types also support operations that allow modifications to be made to item objects interactively.

When a user invokes a procedure to view or modify some piece of the database, one or more heaps may be activated. The heaps are activated for exclusive use – for the duration of the activation, no other process can access the same part of the database. This may seem like a serious restriction, but considering that users were quite able to live with the strictly one-at-a-time access offered by the DEC-20 DBMS, the restriction is actually not too serious. In NDBM, multiple processes *can* simultaneously access parts of the database as long as the parts are in different heaps. More concurrency could be accomodated by using one of the techniques discussed earlier. However, given the nature of the access patterns (infrequent and short), the current scheme seems satisfactory.

4.7 A more ambitious scheme

In this section we describe a scheme for making the application programmer's task considerably easier than it is in the current OM implementation. This scheme involves using special compiler optimization techniques to make certain apparently expensive operations free.

4.7.1 Active References

At the application level, let us replace the concepts of RPointers and RHeaps with a single concept: *active reference* (ARef). At the OM implementation level, an ARef is a T object (i.e. *not* an OM object). ARefs never reside in OM heaps. An ARef is an aggregate – its representation is not immediate, it resides in the transient heap.

An ARef contains an RPointer and an RHeap – but this of no concern to the programmer. We say that an ARef contains an RPointer and RHeap so that we can describe the ARef approach in terms of primitives we have already discussed. These primitives will no longer be used by the programmer.

ARefs are like LPointers in that they completely specify some OM object. ARefs are active in the sense that they apply only to some particular active heap. An ARef is meaningful only in the context of a particular process.

We can introduce a layer of abstraction that uses ARefs instead of RPointers and RHeaps. For example:

```
(DEFINE (AREF-EXAMINE AREF I)
  (MAKE-AREF (RPOINTER-EXAMINE (AREF-RPOINTER AREF)
                               (AREF-RHEAP AREF)
                               I)
             (AREF-RHEAP AREF)))
```

Where `MAKE-AREF` takes an RPointer and an RHeap and makes (i.e. allocates in the transient heap) an ARef. `AREF-RPOINTER` extracts an ARef's RPointer and `AREF-RHEAP` extracts an ARef's RHeap. Thus, `AREF-EXAMINE` takes an ARef and an offset into an object and returns an ARef to the object referenced from the `I`th slot of the object referenced by `AREF`.

`!PAIR-CAR` can be defined in terms of `AREF-EXAMINE` instead of `RPOINTER-EXAMINE`.

```
(DEFINE (!PAIR-CAR OBJ)
  (AREF-EXAMINE OBJ 0)
```

This new !PAIR-CAR now looks more like T's ordinary CAR than the old !PAIR-CAR does because the new one takes just one argument. Thus we have solved the two argument problem without resorting to a pre-processor.

We need to define AREF-DEPOSIT to serve the same function for ARefs that RPOINTER-DEPOSIT serves for RPointers:

```
(DEFINE (AREF-DEPOSIT AREF1 I AREF2)
  (RPOINTER-DEPOSIT (AREF-RPOINTER AREF1)
                    (AREF-RHEAP AREF1)
                    I
                    (AREF-RPOINTER AREF2)
                    (AREF-RHEAP AREF2)))
```

This procedure sets the Ith slot of the object referred to by AREF1 to be the object referred to by AREF2.

But there is a price for the ARef approach. One price is in the extra layer of indirection it introduces. But more importantly it is expensive in terms of storage in the transient heap. To simply extract a field (e.g. the *car*) of an object (e.g. an OM pair) requires an ARef to be allocated in the transient heap. The cost of this is unacceptably high. But there is a way to avoid the cost.

4.7.2 A smart compiler

Consider the normal T expression:

```
(LET ((X (CONS expression-1 expression-2)))
  (+ (CAR X) (CDR X)))
```

It seems clear that since the *cons* cell constructed in this expression is never passed to a procedure that might store away a reference to the cell, a clever compiler that knows the meanings of the procedures CONS, CAR, and CDR could transform the above expression into:

```
(LET ((X-CAR expression-1)
      (X-CDR expression-2)
  (+ X-CAR X-CDR))
```

applying a procedure similar to reduction in strength.

Now consider the expression:

```
(!PAIR-CAR (!PAIR-CAR OBJ))
```

The inner !PAIR-CAR allocates and returns an ARef. This ARef is passed to the outer !PAIR-CAR and then becomes garbage (since !PAIR-CAR will not store the ARef in any object). Based on the definition of !PAIR-CAR, we can rewrite the above expression as:

```
(LET ((P (AREF-EXAMINE OBJ 0)))
  (AREF-EXAMINE P 0))
```

Based on the definition of AREF-EXAMINE, we can rewrite the above expression as:

```
(LET ((P (MAKE-AREF (RPOINTER-EXAMINE (AREF-RPOINTER OBJ)
                                      (AREF-RHEAP OBJ)
                                      0)
                    (AREF-RHEAP OBJ))))
  (MAKE-AREF (RPOINTER-EXAMINE (AREF-RPOINTER P)
                               (AREF-RHEAP P)
                               0)
             (AREF-RHEAP P)))
```

Using the compiler technique described above, since the ARef held in P is not saved away, we know that we can safely eliminate the first MAKE-AREF, resulting in:

```
(LET ((P-RPOINTER (RPOINTER-EXAMINE (AREF-RPOINTER OBJ)
                                    (AREF-RHEAP OBJ)
                                    0))
      (P-RHEAP (AREF-RHEAP OBJ)))
  (MAKE-AREF (RPOINTER-EXAMINE P-RPOINTER
                               P-HEAP
                               0)
             P-HEAP))
```

Now we have allocated only one ARef instead of two. If the original expression is embedded in (say) another !PAIR-CAR, still only one ARef will be allocated as both inner MAKE-AREFs will be eliminated.

If we implement the proposed compiler technique, we can re-introduce the automatic dereferencing of LPointers described in section 4.2.1. Now however, instead of automatically creating an LPointer to return, we create an ARef. We redefine AREF-EXAMINE to check for an ARef's pointing to an LPointer:

```
(DEFINE (AREF-EXAMINE AREF I)
  (LET ((RP (AREF-RPOINTER AREF))
        (H  (AREF-RHEAP AREF)))
    (IF (LPOINTER? RP)
        (WITH-LPOINTER ((P RP H))
          (MAKE-AREF (RPOINTER-EXAMINE P!R P!H I) P!H))
        (MAKE-AREF (RPOINTER-EXAMINE RP H I) H))))
```

To make AREF-DEPOSIT do the right thing in case the two ARefs it is passed refers to objects that are not in the same heap, it must be defined to create an LPointer that case:

```
(DEFINE (AREF-DEPOSIT AREF1 I AREF2)
  (LET ((RP1 (AREF-RPOINTER AREF1))
        (H1  (AREF-RHEAP AREF1))
        (RP2 (AREF-RPOINTER AREF2))
        (H2  (AREF-RHEAP AREF2)))
    (COND ((= H1 H2)
           (RPOINTER-DEPOSIT RP1 H1 I RP2 H2))
          (T
           (RPOINTER-DEPOSIT
                RP1 H1
                I
                (AREF->LPOINTER AREF2 H1) H1)))))

(DEFINE (AREF->LPOINTER AREF H)
  (!EXPORT-RPOINTER (AREF-RPOINTER AREF) (AREF-RHEAP AREF) H))
```

If the heap of the target object (AREF1) is the same as the heap of the source object (AREF2) – the object a reference to which is being deposited – then the slot in the object is simply set to the RPointer to the source object. Alternatively, if the source and target objects are not in the same heap, an LPointer must be created in the target object's heap; the LPointer must point to the source object. AREF-LPOINTER) is simply a procedure that allocates an LPointer to the same object that the ARefs refers to. Note that we can't simply store the ARef in the target object's heap because an ARef is not a process-context independent quantity (because it contains an RHeap) and it does not refer through the heap index.

4.7.3 Active references and heap activation

We can further extend the ARef scheme to make the activation of heaps transparent to the programmer. The general idea is to automically control what heaps are mapped into the process virtual address space. When an object in a heap needs to be examined, the heap has to be activated. If there is room in the virtual address space, the heap is simply mapped. If there is not room, then some already mapped heap must be "bumped" – i.e. forcibly unmapped to make room for another heap.

Recall that the RHeap data structure contains a HID and an RHeapB. The RHeapB is the active heap's base address in the process virtual address space. Suppose that when a heap is bumped, we set the RHeapB field of the RHeap to be *null*. Since only one RHeap structure is allocated for a single active heap, all the ARefs will refer to a heap via a single RHeap structure. Thus, we can modify AREF-EXAMINE (and similarly AREF-DEPOSIT) to be:

```
(DEFINE (AREF-EXAMINE AREF I)
  (LET ((RP (AREF-RPOINTER AREF))
        (H  (AREF-RHEAP AREF)))
    (IF (NULL? (RHEAP-BASE H))
        (REACTIVATE-HEAP H))
    (IF (LPOINTER? RP)
        (!WITH-LPOINTER ((P RP H))
          (MAKE-AREF (RPOINTER-EXAMINE P!R P!H I) P!H))
        (MAKE-AREF (RPOINTER-EXAMINE RP H I) H))))
```

REACTIVATE-HEAP simply remaps the heap indentified by the RHeap's HID field and updates the RHeap structure's RHeapB field to contain the address at which the heap is remapped. Note that REACTIVATE-HEAP takes an RHeap, while ACTIVATE-HEAP takes a HID. The only times that ACTIVATE-HEAP would be called is in the case of an LPointer being dereferenced (i.e. as a result of executing !WITH-LPOINTER expression in AREF-EXAMINE), or in some "first reference" case.

ACTIVATE-HEAP needs to be modified to check the amount of free virtual address space, and deactivating heaps if necessary to make room. Ideally, heaps should be deactivated using a "least recently used" (LRU) strategy. Supporting LRU would require exporting some page reference information from Aegis. Alternatively, a simple active heap FIFO might be sufficient to manage the address space. This is an area for future research.

4.7.4 Object allocation

It is still the responsibility of the programmer to decide in what heap an object should be placed. The allocation procedures still take an argument specifying in what heap the new object should be created. There is not a "right" or "wrong" place to put an object. Rather there are more or less optimal places. The optimal placement of an object is one that minimizes the number of LPointers to the object. That is, in general, an object should be place in the heap that contains the most references to the object. Placing an object in a sub-optimal place will not cause a program to behave

incorrectly; it will simply increase the execution time of the program and the amount of heap space used.

4.7.5 Benefits and costs

The ARef approach is not in conflict with the approach of storing objects in multiple heaps and having two kinds of references. However, the ARef approach simplifies the application programmer's job since it relieves him of the chores of:

- Following the RPointer/RHeap argument convention,
- Managing LPointers, and
- Activating and deactivating heaps.

The pre-processor approach eliminates only the first of these chores. But even in that chore, it imposes more work on the programmer than does the ARef approach.

The ARef approach has two main costs. First it requires a sophisticated compiler that applies the optimization discussed above. The compiler must reliably detect the cases that can be optimized. If it fails to detect a case, an unnecessary ARef will be allocated. If the case is in the middle of a loop, many unnecessary ARefs may be allocated. The investigation of the compiler techniques involved here are beyond the scope of this work.

Note that the cost of the sophisticated compiler is in both compiler development and compiler execution time. The former cost is paid just once, but it is high enough that we were not willing to pay it for this project. The latter cost is the increased execution time incurred by the logic that detects the optimization we have described. However, this cost can be reduced by not applying the optimization on versions of the procedures that are in the debugging phase. Once debugging is complete, the expensive compilation can be performed – once.

Another cost of the ARef approach is that the cost of the accessors goes up. `AREF-EXAMINE` has two more tests – one to see if an ARef refers to an LPointer, and one to insure that the heap is active – than `RPOINTER-EXAMINE`. This cost is in both code size and execution time. Given the size of the definition of `AREF-EXAMINE`, we are unwilling expand it inline at each occurrence of an accessor. The alternative is to use a procedure call. If we do this, the cost of the ARef approach is only execution time. Note that compared with other existing object-based systems (e.g. Smalltalk and Hydra), the cost of accessing a slot in an object is still fairly cheap.

Chapter 5

Conclusion

5.1 Reviewing the problems and their solutions in OM

In chapter 2 we described the problems that arise in a system that needs to store data permanently. We will now review the problems discussed there and how OM addresses them.

5.1.1 Integrity and atomicity

OM guarantees the integrity of data against logical program error by presenting a consistent programmer interface. This interface insures that programs can access data in heaps only using the primitives that insure integrity.

OM does not address the problems that result from hardware errors or disasterous software errors (e.g. system crashes). Thus, the potential for loss of data integrity is present if such errors occur. We feel that this limitation does not make OM unusable since users already deal with this sort of loss of data due to such errors.

OM does not support atomic operations. However, we see the current OM system as a vehicle on which systems that support atomic operations can be built.

5.1.2 Abstraction

OM supports abstract access to data using the object-oriented programming model and the type system we described. This allows programmers to ignore issues of disk and file formatting. Application programs access data using operations that are logical and abstract.

5.1.3 Storage control

Storage is controlled in OM using the heap model and garbage collection. The time required to allocate a piece of storage (excluding garbage collection overhead) is small. Heaps can be garbage collected independently making the use of garbage collected storage feasible. The heap model seems to be a natural one for the class of application programs whose data structures are naturally partitionable.

5.1.4 Sharing and concurrency

In OM, objects can be shared among users and applications that use the interface presented by OM.

Since heaps are based on DOMAIN files which are page-faulted on demand, only those objects that need to be accessed are ever read into main memory.

OM supports concurrency as well as the DOMAIN system does. That is, application programs can use the DOMAIN synchronization primitives to control concurrent access. For highly concurrent processes, we suspect that these primitives are too expensive.

5.1.5 Security

OM uses the DOMAIN access control primitives to insure security at the heap level. Access to individual objects can not be controlled. For the kinds of applications we have in mind for OM (e.g. the ones we described as sample uses of OM) this restriction is not a serious problem.

5.1.6 Reliability

OM does not address issues of reliability.

5.1.7 Performance

In the design and implementation of OM, we have stressed performance over reliability and availability. We built a system that makes accessing permanent objects nearly as cheap as non-permanent objects. In using OM, programmers do not need to use special techniques (e.g. buffering) to increase performance.

5.1.8 Reference

OM has two kinds of references: local (RPointers) and non-local (LPointers). Local references are small and fast to dereference. Non-local references are larger and more expensive than local references. OM's local references are smaller and cheaper than the references used in many permanent object systems. The combination of RPointers and LPointers allow programs to be as efficient as in conventional programming systems in which all the objects are in a single (relatively small) address space, while supporting a very large number of permanent objects.

Having two kinds of references creates some problems for the programmer. We outlined several techniques for making the fact that there are two kinds of references nearly transparent without giving up the advantages of the two reference scheme.

5.2 Design Philosophy

We approached the problem of a permanent object storage system with a very practical orientation. We used existing hardware and operating system software. We based the programming environment on an existing progamming language. While this limited what our system could do, it enabled us to build a real system in which we could build real application programs.

Bibliography

[1] Guy T. Almes. Garbage collection in an object-oriented system. Technical Report CMU-CS-80-128, Computer Science Department, Carnegie-Mellon University, June 1980.

[2] Guy T. Almes. Integration and distribution in the Eden system. Technical Report 83-01-02, Department of Computer Science, University of Washington, January 1983.

[3] Guy T. Almes. The evolution of the Eden invocation mechanism. Technical Report 83-01-03, Department of Computer Science, University of Washington, January 1983.

[4] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The Eden system: a technical review. Technical Report 83-10-05, Department of Computer Science, University of Washington, October 1983.

[5] *DOMAIN System Command Reference Manual.* Apollo Computer, Incorporated, Chelmsford, Massachusetts, 1983.

[6] *DOMAIN System Programmer's Reference Manual.* Apollo Computer, Incorporated, Chelmsford, Massachusetts, 1983.

[7] Malcolm Atkinson, Ken Chisholm, and Paul Cockshott. PS-algol: an Algol with a Persistent Heap. Technical Report CSR-94-81, Department of Computer Science, University of Edinburgh, December 1981.

[8] Malcolm Atkinson, Ken Chisholm, Paul Cockshott, and Richard Marshall. Algorithms for a Persistent Heap. Technical Report CSR-109-82, Department of Computer Science, University of Edinburgh, April 1982.

[9] M.P. Atkinson, K.J. Chisholm, and W.P. Cockshott. CMS – A Chunk Management System. Technical Report CSR-110-82, Department of Computer Science, University of Edinburgh, April 1982.

[10] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, R. Morrison. PS-ALGOL Papers. Technical Report Persistent Programming Research Report 2, Department of Computer Science, University of Edinburgh, May 1983.

[11] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM* 21(4):280-294, April 1978.

[12] Stoney Ballard, Stephen Shirron. *The design and implementation of VAX/Smalltalk-80.* Addison-Wesley, 1983, pages 127-150. Smalltalk-80: Bits of History, Words of Advice.

[13] *Unix Programmer's Manual, Virtual VAX-11 Version.* Seventh edition, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1980.

[14] Peter B. Bishop. Computer systems with a very large address space and garbage collection. Technical Report MIT/LCS/TR-178, Laboratory for Computer Science, Massachusetts Institute of Technology, May 1977.

[15] Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Surveys* 13(3):, September 1981.

[16] Ellis Cohen and David Jefferson. Protection in the Hydra operating system. In *Proceedings of the Fifth Symposium on Operating Systems Principles*, pages 141-160. ACM Special Interest Group on Operating Systems, November 1975.

[17] L. Peter Deutsch. Efficient implementation of the Smalltalk-80 system. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages.* Association for Computing Machinery, January 1984.

[18] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM* 21(11):966-975, November 1978.

[19] *OZ User Documentation.* Yale University, Department of Computer Science, New Haven, CT, 1983. Internal documentation.

[20] R.S. Fabry. Capability-based addressing. *Communications of the ACM* 17(7):403-411, July 1974.

[21] Adin D. Falkoff, Kenneth E. Iverson. The evolution of APL. In *Proceedings of the ACM SIGPLAN History of Programming Languages Conference.* Association for Computing Machinery, June 1978.

[22] Adele Goldberg and David Robson. *Smalltalk-80: The language and its implementation.* Addison-Wesley, 1983.

[23] Maurice Herlihy. Transmitting Abstract Values in Messages. Technical Report MIT/LCS/TR-234, Laboratory for Computer Science, Massachusetts Institute of Technology, May 1980.

[24] M. Herlihy and B. Liskov. A Value Transmission Method for Abstract Data Types. *ACM Transactions on Programming Languages and Systems* 4(4):527-551, October 1982.

[25] Carl Hewitt. The Apiary network architecutre for knowledgable systems. In *Conference Record of the 1980 LISP Conference.* Stanford University, August 1980.

[26] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM* 26(6):419-429, June 1983.

[27] Paul Hudak. Distributed task and memory management. In *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 277-289. Association for Computing Machinery, 1983.

[28] Ted Kaehler and Glenn Krasner. *LOOM - Large object-oriented memory for Smalltalk-80 systems.* Addison-Wesley, 1983, pages 251-270. Smalltalk-80: Bits of History, Words of Advice.

[29] K.C. Kahn, W.M. Corwin, T.D. Dennis, H. D'Hooge, D.E. Hubka, L.A. Hutchins, J.T. Montague, F.J. Pollack, M.R. Gifkins. iMax: A multiprocessor operating system for an object-based computer. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 127-136. ACM Special Interest Group on Operating Systems, December 1981.

[30] Donald E. Knuth. *The Art of Computer Programming, Volume 1, Fundamental Algorithms.* Addison-Wesley, 1969.

[31] Glenn Krasner, ed. *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley, 1983.

[32] Edward D. Lazowska, Henry M. Levy, Guy T. Almes, Michael J. Ficher, Robert J. Fowler, and Stephen C. Vestal. The architecture of the Eden system. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 148-159. ACM Special Interest Group on Operating Systems, December 1981.

[33] Paul J. Leach, Bernard L. Stumpf, James A. Hamilton, and Paul H. Levine. UIDs as internal names in a distributed file system. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 34-41. Association for Computing Machinery, 1982.

[34] Paul J. Leach, Paul H. Levine, Bryan P. Douros, James A. Hamilton, David L. Nelson, and Bernard L. Stumpf. The architecture of an integrated local network. Technical Report, Apollo Computer, Incorporated, 1983.

[35] Henry M. Levy. A comparitive study of capability-based computer architectures. Master's thesis, University of Washington, 1981.

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

[37] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems* 5(3):381-404, July 1983.

[38] Allen William Luniewski. The architecture of an object based personal computer. Technical Report 232, Laboratory for Computer Science, Massachusetts Institute of Technology, December 1979.

[39] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual.* The M.I.T. Press, Cambridge, Massachusetts, 1965.

[40] MC68000 16-bit microprocessor user's manual. Motorola, Incorporated, 1980.

[41] F. J. Pollack, K. C. Kahn, and R. M. Wilkinson. The iMAX-432 object filing system. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 137-147. ACM Special Interest Group on Operating Systems, December 1981.

[42] Data Curator Group. *POMS Manual.* Department of Computer Science, University of Edinburgh, 1983. Data Curator Documentation.

[43] G. Radin. The 801 Minicomputer. In *Proceedings of the ACM SIGARCH/SIGPLAN Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 39-47. Association for Computing Machinery, March 1982.

[44] Jonathan A. Rees and Norman I. Adams IV. T: a dialect of Lisp or, lambda: the ultimate software tool. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming.* Association for Computing Machinery, August 1982.

[45] Jonathan A. Rees and Norman I. Adams IV. Experience With an Implementation of a Lexically-Scoped LISP. Internal memorandum, Yale University Department of Computer Science.

[46] Jonathan A. Rees and Norman I. Adams IV, and James R. Meehan. *The T Manual.* Yale University, Department of Computer Science, 1984.

[47] Brian K. Reid and Janet H. Walker. *Scribe User Manual.* Computer Science Department, Carnegie-Mellon University, 1978.

[48] D. M. Ritchie and K. Thompson. The Unix time-sharing system. *Communications of the ACM* 17(7):365-375, July 1974.

[49] Alan Snyder. A machine architecture to support an object-oriented language. Technical Report MIT/LCS/TR-209, Laboratory for Computer Science, Massachusetts Institute of Technology, March 1979.

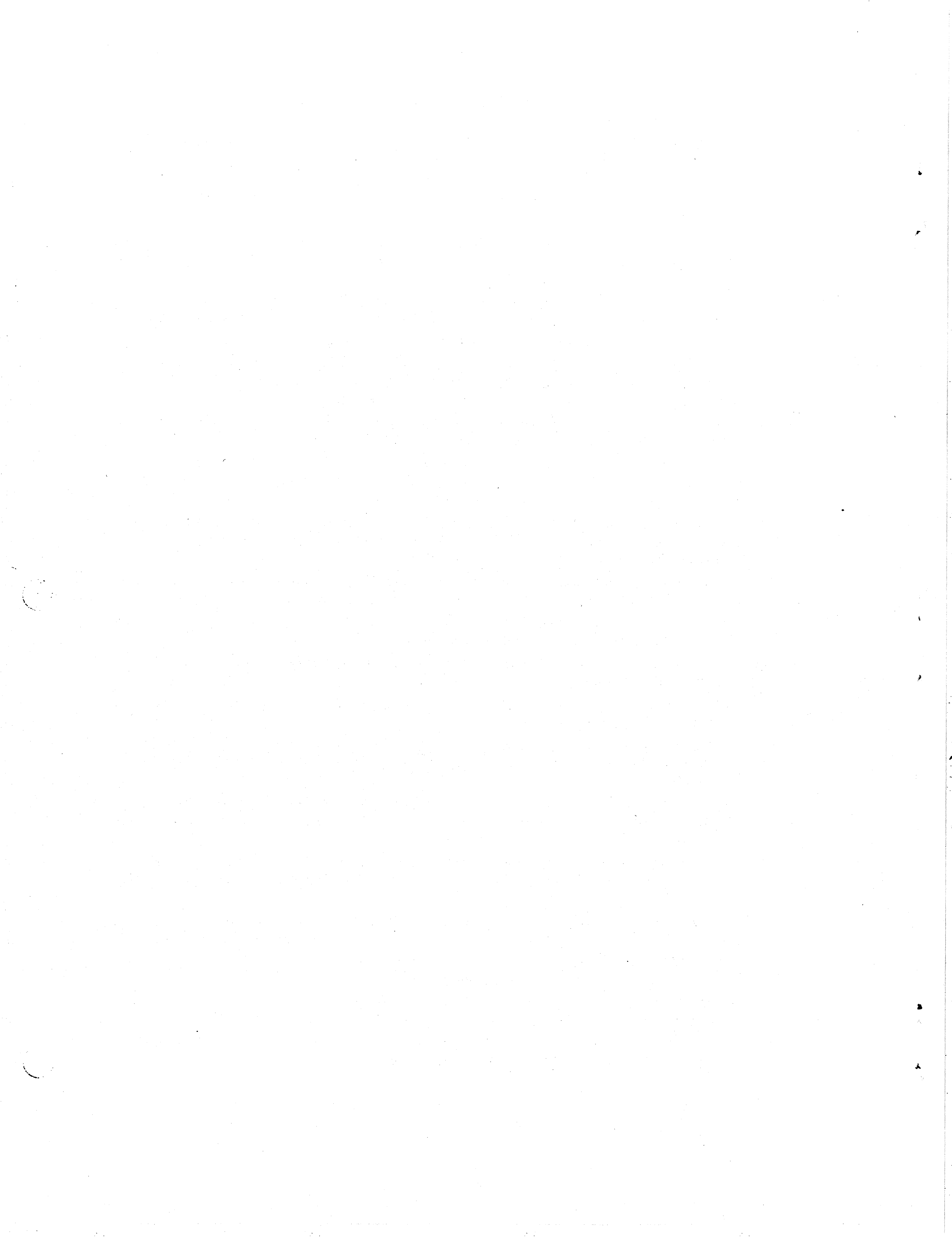
[50] Karen Sollins. Copying Complex Structure in a Distributed System. Technical Report MIT/-LCS/TR-219, Laboratory for Computer Science, Massachusetts Institute of Technology, May 1979.

[51] James William Stamos. A large object-oriented virtual memory: grouping strategies, measurements, and performance. Technical Report SCG-82-2, Xerox Palo Alto Research Center, May 1982.

[52] Guy Lewis Steele, Jr., and Gerald Jay Sussman. The revised report on Scheme, a dialect of Lisp. AI Memo 452, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, January 1978.

[53] Daniel Weinreb and David Moon. *Lisp Machine Manual.* Third edition, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1981.

[54] W. Wulf, R. Levin, C. Pierson. Overview of the Hydra operating system development. In Proceedings of the Fifth Symposium on Operating Systems Principles, pages 122-131. ACM Special Interest Group on Operating Systems, November 1975.



DISTRIBUTION LIST

Office of Naval Research Contract N00014-82-K-0154
Michael J. Fischer, Principal Investigator

Defense Technical Information Center
Building 5, Cameron Station
Alexandria, VA 22314
(12 copies)

Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Dr. R.B. Grafton, Scientific
Officer (1 copy)

Information Systems Program (437)
(2 copies)

Code 200 (1 copy)
Code 455 (1 copy)
Code 458 (1 copy)

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106
(1 copy)

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D.C. 20375
(6 copies)

Office of Naval Research
Resident Representative
715 Broadway, 5th Floor
New York, NY 10003
(1 copy)

Dr. A.L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
Code RD-1
Washington, D.C. 20380
(1 copy)

Naval Ocean Systems Center
Advanced Software Technology Division
Code 5200
San Diego, CA 92152
(1 copy)

Mr. E.H. Gleissner
Naval Ship Research and Development Center
Computation and Mathematics Department
Bethesda, MD 20084
(1 copy)

Captain Grace M. Hopper
Naval Data Automation Command
Washington Navy Yard
Building 166
Washington, D.C. 20374
(1 copy)

Defense Advance Research Projects Agency
ATTN: Program Management/MIS
1400 Wilson Boulevard
Arlington, VA 22209
(3 copies)