OBJECT STORAGE AND INHERITANCE FOR SELF, A PROTOTYPE-BASED OBJECT-ORIENTED PROGRAMMING LANGUAGE

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

ENGINEER

By

Elgin Hoe-Sing Lee

December 1988

Acknowledgments

This thesis would not have been possible without the efforts of the other members of the SELF project. I especially thank my adviser David Ungar for more things than I can count. To name but a few: for his patience and support as this thesis went on much longer than anticipated, for his careful reading and constant prodding to make this work better, for sharing his insights on object-oriented languages and implementations, for creating a powerful and exciting language, and last but not least, for supplying us with an official SELF espresso machine—complete with demitasse cups!

I also thank Craig Chambers, my office mate and the mastermind of the SELF compiler, for sharing his knowledge and insights, and for being there to tell me when I'm completely addled.

I would like to acknowledge that this research has been supported by a National Science Foundation Presidential Young Investigator Grant # CCR-8657631, and by IBM, Texas Instruments, NCR, Tandem Computers, Apple Computer, and Sun Microsystems.

The final portions of this thesis were written while I was an employee at ParcPlace Systems. I would like to thank Glenn Krasner, Peter Deutsch, and everyone else at ParcPlace for bearing with me while I was putting in the last burst of effort on this thesis.

Finally, I dedicate this thesis to my parents, who have patiently helped me through the years with their love and care.

Table of Contents

Chapter 1

Introduction

It is by no longer having to think about things that civilisation progresses.

—Joseph E. Stoy

1.1. Overview

SELF† [UnS87] is a prototype-based object-oriented programming language designed for exploratory programming. To support exploratory programming, SELF provides a number of features which enhance programmer productivity but which may entail significant run-time costs: automatic storage reclamation, dynamic binding of polymorphic procedure calls, run-time typing, unification of procedures and variables, multiple inheritance, and dynamic inheritance. The costs of those features could be forbiddingly high in a straightforward implementation of SELF. The processing power and memory available in today's computers, however, enable the use of more sophisticated implementation techniques like generation scavenging and in-line method caching, which reduce the run-time costs sufficiently to make such language features practical.

Our SELF implementation addresses run-time costs both with compile-time techniques like custom code compilation and with run-time techniques like in-line method caching and generation scavenging. This thesis deals only with the run-time implementation, specifically in the areas of object inheritance and storage management. The compilation techniques, although essential for an efficient

† The logo SELF will be used in this thesis when referring to the language and implementation in order to avoid confusion with other uses of the term "self."

implementation, lie outside the scope of this thesis and will not be discussed except as they relate to the object inheritance and storage mechanisms.

Much of the work described in this thesis builds upon previous research on high performance implementations of the Smalltalk-80™† [GoR83, Gol84] programming system. Unlike Smalltalk, however, SELF is based on prototypes, not classes. As a result, techniques like in-line caching, which originated in Smalltalk systems, had to be adapted for a prototype-based system. One of the principal contributions of this thesis is to demonstrate how techniques for efficient implementation of class-based object-oriented programming languages can be applied to an implementation of a prototype-based language.

1.2. Enhancing programmer productivity

The idea of applying computational resources to ease the task of programming has been around since the start of the computer era. The use of an assembler program, for instance, enabled programmers to work with convenient mnemonics for machine instructions rather than raw bits of binary data. Later on, languages such as FORTRAN, COBOL, and ALGOL allowed programmers to work with higher level abstractions than simply machine instructions; such languages depended on compiler programs that translated from the abstractions used by the programmer to the low-level machine instructions executable by the hardware. More recently, programming systems like Smalltalk-80 have applied computing power to support programmers through all stages of the programming process, from object-oriented mechanisms at the language level that ease the task of program composition to sophisticated programming tools in the environment that help programmers understand the

† Smalltalk-80™ is a trademark of ParcPlace Systems. In the remainder of this thesis, the term "Smalltalk" will be used to refer to the Smalltalk-80™ programming language.

structure of complex programs. The goal of the SELF research project is to produce a similar programming system based on the SELF language.

The level of programmer support provided by systems like Smalltalk and SELF is the exception rather than the norm. It is an agonizing reality that most commercial programming is still done with little programming support. In some areas—tools for the programming environment, for example—the matter is probably just one of development lagging research. With the passage of time, the commercial world can reasonably be expected to catch up and offer sophisticated programming environments as a matter of course. In the area of programming languages, the path of progress is more uncertain. Despite a powerful and flexible language and a highly supportive environment, Smalltalk has been slow in gaining general acceptance—at least partly due to the perception that it provides too much support for the programmer at run-time and is therefore hopelessly inefficient. Run-time efficiency is still a shibboleth for many programmers.

Although the concern with run-time efficiency is well justified for many applications, there are a number of areas for which optimizing the time and effort spent programming is more important than optimizing the time spent actually running the program—the reverse of the approach taken by conventional compiler-based programming systems.

One such area is rapid prototyping, in which the form of the actual end product is not precisely known. The point of prototyping is to allow the programmer to explore variations in order to discover what form the end product should take. User interface design is an example of an area where this process of exploration is important. In rapid prototyping, the key to programmer productivity is ensuring that both creating an initial prototype and revising it can be accomplished quickly and painlessly.

Systems optimizing programming effort are also well-suited to the creation of programs to solve small-scale problems. Some examples might be programs to explore the properties of Ackerman's function or to demonstrate principles of elementary physics. Because the writer of such a program may not be programmer by trade, the programming system should make programming quick and painless so that the writer can concentrate on solving the problem at hand rather than on getting the program to run.

Languages and environments that reduce the amount of time and effort spent in programming are ideal for such situations, even if some run-time overhead is incurred. Programs might run more slowly than if written in a traditional programming language, but the absolute speed of programs is not always critically important. An analogous situation occurs in the publishing domain: documents produced through desktop publishing are usually inferior in appearance to those produced by professional typesetters, but that does not seem to have greatly affected the success of desktop publishing. The important point is that the documents look "good enough." Similarly, what matters in the programming arena is not exactly how fast programs run but whether they run quickly enough to serve their purpose; if that condition holds, then the time and effort saved during programming amply justify the run-time cost.

The problem for programming systems is to enhance programmer efficiency in a way such that the resulting programs run quickly enough for their purposes. That problem is not trivial: early experiences in implementing Smalltalk [Kra83] demonstrated that straightforward implementations were only marginally usable on conventional hardware. Later Smalltalk implementations using more sophisticated run-time techniques like in-line method caching and generation scavenging

demonstrated much better performance [DeS84, CaW86, Ung86]. Our SELF implementation combines such run-time techniques with compile-time optimizations to produce a high-performance system that according to preliminary measurements is within a factor of 4 from optimized C on the Sun-4 [ChU88]. These experiences illustrate the importance of efficient implementation techniques for languages like Smalltalk and SELF.

1.3. Benefits and costs of language features

To establish the utility of the work in this thesis, it is useful to first consider the reasons underlying some of SELF's more computationally expensive features. What do such features provide for the programmer that justify run-time inefficiency in a straightforward implementation and extra complexity in an efficient one? Identifying potential costs also motivates the design choices made in SELF and in related implementations. In this section, I will examine the benefits and costs of several significant features of SELF. Those features will be treated here in general terms; specific details of their inclusion in SELF will be deferred until Chapter 3.

1.3.1. Object-oriented programming

The basic ideas behind the object-oriented programming model—objects and inheritance—originated with the Simula programming language [DaN66]. Simula also introduced classes, which describe the implementations of similar groups of objects. Classes, objects, and inheritance were further developed and extended by Smalltalk, which applied a uniform approach to the object model. Every part of the Smalltalk system, including classes and activation records, was a full-fledged object; and every procedure call was a message sent to an object, which determined for itself what action to take in response. Smalltalk objects also provided information hiding,

since the internal state of an object was only publicly accessible via the procedures defined for that object.

Various definitions of object-oriented programming have been offered [Weg87a, HaN87, DaT88], differing principally on whether classes, inheritance, and information hiding are integral to the model. The approach taken by SELF is that objects and inheritance form the essence of object-oriented programming. Classes and information hiding are regarded as orthogonal concepts, neither of which is supported by SELF.

1.3.1.1. Objects

An object consists of both state (represented by variables) and behavior (represented by the operations defined on the object). Objects support decentralized definitions for generic operations. For example, a generic "print" operation that supports both strings and integers would be implemented in a conventional programming language as a single procedure that must understand the representations of both types. In contrast, the object-oriented approach would specify separate "print" procedures for strings and integers. Strings would know how to print themselves but need know nothing about the internal representations of integers, and vice versa.

The actual procedure invoked to perform an operation depends on the identity of the first argument. In Smalltalk terminology, also adopted by SELF, the generic operation is denoted by a *message* and the first argument is termed the *receiver* of the message. Computation results from sending a message to an object, which responds by invoking its actual procedure (or *method)* for that message. Such message sends are effectively procedure calls that are resolved just before they are invoked. A given

message send may invoke different procedures at different times, depending on the identity of the receiver at the time.

In class-based languages like Smalltalk, all objects belong to classes. An object is said to be an *instance* of its class, which defines its format and behavior. The format for an object describes the layout of its data fields, termed *instance variables*. An object's behavior, also known as its message protocol, specifies which procedures are executed in response to messages. The values stored for the instance variables of one instance of a class are completely independent of the values stored for any other instance of the same class. In contrast, all instances of a class share the same procedures.

1.3.1.2. Inheritance

An inheritance mechanism allows an object to inherit default definitions of attributes that it does not directly define, such as operations, format, or state. There are two principal types of inheritance mechanisms, one based on classes and the other based directly on objects.

Class-based inheritance allows operations for an object that are not directly specified by its class to be inherited from other classes, termed the *superclasses* for its class. An inheritance hierarchy for classes determines how default operations are inherited. Consider, for example, bordered windows, which support all of the operations of plain windows except that the display routine draws a border around each window. A bordered window class might be defined as a subclass of a plain window class; it would inherit all of the procedures of the plain window class except for the display procedure, which is overridden to draw the window border in addition to the rest of the window.

In a class-based system, format is inherited as well as behavior. An object contains not only the instance variables defined for objects of its class but the instance variables defined for objects of its superclasses as well. Only the existence of instance variables is inherited, not their actual values. For example, the plain window class might define instance variables for screen coordinates, which would be inherited by the bordered window class. As a result, all instances of bordered windows would include screen coordinate instance variables, but the values of those variables would be independent of the values for any plain window instance.

Object-based inheritance [HaN87], also termed delegation [Lie86, Weg87a] and classless inheritance [Weg87b], allows objects to inherit state and behavior directly from other objects. Unlike class-based inheritance, object format is not automatically inherited. For example, consider a bordered window object inheriting from a plain window object. In the object-based inheritance model, the fact that the plain window contains data fields for screen coordinates does not imply that the bordered window object must also contain such data fields. Although format is not inherited, object-based inheritance mechanisms like SELF's allow data values to be inherited: if the bordered window does not specify its own screen coordinates, it would inherit the values for the plain window. This ability to inherit state is one of the most important differences between class-based and object-based inheritance.

1.3.1.3. Benefits

The object-oriented approach fosters code reuse by providing dynamic binding of polymorphic messages. Dynamic binding supports the creation of procedures that depend only on the protocol of operations supported by their arguments, not on the actual identities or types of the arguments. For example, a routine that prints all the objects in a list need not know whether those objects will be integers, strings, or

complex numbers. By deferring the binding of its embedded "print" message until run-time, when the type of object to be printed is known, the routine could be reused for lists of strings, lists of integers, or lists of any mixture of objects supporting the print message.

Inheritance mechanisms foster both reusable and malleable code. Inheritance provides the ability to program incrementally, which can dramatically reduce programming time because the programmer would only need to specify differences from previously written routines. The inheritance mechanism also encourages well-factored programs, in which behavior common to distinct data types are factored out and placed in a shared superclass or ancestor. A single change in a routine would then automatically propagate to all objects inheriting it, making sweeping changes possible with minimal effort.

Polymorphism and inheritance therefore provide the ability to quickly create and revise programs, an ability which is a hallmark of the model and one of the principal reasons for the model's burgeoning acceptance. However, languages designed for type security—like C++ [Str86], Trellis/Owl [SCW85], and Eiffel [Mey87]—limit that ability by requiring both polymorphic messages and inheritance to adhere to type constraints. In such languages, all objects referenced in code are given types; the dynamic binding of an operation for an object is constrained to choose only those methods defined for the object's type and subtypes. A print method defined on a list of numbers would not work on a list of strings, even if strings support the print operation. Though type constraints are beneficial for security, verification, and other concerns of programming in the large, they inhibit the reuse of code by requiring the programmer to decide when he writes a routine the exact domain for which that routine is applicable. Such a decision forces the programmer to focus on unnecessary

details while programming and requires him to forecast all possible reuses of the code in order to correctly make typing decisions.

The power of inheritance is likewise limited in type-secure languages. For such languages, inheritance is a mechanism for subtyping, not for reusing code. A type inherits all the operations of its supertypes and can only extend or override them; exclusion of operations is prohibited. Such a constraint provides type security: a type can always be used wherever one of its supertypes can be used, and that constraint can be checked statically at compile time. Though such a constraint provides type security, it also inhibits flexibility, since it does not support inheriting operations solely for the sake of reusing code; objects cannot inherit only a subset of the message protocol of its parent. Such inheritance for code reuse is used, for example, in the Smalltalk collection hierarchy [GoR83], where Dictionary inherits from Set solely on the basis of similar implementations.

SELF does not support types because it was principally designed to foster rapid code production. For such a domain, the flexibility of untyped polymorphism and inheritance outweighs the benefits of type security.

1.3.1.4. Costs

Although dynamic binding of polymorphic procedure calls and variable types supports reusability, it is also one of the primary sources of run-time cost in object-oriented programming. A straightforward implementation of message lookup, for instance, could traverse a potentially large multiple inheritance graph on every message send. The problem is even more pronounced in SELF: since both procedure invocations and variable access use the same late-binding mechanism, a traversal of the inheritance graph could be required for every variable access as well as for every procedure invocation.

The cost of dynamic binding caused languages like Simula-67 and C++ to use static binding by default and to require the programmer to explicitly specify which procedures should be bound at run time. Though static binding reduces the cost of a procedure call, it also conflicts with the object-oriented programming model. In the object-oriented model, objects determine for themselves the manner in which they respond to messages. With static binding, on the other hand, the actual procedure invoked for a message depends not on the identity of the receiver but on the type declared for the receiver in the code sending the message. Consider, for example, a screen redisplay routine which runs through a window list and redisplays all the windows. The body of such a routine might keep the current window in a variable and redisplay the window by sending the display message to the variable. With static binding, if the window variable is declared to be of type "plain window" then the plain window display procedure will always be invoked, even if the window variable actually refers to a bordered window which overrides that procedure. The benefits of object-oriented programming are therefore diluted in languages that use static binding for efficiency reasons.

The multiple inheritance semantics of SELF can require more of the inheritance graph to be traversed than is necessary in other languages. Normal inheritance in SELF is unordered, requiring a search of all parents in order to detect conflicting methods; in comparison, languages such as Flavors [Moo86], LOOPS [BoS83], and CLOS [BDG88] provide ordered inheritance, in which the inheritance lookup can terminate after finding the first matching method. SELF's multiple inheritance semantics also includes an implicit disambiguation rule that makes the inheritance lookup more complex than those of other languages supporting unordered lookup, like Eiffel and Trellis/Owl.

An additional cost stems from the existence of dynamic inheritance, which allows an object to change its inheritance at run-time. Though dynamic inheritance entails minimal additional cost in a straightforward implementation of late-binding, it causes complications for more sophisticated implementations, which typically cache binding information and which therefore must take care to invalidate appropriate portions of the cache when dynamic inheritance occurs.

Finally, well-factored object-oriented programs tend to be composed of many short procedures, resulting in frequent procedure calls. That characteristic can be expensive for architectures with substantial procedure call overhead.

1.3.2. Automatic storage reclamation

Automatic storage reclamation removes the burden of storage reclamation from the programmer and places it on the run-time system, which automatically reclaims storage for inaccessible objects.

1.3.2.1. Benefits

Automatic storage reclamation allows the programmer to concentrate on the design of his program without the distraction of having to worry about reclaiming storage. In addition, a program using automatic storage reclamation is likely to be developed more quickly: not only does the programmer not have to implement storage management, but he does not have to spend time debugging it. Automatic storage reclamation eliminates a significant class of programming errors: dangling references and storage leaks.

1.3.2.2. Costs

Automatic storage reclamation can consume a substantial amount of run-time. Shaw's analysis [Sha88] of the results from Gabriel's LISP benchmarks, for example,

showed that the average amount of total execution time spent reclaiming storage was 38% for the 36% of the benchmarks which reported any garbage collection. The overhead for storage reclamation in early Smalltalk implementations ranged from 7% to 40% of total CPU time [BaS83, UnP83, FaS83]; as observed by Ungar [Ung86], these times were for interpreted systems, and the run-time overhead could be expected to be worse for a compiled implementation.

Where the costs lie depends on the method of storage reclamation. A great deal of research has been done for automatic storage reclamation, and the most recent techniques require lower overhead than indicated in the previous paragraph. Ungar's generation scavenging scheme for SOAR, for example, reduced the CPU time overhead to around 3% in a compiled implementation [Ung86]. Our SELF implementation uses a version of generation scavenging.

1.3.3. Summary

Though object-oriented programming, dynamic binding of polymorphic procedure calls, and automatic storage management enhance programmer productivity, they add a significant amount of run-time cost. Straightforward implementations of the Smalltalk-80 system on stock hardware were found to be inadequate [McC83a]. Run-time costs might be expected to be even worse for SELF, since it includes all of the costly run-time features of Smalltalk plus the space overhead of prototypes and the ability for objects to change their inheritance characteristics at run time via dynamic inheritance. Techniques to efficiently deal with these costs are essential for a successful implementation of SELF and are the subject of this thesis.

1.4. Overview of the implementation

The implementers of SELF were Craig Chambers, David Ungar, and myself. David Ungar supplied general vision and the overall design of the implementation; he was also the principal architect of the memory system. Craig Chambers was primarily responsible for the compiler, while I wrote the storage manager and inheritance routines. All three project members took part in implementing the memory system substrate.

Though the SELF language stresses flexibility and expressive power over run-time efficiency, performance and efficiency were major concerns for our implementation efforts. The implementation is therefore substantially more complex than a straightforward one. To help reduce that complexity, we chose to write the bulk of the implementation in C++ [Str86], a high-level object-oriented programming language that produces reasonably efficient code. One side note is that though we chose C++ because we thought that its object-oriented features would help us produce a cleaner and simpler implementation (which it did to some degree), taking advantage of such C++ inline procedures actually added a large amount of complexity, principally in organizing header files to support our use of inlining.

Our SELF implementation currently runs on the Sun-3 (MC68020) and Sun-4 (SPARC) workstations. A port to the Macintosh-II is presently being undertaken by Lloyd Chambers.

1.5. Organization of the thesis

Chapter 2 examines some previous work in language implementations. It focuses primarily on Smalltalk implementations, since they were the source of most of the run-

time techniques used in the SELF implementation. The chapter also reviews previous research for automatic storage reclamation.

Chapter 3 describes the semantics of the SELF language. The chapter explains the language in detail so that the reader may understand the precise semantics implemented by the techniques described by the following chapters.

Chapter 4 describes the architecture of the SELF memory system. The chapter covers the layout of SELF objects and the composition of the object memory spaces.

Chapter 5 describes the techniques used to implement object storage management, focusing on the generation scavenger and the mark-and-sweep garbage collector.

Chapter 6 examines the implementation of object inheritance in SELF. The chapter covers simple inheritance, multiple inheritance, and dynamic inheritance.

Chapter 7 summarizes the main points of the thesis and concludes.

Appendix A describes SELF syntax.

Appendix B presents performance data gathered from measurements of the memory system.

Chapter 2

Previous Work

2.1. Introduction

SELF shares many characteristics with Smalltalk: object-oriented programming, late binding of procedure calls, automatic storage management, and user-definable control structures. For that reason, our implementation draws heavily on previous research on building high-performance Smalltalk implementations. Accordingly, the first part of this chapter will focus on Smalltalk implementation efforts.

SELF also shares one characteristic with many other programming languages designed for interactive programming: it removes the burden of storage management from the programmer by providing automatic storage reclamation. The chapter ends with a review of the large body of previous research on efficient algorithms for automatic storage reclamation.

2.2. The early years of Smalltalk-80

Early versions of Smalltalk-80 were implemented by researchers at the Xerox Palo Alto Research Center on Xerox Altos, Dorados, and Dolphins. In 1980, four other companies (Apple Computer, Digital Equipment Corporation, Hewlett-Packard, and Tektronix) implemented Smalltalk systems [Kra83] based on a formal specification [GoR83] of a stack-oriented Smalltalk Virtual Machine, which provided a detailed low-level description of an interpreted Smalltalk system.

The specification included:

- *Byte-code interpretation.* The Smalltalk Virtual Machine was specified to interpret a byte-coded instruction set specialized for Smalltalk.

- *Tagging*. An object pointer consisted of a 15-bit content field and a one bit tag, indicating whether the content field represented a 15-bit signed integer or a 15-bit object memory reference.
- *Object table indirection.* An object reference was represented as an index into an object address table. Each table entry contained the actual heap address of the object and some bookkeeping information. Use of the object table supported quick compaction, increased the amount of memory that could be referenced through a 15-bit pointer, and enabled quick performance of the Smalltalk *become* primitive that switched all references from one object to another.
- *Method cache.* A method cache was used to avoid performing a full method search on each message send. Resolving a message send involved hashing the class of the receiver with the message selector and using the resultant value as a key into a global hash table of methods; only if this probe into the method cache failed would a full search take place.
- *Special messages*. Primitive operations like "+"are specially treated by handled by the interpreter. The "+" message, for example, is usually sent to integers, so the interpreter performs in-line integer addition for this common case, only resorting to a full method search only if the arguments are not integers.
- *Reference counting*. Storage management was performed using reference counting. Since reference counting could not reclaim circular structures, a mark-and-sweep garbage collector was also specified to reclaim such structures when necessary.

- *Heap allocation of activation records*. Method and block activation records (called *contexts* in Smalltalk terminology) were allocated on the heap in the same manner as all other Smalltalk objects. Heap allocation is required in the general case because Smalltalk activation records could be allocated in non-LIFO fashion, due to the semantics of Smalltalk blocks and to the ability to capture any method activation as a full-fledged object.

The Xerox Dorado system [Deu83] was the first Smalltalk system to exhibit acceptable performance. It achieved its performance by taking advantage of its hardware platform: the Dorado was a high-performance (70 ns cycle time) ECL personal computer with a large (4K) microprogram store. The majority of the Smalltalk-80 virtual machine was implemented in microcode, including the byte-code interpreter, the majority of the storage management code, and a sizeable number of primitives.

Implementations for stock hardware adhering to the formal specification exhibited disappointing performance. For instance, two early implementations by Hewlett-Packard and Tektronix [FaS83, McC83b] written in high level languages (C and Pascal) and running on stock hardware (Motorola 68000 and VAX-11/780) performed dismally on McCall's macro-benchmarks [McC83a]; they turned in performances ranging from 1% to 4% of Smalltalk performance on the Xerox Dorado, which at that time the only implementation whose performance was considered adequate [UBF84, Ung86].

The DEC assembly language implementation [BaS83] differed from the specification in a few important respects. First, a modified Baker incremental copying collector was used for garbage collection. Unlike a reference counting collector, the Baker collector could reclaim circular structures. Finally, method and block contexts,

though still allocated in the heap, were reclaimed specially; that scheme allowed about 85% to 90% contexts to be reclaimed immediately as part of the return operation.

The first incarnation of Berkeley Smalltalk (BS) [UnP83] was written in C and ran on the VAX-11/780. It reduced the cost of reference counting through some of the same methods used in the Xerox Dorado and Dolphin implementations. As in the DEC implementation, contexts were freed immediately on return when possible. BS also used careful coding to reduce the cost of decoding byte codes in software.

The second Tektronix MC68000 implementation [Wir83] provided the best performance of all the early implementations on stock hardware. It was written in carefully tuned assembly language, used deferred reference counting to limit reference-counting overhead, and did not create complete context objects for leaves of the message send tree. Though the best of the lot, it turned in a performance that only ranged from 9% to 22% of the Dorado's in the macro benchmarks.

2.3. Deutsch and Schiffman (PS)

L. Peter Deutsch and Alan Schiffman [DeS84] developed a high performance Smalltalk system (PS) for the MC68000. Their implementation introduced a number of techniques used in subsequent implementations.

- *Dynamic translation.* Unlike previous systems, which were all interpreted, PS dynamically translated Smalltalk virtual machine byte-code methods into native code for direct execution. The native code methods were kept in a cache for subsequent reuse.
- *Stack allocation of activation records.* PS allocated method contexts (procedure activation records) on a stack and normally deallocated them in

strict LIFO fashion. Contexts on the stack were stored in machine-oriented form rather than as full-fledged objects, enabling PS to take advantage of stack-oriented machine instructions. Contexts required to be full-fledged objects would be converted to object form in the heap (possibly passing through an intermediate form on the stack along the way). PS always created block contexts (closures) in the heap since they represent deferred evaluation. Heap-resident contexts could not directly support execution and so were converted back into machine-oriented form on the stack before they could resume execution.

- *In-line caching of method addresses.* After a message send was resolved via a run-time method search, PS dynamically linked a call to the target method in-line, so that subsequent sends at the same call site would directly invoke the method last invoked. The prologue of the method needed to verify that the current receiver's class was the same as the last. If so, then the main body of the method would be executed; otherwise, a full method search would be undertaken and the process repeated. In-line caching was based on the observation that 95% of the time a given send would invoke the same method that it did the last time [DeS84, DAm83].
- *Deutsch-Bobrow deferred reference counting* [DeB76]. The Deutsch-Bobrow techniques were reported to eliminate approximately 85% of the reference counting overhead in a standard implementation.

The Deutsch-Schiffman system was the first Smalltalk system for a commercial, non-microprogrammable computer to demonstrate reasonable performance relative to the Dorado. It received a Xerox performance rating of 44% of a Dorado on a Sun-2

(MC68010) and 109% of a Dorado on a Sun-3 (MC68020), based on the 13 Smalltalk macro-benchmarks and two of the micro-benchmarks. [Bay85, Ung86].

2.4. Suzuki and Terada

Independently of Deutsch and Schiffman, Norihisa Suzuki and Minoru Terada [SuT84] developed software techniques to improve the run-time performance of Kiku, a typed version of Smalltalk implemented on the MC68000.

- *Static binding of special cases*. In addition to a standard method cache, the Kiku implementation addressed the costs of late binding by statically binding message sends in a number of special cases: sends to *super*, sends to *self* when the message is not defined in a subclass of the class containing the sending method, and sends to *here* (a Kiku-specific keyword indicating that the search should begin in the class that contained the sending method).
- *Optimized handling of sends within the same class*. Another message binding optimization compared the class of the receiver with the class containing the sending method; if the two classes were the same, then the proper method could be directly invoked without going through the full method search. That optimization was based on an observation that more than 90% of messages invoke methods in the same class as the sender.
- *Deutsch-Bobrow deferred reference counting* [DeB76]. Suzuki and Terada adopted a variation of the Deutsch-Bobrow algorithm.
- *Stack allocation of activation records*. The implementation reduced context allocation and deallocation overhead through delayed retention of contexts. Contexts are normally allocated and deallocated in strict stack

discipline. While on the stack, contexts are not full-fledged Smalltalk objects. If the top context must be used as a full-fledged object or a process switch occurs, all the contexts in the stack are moved into the heap. To bound the time of that operation, the stack is fixed size; on overflow, contexts are moved into the heap in FIFO order.

2.5. Berkeley Smalltalk II

The second version of Berkeley Smalltalk [Ung86], which ran on the Sun-2 (MC68010), incorporated several improvements over its predecessor:

- *Generation scavenging*. Storage reclamation was performed using Ungar's generation scavenging algorithm, a stop-and-copy collector segregating objects based on lifetimes. The BS II implementation used two generations. Generation scavenging required only 1.5% CPU time overhead on a Sun-3, less than a fourth of the time taken by its nearest competitor, the modified Baker collector in the DEC implementation, which took 7%. (Both those overhead measurements were for interpreted systems; for comparison, deferred reference counting in PS required around 11% overhead [Ung86] in a faster, compiled system.)
- *Direct pointers*. Object pointers (after tag removal) were direct references to objects; no object table was used. Direct pointers allowed full use of virtual address space and eliminated the indirection required for every object field reference in a system with an object table. The use of generation scavenging eliminated the need of an object table for compaction.

Despite those improvements, BS II still exhibited modest performance. It was rated at 14% of a Dorado [Bay84, Ung86], compared to PS's rating of 44% on the same hardware.

2.6. SOAR

The SOAR (Smalltalk On A RISC) project [UBF84, Ung86] at UC Berkeley designed and built a high performance Smalltalk system for a specially designed reduced instruction set NMOS microprocessor. Unlike the Dorado, which executed Smalltalk byte codes in microcode, SOAR was designed to run a compiled version of Smalltalk. The SOAR microprocessor provided a number of language-specific hardware features, including tagged arithmetic instructions, register windows, hardware support for garbage collection, parallel register clearing, byte insert/extract instructions, vectored traps, and tagged immediate data. The SOAR project also utilized a number of important software techniques to achieve high performance:

- *Compilation to a low-level instruction set.* SOAR used a low-level RISC instruction set with a few language-specific features, an approach that distinguished SOAR from other hardware projects for Smalltalk, which provided hardware support for byte-code emulation.
- *In-line caching of method addresses.* SOAR adopted the Deutsch-Schiffman in-line cache. Measurements indicated that the in-line was the most important software feature aside from compilation; SOAR was estimated to be 33% slower without it.
- *Stack allocation of contexts.* Contexts were normally allocated in LIFO form on a stack and moved into stable form on the heap as necessary. SOAR provided multiple overlapping on-chip register windows to cache

the context stack, reducing the cost of a procedure call. Addressable registers allowed pointers to be generated to contexts in the register file. Hardware trap mechanisms aided detection of blocks which needed to be moved into the heap. Ungar also showed how software could eliminate the need for addressable registers, at a cost of a 3% performance penalty.

- *Generation scavenging*. SOAR used the same generation scavenging algorithm as BS II. Despite running Smalltalk programs ten times faster than BS II, SOAR's scavenging overhead was only 3%—less than a third the cost of deferred reference counting, its nearest competitor for a compiled Smalltalk implementation.
- *Direct pointers*. Like BS II, SOAR did not use an object table.

The SOAR system exhibited good performance relative to the Dorado: although SOAR's projected cycle time was over 5 times slower than the Dorado's (400 ns versus 70 ns), simulations demonstrated median performance at 107% of the Dorado over five of the standard Smalltalk macro-benchmarks. (The final NMOS SOAR chip had a cycle time of 510 ns due to an unforeseen critical path to memory.)

Two of SOAR's hardware ideas—register windows and tagged arithmetic—have been incorporated in the Sun SPARC, a commercial RISC processor used in the Sun-4.

2.7. Sword32

Researchers at the University of Tokyo produced Sword32 (also known as Katana-32) [SKA84], a 32-bit VLSI microprocessor designed to efficiently emulate the Smalltalk-80 byte-coded instruction set in microcode. Hardware features included byte-code dispatching support and an on-chip register file for contexts. Sword32 grew out of the work done by Suzuki and Terada [SuT84] and included similar software approaches, including Deutsch-Bobrow deferred reference counting

and stack allocation of contexts (using the on-chip register file). Though an object table was used, object pointers were not indices into the table but direct references to the entries.

2.8. Caltech Object Machine

The Caltech Object Machine (COM) [DaK85] was designed by researchers at the California Institute of Technology for the efficient execution of late-binding object-oriented programming languages like Smalltalk. The principle features of COM were hardware approaches to context management and late binding of message sends.

Contexts were fixed in size and allocated from a free list referenced by a dedicated register. A dual-ported set associative cache keyed on the absolute addresses of contexts provided fast context allocation, deallocation, and access. Unlike the SOAR register windows, contexts in the COM cache were not required to be contiguous.

An associative global method cache supported late binding of messages in hardware. Instructions were abstract; the effect of any instruction depended on the class of its operands.

Other hardware features included floating point addresses, tagged memory, and three level addressing. Floating point addresses provided variable length segment and offset fields, which supported both large numbers of small objects and small numbers of large objects. The only software technique indicated was the use of a free list for the context cache, which allowed arguments to be directly stored into the next context block.

2.9. Swamp

Researchers at the University of Toronto designed Swamp (Smalltalk Without All that Much Pipelining) [LGF86], a microprogrammed processor directly executing Smalltalk byte code. Hardware features included a cache of non-overlapping contexts; tag checking for both arithmetic and stores (to support garbage collection); and a hardware method cache. Software approaches were similar to SOAR's and included stack allocation of contexts, a generation-based garbage collector (with eight generations rather than SOAR's two), and direct object pointers.

The method cache was essentially a hardware implementation of the common Smalltalk global method cache. Unlike the COM cache, the Swamp method cache was keyed by a hash of the receiver and message, rather than their concatenation.

Swamp was implemented using a standard bit slice ALU and sequencer, TTL MSI, and NMOS LSI RAMS. It operated at a cycle time of 136 ns and was estimated to run from 44% to 89% faster than its closest competitor on five micro-benchmarks on an absolute basis. When performance was compared on the basis of normalized cycle times, however, the estimated Swamp performance was 28% to 52% slower than SOAR.

2.10. Tektronix 4406

The Tektronix 4406 Smalltalk system was a high-performance interpreter-based Smalltalk implementation for the MC68020 [CaW86]. Major features of the implementation included direct object pointers, a generation-based garbage collector with seven generations, and stack allocation of contexts. Object formats were also revised for a cleaner implementation.

Despite being an interpreted system, the Tektronix 4406 exhibited reasonable performance relative to the Dorado. Comparing the benchmarks for the 4406 with those for the Dorado cited by Ungar [Ung86], the 4406 system performed the standard Smalltalk compiler benchmark at 78% of a Dorado, as compared to the initial performance of 80% reported for the Deutsch-Schiffman system that compiled to 68020 native code. The median of the five macro-benchmarks cited by Ungar was 67% of a Dorado for the 4406, compared to 99% for the Deutsch-Schiffman system.

2.11. Review of storage reclamation algorithms

A number of different types of storage reclamation algorithms have been developed over the years. Three principal approaches to storage reclamation have been used: mark-and-sweep, reference counting, and moving (scavenging).

2.11.1. Mark-and-sweep

A mark-and-sweep garbage collector operates in two phases. In the first phase, the collector starts at the base registers accessible to the program and from there recursively marks all accessible objects, usually by setting a bit for each accessible object. In the second phase, the collector sweeps through all of program memory and reclaims all unmarked objects, typically by adding them to a free list.

The basic form of a mark-and-sweep garbage collector was first described by John McCarthy in 1960 for his LISP programming system [McC60]. Since then, many variations on mark-and-sweep collection have been introduced. Some of these variations are restricted to cases where all objects are the same size, as is the case with LISP cons cells. This section will concentrate on algorithms applicable to variable-sized objects.

2.11.1.1. Marking phase

A number of variations to the marking phase have been proposed [Knu73, Sta80, Coh81]. A straightforward recursive version of the basic marking algorithm requires stack space to store the state of the recursion. A more efficient non-recursive version may be written using an explicit stack that only stores object references and not return addresses.

Many early implementations were concerned with the amount of available memory during a garbage collection. In the absence of virtual memory, garbage collection may take place without sufficient memory to hold a stack for the marking phase. To enable garbage collection with limited available memory, various algorithms have been designed that reduce the amount of memory needed for marking at the cost of greater processing time.

One such algorithm is the Deutsch-Schorr-Waite pointer reversal algorithm [ScW67], which avoids the use of a stack by reversing pointers during the graph traversal of the marking phase until a leaf or previously marked node is found. The pointer reversal can then be undone to back up one node, and the graph traversal may continue from that point, until eventually all nodes have been marked and all reversed pointers have been restored. The Deutsch-Schorr-Waite marking algorithm requires greater processing time than a stack-based scheme, so various hybrid algorithms have been proposed which use a fixed-size stack until overflow and then use pointer reversal techniques [ScW67, Coh81].

2.11.1.2. Sweeping phase

In the sweeping phase, the collector scans linearly through the heap, adding unmarked objects to the free list and unmarking marked objects. Although that operation reclaims all unused storage, it leaves the heap filled with holes and

suffering from fragmentation. In addition to simply reclaiming storage, the sweeping phase can also compact the heap to remove unused space between objects surviving the garbage collection. Although compaction is an expensive operation, it is important for good memory performance in a virtual memory environment, for which the problem is not so much one of reclaiming storage to avoid memory exhaustion but of compacting active storage to avoid thrashing [FeY69]. As a further refinement, a garbage collector designed for a paged virtual memory environment can reorganize objects during compaction to improve locality of reference.

Jacques Cohen classifies compaction algorithms into three categories based on the relative positions of objects after compaction [Coh81].

- *Arbitrary* compaction algorithms may leave objects in an arbitrary order after compaction.
- *Linearizing* compaction reorganizes memory so that objects which point to each other are adjacent or nearby after compaction; such compaction algorithms can improve paging performance in a virtual memory environment. The moving garbage collectors discussed in section 2.11.3 perform linearizing compaction.
- *Sliding* compaction moves objects toward one end of the address space without changing their linear order. Compacting mark-and-sweep garbage collectors that deal with variable-sized objects are generally of the sliding type.

The various compaction algorithms principally differ on how they adjust pointers in the heap to refer to the new object locations following compaction. One simple compaction method would be to build an object relocation table describing data movement during compaction. After compaction, a linear scan through the heap would

relocate pointers based on the information in the object table. Cohen [Coh81] presents a good review of more sophisticated compaction algorithms.

One such algorithm, due to Haddon and Waite [HaW67], first scans through the heap performing the actual compaction and building a break table describing the initial address and size of each hole. A second scan then updates pointers based on the break table. No extra storage is needed for the break table, since the space available in the holes can be shown to be large enough to store the table (which may at times move from one hole to a larger one during compaction).

Compaction in the LISP 2 system, described by Knuth [Knu73, pp. 602-603] takes three passes through the heap. The first pass links all the holes together, the second pass adjusts pointers based on the hole list, and the final pass performs the actual compaction. Various refinements to that scheme deal with speeding up the pointer adjustment pass, through such techniques as building break tables [Coh81].

F. Lockwood Morris developed a compaction algorithm based on pointer reversal [Mor78]. Rather than relocating pointers based on the location and size of holes, Morris's algorithm uses pointer reversal to link all pointers to an object into a list headed by the first word of that object. When an object is moved, all references to it are updated by undoing the pointer reversal. A problem with simple pointer reversal is that the first word for an object may function as either a list head for the object's relocation list or as a reference to another object which must be placed into a relocation list for another object. Morris's algorithm resolves that problem by observing that if all pointers pointed in the same direction (such as from lower to higher addresses), processing objects in that direction would enable each such problematic location to be finished in its role as list head before it plays the role of object reference. The algorithm therefore performs two scans of the heap, one scan

processing only forward pointers and the other only backward pointers. The second scan also performs the actual compaction, undoing the pointer reversal after moving each object.

2.11.1.3. Evaluation

The principal advantages of mark-and-sweep garbage collection are:

- All allocated storage may be used by the user, which may be important for systems without virtual memory.
- All inaccessible storage is reclaimed, a point that is not true for reference-counting and moving schemes.

Along with those advantages, however, come two significant disadvantages:

- All user computation must stop during the collection, leading to a long pause time while the collection occurs. Such a long pause may be intolerable for interactive programming systems.
- Mark-and-sweep collection may also exhibit poor virtual memory performance. In the absence of linearization, the marking phase may reference memory pages in arbitrary order, resulting in thrashing and poor performance.

2.11.2. Reference counting

The use of reference counting for storage reclamation was first proposed by George Collins [Col60]. In its simplest form, a reference-counting system keeps track of the number of immediate references held to each object. The reference count for an object is incremented whenever a new pointer to the object is stored and decremented whenever a pointer to the object is overwritten. When an object's reference count reaches zero, storage for the object is reclaimed. If the reclaimed

object itself refers to other objects, the reference counts for those objects are decremented, and objects whose reference counts become zero as a result of that operation are recursively deallocated.

2.11.2.1. Cyclic garbage

Reference counting by itself does not reclaim all inaccessible storage. A cyclic structure (such as an object referring to itself) will have non-zero reference counts even if no pointer to it exists from the outside. As a result, simple reference counting cannot reclaim storage for inaccessible cyclic structures. More complex forms of reference counting are able to reclaim some forms of cyclic garbage at the cost of more processing overhead [Sta80, Bro85]. D. R. Brownbridge [Bro85] describes a general cyclic reference counting algorithm, but it incurs substantial overhead (two extra reference counts and two extra bits per object, and one extra bit in each pointer) and still requires a localized mark-and-sweep collection for each reclaimed cyclic structure.

2.11.2.2. Recursive freeing

One of the strong points of reference counting is that the time for storage reclamation is distributed over the course of program computation, unlike the mark-and-sweep collectors which produce lengthy and noticeable pauses in computation to perform garbage collection. Recursive deallocation of objects, however, may also produce lengthy pauses in computation in a reference counting system. To avoid such pauses, some reference counting systems defer recursive freeing until necessary (e.g., [Wei63]). An object whose reference count reaches zero is added to the free list, but its own references are not processed until the object is removed from the free list.

2.11.2.3. Deferred reference counting

Simple reference counting entails a substantial amount of overhead to keep the reference counts up to date. That overhead includes both the space overhead of storage for the reference counts and the time overhead of updating reference counts on each memory store operation. L. Peter Deutsch and Daniel G. Bobrow [DeB76] developed a deferred reference counting algorithm for LISP that reduces both forms of overhead.

Space overhead is reduced through the observation that most objects in a LISP system have a reference count of one. Deutsch and Bobrow take advantage of that observation by only storing reference counts for objects with multiple references. Such objects are entered along with their reference counts into a hash table called the multireference table (MRT). Objects with a reference count of zero are stored in a zero count table (ZCT). Objects not in either the MRT or ZCT are presumed to have a reference count of one. The reference count for an object does not include references to the object from program variables, so objects that are only referred to from variables have a reference count of zero and are stored in the ZCT. Objects referred to from variables are also stored in a variable reference table (VRT). Objects that are reclaimable may then be found by determining which objects are in the ZCT but not in the VRT.

Time overhead is reduced by deferring reference-counting operations. Instead of updating the reference count tables on each transaction that affects them, the algorithm writes the transactions to a transaction file, which is then processed at suitable intervals to update the tables. Since groups of transactions are processed together, transactions that cancel each other need (like a pointer creation and

destruction) not be entered into the tables. Using a transaction file therefore reduces the number of actual reference counting updates that must be recorded.

Although the Deutsch-Bobrow algorithm was specifically designed for LISP, it is applicable to other languages with similar patterns of allocation and deallocation. In particular, it was used in a number of high performance Smalltalk systems, including Dorado Smalltalk [Deu83] and PS [DeS84].

2.11.2.4. Evaluation

The most attractive point of reference counting is that it distributes garbage collection time over the course of computation, rather than stopping all computation for lengthy intervals like mark-and-sweep collection. The lack of lengthy collection pauses is a major advantage for real-time tasks like animation and for interactive programming, where lengthy pauses are an annoying distraction.

One disadvantage is that reference counting cannot reclaim all inaccessible storage. As previously mentioned, general cyclic structures cannot be reclaimed by simple reference counting . In addition, since the size of reference count fields is finite, reference counting cannot reclaim objects whose reference counts overflow, since the actual reference counts for such objects cannot be known. A reference counting system must therefore also include a full trace-accessible garbage collector.

Another disadvantage is that the total effort spent in reference counting is proportional to the number of dead objects. For systems like Smalltalk, in which most allocated objects quickly become garbage, the generational moving collectors discussed in section 2.11.3.1 can exhibit better performance.

2.11.3. Moving collectors

A moving collector operates by moving accessible objects to a new storage area rather than reclaiming inaccessible ones. After all accessible objects have been moved to the new space, the collector reclaims all the space in the old storage area. Moving collectors are well-suited for virtual memory environments because they automatically provide compaction of objects in the new memory space. As a side effect of storage reclamation, most moving collectors also reorganize object storage so that objects are usually stored adjacent to their references.

M. L. Minsky [Min63] proposed the first moving collector, designed for LISP. The collector moves copies of objects along with relocation addresses into secondary or temporary storage. An explicit stack is used to keep track of remaining structures to move. After an object is moved, the original object is replaced with a forwarding pointer to the relocated object in the new memory space. When an object is moved to the secondary storage, the collector follows the forwarding pointers for any embedded references when constructing the embedded references of the copy. As a result, shared objects will remain shared in the new memory space. After all accessible objects are written out, the second pass reads the previously written information to build the compacted object structure in the new memory space (which could be the same as the old memory space). After collection, all LISP cdr sequences end up linearly packed.

Robert R. Fenichel and Jerome C. Yochelson [FeY69] developed a moving collector specifically designed for use with virtual memory. Their collector divided the object storage space into two semispaces. At any time, only one semispace is used for object storage. Following Baker's terminology [Bak78], that semispace will be referred to as the *from* space and the other as the *to* space. When a collection is

initiated, accessible objects are moved in depth-first order from the *from* semispace to the end of the *to* space, providing compaction and object reorganization for free. Forwarding pointers are left in the place of previously moved objects to allow the collector to retain object sharing in the moved object structure. After all accessible objects are moved, the sense of the semispaces is reversed: the former *from* space, now containing only garbage, is considered to be the new *to* space, and the former *to* space is considered to be the new *from* space. New objects are allocated in the new *from* space until the next collection commences. The Fenichel-Yochelson algorithm used recursion and an implicit stack to keep track of the state of the collection.

C. J. Cheney [Che70] refined the Fenichel-Yochelson algorithm by eliminating the need for recursion and a stack. Objects directly accessible by program variables are first moved from *from* space to *to* space, as in the Fenichel-Yochelson algorithm. Objects in *to* space are then scanned from the beginning to the end of *to* space. During the scan, if any references to objects in *from* space are found, those objects are moved to the end of *to* space, with the corresponding references updated. The scan of *to* space may therefore cause *to* space to grow in size and the end of the space to move. The collection ends when the scanning pointer reaches the end of *to* space. No stack, either implicit or explicit, is needed to keep track of the remaining objects to be moved; the objects in *to* space serve that function ably. Because *to* space is scanned from beginning to end, the objects end up compacted in breadth-first order after a collection.

The previously mentioned moving collectors stop all computation during the collection, causing potentially lengthy pauses in computation. Henry G. Baker, Jr. [Bak78] addressed that problem by refining the Cheney collector to create an incremental moving collector that interleaved collection with computation. Rather

than collecting all accessible objects at once, the algorithm distributes collection by causing each object allocation to move a fixed number of accessible objects into *to* space before the allocation takes place. To the user program, all objects appear to reside in *to* space. During program execution, memory references to objects in *from* space must be detected so that those objects may be moved into *to* space before the program continues. References to previously moved objects are altered to follow the proper forwarding pointers.

2.11.3.1. Generation-based collection

One of the advantages of moving collectors is that the collection time is independent of the number of inaccessible objects—a point that is not true of either the mark-and-sweep or reference-counting collectors. The collection time is therefore short when the number of inaccessible objects in *from* space is much greater than the number of objects that must be moved into *to* space.

Henry Lieberman and Carl Hewitt [LiH83] refined Baker's algorithm to minimize the number of objects that survive a collection. They observed that most LISP objects live for a short amount of time and then die, becoming inaccessible. Objects that have survived for a long time, on the other hand, are likely to continue to live for a good while longer. Lieberman and Hewitt used those heuristics to guide collection.

Objects are divided into generations based on age, and each generation is stored in a separate pair of Baker semispaces. Since objects in younger generations are more likely to die, collecting younger generations is likely to reclaim more space for less effort. Each generation is therefore collected separately, with younger generations being collected more frequently than older ones. Objects that survive long enough are moved to older generations. When a object is moved, all pointers to that object must be updated. To avoid having to scan older generations to update

pointers when an object moves, no object may contain a direct pointer to an object in a younger generation—pointers to younger object must go through an entry table for the younger generation. After a generation is collected, pointers to objects of that generation may be updated by collecting younger generations and altering pointers in the proper entry table. The entry table must be maintained by software checks on memory stores.

Based on the work of Lieberman and Hewitt, David A. Moon developed an incremental generation-based collector for the Symbolics 3600 that uses special hardware to maintain the tables of references to the youngest generation and to handle references to *from* space that must be routed through the appropriate forwarding pointers [Moo84].

Stoney Ballard and Stephen Shirron [BaS83] implemented a moving collector for Smalltalk based on the Baker algorithm and incorporating elements of the Lieberman-Hewitt algorithm. The collector employed two generations. The newest generation was stored in a pair of semi-spaces managed using Baker's garbage collector. The oldest generation was stored in a static object region that is only collected (by a mark-sweep collector) when the user saves a snapshot of his system. Static objects that contain references to dynamic objects are remembered by an exit table to enable the Baker collector to preserve the dynamic objects at collection time. The exit table is maintained by software checks on each memory store. Objects in the dynamic region that survive a specified number of Baker collections are eligible for tenuring into the static object space, but such tenuring only takes place when the user makes a snapshot.

David Ungar [Ung86] designed a stop-and-collect (non-incremental) generation-based moving collector for Smalltalk called Generation Scavenging. Generation

Scavenging uses two generations, termed the *old* and *new* regions. The *new* region is subdivided into three subregions, which will be termed *eden*, *from* space, and *to* space. To minimize paging, new objects are always created in *eden*, which is expected to always remain in main memory. *From* space holds objects that have survived at least one scavenge (move into *to* space). During a collection, accessible objects are moved from *eden* and *from* space into *to* space, or into *old* space if the object is older than the tenuring threshold. At the end of the collection, the *from* and *to* spaces are interchanged. To avoid having to scan all of *old* space to update pointers when an *new* object is scavenged, *old* objects containing references to *new* objects are stored in a table called the remembered set. Updating pointers in the *old* generation may be accomplished by only looking at objects in the remembered set. The remembered set is maintained by software checks on memory stores. Unlike other generation based collectors, Generation Scavenging never scavenges *old* space, which contains the oldest generation. Instead, a full mark-and-sweep garbage collector runs at infrequent intervals to reclaim garbage in *old* space.

David Ungar and Frank Jackson studied the problem of tenuring objects in Generation Scavenging [UnJ88]. Tenuring objects too liberally reduces pause time for scavenging but wastes space because prematurely tenured objects will die soon after being moved to *old* space, resulting in unused storage that cannot be reclaimed using scavenging. Tenuring objects too conservatively reduces wasted space but produces longer pause times, as more surviving objects are moved back and forth by successive scavenges. Based on empirical data, Ungar and Jackson proposed a demographic feedback-mediated tenuring policy to address the tenuring problem.

2.11.3.2. Evaluation

Moving collectors possess a number of desirable characteristics:

- They automatically provide compaction, which is highly desirable in a virtual memory environment. Objects are typically compacted in either depth-first or breadth-first order, leading to improved locality of reference over random object ordering.
- They can also reclaim general cyclic structures, which is not true of reference-counting algorithms.
- The time required for a collection depends only on the number of survivors; it is independent of the number of inaccessible objects reclaimed. That characteristic results in shorter collection times than in the other methods when the number of surviving objects is much less than the number of inaccessible objects.

Generation-based moving collectors take advantage of the last point to produce shorter collection times than other methods. Generation-based reclamation allows the collector to concentrate on objects with the highest mortality rate. Since the number of survivors per collection is much less than the number of inaccessible objects reclaimed, the time required to perform a collection is low.

Another issue is the use of incremental versus non-incremental collection. The Baker, Lieberman-Hewitt, and Moon algorithms are incremental, while Generation Scavenging is not. Incremental garbage collectors distribute the time for storage reclamation over the course of program execution, reducing the possibility of noticeable pauses. Distributing collection overhead, however, comes at the cost of greater overall reclamation overhead because such incremental moving collectors must check all memory loads to detect whether referenced objects are in a *from* space, so that the scavenger may move such objects into the corresponding *to* space or into an older generation. For non-real-time applications, a stop-and-collect

algorithm like Generation Scavenging appears preferable, offering lower overall overhead with pauses that are unnoticeable in normal interactive applications (mean pause time of 19 ms every 2.3 seconds in SOAR) [Ung86]. Generation Scavenging with a demographic feedback-mediated tenuring policy appears to offer the best performance for storage reclamation in an interactive programming environment. It is the scheme adopted by SELF.

2.12. Summary

Three areas in the Smalltalk system have been identified as performance bottlenecks: automatic storage reclamation, late binding of message sends, and context management. Those performance bottlenecks also apply to SELF. The following table compares selected Smalltalk systems to SELF, emphasizing the software approaches used to reduce the three performance bottlenecks.

SYSTEM	TYPE	OBJECT TABLE	GARBAGE COLLECTOR	METHOD SEARCH	CONTEXT MANAGEMENT
PS	dynamic compiler	yes	reference counting†	in-line cache	stack
BS II	bytecode interpreter	no	generation scavenger	method cache	heap (free list)
SOAR	static compiler	no	generation scavenger	in-line cache	stack
Sword32	bytecode emulator	yes	reference counting	method cache	stack
Swamp	bytecode emulator	no	generation scavenger	method cache	stack
Tektronix	bytecode interpreter	no	generation scavenger	method cache	stack
SELF	dynamic compiler	no	generation scavenger	in-line cache	stack

† Later versions of PS used a generation scavenger.

Our SELF implementation incorporates techniques developed in the Deutsch-Schiffman, BS II, and SOAR systems to address those bottlenecks: generation scavenging, an in-line method cache, and stack allocation of activation records. Minor changes were required to adapt these techniques for use in a prototype-based language and to mesh well with SELF's customized code compilation approach.

The Sun-4 version of SELF also takes advantage of the SPARC register windows and tagged arithmetic instructions—two features contributing substantially to SOAR's performance.

Later chapters will examine in more detail the issues involved in applying the Smalltalk implementation techniques to SELF.

Chapter 3

The SELF Language

3.1. Introduction

This chapter describes the SELF programming language in detail to make clear the exact semantics implemented by the techniques described in later chapters. The description of SELF in this chapter differs in some respects from the original one by David Ungar and Randall Smith [UnS87]. As we implemented SELF and began to program in it, we (David Ungar, Craig Chambers, and Elgin Lee) noted a number of ways in which it could be improved. The description in this chapter incorporates our changes, which range from minor points of syntax to major additions like ordered multiple inheritance.

The chapter is divided into two sections. Section 3.2 examines the two basic characteristics distinguishing SELF from conventional object-oriented programming languages: use of prototypes instead of classes and reliance on message-passing for both procedure invocation and state access. Section 3.3 describes the semantics of SELF, focusing first on the basics (objects and messages) and then surveying more advanced features (blocks, multiple inheritance, and dynamic inheritance).

3.2. The basis of SELF: prototypes and message-passing

SELF is a simple and expressive programming language for exploratory programming. It owes its greatest debt to Smalltalk, which demonstrated the power of a uniform model of communicating objects. Ungar and Smith observed that Smalltalk could be simplified by eliminating classes and by using message-passing for both procedure invocation and state access; such simplifications would produce a

purer form of the object-oriented programming model, one which would be both easier to understand and more flexible. That observation was the basis for the SELF language design.

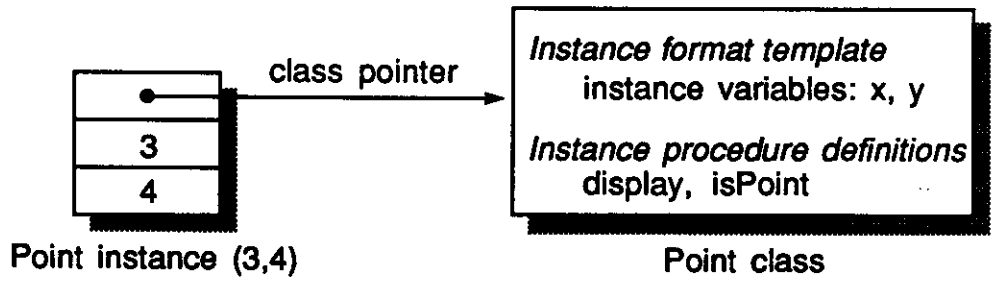
3.2.1. Classes versus prototypes

SELF uses the prototype-based object model because the model's greater simplicity and flexibility are significant advantages for exploratory programming. Before examining more closely the argument for prototypes, it is useful to first take a closer look at classes—particularly because prototype-based systems originally arose as a reaction to class-based systems.

3.2.1.1. Classes: characteristics and usage

The class-based object model deals with two types of entities: classes and instances. A class embodies the abstract specification of the format and behavior of a group of similar objects, termed its instances. Every object must belong to a class, but classes are not required to have any instances.

For example, all Cartesian points in Smalltalk belong to the `Point` class:

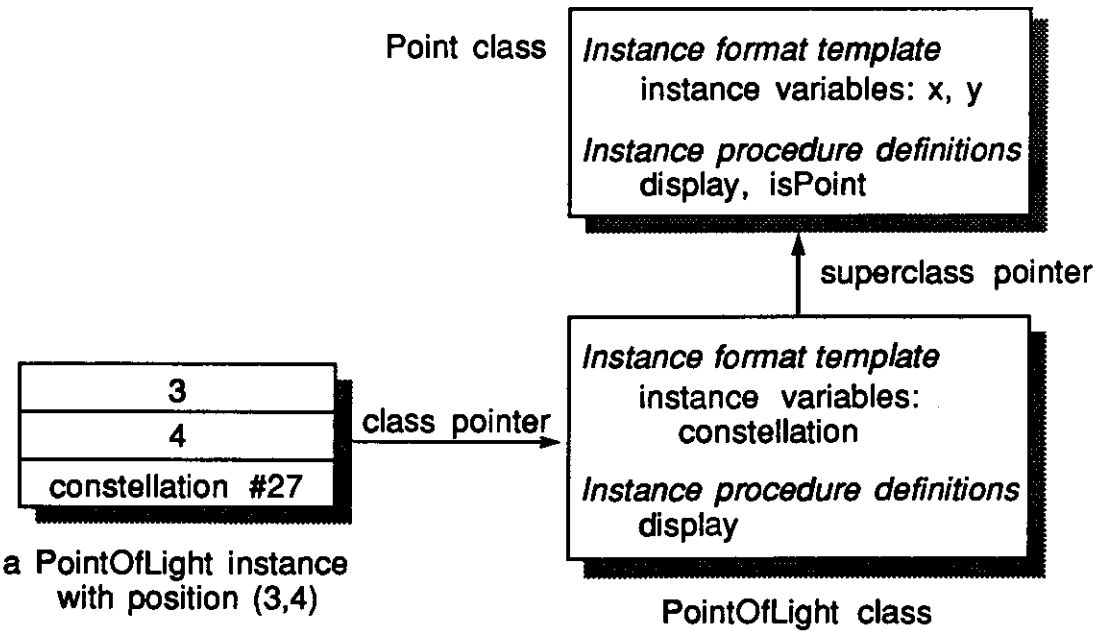


The `Point` class defines the data representation of all point instances: each point contains two data fields—termed instance variables—specifying its x- and y-coordinates. The class also defines how its instances respond to messages—which code is executed, for example, when a point receives the `display` or `isPoint` message. Although the example above does not define one, a class could also define

class variables containing data common to all instances of the class.

A new object is created by requesting an instance of the desired class. In an uniformly object-oriented language like Smalltalk, such a request takes the form of an instantiation message sent to the class. In languages like C++, however, where classes are not themselves full-fledged objects, object creation cannot be expressed through message-passing—a class cannot be the receiver of a message—so it is accomplished by another mechanism, the `new` operator.

Conventional class-based languages use an inheritance mechanism to share formats and behavior between groups of objects related in a class-superclass hierarchy. For example, suppose that one wishes to define the format and behavior of points of light, which are similar to normal `Point` objects except that they are displayed differently† and tend to cluster in groups of a thousand. To share common format and behavior between points of light and normal points, a `PointOfLight` class could be created as a subclass of the `Point` class:



† A point of light might, for example, be displayed as a rather nebulous and diffuse area of incandescence roughly centered at its x-y position.

The `PointOfLight` class inherits the instance format template of its superclass, `Point`. `PointOfLight` also contains a local instance template; such a local template can extend an inherited one but cannot override it. Since `PointOfLight` is a subclass of `Point`, it must inherit all the `Point` instance variables (`x` and `y`). The local instance template extends the inherited `Point` template with an instance variable named `constellation`, which identifies the cluster to which an instance belongs. The local and inherited instance templates jointly define the format of individual `PointOfLight` instances.

A class also inherits by default all the instance procedures of its superclasses. In the example above, the `PointOfLight` class would inherit by default the `display` and `isPoint` procedures from `Point`. Unlike the case for instance variables, a class may override a procedure defined in its superclass. The `PointOfLight` class inherits only the `isPoint` procedure from its superclass; it overrides the generic `Point display` procedure with a procedure specialized for `PointOfLight` objects.

In class-based languages, the inheritance hierarchy consists only of classes; instances do not inherit format or behavior from other instances. That attribute is a major point of difference between class-based and prototype-based languages. As will be seen later, object-based inheritance is one of the principal reasons for the greater flexibility of prototype-based languages.

In addition to determining how behavior is shared, a class-based inheritance hierarchy also presents a descriptive—typically taxonomic—organization of the program and object domain. Such an organization may improve the manageability of large systems. Peter Wegner [Weg87b] has argued that class-based inheritance hierarchies provide one of the best notations available for the organization of complex systems.

3.2.1.2. The metaclass problem

Though classes may be a useful structuring mechanism, they add complexity to the object model. In a uniformly object-oriented system, everything is an object, from procedure activations to classes. The clearest advantage of the uniform approach is simplicity: every part of the system is accessed in the same way. Besides presenting a simple model of the system, being uniformly object-oriented also makes the system malleable and accessible—programs can easily inspect and modify substantially all parts of the system using the standard object mechanisms of the language.

In a class-based system, the fact that a class is an object means that it too must belong to a class (termed a *metaclass*), which is also an object and must belong to a class, ad infinitum. Metaclasses are not the result of an idle exercise in philosophy; they supply useful functionality in a class-based, uniformly object-oriented system. For the previously mentioned `Point` class, for example, a new point would be created by sending the `new` instantiation message to the `Point` class. Since the message protocol for an object is defined by its class, the definition of how the `Point` class responds to the `new` message must be found in the metaclass for `Point`. A metaclass can also be a useful place to define procedures, such as class initialization routines, that only operate on class-specific data in class variables or class instance variables.† Though metaclasses can be useful, empirical studies have found them to be the greatest obstacle encountered in learning Smalltalk [OSh86, BoO87]. Students and teachers in those studies were unable to understand metaclasses and worried at length about where the seemingly endless chain of metaclasses stopped.

---

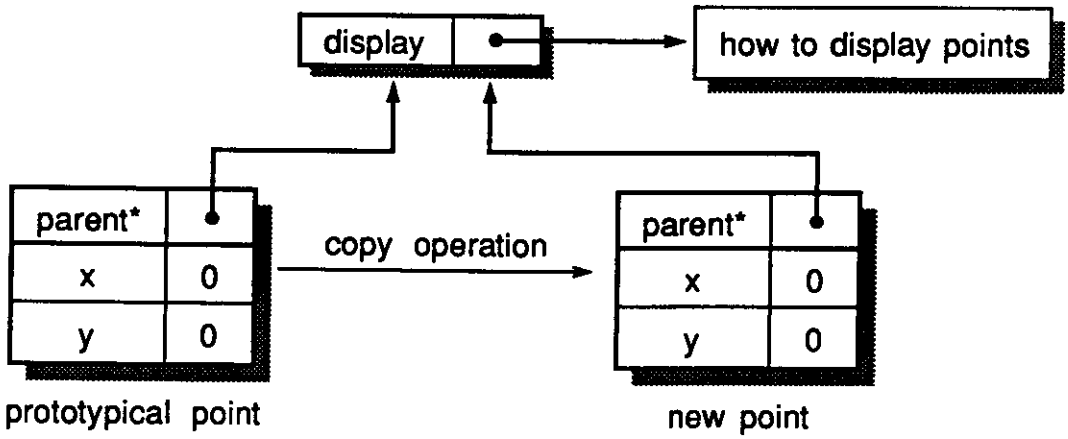
† Since a class is a full-fledged object, it can itself contain instance variables, as specified by the metaclass. Unlike the case with class variables, only the class, and not its instances, can access its class instance variables. The existence of class instance variables in addition to class variables is another complication engendered by the use of metaclasses.

3.2.1.3. Solving the metaclass problem by simplifying classes

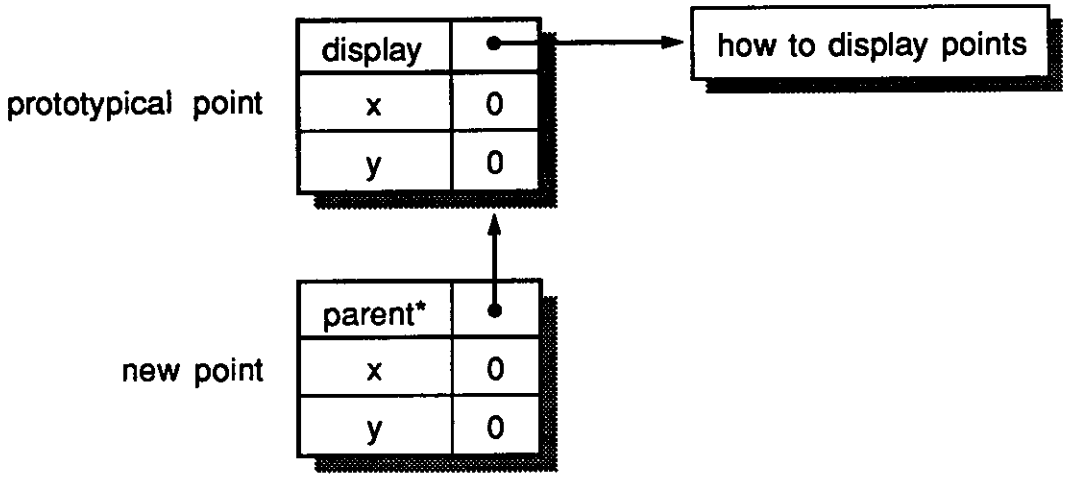
To reduce the learning problems posed by metaclasses, Alan Borning and Tim O'Shea [BoO87] proposed a simplification of Smalltalk (called Deltatalk) that eliminated metaclasses: classes would all belong to the single class `Class`, which would be an instance of itself. The price of that simplification was a reduction in functionality. Since all classes shared one `new` procedure (defined in `Class`), classes could not specify how instance variables should be initialized. Their simplification also left no convenient place to define procedures accessing or initializing data common to all instances of a class.

3.2.1.4. Prototypes: a simple solution to the metaclass problem

Another approach to the metaclass problem replaces the abstract concept of classes with the more concrete notion of prototypes. That approach is the one taken by SELF. (Others exploring the use of prototypes include Borning [Bor81, Bor86], Smith [Smi86], and Henry Lieberman [Lie86].) In such a prototype-based system, objects specify their own format and behavior, and objects may inherit state or behavior from any other object. A new object is created by copying another object (both state and behavior); the object so copied is termed a *prototype* for the new object:



In some prototype-based systems, like ObjectLisp [Dre85] and Borning's proposed prototype-based language [Bor86], certain objects are distinguished as prototypes, and new objects are created as children of their corresponding prototypes:



That approach allows changes in a prototype's behavior to be immediately reflected in all objects created from it, but makes the prototype special and akin to a class. In SELF, any object may be used as a prototype; the programmer is responsible for explicitly factoring common behavior into a parent shared by all the copied objects.

Prototypes provide a simple, easily understandable object model that supports the initialization of individual and shared state, functionality that was eliminated by the simplified class scheme. Instantiation by copying guarantees that the new object will be initialized with the same instance values as its prototype, while object-based inheritance allows state and behavior shared by a group of objects to be placed in a shared ancestor, which may respond to messages just as any other object does.

3.2.1.5. The benefits of concreteness

In addition to solving the metaclass problem, the prototype model may more closely match the way in which people acquire knowledge. Lieberman [Lie86] argued

that a class-based system forces the programmer to think in terms of abstract general principles first, while a prototype-based system allows the programmer to create individual concepts first and defer generalization until later in the programming process. The concreteness of the prototype approach seems to correspond more closely with human knowledge acquisition, which deals with specific examples first and generalizes from them later. The differences between using concrete prototypes and abstract classes may be most significant for areas like rapid prototyping, where programming with classes may force the programmer to classify prematurely, resulting in an object hierarchy that is more rigid and harder to make changes in. For that reason, prototypes may be more appropriate than classes for rapid prototyping.

3.2.1.6. The benefits of object-based inheritance

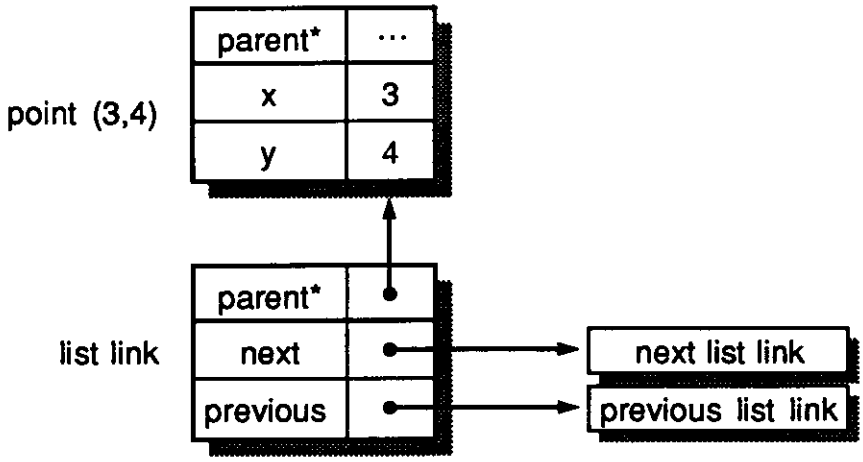
Prototypes support a more dynamic object organization than classes. Gul Agha and Carl Hewitt [AgH87], in a discussion on actors also relevant for prototypes†, note that objects in a class-based system are permanent members of their classes because the behavior of an object is fixed for its entire lifetime to be the same as for all other instances of its class. In systems with object-based inheritance, on the other hand, an object may change its inheritance and hence its behavior at any time without requiring a similar change in any other object. Dynamically reconfigurable hierarchies may therefore be better expressed with prototypes than with classes.

Another benefit of object-based inheritance is the ability to inherit state as well as behavior. The ability to inherit state enables patterns of sharing that are difficult to express in conventional class-based systems. As one example, a linked list of

---

† Prototype-based systems are similar in some respects to actor systems: objects are classless, and objects may inherit behavior from other objects. Actors, however, are also active objects akin to processes, supporting fine-grained concurrency.

points could be implemented using a link object that inherits the state of its point data:



The result is a linked list whose link nodes can be directly used by any procedure that expects to operate on points. Later in this chapter, other examples of object-based inheritance will be used to illustrate specific aspects of SELF semantics.

3.2.1.7. Comparing classes and prototypes

SELF uses the prototype-based approach because its advantages—simplicity, dynamism, and flexibility—are especially important for the rapid-prototyping domain to which SELF is targeted. The flexibility and dynamism of prototypes are due in great part to the benefits of object-based inheritance, which allows objects to inherit state from other objects and to change their inheritance dynamically.

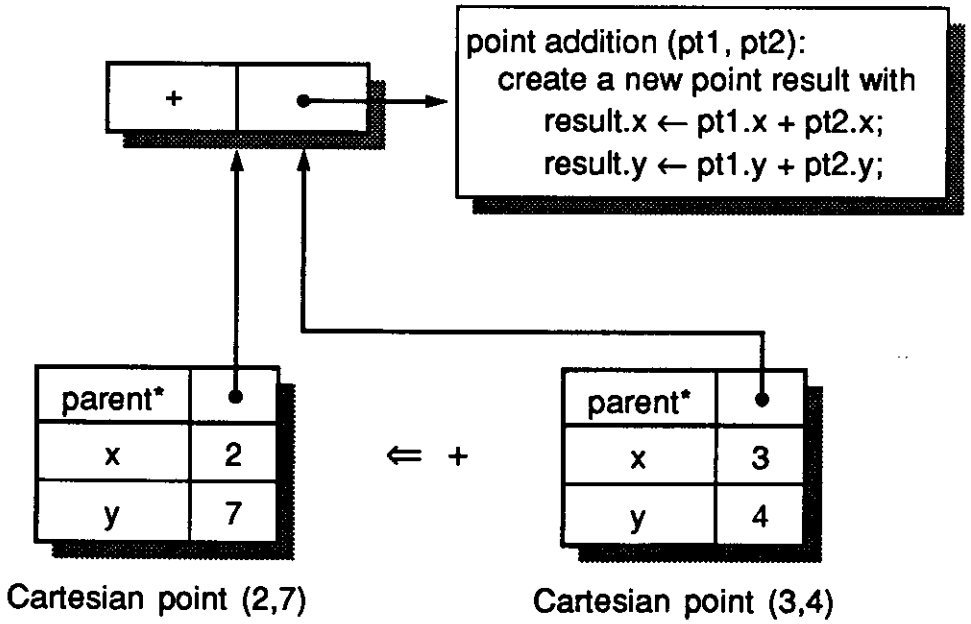
Prototypes do not provide all the benefits of classes, however. Classes can be useful in structuring large programs, where the need for order and maintainability may be greater than the need for flexibility. Some systems attempt to provide the advantages of both classes and prototypes. Actra [LTP86, LaL86] adds the structural clarity of a class-based specification hierarchy to a prototype-based substrate. Hybrid [Ste87] adds flexibility to classes by requiring instances to adhere only to the minimal template specified by their classes; an instance may carry

attributes in addition to those required by its class. Mixed systems like Hybrid and Actra provide a compromise between the structure of classes and the simplicity and flexibility of prototypes.

3.2.2. Unifying state and behavior

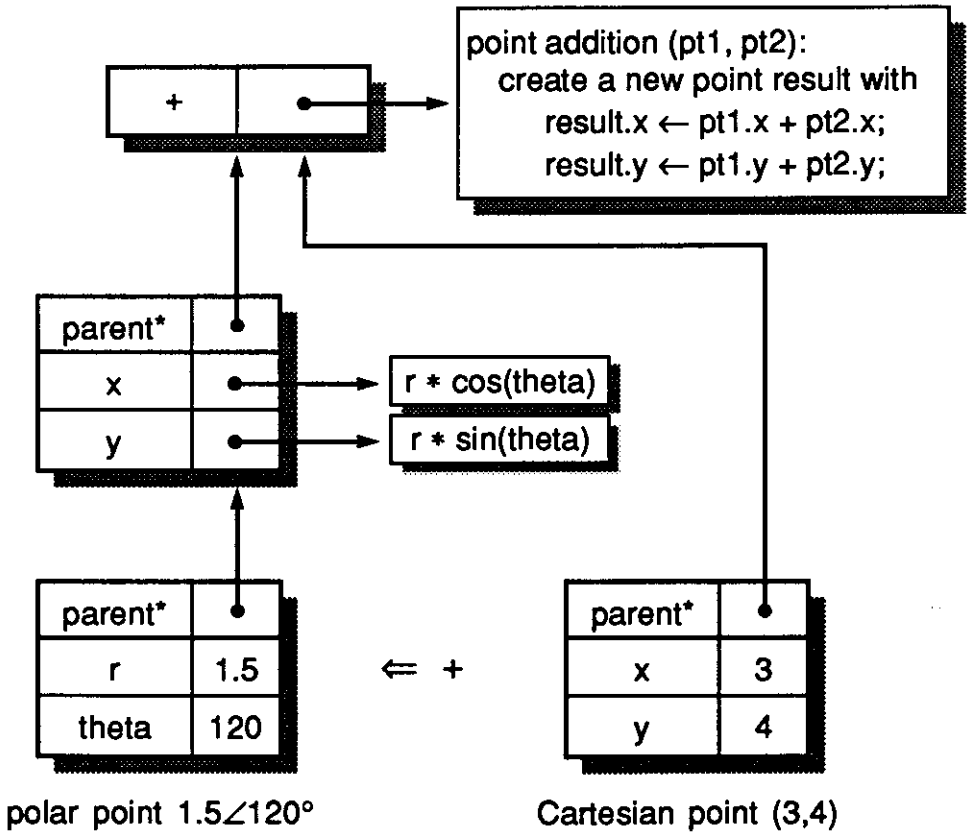
Aside from its use of prototypes, SELF may be principally characterized by its unification of state and behavior. No distinction is made in SELF code between variable access and procedure invocation; both are accomplished through the same message sending mechanism. Using a single mechanism for both variables and procedures promotes code reusability because a given message send in a routine may be either a state access or a procedure call in different circumstances.

Consider an addition routine for points, which produces a new point whose x- and y-coordinates are the sums of the corresponding coordinates of the arguments:



The pseudo-code syntax "`pt1.x`" is used to denote the operation retrieving the x-coordinate of the argument `pt1`. In conventional programming languages, that

operation would be a hard-coded data-field access; the point addition code would work correctly only for Cartesian points containing `x` and `y` data fields. On the other hand, if the `pt1.x` operation is treated as sending the `x` message to `pt1`, then the point addition code no longer depends on the format of its arguments. The addition procedure requires only that its arguments must return the values of their Cartesian coordinates when sent the `x` and `y` messages; in particular, it does not depend on whether the arguments respond to those messages with state accesses (for Cartesian points) or with procedure calls calculating the proper coordinate values (for polar points):



Using message-passing as the underlying mechanism behind both state accesses and procedure invocations allows a single point addition routine to operate correctly for both Cartesian and polar point arguments.

Unifying state and behavior also enables a programmer to change the binding of a message between a variable access and a procedure call without changing code in all the routines sending the message. Hiding the implementation of the message from its senders makes changing that implementation a much less formidable task.

A similar effect may be achieved in conventional object-oriented programming languages by using a convention that all access to variables occur through state access functions, which would be the only places directly accessing the variables. While following such a convention would also promote code reusability, the possibility would remain that someone would choose to flout the convention and embed direct variable accesses in his code. The SELF approach of eliminating the distinction between variables and procedures in the language ensures that all code may be easily reused for different implementations of messages.

3.3. SELF semantics

This section describes the semantics of SELF. It is intended to be an introduction rather than a reference manual, so language features will be introduced informally and explained through detailed examples. The presentation generally eschews matters of syntax. Some of the examples do, however, involve written SELF code. In general, SELF is similar to Smalltalk in syntax, so readers familiar with Smalltalk should be able to follow the examples with little difficulty. For those who are not conversant with Smalltalk, or who desire more precise syntax definitions, appendix A describes SELF syntax in detail.

Familiarity with Smalltalk terminology will also be helpful. In particular, the term *method* will be used to refer to an executable object invoked to handle a message send. Most of the time, "method" is synonymous with "procedure." Since SELF

unifies variables and procedures, however, a method invocation could produce a variable access rather than a procedure call.

3.3.1. Objects in SELF

Objects are the building blocks out of which SELF programs are formed. An object in SELF is similar to an object in other object-oriented programming languages, in that it embodies state and behavior in a single entity. The concept of an object in SELF, however, also incorporates the idea of a function: each object has code that is executed when the object is evaluated. More specifically, a SELF object may consist of a set of named slots containing object references (termed *oops* for historical reasons†), SELF code to be executed when the object is evaluated, an indexable array of object references, and an indexable array of bytes. Those elements need not all be present in an object; an object without explicitly specified code, however, is treated as if its code were a constant function that returned the object itself upon evaluation. As a result, every SELF object may be regarded as a function.

3.3.1.1. Slots

Slots provide the basic mechanism by which both function invocation and state access are accomplished. Slots are essentially object holders. A slot is characterized by a name and a value (the contents of the slot). Message lookups use the slot name to decide if a looked-up message matches the slot. Message semantics will be considered in more detail in section 3.3.2; for now, the important point is that when a message matches a slot, the object contained in that slot is evaluated. That behavior allows SELF slots to play the roles taken by functions and variables in other languages.

---

† Oop stands for "object-oriented pointer."

The contents of a slot determine whether it behaves like a function or a variable:

SLOT CONTENTS	SLOT ROLE	SLOT EVALUATION SEMANTICS
object without code (static)	variable	returns contents of slot
object with code (dynamic)	function	function call using slot contents as a prototype method activation record

A variable is obtained by using a slot containing an object that does not bear explicitly specified SELF code; such objects will be referred to as *static* objects. A message send matching the slot returns the result of evaluating the slot contents. Since a static object behaves like a constant function, the evaluation returns the static object itself. The message send in this case is equivalent to a state access, where the slot plays the role of a variable.

A function is obtained by using a slot containing an object bearing explicitly specified code; such objects will be referred to as *dynamic* objects. A dynamic object acts like a cross between a prototype method activation record and a function definition, with its slots defining the arguments and local variables and its code representing the body of the function. A message send matching a slot containing a dynamic object causes the object's code to be evaluated in a new method activation context. The dynamic object's code can itself send further messages upon evaluation. The result of the last of those messages is returned as the result of evaluating the dynamic object. A message send that matches a slot containing a dynamic object is therefore the equivalent of a function call. Section 3.3.3 will present an example explaining method activation in more detail.

Besides the general roles of variable and procedure, slots are used for a number of specialized roles. *Parent slots* indicate where to inherit operations that are not

locally defined in the containing object. Aside from their special treatment during message lookup, however, parent slots are the same as any other slot; in particular, they share the same name space as normal slots and will behave like any other slot when a matching message is received.

Argument slots are initialized with the appropriate message arguments when their containing object is evaluated. SELF currently does not allow a user-definable slot to be both a parent and an argument slot, although some automatically defined slots (such as `self`) do carry both parent and argument attributes.

Assignment slots contain the assignment primitive, which is used to store values into slots. The assignment primitive is an object with one argument slot. When the primitive is invoked, the message argument is stored into the slot whose name is the same as that of the assignment slot excluding a trailing colon; the receiver is then returned as the result. Assignment slots must reside in the same object as their corresponding data slots. Slots without corresponding assignment slots are considered to be read-only and serve the same purpose as named constants in other languages. No argument slot may have an assignment slot; all arguments in SELF are read-only.

The term "assignment slot" is something of a misnomer, since the assignment slot is not a special type of slot. Unlike parent or argument slots, which are distinguished by having the parent or argument slot attributes, an assignment slot is simply a normal slot that contains the assignment primitive. Although slightly misleading, the term "assignment slot" is used as a convenience because such slots serve a common and useful function in SELF.

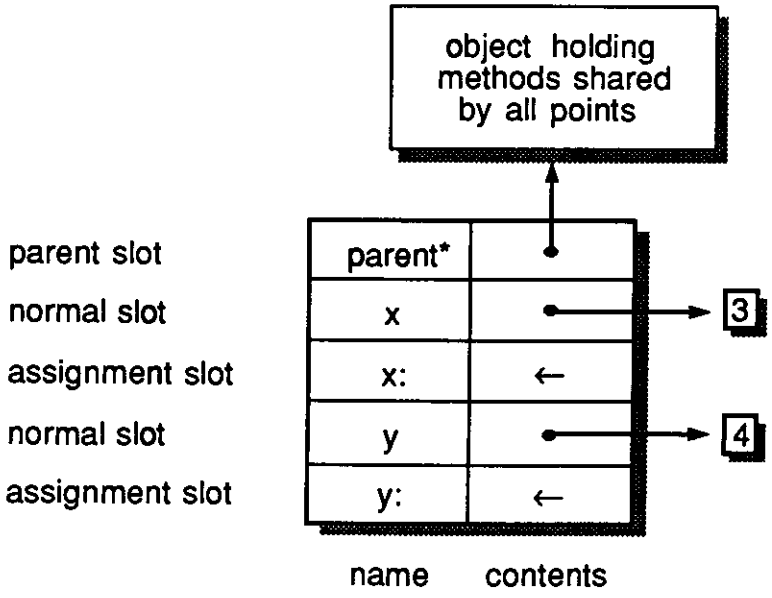
3.3.1.2. Indexed access

While slots provide named access to state or behavior, they do not support the indexed access required for arrays. To support array-like behavior, a SELF object may contain a zero-origin, one-dimensional array. Such an array is variable-length and dynamically sizeable; in particular, objects without arrays are essentially objects with arrays of length zero, which may be grown to non-zero size at any time. Both array access and resizing are accomplished via primitive operations supplied by the implementation. Both arrays of oops and bytes are supported; which type is used depends on which primitive operations are used to create and access the arrays.

Some aspects of SELF arrays may change in the future. Although SELF objects may currently contain both an oop array and a byte array, that capability may be eliminated. The ability to resize primitive arrays dynamically may also be removed; dynamically resizable arrays could still be implemented as a user-level, non-primitive data type built using the fixed-size primitive arrays.

3.3.1.3. An example SELF object

Consider a possible representation for a Cartesian point in SELF:



The object has five slots: normal slots named "x" and "y" holding the x- and y-coordinates for the point, associated assignment slots "x:" and "y:" containing the assignment primitive (indicated by a left arrow), and a parent slot (denoted by an asterisk) named "parent" indicating that methods for the point may be inherited from a parent object shared by all points. The point contains neither an array nor explicitly specified code. It functions as a simple static object with two assignable fields.

Two aspects of this example deserve further mention. First, the parent slot is named "parent" merely as a convention; parent slots may be named with any legal slot name. Second, the example illustrates a common and useful idiom in SELF programs: methods common to a number of objects are placed into a parent shared by those objects. The shared parent plays a role similar to a class in storing common operations but differs in not dictating format. SELF objects playing such a role are termed *traits* objects.

3.3.1.4. Object creation

SELF is a prototype-based system, so new objects are obtained by *cloning* existing ones. In the point example, a new point would be obtained by cloning an existing point and assigning new values for the x and y slots; the parent slot would be unchanged. The cloning operation is a shallow copy, which copies embedded object references so that the new object's slots and oops array will contain the same objects as the old. For comparison, a deep copy would copy embedded objects whole, so that the new object will contain objects that are equal in value to but distinct in identity from the corresponding objects in the old one. SELF's shallow copying semantics provide a simple and computationally inexpensive model for object creation.

3.3.1.5. Specialized objects

Although all objects are formed of the same material (slots, indexed arrays, and code), certain types of objects occur frequently enough in specialized roles to merit distinct terminology and syntax. Some of those are conventional, such as integers, real numbers, and strings. Others are more particular to SELF:

- *Inner methods.* An inner method is a dynamic-object literal embedded in the body of a SELF method definition. It is similar to a parenthesized subexpression in a conventional language but differs by also introducing a new lexically nested scope.
- *Outer methods.* An outer method is a dynamic-object literal written as a slot initializer. Like a Smalltalk method, it is the equivalent of a function in conventional languages, with the exception that nested messages are scoped through the receiver and its parents.
- *Blocks.* A block is a SELF closure. The code enclosed in the block is not evaluated until the block receives a `value` message asking it to evaluate itself. A SELF block is similar to a Smalltalk block, but some subtle differences exist.

The semantics of inner methods, outer methods, and blocks will be described in more detail later in the chapter.

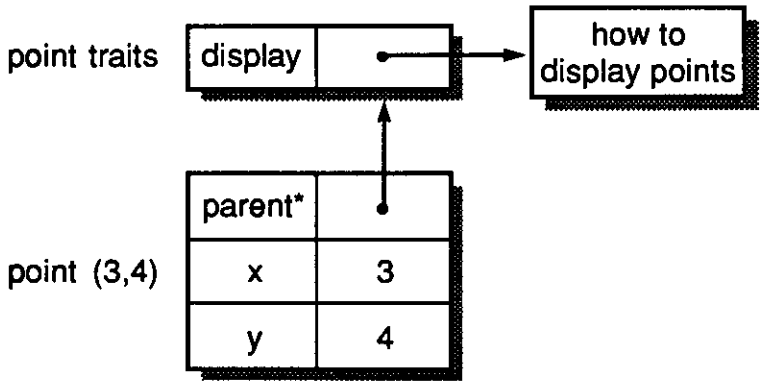
3.3.2. Messages

As in other object-oriented programming languages, computation in SELF results from sending messages to objects, which respond by performing some action and returning a value for each message. Unlike most other object-oriented programming languages, SELF extends the domain of message-passing to encompass all computation, including state retrieval and modification.

3.3.2.1. Responding to a message

When an object receives a message, it responds by looking for a local slot with the same name as the message. If a matching slot is found, the object contained there is evaluated and the result returned; otherwise, the search for a matching slot is continued recursively in the object's parents. In the simple case where there is only one parent (single inheritance), the recursive search may either find one matching slot or none. If one is found, then the object found in the slot is evaluated as before and the result returned. If no matching slot is found, then an error is indicated and computation aborts.

Consider a simple example involving messages sent to a point object:



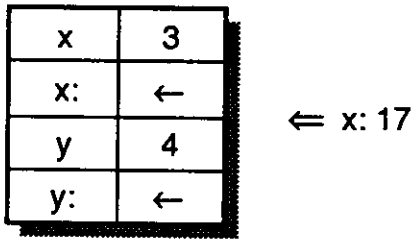
If an `x` message is sent to the point (3,4), a matching slot is found in the point itself, and the result of the message send is the integer object 3. If, on the other hand, a `display` message is sent, the local search for a matching slot fails, so the search continues in the point's parent (the point traits object), which holds operations shared by all points. The point's parent does contain a `display` slot, so the object contained there (the procedure for displaying points) is evaluated, displaying the receiver of the message.

Finally, if the `parent` message is sent to the point, the matching slot is found locally. Although the slot is a parent slot, it behaves just like any other slot that matches a message, so the object contained in the slot is evaluated and the result returned. Since the point traits object is static and does not contain explicitly specified code, the result of the evaluation is the traits object itself. Sending the `parent` message to the point (3,4) therefore produces the point's parent, the point traits object.

At first glance, the reason for the point traits object being static may be unclear. After all, it does appear to carry code—the code for displaying points. That code, however, is not part of the traits object itself but of a method object contained in its + slot. Since the point traits does not directly bear code, it is a static object. If the `parent` slot did contain a dynamic object, the code for that object would be evaluated in response to the `parent` message but would have no bearing on the inheritance semantics; searching a parent for a matching slot does not imply evaluation of that parent.† Parent slots therefore behave just like other slots except when they are used during a search for inherited operations.

3.3.2.2. Assignment messages

A second example demonstrates how assignment is accomplished in SELF:



† Evaluating parents on a message search was considered but rejected for increasing complexity with too little gain.

In this example, the point `(3, 4)` receives the message `x:` with the argument 17. The `x:` slot contains the assignment primitive, so the integer object 17 is stored into the point's `x` slot. The effect of the message send is to change the point's internal state to represent the point `(17, 4)`, which is then returned as the result of the assignment.

3.3.2.3. Primitive messages

As described so far, a message send may either access state—returning a static object or invoking the assignment primitive—or evaluate a dynamic object, resulting in further message sends. In order for anything other than state access to occur, a message send must eventually invoke a primitive operation, such as arithmetic or input/output. In SELF, messages whose names begin with an underscore ('_') are interpreted as primitive messages, which invoke low-level operations defined by the SELF virtual machine rather than operations found through the normal message search. For instance, sending the message `_IntAdd:IfFail:` performs primitive integer addition, invoking a user-specified error handler if the arguments are not integers.

3.3.2.4. Implicit-receiver messages

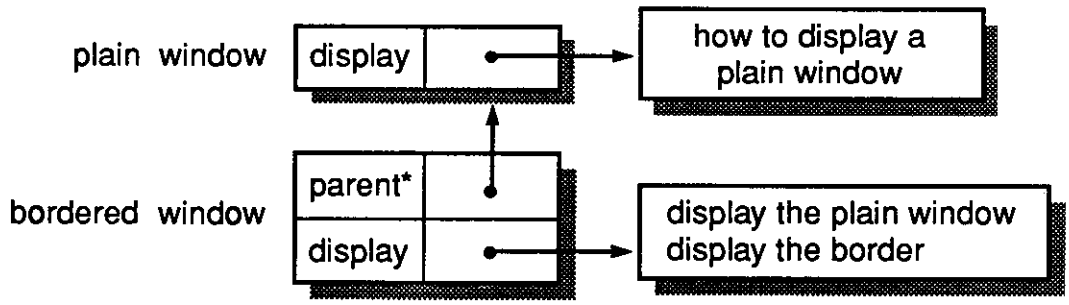
All the messages in the preceding discussion were sent to explicit receivers. A message may also be sent without an explicit receiver, in which case the receiver is implicitly assumed to be `self`, the receiver of the currently executing outer method. An implicit-receiver message differs from a message sent explicitly to `self` in beginning the method search at the currently executing method activation context rather than at the implicit receiver `self`. Starting at the current method activation context allows nested scoping to be implemented with inheritance. When the proper

method is found, the current receiver is retained as the receiver during execution of the newly found method.

3.3.2.5. Super messages

When a method search proceeds up a parent path, a matching slot will mask matching slots higher in the path. Although allowing an object to override a method defined by an ancestor is essential for incremental programming, the method initially responding to the message may at times itself wish to invoke the overridden method.

Consider a bordered window object, created as a child of a plain window:

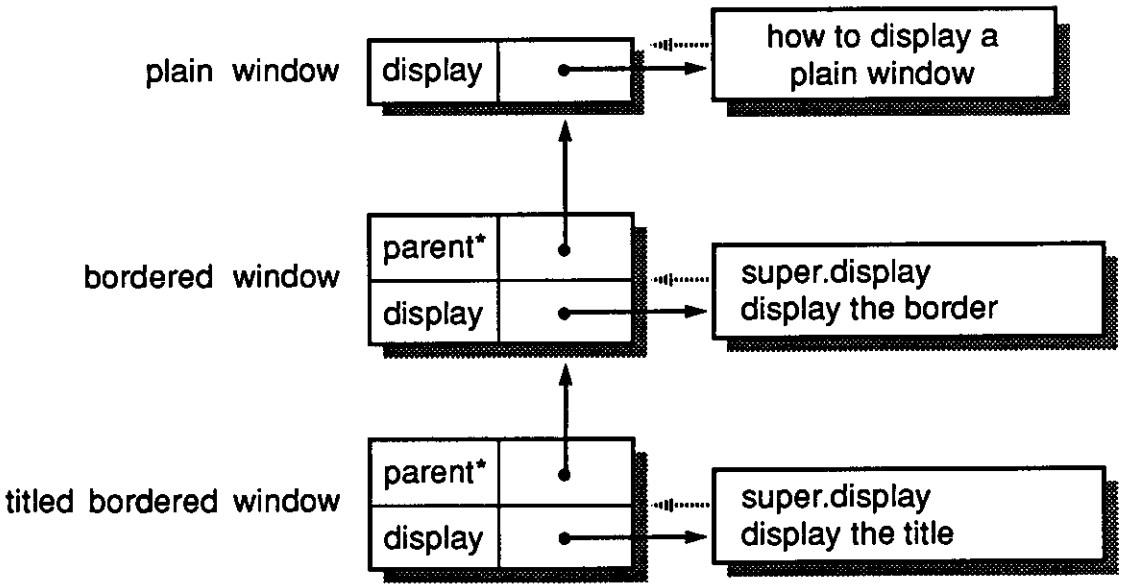


The bordered window should be displayed differently than the plain window, so it defines a local `display` method that first displays the body of the window and then draws in the window border. In order to display the window body, however, the bordered window `display` method needs to invoke the overridden `display` method. To do so, it sends a `super.display` message, indicating that the message lookup should find the overridden method in the parent. (The term `super` is adopted from Smalltalk, in which a message sent to `super` is looked up beginning in the superclass of the class containing the sending method.)

More precisely, in SELF a message send qualified as a super send is looked up starting at the parents of the send's *method holder*. The method holder is the object whose slot contains the outer method that sent the message. In the example above, the method holder for the `super.display` send is the bordered window object,

since that object holds the `display` slot containing the method with the `super.display` message send. The super send is therefore looked up starting at the single parent of the bordered window. (If the method holder has multiple parents, all the parents are searched, with conflicts resolved using the multiple inheritance rules discussed in section 3.3.6.)

The concept of a method holder is necessary to allow the starting point of the method search to be independent of the current receiver. Such independence enables creation of a chain of overriding methods, each invoking the method that it overrode:



A message sent to an explicit receiver cannot be a super send; only super sends of implicit-receiver messages are allowed. That restriction preserves object modularity because no message sent to an object from the outside can use the super mechanism to bypass the object's public interface and directly invoke an overridden method in one of its ancestors.

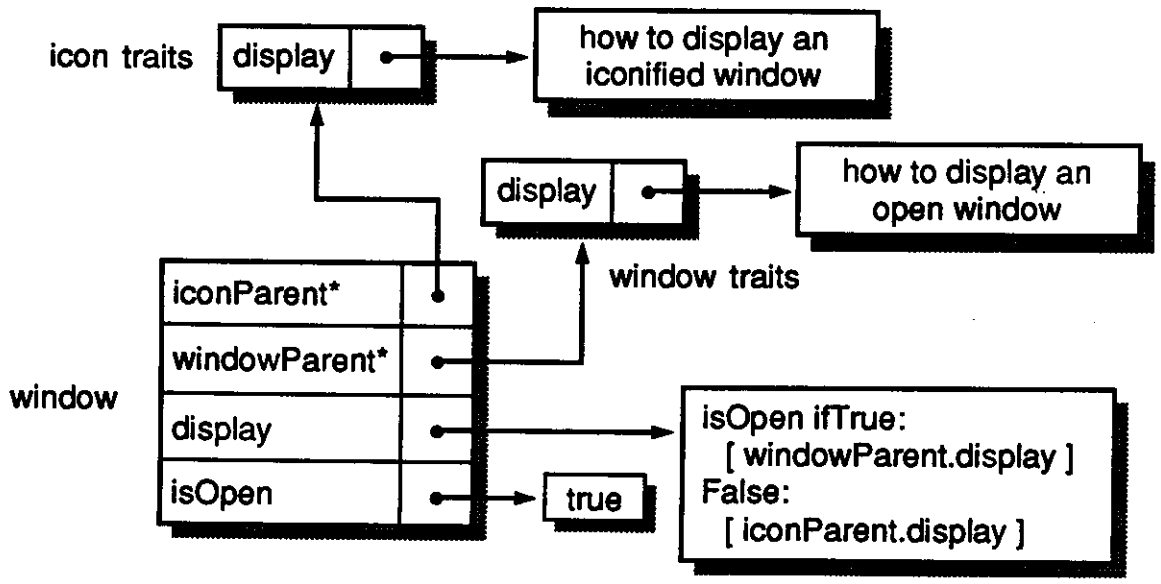
Like other implicit-receiver messages, a super send retains the current receiver `self` as its receiver. The method search for a subsequent implicit-receiver message

sent by the method handling the super send will proceed through the current receiver, which will have the opportunity to handle the message locally, possibly overriding a method defined in an ancestor.

3.3.2.6. Delegated messages

In some circumstances when the method holder has multiple parents, the super send, which searches all the parents for a matching slot, does not express the desired lookup behavior. Rather than search all the parents, the desired behavior may be to search a specific parent of the method holder. For that reason, SELF allows a message to be delegated to an explicitly specified parent of the method holder, so that the method search begins at that parent and ignores all the others. Like a super send, a delegated message must be sent to the implicit receiver `self`.

Consider the case of a window that may be either open or closed. If the window is open, it is displayed normally; otherwise, it is displayed as an icon. This behavior can be implemented using a window that inherits open and iconified window routines through different parents:



The window inherits routines from both the icon and window traits. (For brevity, only

the `display` slots of the icon and window traits are shown.) The window `display` routine checks to see if the window is open. If so, it delegates the `display` message to the `windowParent`, otherwise it delegates to the `iconParent`.

It is important to note that the delegation operation is not the same as simply resending the `display` message to either the `iconParent` or `windowParent`. If the message were simply resent, the receiver of the resent message would be either the `iconParent` or `windowParent`, with the result that implicit-receiver messages sent by the icon or window display routine would be looked up through the new receiver but not the window object, which was the receiver of the original `display` message. By delegating the `display` message instead, the message's receiver is the same as the current receiver, the window object. As a result, the window object will have a chance to intercept subsequent implicit-receiver messages before the method search reaches its parents.

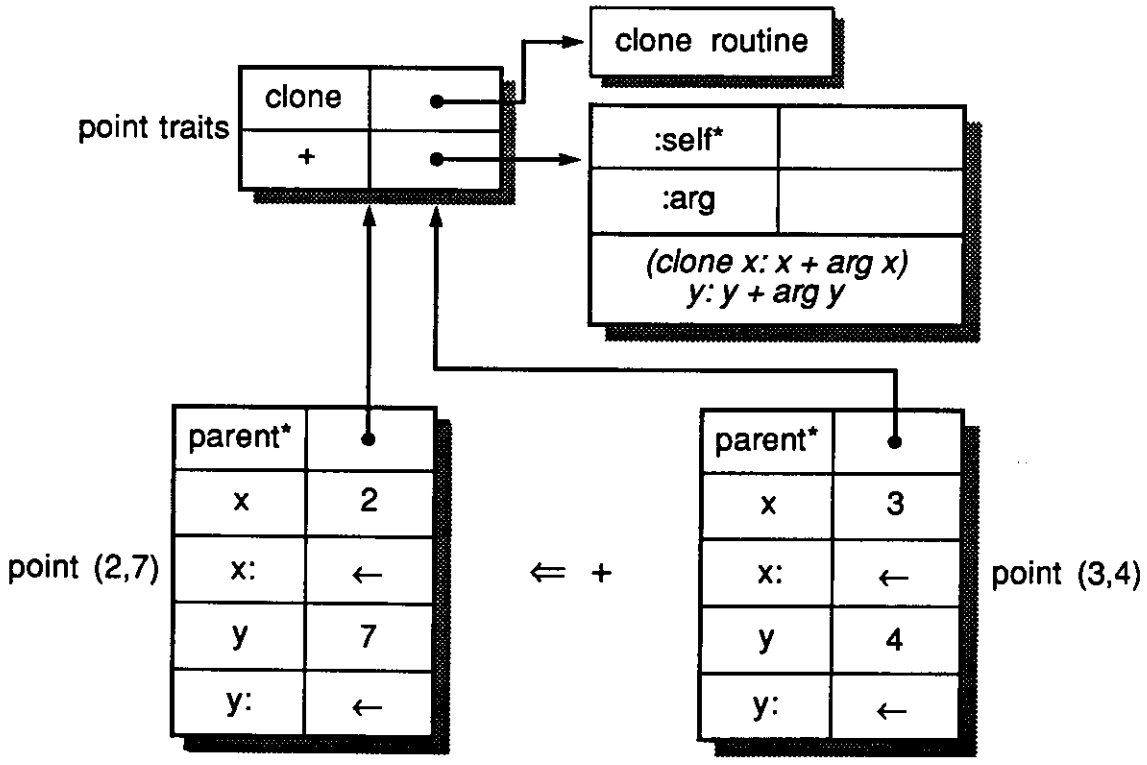
Another subtlety is that a message send is not used to find the object to delegate to. The named delegate slot must exist in the method holder; it cannot be inherited from an ancestor. Additionally, the contents of the delegate slot are not evaluated. If the delegate slot contains a dynamic object, then the method search will begin at that object itself, not at the result of evaluating the dynamic object.

3.3.3. Activation of outer methods: a detailed example

In SELF, methods are simply objects, and invoking an outer method is done by sending a message to an object, finding a matching slot, and evaluating the object found within. For a static object, evaluation is simple: the object itself is returned as the result. For a dynamic object, the situation is more complex, since method activation must support recursion and reentrancy, and it must also set up the proper binding environment for implicit-receiver messages.

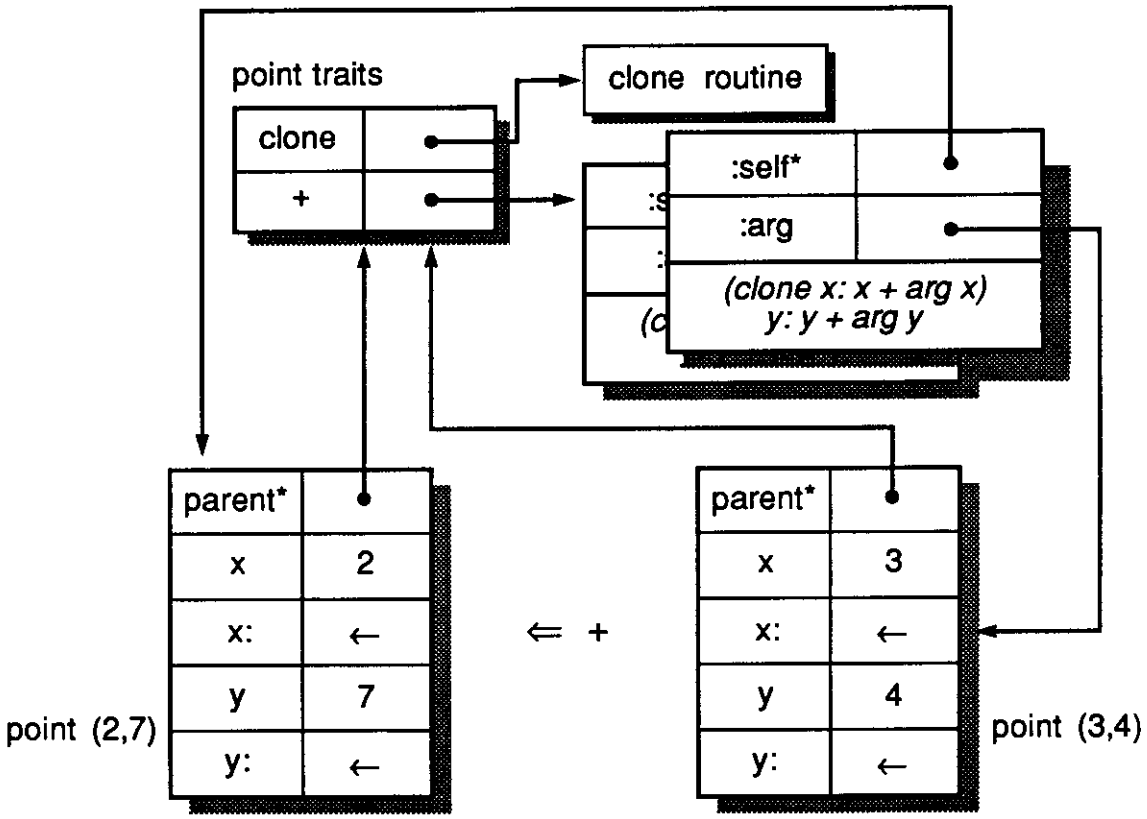
Consider what happens when an outer method is evaluated as a result of a message send. The outer method is first cloned, providing new versions of the argument slots and local slots in order to support reentrancy and recursion. Next, the argument slots of the clone are filled in with the message arguments, including the receiver as an implicit first argument named `self`. The `self` argument slot is also a parent slot, which extends the binding environment to include the receiver and its parents. Finally, after the environment has been set up, the code of the object is executed in the context of the clone, which causes the method search for a message sent to an implicit receiver to begin at the clone of the prototype activation object.

As a specific example, consider the case where the + message is sent to the point `(2,7)` with the point `(3,4)` as the single explicit argument. The situation just before method activation is:



The method search has found the matching slot for the + message in the parent of the receiver. The object contained there has two argument slots, one for the implicit

argument `self` and the other for the explicit argument `arg`; it also has explicit SELF code for evaluation. The contents of the argument slots are unspecified and irrelevant, as they would be replaced with the message arguments on method activation. After method activation, the situation becomes:

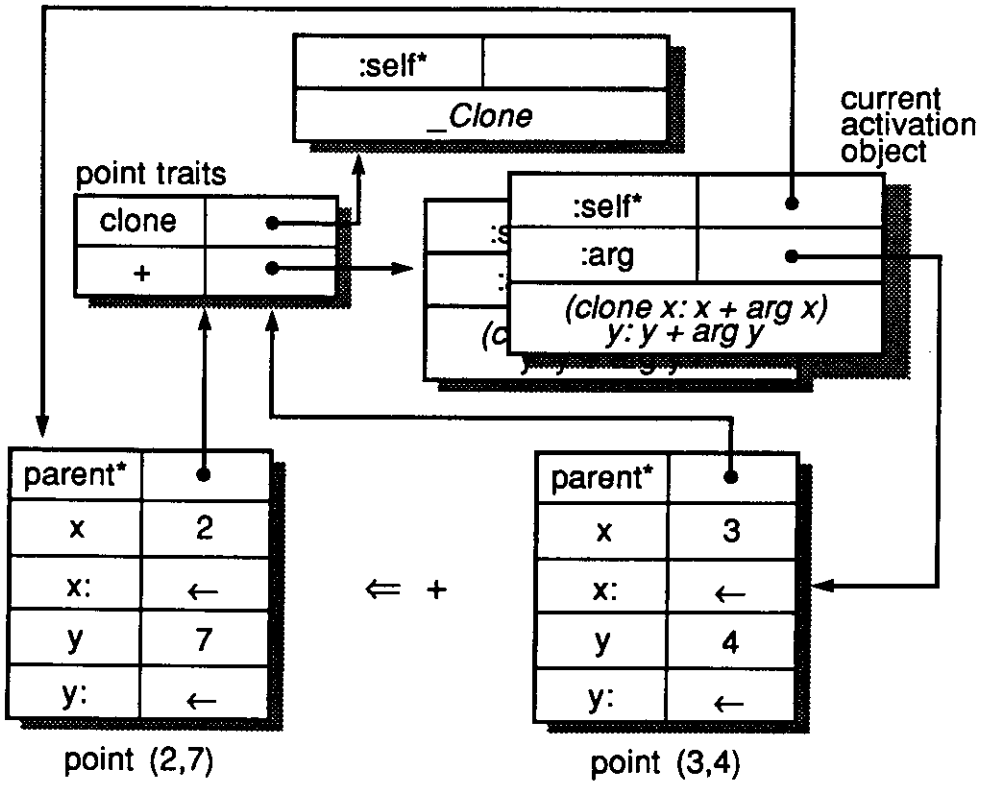


The + method object, which serves as a prototype activation record, has been cloned and the two argument slots of the clone filled in. The `self` argument slot has been initialized to refer to the point `(2,7)`, the receiver of the + message, and the `arg` slot to refer to the point `(3,4)`, the argument of the message. After the arguments are initialized, the code "`(clone x: x + arg x) y: y + arg y`" is executed in the context of the new method activation clone. That code creates a new point by cloning the receiver and then stores new values of x and y into the new point, the new values being the sums of the corresponding values in the receiver and argument.

3.3.4. Scoping through inheritance with implicit-receiver messages

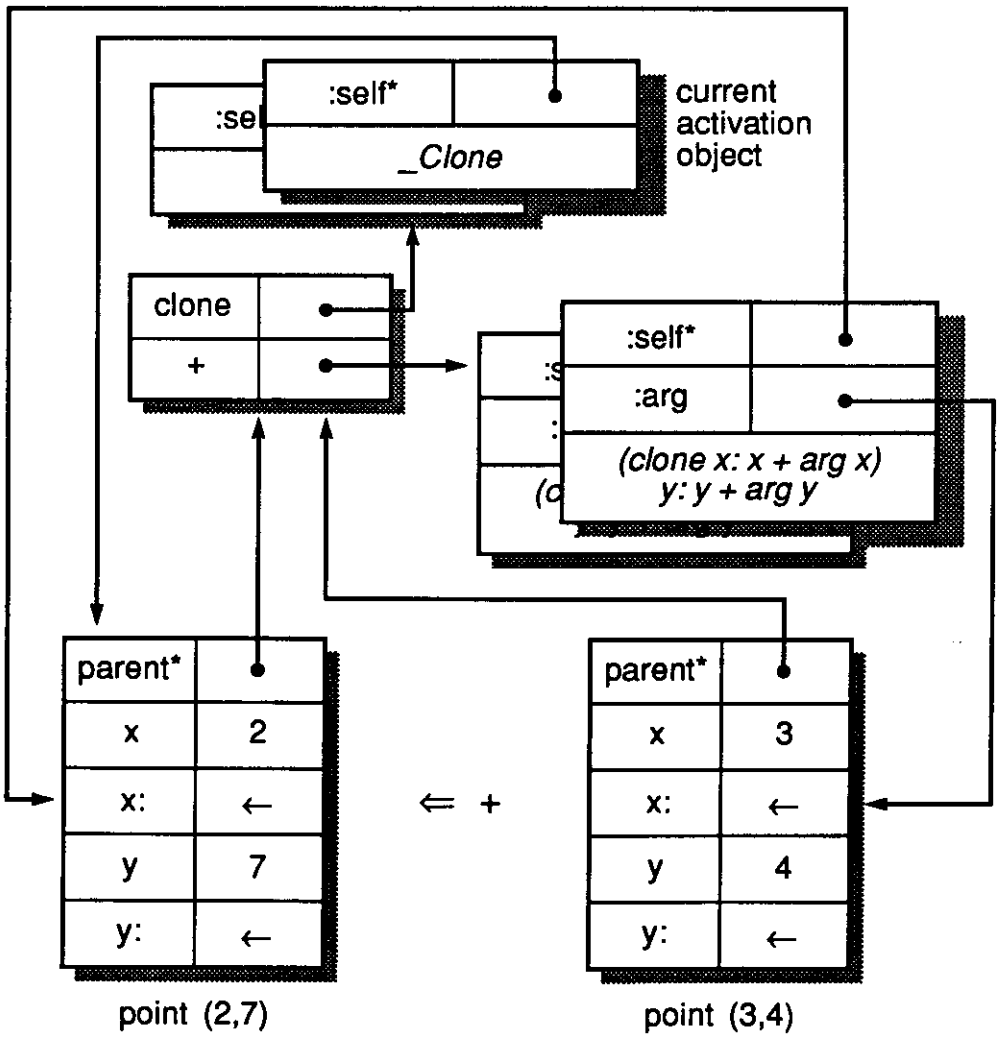
Passing the message's receiver implicitly in a parent argument slot allows the inheritance mechanism to handle scoping of implicit-receiver message sends. When a message is sent to an implicit receiver, the method search begins at the current execution context. If the current method activation context does not contain a slot matching the message, the method search continues in its parent, which for an outer method activation is the implicit parent `self`. When method found by the method search is invoked, the current receiver (which is contained in the `self` slot of the current activation context) is passed as the new `self` argument. As a result, the binding environment for non-local (free) implicit-receiver messages is the same for the original and newly invoked method contexts: the portion of the inheritance hierarchy accessible through the original receiver.

For a detailed example, consider again the case of the + method:



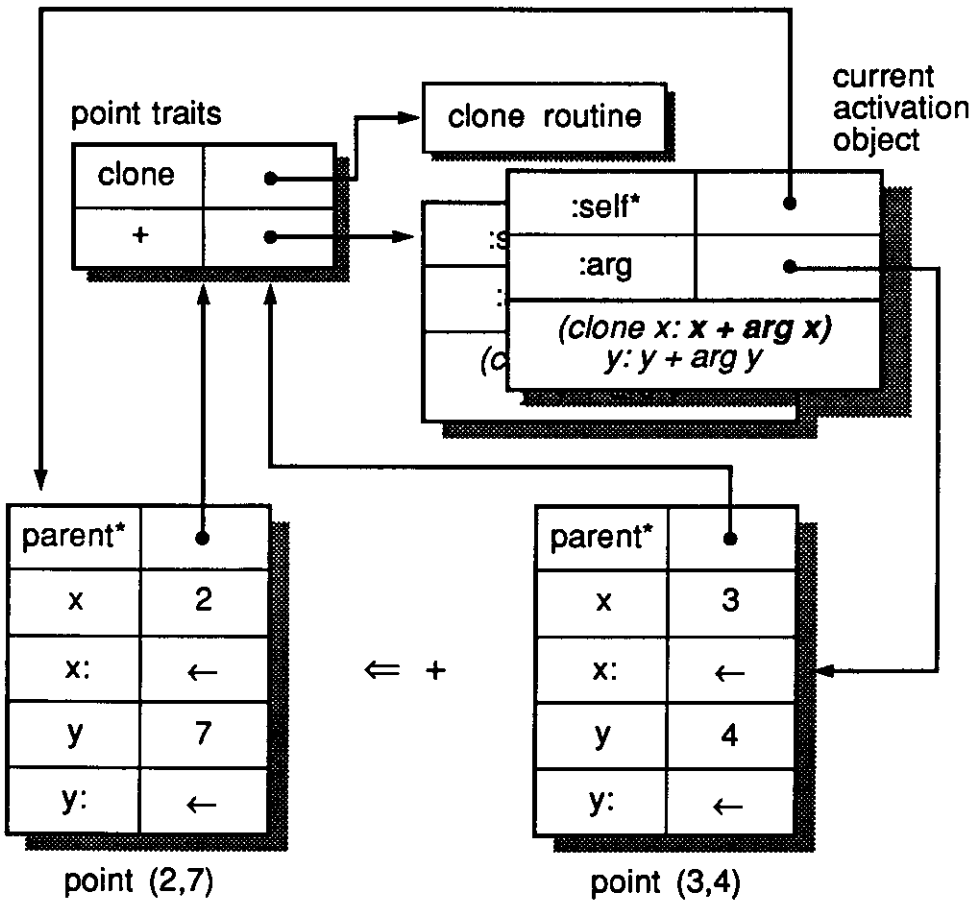
The first message sent in the + method is `clone`. The `clone` message does not have an explicit receiver, so the method search begins in the current activation object. As no matching slot is found there, the search proceeds through the `self` parent and continues in the receiver and its ancestors, ending when the matching slot is found in the point traits object.

The object found in the `clone` slot is cloned and the `self` argument slot initialized to the current receiver (the object found in the `self` slot of the current activation object), resulting in the following situation:



The `clone` method object is then evaluated. The activation object for the `clone` routine becomes the current execution context and its code executes, invoking the primitive method `_Clone` to perform the actual cloning of the receiver.

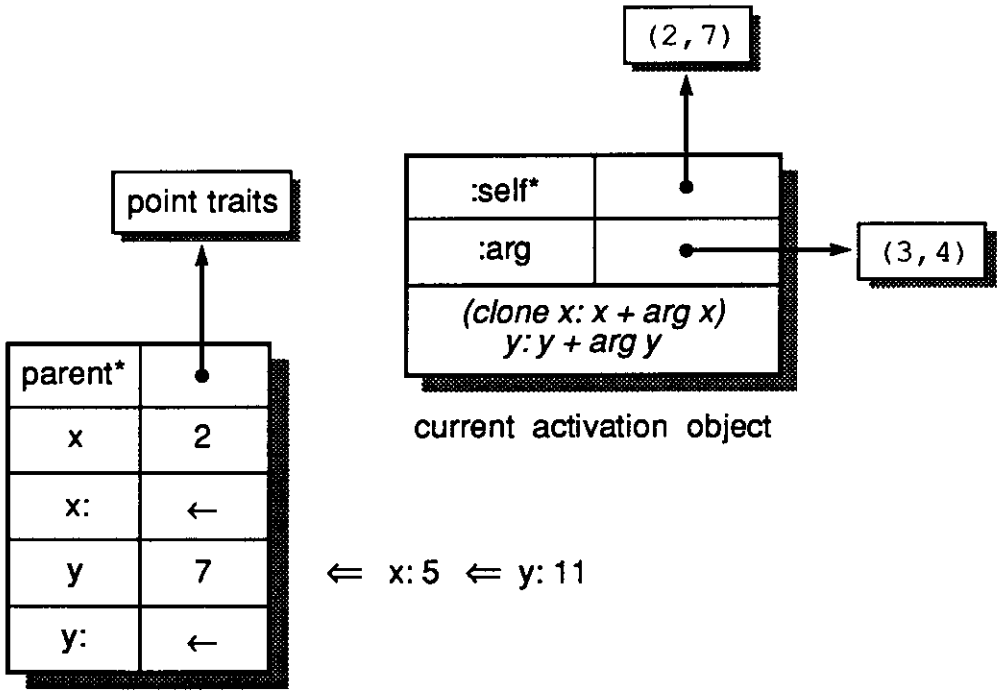
For an example combining message sends with both implicit and explicit receivers, consider how the code fragment "`x + arg x`" is handled:



The first `x` message is sent to an implicit receiver, so the method search begins in the current activation object, proceeds through the `self` parent, and finds the `x` slot in the point `(2,7)`. The resulting evaluation returns the integer `2`. The `arg` message also is an implicit-receiver send, so its method search begins in the current activation object. The matching slot is found there, and the point `(3,4)` is returned. That point is then the explicit receiver for the second `x` message, so its method search starts

there rather than in the current activation object. As a result, the second x message returns 3, as opposed to the integer 2 returned for the first x message. Finally, the + message is sent to 2 with the argument 3. Since an explicit receiver is involved, the message search begins at the integer 2, not the current activation object. As a result, the integer + method (which eventually invokes the primitive `_IntAdd:IfFail:`) is used rather than the point + method. The final result of "`x + arg x`" is hence `5`.

Similarly, the message sequence "`y + arg y`" returns 11. The situation is then:



The only remaining actions are sending the "`x: 5`" message to the new point and then the "`y: 11`" message to the result. The "`x: 5`" message is sent to an explicit receiver, the new point (cloned from the original receiver), so the method search begins there and finds the assignment slot for x. The effect of the send is to store 5 into the x slot of the point, which becomes `(5,7)`. The assignment primitive

returns the receiver, so the result of the message send is the point `(5,7)`. Finally, the message "`y: 11`" is sent to the result of the last send, storing `11` as the new y-coordinate and returning the point `(5,11)` as the final result of the point + method.

Two kinds of scoping may be discerned from the preceding examples. Messages sent to an explicit receiver are scoped through the receiver, as in a conventional object-oriented message send. Messages sent to an implicit receiver are first scoped lexically and then through the current receiver. Scoping through the current receiver is similar to the binding of messages sent to `self` in Smalltalk: the messages are sent to the dynamic receiver for the currently executing method. Since the method search begins in the current execution context, however, slots defined locally for the current method override those defined in the current receiver and its ancestors. The result is essentially lexical scoping.

3.3.5. Nested scopes

Like many other languages, SELF provides nested lexical scopes. As with top-level scoping, the semantics of nested scopes are described in terms of inheritance. When a nested scope is entered, a new SELF scope activation object is created as a child of the enclosing scope, and then execution proceeds in the context of the new activation object. Two types of nested scope are supported. *Inner methods* correspond to the nested scopes of conventional block-structured languages like Pascal or Algol. *Blocks* defer evaluation; they are closures in the lexical binding environment.

3.3.5.1. Inner methods

An inner method is written as a parenthesized expression with an optional slot list; it behaves like a method executed in line. When the expression is evaluated at run time, the inner method is cloned, and the parent of the clone is set to be the

current activation object, which represents the enclosing lexical scope. The clone then becomes the current activation object, and its code is executed.

For example, suppose we wish to calculate the probability density function for a normal distribution:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right]$$

The following SELF method implements the desired density function:

:self*	
:mu	
:sigma	
*((2 * pi) squareRoot * sigma) inverse * exp: -0.5 * (| t | t: - mu / sigma. t * t)*	

The function definition contains an embedded parenthesized subexpression that introduces a nested lexical scope:

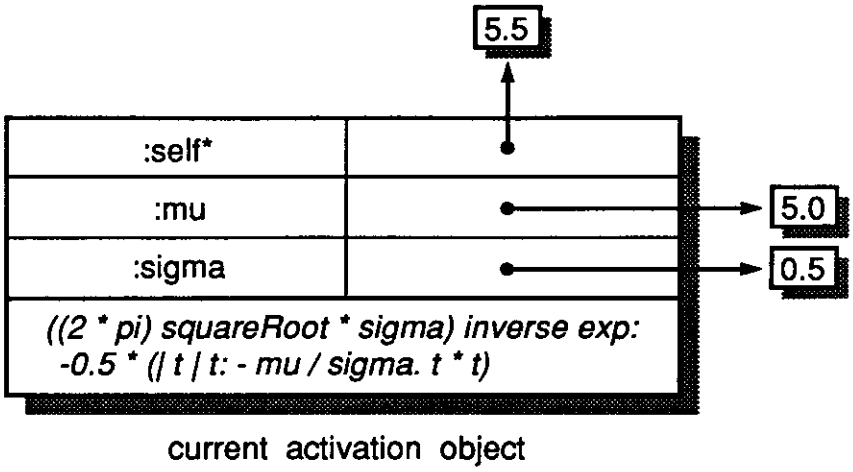
```
(| t | t: - mu / sigma. t * t)
```

The `t` slot defined in that scope is only accessible during the execution of the subexpression. Messages not defined in the innermost scope are lexically resolved. In the example, the `mu`, `sigma`, and implicit receiver (`self`) in the inner method refer to the argument slots of the outer method, so the indicated subexpression is equivalent to

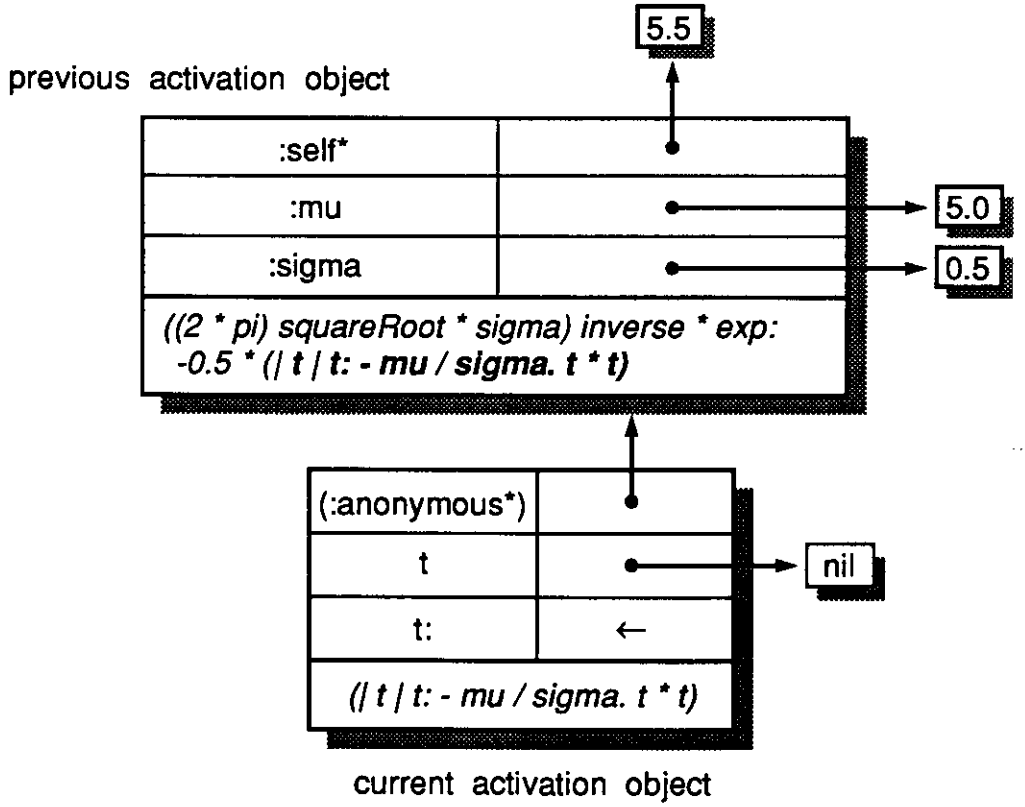
```
(self - mu / sigma) * (self - mu / sigma)
```

except that "`self - mu / sigma`" is only evaluated once.

Consider the case where the receiver is 5.5, `mu` is 5.0, and `sigma` is 0.5:



When the inner method (parenthesized subexpression) introducing the nested scope is evaluated, it is cloned, the clone becomes the current activation object, and the parent is set to be the enclosing lexical scope:



(The first slot of the inner method is parenthesized because it is anonymous and

inaccessible at the SELF language level.) Execution commences in the new activation object, which evaluates the expression

```
(| t | t: - mu / sigma. t * t)
```

The first statement subtracts `mu` (bound to 5.0) from `self` (bound to 5.5), divides the result by `sigma` (bound to 0.5), and stores that result (1.0) into `t`. The next statement multiplies `t` by itself, resulting in a final value of 1.0. That value is returned as the result of evaluating the inner method. Execution resumes in the context of the previous activation object, where the newly returned value (1.0) is used as the argument to the `exp:` message send (which raises the transcendental number *e* to the power of its argument).

The semantics for an inner method are similar to those for an outer method invoked by an implicit-receiver message send. The principal difference is that the parent scope is the lexically enclosing scope rather than the current receiver (contents of the `self` slot). Because no new `self` slot is defined, execution in the inner method uses the same receiver as the enclosing outer method.

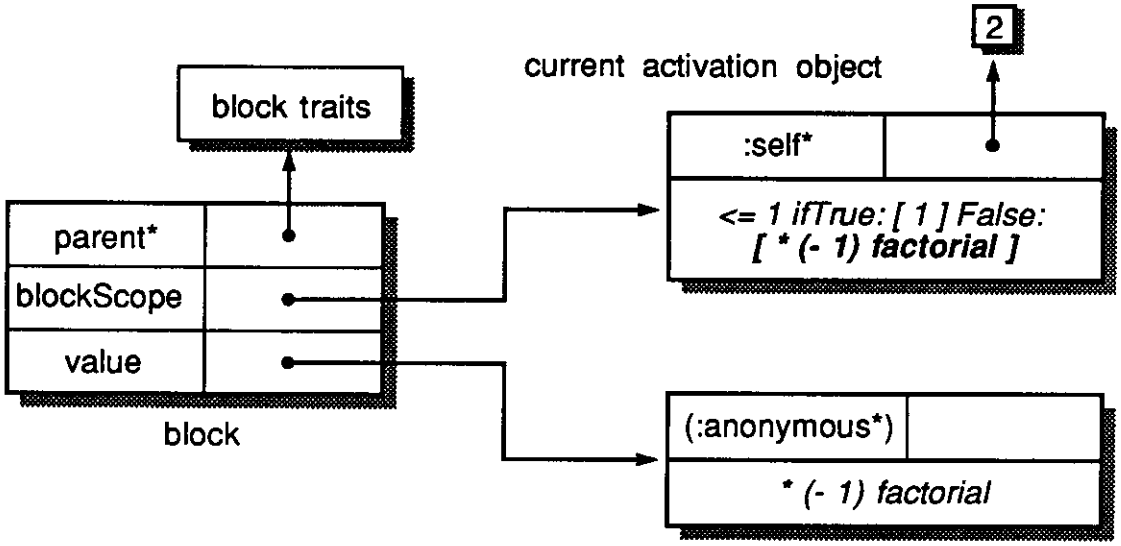
3.3.5.2. Blocks

A block is a SELF closure. It behaves like an inner method whose evaluation is deferred until explicitly invoked by receipt of a `value` message. A block is written similarly to an inner method except that the parentheses are replaced by square brackets ('[' and ']').

For example, the factorial function for non-negative integers might be written as

```
factorial = (
     <= 1 ifTrue: [ 1 ] False: [ * (- 1) factorial ]
)
```

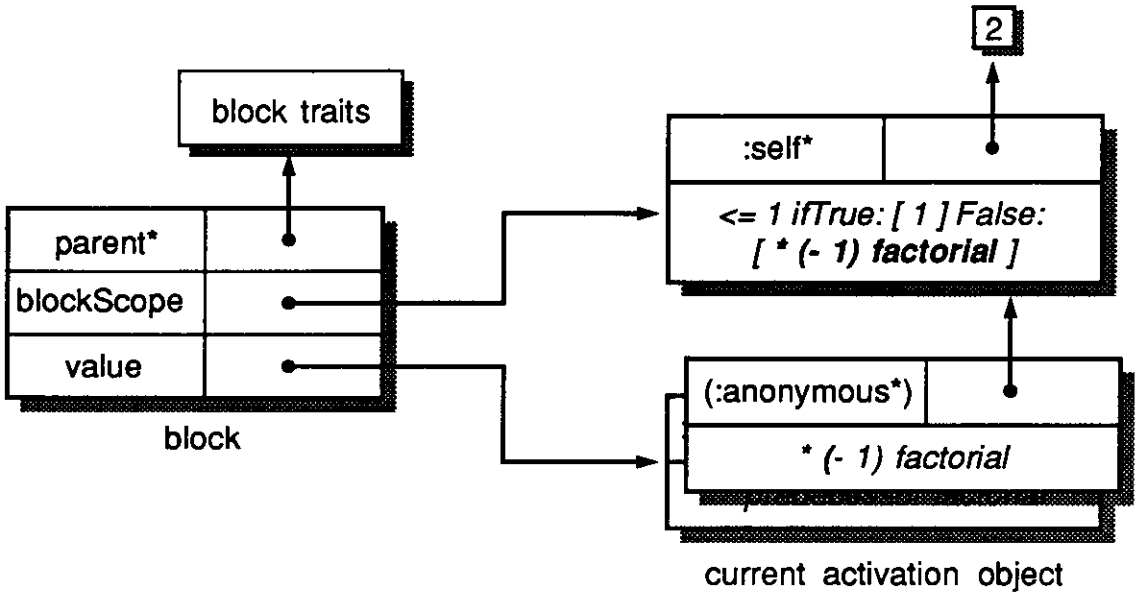
Consider invoking the `factorial` method on the integer `2`. What happens when the block literal is evaluated? The block is cloned, and the enclosing scope of the clone is set to the currently executing activation object (the `factorial` outer method):



The block is a static object, so the result of the block literal evaluation is the newly cloned block itself. The parent of the block is not the enclosing scope but the block traits object that contains the message protocol for all blocks, including such methods as `whileTrue:`, `loop`, and `untilTrue:`. As a result, messages sent to the block are not scoped through the enclosing lexical scope of the block; the block acts like an entirely independent static object.

When the block receives the `value` message, the method in the `value` slot is cloned and the argument is filled in, similarly to outer method activation. Unlike an outer method, the `value` method contains no `self` slot, and the implicit parent argument slot is filled in with the contents of the `blockScope` slot rather than with the receiver of the `value` message (which would be the block itself). The `value`

method then becomes the current activation object, and its code is executed:



Because the anonymous parent slot points to the original activation object, implicit-receiver messages in the block are lexically scoped. In example above, lexical scoping is only used to obtain the contents of the original receiver `2`, which is used as the receiver of the "`*`" and "`-`" implicit-receiver messages. More generally, lexically scoped local data slots may also be accessed from within a block:

```
printMin: i = (
     < i ifTrue: [ print ] False: [ i print ]
)
```

In this example, the `i` message embedded in the false block will be bound to the argument `i` in the lexically enclosing scope, no matter what the binding environment is when the block finally receives the `value` message.

Blocks may contain arguments which are filled in when the block is invoked. Instead of `value`, the invocation message for a block with one argument is `value:`,

for two is `value:With:`, for three is `value:With:With:`, and so forth. Since the block is cloned before execution, new copies of the local and argument slots are created for every send of `value` to the same block. As a result, SELF blocks are reentrant, unlike original Smalltalk-80 blocks.†

After a block `value` method has finished execution, the result of the last statement in the block is returned, and execution resumes from the point at which the `value` message was sent—not from the point of definition of the original block. For example, the `loop` method for blocks is defined as

```
loop = ( value. _Restart )
```

where the `_Restart` primitive restarts in place the lexically closest executing method. (The SELF compiler generates a branch back to the beginning of the method when the `_Restart` primitive is encountered.) Each time through the loop, the block (receiver of the `loop` message) is invoked by the `value` message; after the block completes execution, the `_Restart` primitive is executed.

3.3.5.3. Non-local return: the ^ operator

The ^ (non-local return) operator returns from the closest lexically enclosing top-level (non-nested) method. It takes an expression as an argument and returns the result of evaluating that expression. The ^ operator may only be written at the beginning of the last statement in a method's code body.

When used in an outer method, the ^ operator produces exactly the same behavior as simply falling off the end of the method. That is,

```
legalMethod = ( 3 + 4. 5 - 2. ^ 7 * 8 )
```

† Recent versions of Smalltalk-80 support reentrant blocks.

is equivalent to

```
legalMethod = ( 3 + 4. 5 - 2. 7 * 8 )
```

In both cases, the integer 56 is returned to the site from which the `legalMethod` message was sent.

The ^ operator is more useful when embedded in a nested scope (inner method or block). Consider the definition of `whileTrue:` for blocks:

```
whileTrue: body = (
     [ value ifFalse: [ ^ nil ]. body value ] loop
)
```

The receiver block (the boolean condition) is evaluated each time through the loop. If the condition is true, the loop body (b) is evaluated. If the condition is false, the nil object is returned from the lexically enclosing outer method (`whileTrue:`), terminating the loop.

Non-local return may also be used for inner methods, though with behavior that may be more curious than useful. For example, the expression

```
5 + (^ 3 + 4) * 7
```

returns 7 from the enclosing outer method.

SELF currently follows Smalltalk in allowing a top-level method activation to be returned from only once. Once an outer-method activation has been returned from (either by executing the last statement in the outer method or through a non-local return in a nested block or inner method), no nested blocks may subsequently execute a non-local return.† Smalltalk is currently more flexible than SELF on this point.

† Unlike Smalltalk, the current implementation of SELF does not check this constraint. Invoking a non-local return from a method activation that has already returned is likely to crash the implementation.

While the current implementation of SELF does not allow a block to be invoked once its enclosing outer method returns, Smalltalk allows blocks that do not perform non-local returns to be invoked even if their home (enclosing) method activations have already returned; such a non-LIFO block can access local variables in its (already returned) home method context. SELF block semantics in this area are unsettled. We are considering a wide range of possible extensions to current SELF block semantics: allowing invocation of non-LIFO blocks provided they do not perform non-local return or access state in their enclosing outer-method activations, allowing such non-LIFO blocks to access state in enclosing method activations that have already returned, and allowing non-LIFO blocks to perform non-local returns from outer-method activations that have already returned. The last solution would provide the full power of continuations, enabling the construction of complex control structures such as non-blind backtracking [HaF87].

3.3.6. Multiple inheritance

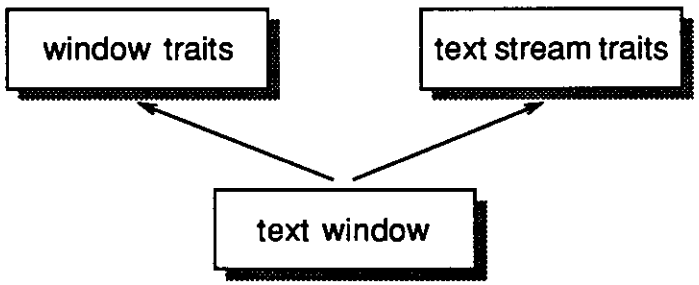
Multiple inheritance allows an object to inherit operations from multiple parents. It can be used both to combine unrelated classes of behavior (e.g., combining writer and artist traits to produce a cartoonist) and to mix in overriding behavior (e.g., overriding a comparison operation to use identity rather than value). Most languages support either one or the other of those uses. Languages that support combination of unrelated traits (e.g., Trellis/Owl [SCW85]) typically provide unordered inheritance, in which all parents (or superclasses) are treated equally; inheriting matching operations from multiple parents is considered to be an error. On the other hand, languages that support mix-ins (e.g., Flavors [Moo86] and CLOS [BDG88]) typically provide ordered inheritance, in which parents (or superclasses) are searched

in a fixed order, invoking the first matching operation is used. (Languages like Flavors and CLOS also allow multiple matching operations to be combined using some form of method combination.)

SELF supports both uses of multiple inheritance by providing both ordered and unordered inheritance. Parents are given priority levels which are indicated by the number of asterisks following their names. (Lower numbers of asterisks indicate higher priority.) Parents in higher priority levels override those in lower levels, and parents in the same priority level are treated equally.

3.3.6.1. Combining unrelated objects

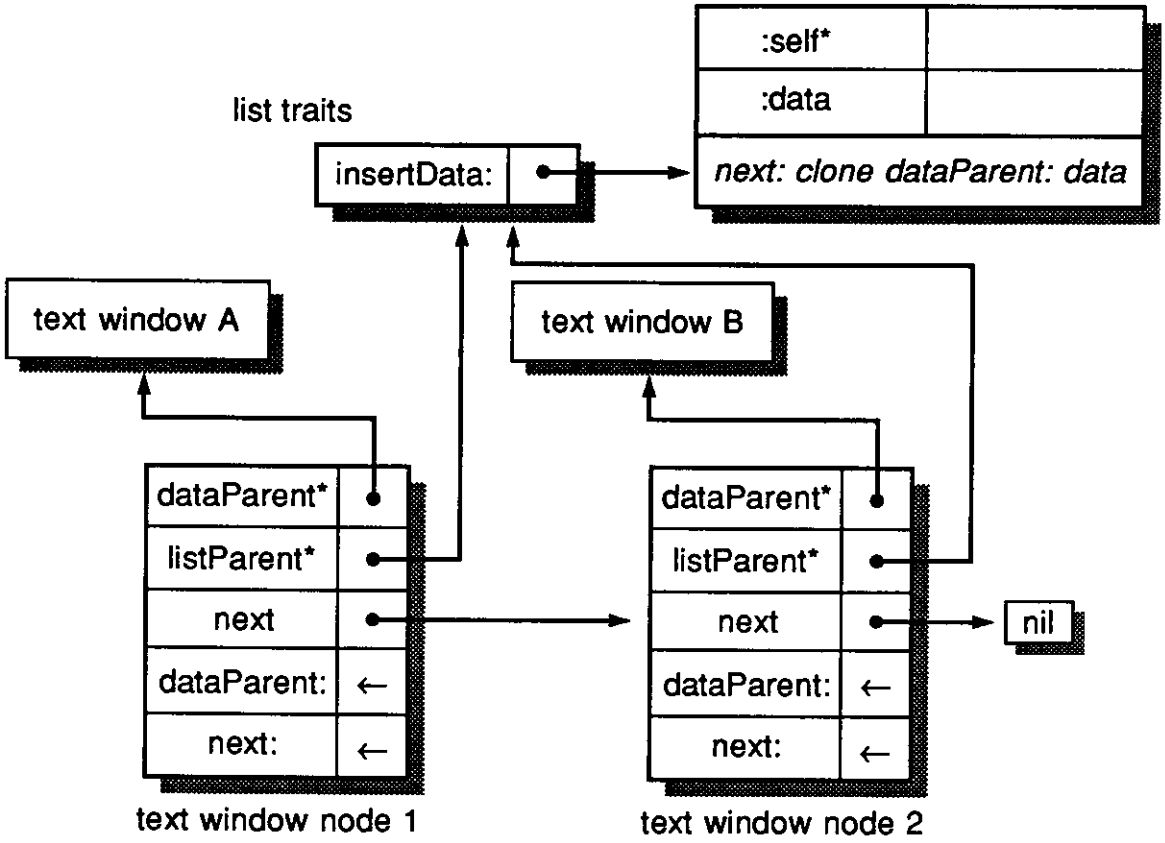
Unordered inheritance is useful for combining unrelated objects, for which any name conflicts are probably unintentional and should be flagged as errors. A fairly standard example (adapted from a paper by Daniel Halbert and Patrick O'Brien [HaO88]) is a text window (supporting stream input and output), which might be implemented by combining traits for plain windows and for abstract text streams:



The resulting text window would support all the operations of both windows and text streams.

The use of prototypes in SELF rather than classes increases the utility of multiple inheritance, since it can be used to combine objects holding state. Consider, for example, implementing a linked list of text windows. In SELF, a text window list node may be created by combining a full-fledged text window object (corresponding to a

text window instance in a class-based language) with a list node object:

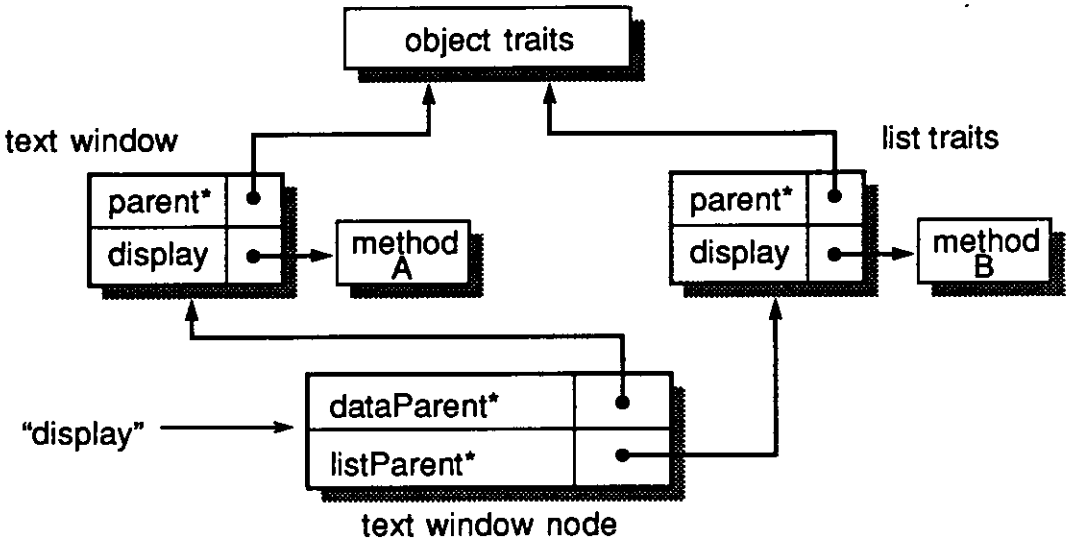


Such a combination list node inherits from both its data object and the list traits object. As indicated by the diagram, the list is maintained by a data insertion method that creates a new list node for a data object and inserts the new node into the list. The advantage of this approach is that each text window list node understands the message protocol of both text windows and list nodes and may be used in any place that expects one or the other.

3.3.6.2. Conflict handling in unordered inheritance

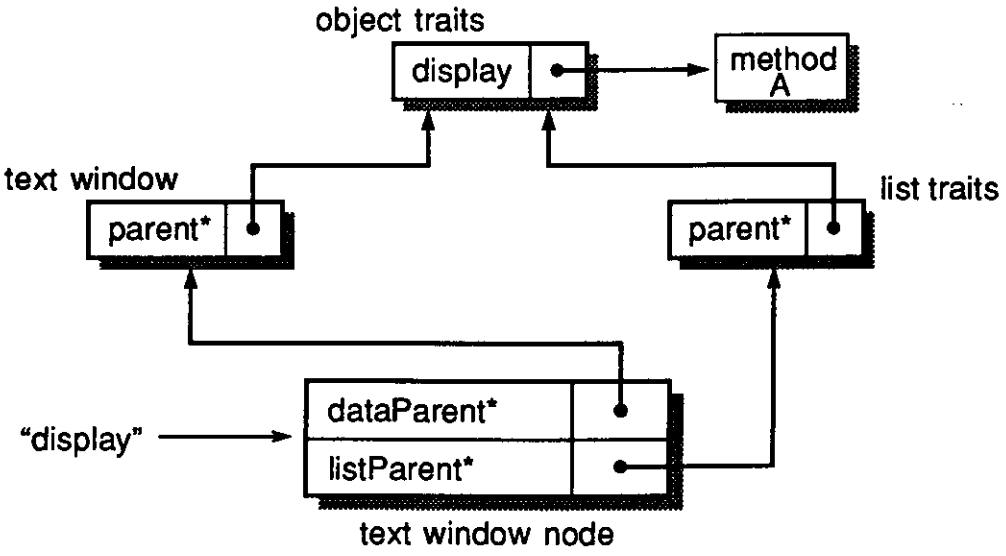
What happens when matching slots for a message are found along more than one parent path? In the normal case of unordered parents, inheriting different operations is regarded as an error and is flagged as such at run time when the message is received. Finding the same matching slot through different paths is allowed.

Consider the following situation:



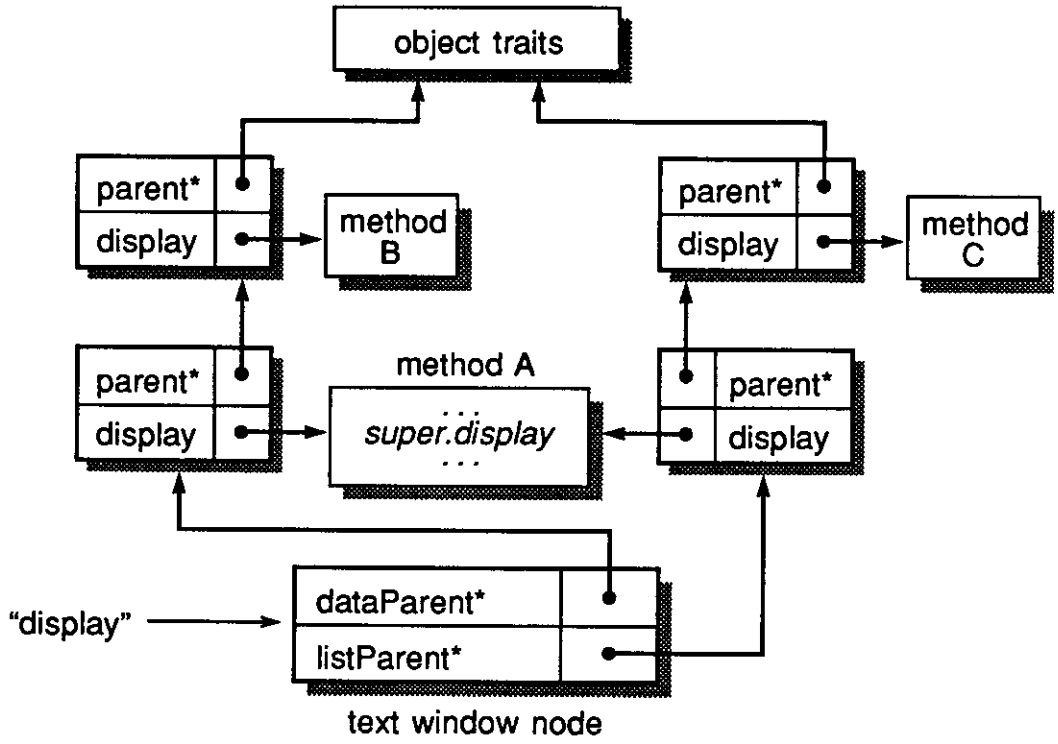
The text window node inherits two different `display` operations, one from its `dataParent` and the other from its `listParent`. When the text window node receives the `display` message at run time, the system reports a conflict because either method A or method B might be invoked.

If the same `display` slot were found along different parent paths, no conflict would exist. Consider the following:



Even though the text window node inherits `display` operations along both the `dataParent` and `listParent` paths, they are contained in the same slot, so the message handler is unambiguous. The text window can therefore respond to the `display` message by invoking method A.

One subtlety of method inheritance is that inheriting the same method along multiple parent paths is a conflict unless the method holders are the same. Consider:



The text window node inherits the same display method (method A) from both its `dataParent` and `listParent`. Although the same method is inherited, the `display` message to the text window node is ambiguous because method A behaves differently depending on whether the `dataParent` or `listParent` is regarded as its method holder. The "`super.display`" message in method A binds in one case to method B and in the other case to method C.

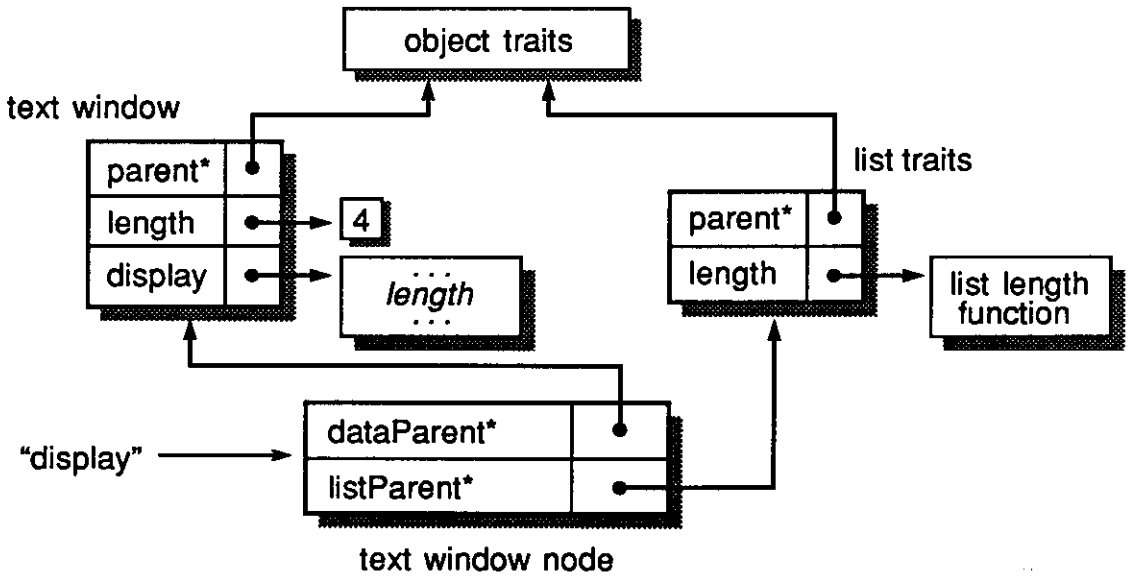
The simple rule for inheritance conflicts is that finding distinct matching slots along multiple parent slots is considered a conflict. The contents of those slots is

irrelevant. In the preceding example, the `display` message would be ambiguous even if method A never sent a message depending on its method holder. It would be ambiguous even if method A contained no code at all—if method A were actually the integer 3, for example. This approach is more conservative than necessary and may be changed in the future to only report an error when a conflict actually arises.

3.3.6.3. The sender path constraint

The simple rule presented in the preceding section treats an unreasonably large number of message sends as ambiguous. A large class of such conflicts may be avoided through a sender path constraint.

Consider the following situation:



In this example, the `display` message is unambiguous but the code for the `display` method contains an implicit-receiver send of `length` that is considered ambiguous by the simple rule for inheritance conflicts presented in the preceding section—the method search for `length` proceeds through the text window node (the receiver of the `display` message) and then finds matching `length` slots in both the `dataParent` and `listParent`. Though an apparent conflict exists, that conflict is

unreasonable—the implicit-receiver send of `length` of the window `display` method is meant to obtain the length of a window on the screen, not calculate the length of a linked list.

Another way of looking at this situation is to observe that the text window node can be viewed in two perspectives, either as a window or as a list node. Any implicit-receiver messages sent by a method in one perspective should therefore be preferentially bound to another method in the same perspective. In the example above, that approach would cause the `length` message to be bound to the window length data slot, not the list length calculation.

SELF supports such a view of multiple inheritance. The method search for implicit-receiver messages gives preference to slots found on a path containing the method holder for the sender of the message. That preference is not a strict constraint: if no matching slot is found on the sender path, the method search may bind a message to a slot not on the sender path.

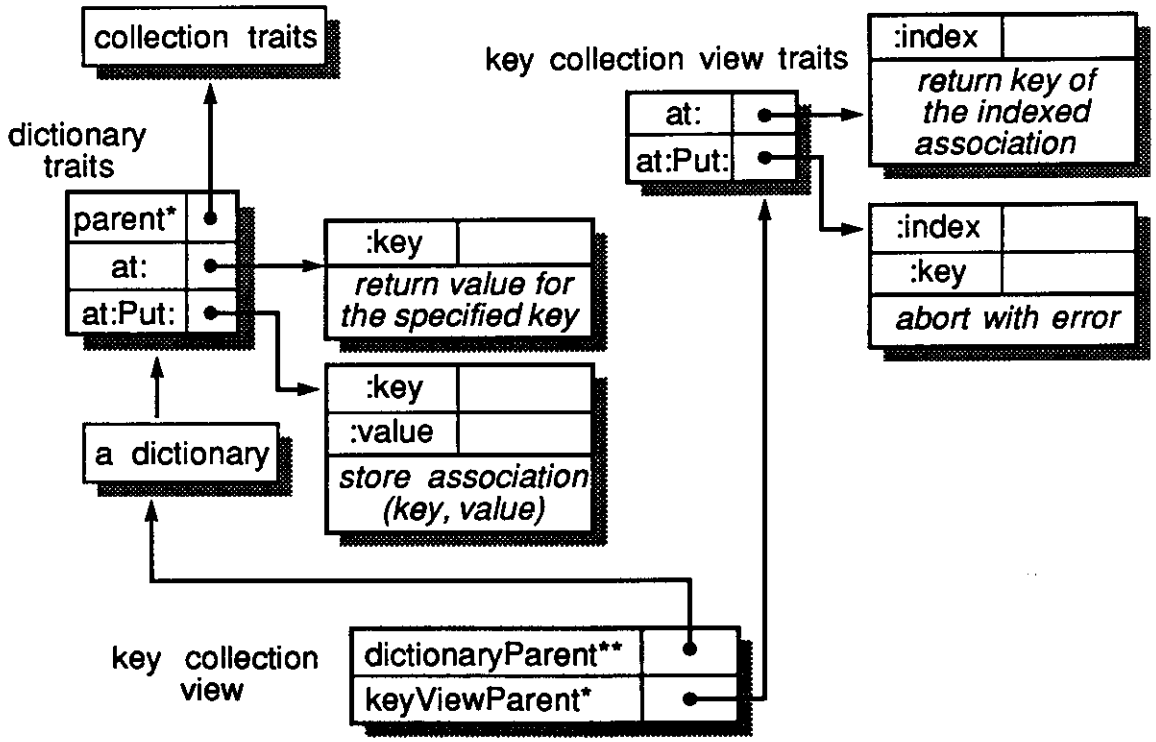
In the example, the sender of the `length` message is the `display` method, whose method holder is the `dataParent`. Since the window `length` slot is in an object on a parent path containing the method holder but the list node `length` slot is not, the method search binds the `length` message to the window `length` slot.

3.3.6.4. Ordered inheritance

The unordered inheritance semantics presented in the previous sections are well suited to the combination of unrelated objects but do not support the use of multiple parents to mix in overriding behavior. To enable such overriding behavior, SELF allows the programmer to specify explicit priorities for parent slots. Priority levels are indicated by the number of asterisks following the name of a parent slot, with priority level one (the normal case) representing the highest priority. Larger number

of asterisks therefore indicate lesser priority levels. The SELF method search checks parents in priority order, so if a matching slot is found for one priority level, no parents of lesser priority are searched.

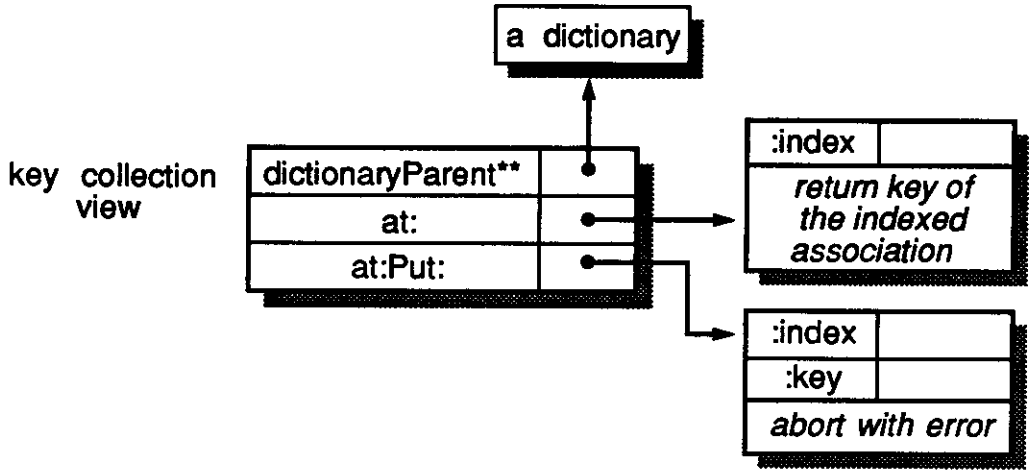
For example, consider a dictionary implemented as a collection of associations, where an association is a mapping consisting of a key and a value. Although such an object normally behaves as a collection of values indexed by keys, it might at times be more conveniently viewed as a dictionary with the mapping reversed (keys indexed by values), as a collection of the associations, as a collection of only the keys, or as a collection of only the values. A SELF program can form an alternative view of such a dictionary object through the use of ordered inheritance:



The example above illustrates the use of a view object that provides a perspective of a pre-existing dictionary as a read-only collection of its keys. The view object inherits state and behavior from the dictionary and also overriding definitions of `at:` and `at:Put:` through a higher priority parent. The `at:` method is overridden to

return the key for an association stored at a specified numeric index. The `at:Put:` method is overridden to disallow alterations to the dictionary through the view.

Similar functionality can be achieved without ordered inheritance by placing the overriding slots in the view object itself:



Such an approach is inferior to the ordered inheritance approach. In the ordered inheritance case, a change made to the key collection traits object is immediately reflected in all pre-existing key collection views. For example, if the programmer adds a new `removeKey:` slot to the key collection traits object, the new message will be understood by all pre-existing key collection views, since they all inherit their behavior from the same key collection traits object. This is not true of the unordered inheritance approach because all pre-existing views would be independent; a change made to one would have no effect on any other view.

3.3.6.5. Combining ordered and unordered inheritance

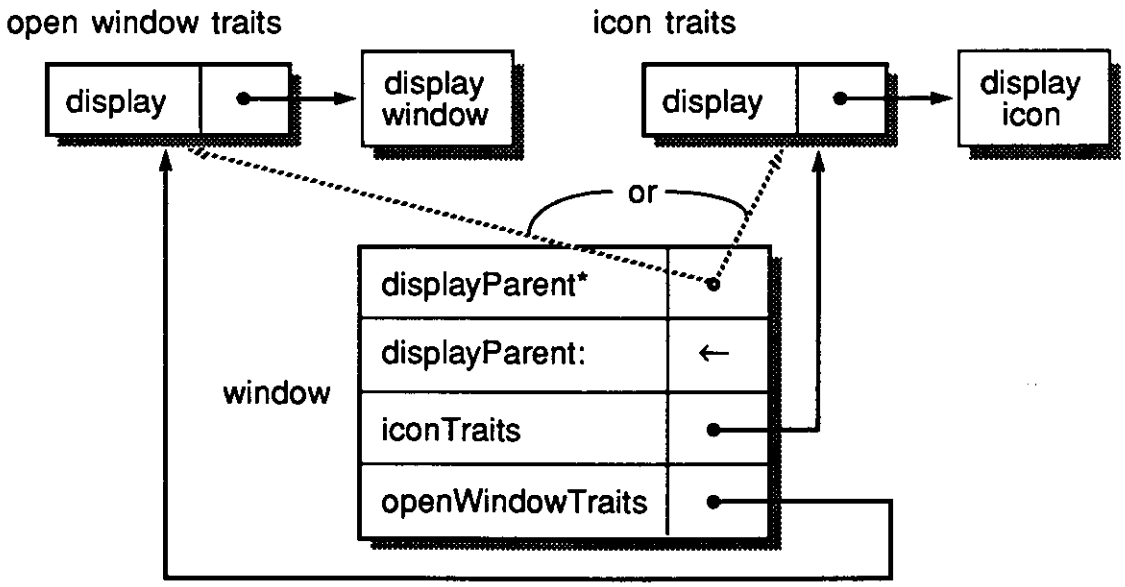
Since parent priorities do not have to be unique, ordered and unordered inheritance can be combined by simply specifying groupings of parent slots based on priority level. Parents in the same priority group would carry the same priority, and all parents in a priority group would carry higher priority than any parent in a lesser

priority group. The method search examines parent groups in order from highest (one asterisk) to lowest priorities. Within a priority group, name conflicts are dealt with using the rules for unordered inheritance. If an unambiguous matching slot is found for a priority group, no lesser priority groups are searched.

3.3.7. Dynamic inheritance

SELF objects can be designed to change their inheritance dynamically at run time. The language mechanism for dynamic inheritance is simple: an object is specified with one or more assignable parents, which may be assigned to at run-time just like any other assignable slot.

For example, a window object might inherit different operations depending on if it is open or iconified; it could respond to a `display` message either by displaying an open window or an icon. Such behavior may be implemented as follows:



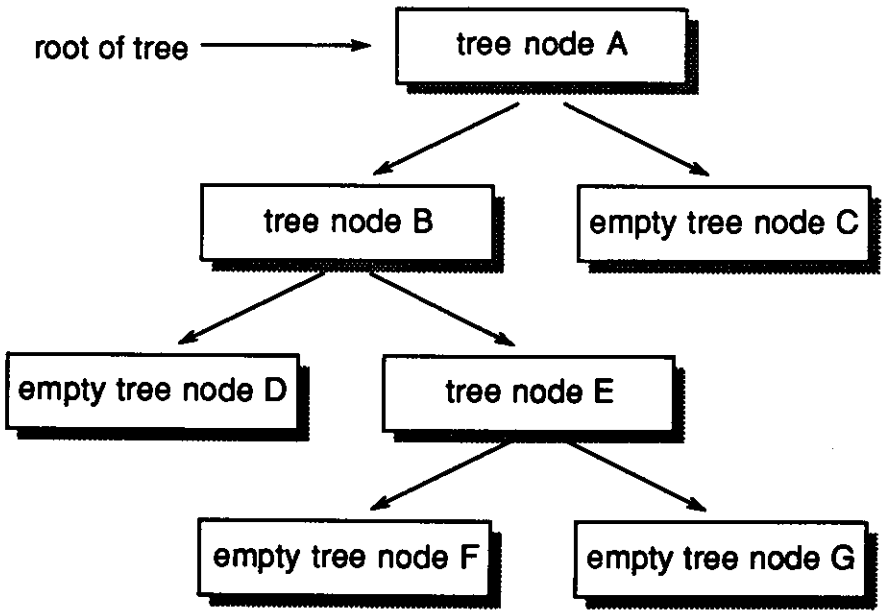
The window object inherits its `display` method through an assignable parent slot named `displayParent`. When the window is open, the `displayParent` slot is set to the open window traits object, and the window inherits a `display` method

which displays the open window on the screen. When the window is iconified, the `displayParent` slot contains the icon traits object, and the inherited `display` method displays the icon of the window on the screen. Changing the parent is accomplished through simple assignment. For example, when mutating from open window to iconified form, the code

```
displayParent: iconTraits
```

may be used to change the `displayParent` to the appropriate object.

Dynamic inheritance may be used to inherit state as well as behavior. For example, consider a possible SELF implementation of binary trees. A binary tree can be regarded as a hierarchical structuring of tree nodes, where each tree node consists of some data and references to its left and right subtrees. Trees can be empty; an empty tree holds no data and has no children. The following diagram illustrates an example of a binary tree.

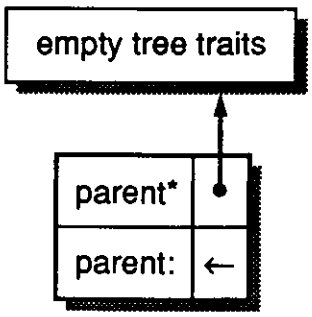


In the example above, A is the root (top-level) node of the tree; it has two children, the subtrees whose roots are B and C. C is an empty tree node, indicating that A's

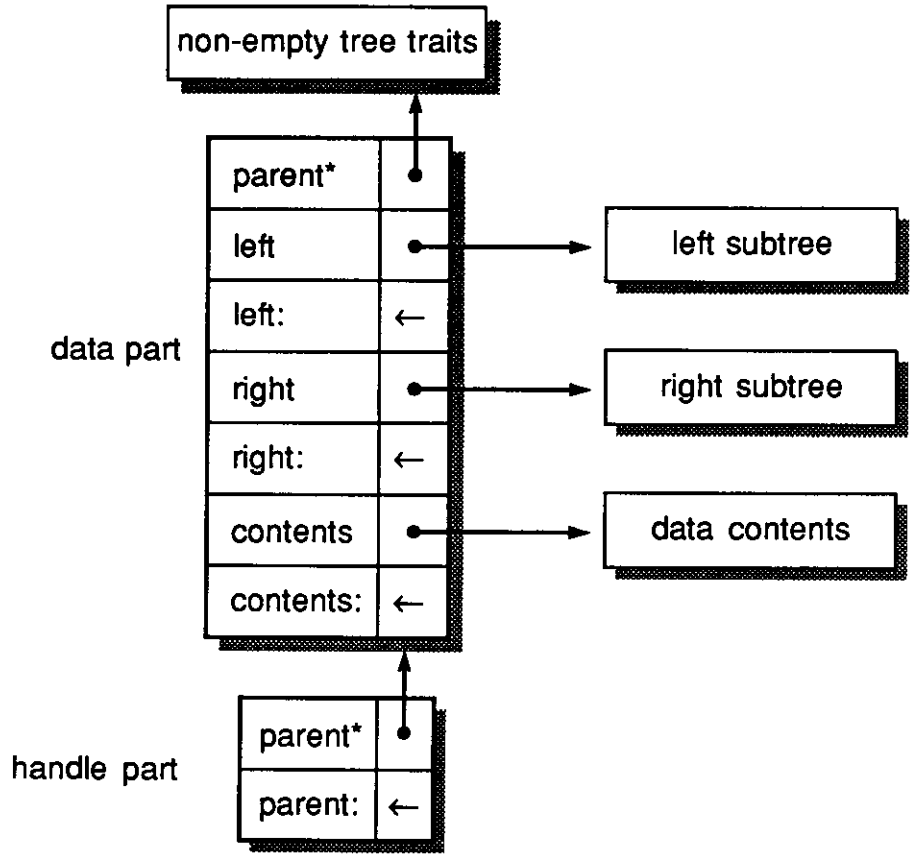
left subtree is empty. Tree node B itself has two children, an empty left subtree (D) and a right child rooted by E, whose children are empty trees (F and G).

Empty tree nodes do not respond to messages in the same way as non-empty tree nodes. Empty tree nodes cannot respond to `left` and `right` messages, which are used to determine the left and right children of a non-empty tree node. Empty and non-empty tree nodes also differ in the way that they respond to messages that access data contents. For the `includes:` message (which checks whether the subtree rooted at the receiver contains the specified data), an empty tree node always return false, while a non-empty tree node must check its contents and possibly the contents of its child subtrees. For the `insert:` message (which inserts the indicated data into the receiver subtree), an empty tree node can insert the data in itself (transforming into a non-empty node), while a non-empty tree node must determine the proper place to insert the data within its subtree.

The difference in message response between empty and non-empty trees can be directly expressed in SELF by making them inherit from different traits objects. The empty and non-empty tree traits objects would contain different implementations of operations like `includes:` and `insert:`. Each empty tree routine can be written with the knowledge that it will only be invoked for empty trees, and each non-empty tree routine can similarly be written specifically for non-empty trees. The following diagram shows how an empty tree node could be represented in SELF:



A non-empty tree node can be represented as follows:



The non-empty tree node is represented as two objects, the handle and the data, in order to facilitate transformation between empty and non-empty tree nodes. When data is added to an empty tree node, the node can transform itself into a non-empty node by creating a data part with the proper data contents and changing its parent to be the newly created data part. Similarly, when the last data object is deleted from a non-empty tree, the root tree node transforms itself into an empty node by changing its parent to be the empty tree traits.

Dynamic inheritance is necessary to capture this dynamic tree behavior. Although an object-oriented language without dynamic inheritance could express a static distinction between empty and non-empty tree traits, it could not express the

dynamic changes in behavior necessary to make that distinction useful in implementing routines like `insert:` and `delete:` that could change the message protocol of the receiver. For that reason, a conventional implementation of binary trees without dynamic inheritance would use the same routines for both empty and non-empty tree nodes. Many of those routines must check whether the receiver is an empty or non-empty tree before taking any action.

A similar effect to dynamic inheritance can be achieved in Smalltalk through use of the `become:` primitive, which switches the identities of its receiver and argument so that all references to the receiver are changed to refer to the argument object and vice versa. For example, a Smalltalk implementation of binary trees could use the `become:` primitive to mutate an empty tree instance into a non-empty tree instance. Using dynamic inheritance, however, has a number of advantages over using `become:` in that fashion. One advantage is that the possibility of dynamic inheritance can be readily seen by the programmer because all objects that could change their inheritance will have assignable parent slots. In contrast, no such indication exists showing whether a particular object could be transformed through `become:` in Smalltalk—the control of the transformation is external to the object being transformed, so a given object could be transformed by a `become:` invoked by an arbitrary routine in the system. A related advantage is that dynamic inheritance flags the possibility of a transformation to the compiler as well as the programmer, which a sophisticated compiler could possibly take advantage of to produce better code. Finally, dynamic inheritance may also require less overhead for a mutation in a system without an object table because it avoids a full scan of object memory to switch references. Although `become:` is fast in a system with an object table, such a system is slower in normal operation because all data accesses would require an

indirection through the object table. In a system with dynamic inheritance, only those routines that actually use dynamic inheritance would have to pay the cost to support it.

3.4. Summary

SELF is a prototype-based object oriented programming language unifying state and behavior. The kernel of SELF is built upon a few simple ideas—objects, slots, messages, and inheritance—which are applied uniformly to produce a flexible and expressive language. In addition to those basic features, SELF provides a number of more advanced features, including blocks, multiple inheritance, and dynamic inheritance.

In a straightforward implementation of SELF, however, some of the same features responsible for SELF's power and flexibility could result in unacceptable performance penalties:

- *Prototypes.* A straightforward implementation of prototypes would require at least twice the storage of a class-based system, since each object would require storage for the name and contents of each slot, and possibly argument and parent slot attributes as well. In comparison, in a class-based system, each instance only requires storage for the contents of each instance variable.
- *Garbage collection.* As with Smalltalk and Lisp, SELF provides automatic storage reclamation, a feature which makes the task of programming easier but which can require substantial run-time overhead.
- *Multiple inheritance.* A method search involving multiple inheritance could potentially search a large portion of the inheritance hierarchy. Unlike

languages with strictly ordered, linearized inheritance, SELF cannot end a unordered inheritance method search when a match is found, since another parent path may contain a conflicting or overriding slot. Unlike languages using simple unordered inheritance, SELF cannot always end a method search with an error when conflicting slots are found—if the conflicting slots do not lie on a sender path (a path containing the method holder), another parent path may contain a slot that does lie on a sender path and therefore overrides the conflict. In addition, a SELF method search cannot always stop a traversal of a parent path when a matching slot is found, because it may be necessary to search the ancestors of the object containing the matching slot to determine whether the match lies on a sender path.

- *Dynamic inheritance.* The method-caching techniques used by high-performance Smalltalk systems to reduce the cost of run-time message lookup do not directly apply to objects that are able to dynamically change their inheritance. The principal difficulty is that no simple, uniform test can determine whether a method cache hit is valid, since the determination of the proper method to invoke could depend on the state of assignable parent slots not just in the receiver but in its ancestors as well.

Those points indicate the issues that must be addressed by a high-performance implementation of the SELF programming language, particularly in the areas of object storage and inheritance. The remainder of this thesis describes the current implementation of the SELF memory system and run-time message lookup, which successfully addresses most of the indicated performance issues.

Chapter 4

Memory System Architecture

4.1. Introduction

This chapter describes the architecture of the SELF memory system, which is responsible for managing object storage. Only the static makeup of the memory system—its storage layout, object format, and functional organization—are examined here; the dynamic algorithms used for object management will be considered in the following chapter.

4.2. Primary issues

The primary issues facing the memory system are:

- *storage efficiency for prototypes.* Objects in a straightforward implementation of a prototype-based language—in which all objects are independent and define their own format—would consume a large amount of storage because they would require storage for the name of each slot as well as the contents. In contrast, instances in a class-based language only require storage for the contents of their instance variables. A high-performance memory system for a prototype-based language like SELF should provide storage efficiency comparable to that of class-based languages.
- *object creation and destruction.* Since objects are created and destroyed frequently, both object creation and automatic storage reclamation should be implemented with minimal overhead.

- *object manipulation*. Basic object modification operations like adding, removing, and modifying slots should be handled efficiently.

4.3. Secondary issues

The SELF memory system is designed to support a highly interactive programming environment similar to Smalltalk. The envisioned environment would foster programmer productivity with both substantial programming tools and imperceptible compilation time. At the same time, SELF code should run efficiently—at least as fast as compiled Smalltalk or Lisp code, and preferably comparable to unoptimized C code.

To support both those goals, the memory system must efficiently implement both the basic operations used by the environment (e.g., searching for object references) and by run-time SELF code. The design of the memory system must take these requirements into account, both directly in the case of object searching and indirectly by supporting mechanisms like the in-line method cache, which is used to increase the efficiency of SELF code.

In addition to the primary issues mentioned before, the memory system must therefore also address the following issues:

- *polymorphic messages*. Use of polymorphic messages, which may invoke different operations at different times, increases the likelihood that code may be reused. Polymorphism should involve minimal overhead so that programmers are not discouraged from its use. Mechanisms like the in-line method cache that reduce polymorphism overhead require memory system support.
- *dynamic inheritance*. Dynamic inheritance must be supported as efficiently as possible.

- *searching for object references.* Finding all objects containing a given object is expected to be a common primitive operation used by the programming environment to provide useful information to the programmer. The memory system should be designed to support efficient searching for object references.
- *optimized compilation.* To achieve run-time efficiency, the SELF implementation compiles programs to native code using a number of optimization strategies such as custom code compilation (an approach in which a given method compiles differently depending on the identity of the receiver) and method in-lining (also known as procedure integration). Such optimizations complicate the task of the memory system.

4.4. Compilation strategy

Although the SELF compiler [Cha, ChU88] is outside the scope of this thesis, the SELF compilation strategy deserves mention because it affects a number of aspects of the memory system. SELF programs are first parsed into compact byte codes. When a message would cause a method to run, the byte code for that method is compiled into native code, which is cached for future invocations.

The compiler performs a number of optimizations when generating the native code. The same byte-code method may be compiled into different versions of native code depending on the relative positions in the inheritance hierarchy of the object containing the method and of the receiver that inherits the method. Compiling different versions of code permits most implicit-receiver messages to be resolved at compile time. The compiler can take advantage of early resolution of a message by generating the native code for the bound method in-line in the caller rather than by

generating a procedure call. State access in such cases can also be efficiently generated in-line rather than as a message send.

Those optimizations, along with the use of in-line and external method caches to reduce the cost of messages that the compiler cannot bind statically, carry the penalty that compiled native code may become invalid if a slot is added, removed, or modified. Properly updating previously compiled code in such an event requires extensive use of dependencies. The use of dependencies affects a number of areas in the memory system, as will be indicated in subsequent sections of this and following chapters.

4.5. Organization of object storage

SELF objects are referenced by tagged object pointers (termed *oops*, for object-oriented pointers). An integer is represented immediately by its object pointer; for all other SELF objects, the object pointer indicates the location of word-aligned data storage for the object in memory. All object pointers are the same size, 32 bits in the current SELF implementation.

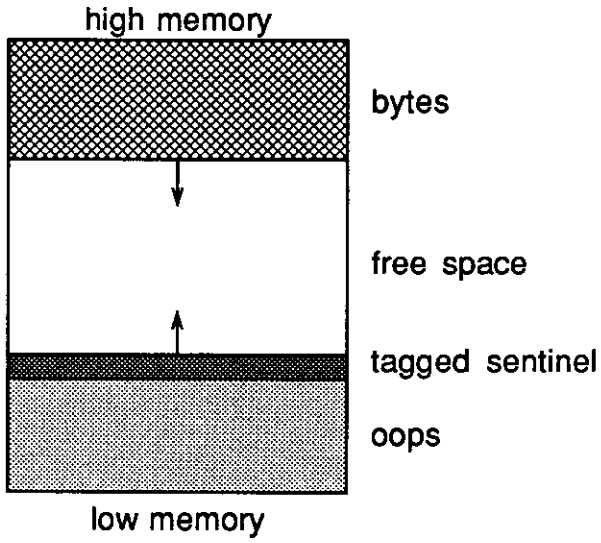
Object pointers do not just denote SELF objects but also lower-level objects such as maps and byte arrays, which are used to implement user-level SELF objects. Many of the operations of the SELF storage manager function at the level of such low-level VM (virtual machine) objects rather than at the higher level of SELF objects. In the remaining discussion of object storage in SELF, the term "object" will principally be used to refer to VM objects.

Most memory objects are stored in one of four object spaces managed by generation scavenging [Ung86]. Those four object spaces will be collectively referred to as the heap. The discussion of object storage in this thesis will focus primarily on

heap objects rather than on objects like native code and method activation records, which are managed differently.†

4.6. Object spaces

Each object space is divided into two sections, one containing only oops and the other containing byte data like strings, byte vectors, small bitmaps, and floating point numbers. The oops portion begins at the bottom of a space and grows upward, terminating with a tagged sentinel word; the bytes portion begins at the top of a space and grows downward:



The arrangement and operation of the four object spaces will be explained in Chapter 5, which examines the storage management algorithms.

4.7. Fast scanning for oops

Segregating bytes and oops, along with the tagging scheme described in the following section, allows operations like searching for an oop and switching all occurrences of one oop to another (a one-way *become* operation) to be accomplished

† Native code is stored in a separate space. Method activation records are stored on the stack or in processor registers (managed using register windows on the SPARC and explicit saves and restores on the MC680x0).

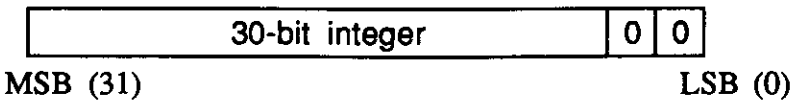
by scanning through the oops portion of the space. Since all elements of the oops portion are the same size (32 bits), the oops portion may be scanned on a word-by-word basis. In contrast, scanning on a per-object basis—necessary if bytes and oops were intermingled—would require extra overhead to determine the format of each object and to avoid searching byte data, which could produce spurious matches.

A typical application of an oop search requires that an operation be performed on each object that contains a matching oop. For example, the programmer may wish to see a list of all objects referring to a given object. As another example, the SELF memory system performs a one-way *become* operation to switch oop references when an object moves as a result of resizing; an object containing a switched oop may need to be remembered for proper operation of generation scavenging. A simple word-by-word scan through oops storage is able to support such containing-object operations because each object begins with a specially tagged header word. Once a matching oop is found, the containing object may be found by searching backwards for the tagged header word.

4.8. Tags

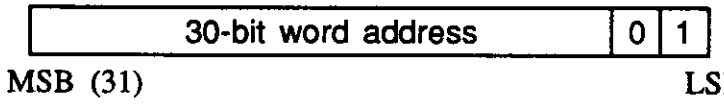
Object pointers are tagged to indicate the basic representations of their underlying objects. The two least significant bits of each 32-bit word are used as the tag bits. The meaning of the tag bits are as follows:

- A pointer with both bits 0 and 1 off is a signed 30-bit two's complement integer whose value is embedded in the remainder of the pointer:



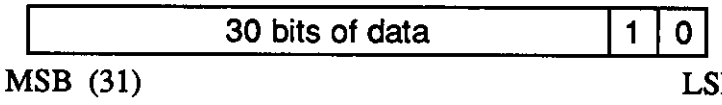
- A pointer with bit 0 on and bit 1 off is a reference to an object in main memory. The remainder of the pointer is interpreted as a 30-bit word

address to main memory:



Such object pointers may refer to objects in the heap or in the stack.

- A pointer with bit 0 off and bit 1 on is used for values that do not represent objects. Such values are called *marks*, since they are typically used for header words marking the beginning of a object in main memory. The 30 non-tag bits in a mark may take on different meanings in different contexts.



- The 11 tag pattern is undefined and unused, so that a word may be determined to be a mark or an object reference by checking a single bit.

4.8.1. Comparison with previous tagging schemes

Early Smalltalk implementations typically used only a single tag bit, indicating whether the oop is a integer or a reference to an object in memory. (Memory pointers in SOAR also carried a three-bit generation tag for storage management [Ung86].) Wirfs-Brock [Wir83] discusses the various alternatives: whether the tag bit is the most significant or least significant bit, and whether integers should be indicated by a tag value of one or zero. The best choices for those options depend on the machine architecture that the implementation runs on.

Shaw [Sha88] looks at tagging schemes for Lisp systems. Lisp systems typically use more tags than Smalltalk, with different tags for various different classes of data objects (conses, fixnums, symbols, floats, strings, etc.). Three encoding schemes are commonly used: low tagging (tags in least significant bits), high tagging (tags in most significant bits), and BIBOP or BIg Bag Of Pages (in

which all objects on a fixed-sized memory page are of the same type, and the page number is used into a table of tags). Again, the best choice depends on the machine architecture. Principal issues in evaluating tagging schemes are the expense of tag removal, the expense of small integer arithmetic, and the reduction in address space. Shaw found high tagging to be best for processors for which the address size is less than the size of a data word, especially if the non-address tag bits are ignored by the addressing hardware. Low tagging is found to be best when objects are required to be aligned (starting, for example, only on four or eight byte boundaries) and the machine architecture uses byte addressing (as is the case for most conventional architectures). The BIBOP approach involves more memory references than the other approaches, and is most appropriate when the address space is limited and would be unacceptably reduced by tag bits, or when bitfield operations are poorly supported by the machine architecture.

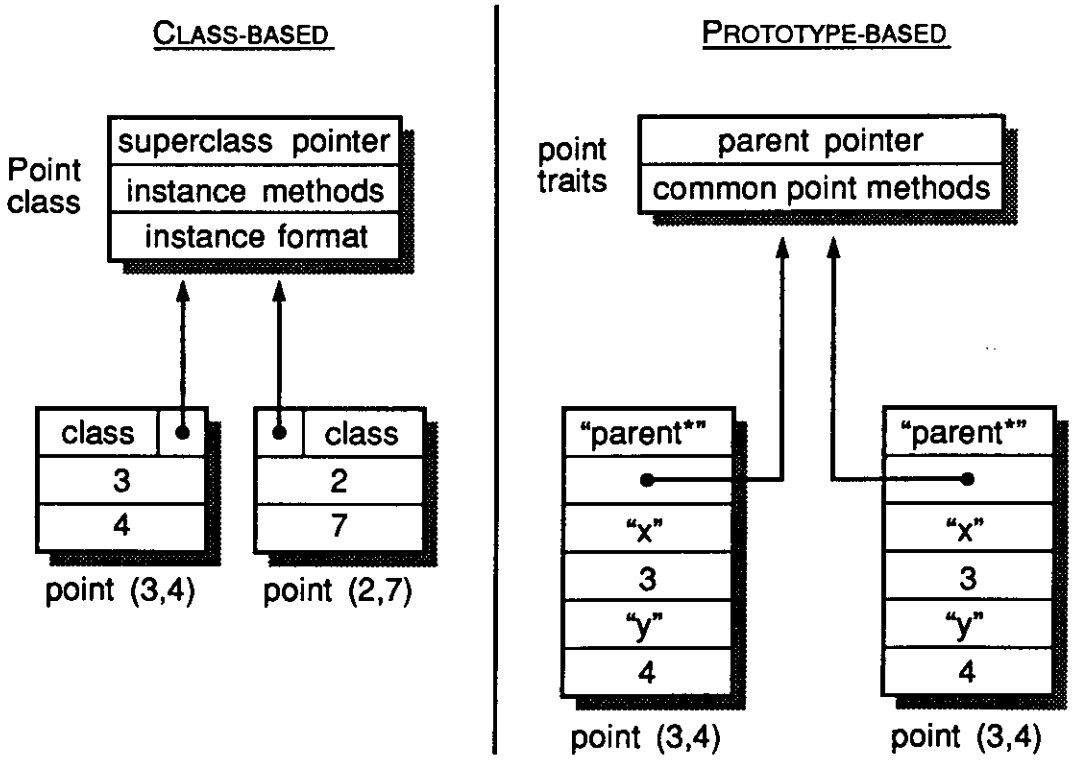
The tagging scheme used by SELF (two bits of low tagging with 00 as the integer tag) was chosen primarily because it agrees with the scheme used by the tagged addition and subtraction instructions on the SPARC, which set the overflow flag if any operand has a non-zero tag. Taking advantage of the SPARC tagged arithmetic instructions permits SELF to perform an integer addition or subtraction operation in a single cycle, only requiring a message send if the tagged instruction fails. Integer arithmetic on the 680x0 is also fairly efficient, since no operand conversion needs to be done for addition and subtraction, and multiplication and division only require one operand to be shifted right two bits. Since all memory oops point to word-aligned objects—making the least significant two bits of an address immaterial—the two tag bits do not affect the amount of addressable object memory.

Tag removal for memory references is also inexpensive. A memory oop (with a

01 tag) can be used directly without conversion by performing the tag removal with a register indirect with displacement addressing mode. To access the third word in an object, for example, the corresponding oop would be placed in a register, and a register indirect memory reference with a displacement of seven bytes would be used, rather than eight bytes if the tag were not present. The overhead for tag removal is minimal for the SPARC, since a register indirect memory reference with a displacement takes the same amount of time as a reference without a displacement. For the 68020, the use of a 16-bit displacement only costs 5% to 18%.

4.9. Object maps

At first glance, an object model based on prototypes would appear to require much more space than a model based on classes. For example, the following diagram compares object storage for Cartesian points in a class-based system and in a straightforward implementation of a prototype-based system:

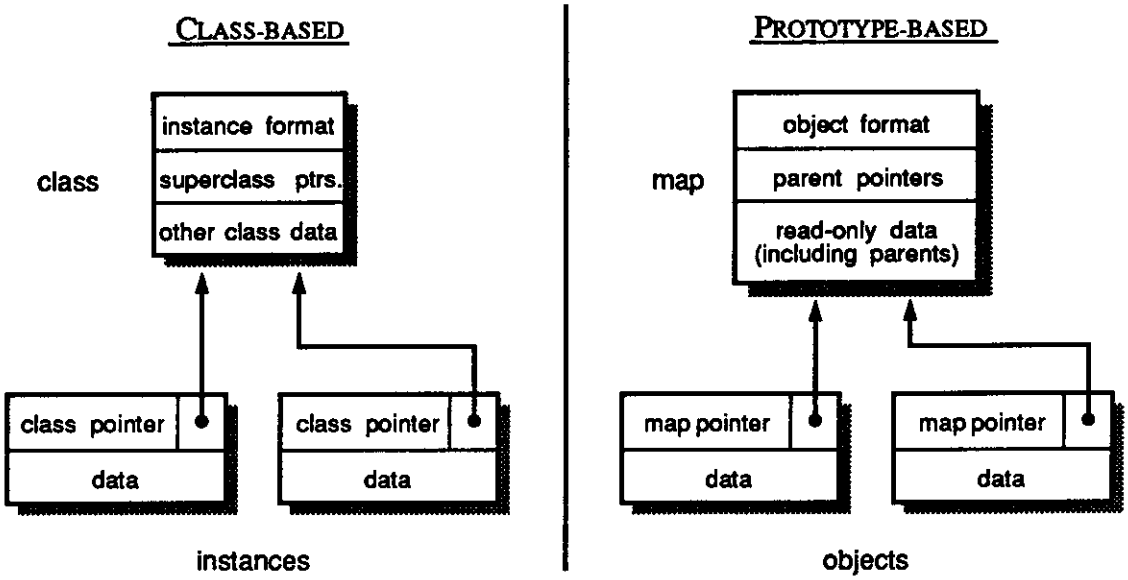


In a class-based system, all instances of a class must have the same format and inheritance attributes, which can be stored in the class and shared by all the instances. Each instance must store only its class pointer and the values of all its instance variables; the total storage for each point instance is therefore only three words of data. In contrast, each object in a prototype-based system determines its own format information, such as the names of its slots. Objects in a straightforward implementation of prototypes would require storage for the names as well as the contents of their slots. As a result, each point object in such a system would consume six words of storage as opposed to three words for each point instance in a class-based system. Since since most classes typically have many instances, a class-based system would appear to require approximately half the space of a prototype-based system.

Our implementation of SELF provides space efficiency comparable to a class-based system by exploiting the phenomenon of *clone families*. Just as many objects in a class-based system are instantiated from the same class, many SELF objects are cloned from the same prototype. Such objects form a clone family. For example, SELF points belong to the same clone family because they are cloned from the same prototypical point object. The membership of a clone family is more general than simply objects cloned from the same prototype: two objects belong to the same clone family if they have been cloned from objects that belong to the same clone family at the time of cloning. Because the cloning operation preserves object format, all members of a clone family share identical formats.

The implementation takes advantage of clone families by factoring out information common to members of the same clone family into a VM object called a *map*. Maps and classes serve a similar function in reducing the space costs for object storage, as

indicated in the following diagram of object storage from the implementation's perspective:



As may be seen from the preceding diagram, maps store more than simply format information: all read-only data shared by members of a clone family are stored in the map. More precisely, the contents of slots without corresponding assignment slots are stored in the map and not in individual members of the clone family.

As an important special case, a non-assignable parent pointer—the most common type—always resides in the map and is shared by all members of the corresponding clone family. Besides saving space, storing parent pointers in that manner ensures that when none of the parents are assignable, objects with the same map must share the same inheritance characteristics. That fact is used by the SELF in-line method cache and native-code generator to efficiently implement potentially polymorphic SELF code [Cha, ChU88].

Although integers are directly represented by their oops and do not have any individual object storage in memory, they have an associated integer map, known specially by the memory system. Since all integers should inherit from the integer

traits object, they each require a parent slot containing the integer traits object. The integer map specifies the contents of that single read-only parent slot shared by all integers. One implication of that solution is that all integers must share the same parent. Although SELF semantics specify that all objects are independent, so that changes to one object can be made without affecting others except through inheritance, integers are an exception: it is not possible to change the contents of the parent slot or to add a slot to one integer without affecting all the others. Other solutions do exist to implement full SELF semantics for integers, but they require more overhead, which we did not feel was justified for typical uses of integers.

Although maps may appear similar to classes, they are strictly an implementation optimization and do not alter the semantics of the language. Maps are completely invisible to the programmer, to whom all objects appear to be completely self-contained. That invisibility is possible because maps are immutable—whenever a programmer changes either the format of an object or the contents of a non-assignable slot, a new map is created with the changed information, starting a new clone family around the changed object. Members of the old clone family are insulated from the change.

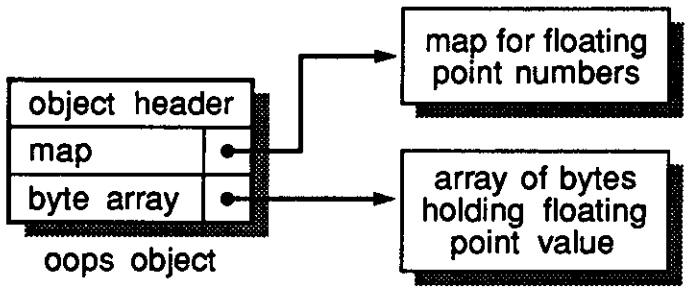
4.10. Components of SELF objects

The memory system deals with objects at a lower level than user-level SELF objects. Three types of such objects exist:

- An *oops object* constitutes the main part of a SELF object, containing not only the object's oops—the slot contents and/or oops array—but the object header, map pointer, and a tagged pointer to a byte array, if any. The oop denoting a SELF object is represented as a tagged (01) pointer to the beginning of the corresponding oops object.

- A *bytes object* contains the byte array for a SELF object. It serves as an extension object for the corresponding oops object.
- A *map* defines the format of a SELF object and stores the contents of all read-only slots. If the object contains code, the map also holds a pointer to the corresponding code object. The map also contains the start of the dependency chains that determine what native-code methods must be updated if any of the slots are changed.

A user-level SELF object may be composed of one or more of those basic VM objects. For example, a floating point number object consists of a map object, a bytes object containing the actual floating point value, and an oops object holding pointers to the map and bytes:



The oop for the floating point number points to the beginning of the oops object.

Some components of a SELF object are themselves user-level SELF objects: slot names are represented by SELF string objects, and the SELF code for an object is represented as a code object that is an extension of the original object. Such an extension code object is itself a full-fledged SELF object, which contains slots, an oops array, and a byte array containing the byte-code representation of the SELF code. A code object may not itself bear code.

The formats of these component objects will be examined in more detail in following sections.

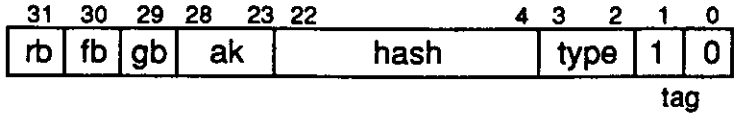
4.11. Object format

This section describes the formats of objects stored in the heap. The formats of such objects heap differ from the formats method activation objects, which are stored on the stack or represented in registers. The formats of method activation objects will not be examined since they depend on the register allocation and method in-lining strategies of the compiler, which will be described in a dissertation by Craig Chambers [Cha] and a paper by Chambers and Ungar [ChU88], both in preparation.

4.11.1. Object header

The oops portion of each object begins with a header word tagged as a mark (10 tag). The header word is the only word in the oops part of an object that is tagged as a mark; all other words are oops that are tagged either as objects stored in memory (01) or as small integers (00). As a result, given any word in the oops portion of an object, the beginning of the following object may be determined by scanning words for the next oop tagged as a mark. Tagging object headers as marks also allows the memory system to do without size fields in oops objects, reducing object storage overhead. The size of an oops object may instead be determined by scanning for the beginning of the next object and determining the distance between the object headers.

In addition to the tag, the object header contains a number of flags for storage management and method lookup, a hash value, an age field for oops objects or bytes kind field for bytes objects, and an object type field:



rb = remembered bit
fb = forwarded bit
gb = union of garbage-collection mark bit and method-lookup cycle-detection bit
ak = union of age field and bytes kind field

The remembered bit and age field are used for generation scavenging. The gb bit is used during garbage collection to mark reachable objects and during method lookup for cycle detection. The fb bit indicates a forwarded object during garbage collection and scavenging. These uses will be discussed in more detail in Chapters 5 and 6.

The hash field is used to provide each object with a semi-unique value suitable for use as a hash table key. The first object is currently allocated with a hash value of 3, and each subsequent object is allocated with a hash value one greater than the previous. After the maximum hash value is reached, the hash allocations wrap around back to the initial hash value of 3. Hash values from 0 to 3 are used to denote special oops:

HASH VALUE	MEANING
0	error indication
1	sentinel
2	bytes object

A mark whose hash is 0 is used internally in the implementation to indicate an error where an oop would normally be returned. Sentinel marks are used in various places in the memory system to terminate searching and copying loops. A bytes hash indicates that the object contains bytes rather than oops.

In place of an age field, the header word for a bytes object contains a field indicating how to interpret the bytes data. The bytes type field is interpreted as follows:

BYTES TYPE	MEANING
0	raw bytes
1	byte code
2	string
3	float

Since a bytes object is always the extension for an oops object, it shares the same hash value and age as its corresponding oops object and does not itself need explicit header fields for a hash and age.

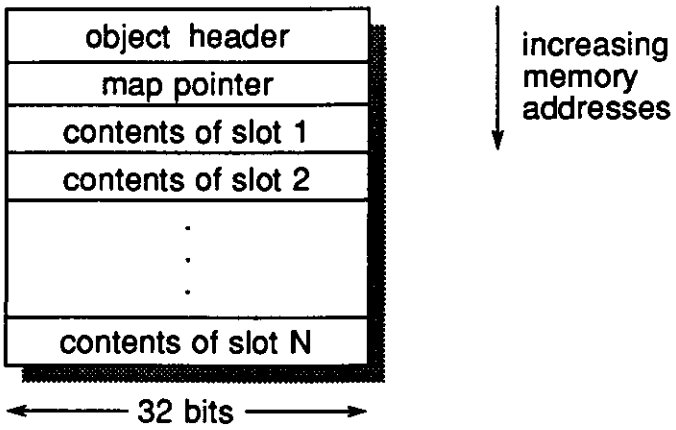
The two-bit object type field is currently used to indicate whether the object is a SELF object or a map:

TYPE BITS	MEANING
01	SELF object
10	map
00	unused
11	unused

The type field is an optimization to reduce the time necessary to determine the object type. The object type could be determined without the type field, but at the cost of two memory references and a comparison (to compare the map against the canonical map for maps) rather than a memory reference and a bit test. The two reserved bit patterns may be used in the future to optimize detection of other common object types, such as strings or floats.

4.11.2. Objects with slots

An object containing only slots is represented by an oops object with the following format in the heap:

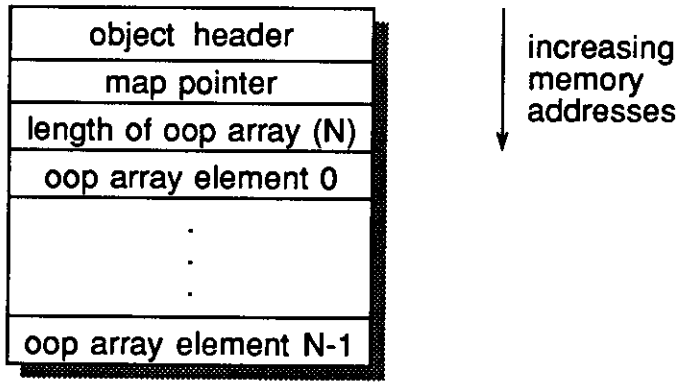


The object begins with an object header word tagged as a mark. Following the object header is the map pointer, which identifies the map defining the object's format. The remainder of the object holds the contents of the assignable slots, in the order indicated by the map.†

Parent slots currently precede non-parent slots, and parents are ordered so that higher priority parents precede parents of lower priority. The ordering scheme minimizes the expense of searching for parents, since they may be searched in priority order through a simple linear search. In retrospect, that optimization seems premature since parent searching time does not seem to be a bottleneck, and the ordering scheme precludes using hashing for slot access.

4.11.3. Objects with oop arrays

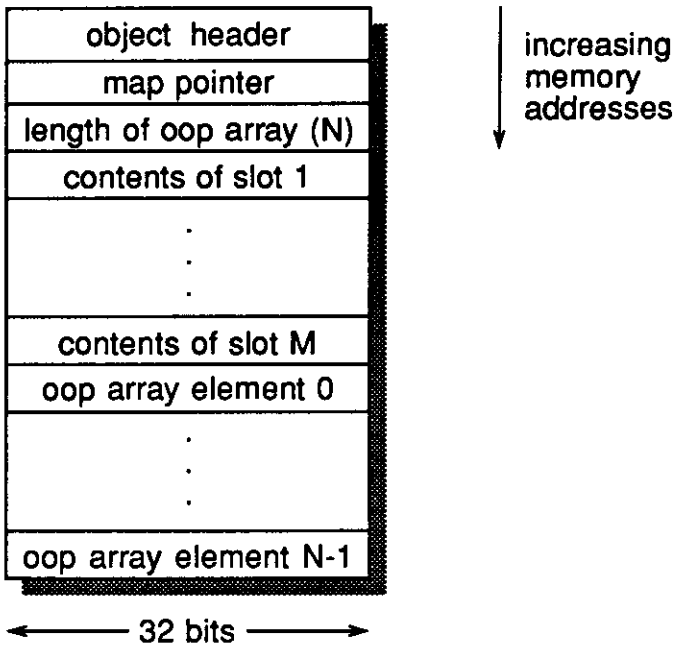
An object containing only an oop array is represented as:



In addition to the header and map pointer, the object contains an oop array and an array length field. The array length is stored in the object so that objects with differently sized oop arrays may share the same map.

† Read-only slots are stored in the map rather than in the object. An object that only contained read-only slots would be represented as an oops object with only an object header and a map pointer.

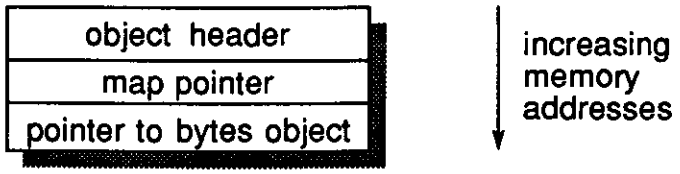
An object containing both slots and an oop array has the following format:



The slots precede the oop array so that the slots have the same field offsets for all objects sharing the same map, even if the objects contain differently sized oop arrays.

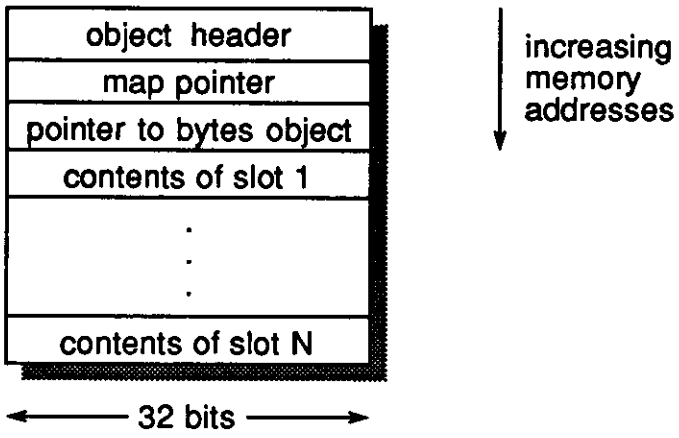
4.11.4. Objects with byte arrays

An object with a byte array is represented with an oops object containing a pointer to a bytes object containing the byte array:

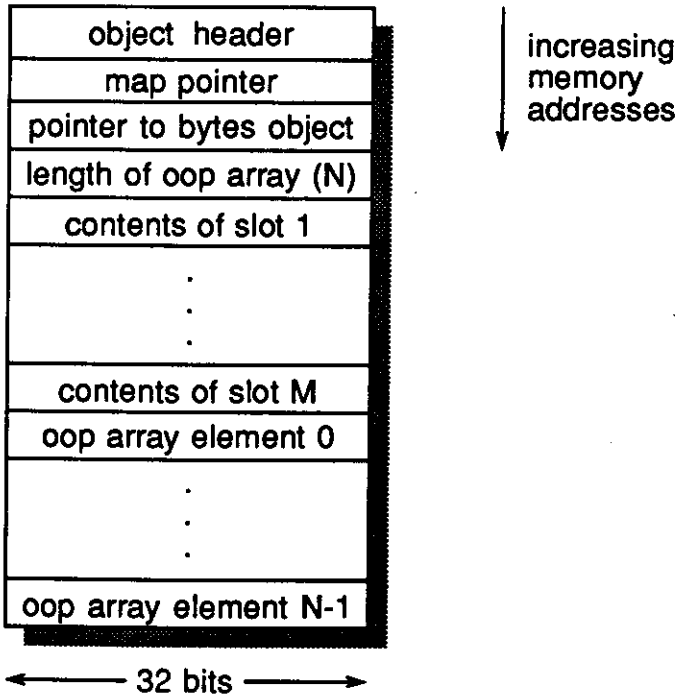


The bytes object pointer is tagged as a memory oop (01 tag).

An object containing both an byte array and assignable slots has the following format:

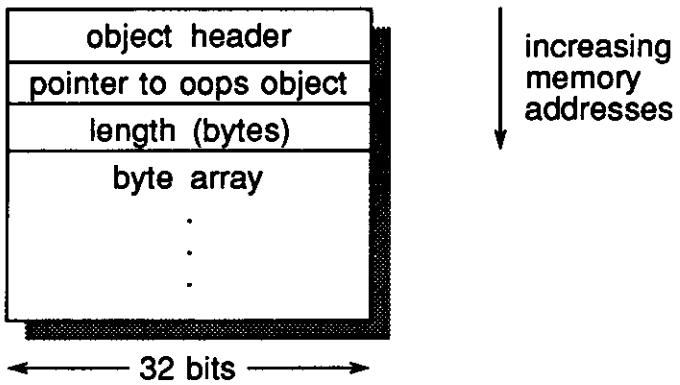


An object containing an oop array, byte array, and assignable slots is represented as:



4.11.5. Bytes objects

A bytes object consists of an object header, a tagged pointer to the corresponding oops object, a length field, and a byte array:



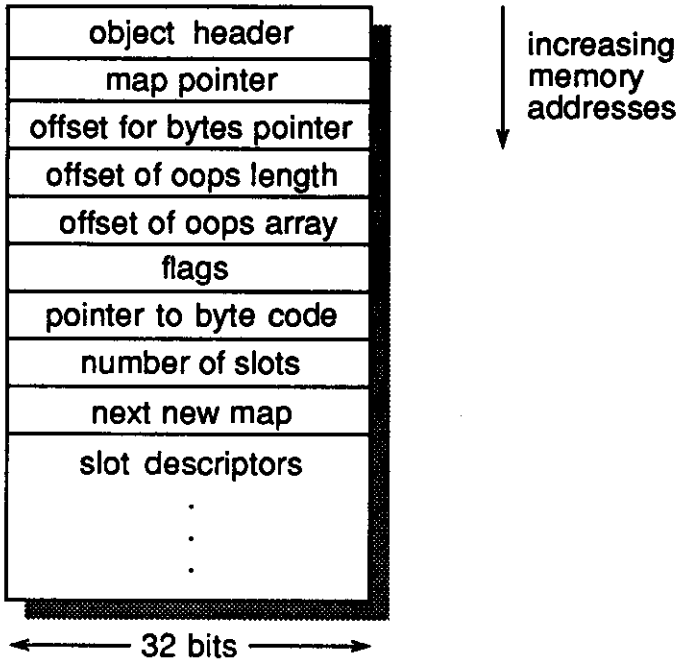
As indicated previously in section 4.11.1, a field in the object header determines whether the bytes object is interpreted as a simple byte array or as a special bytes object (byte code, string, or float).

The length field specifies the logical length of the byte array, not of the entire bytes object. Although the byte array may be of any length, the actual amount of space allocated is an integral number of four-byte words. The end of the bytes object may therefore contain zero to three filler bytes that are not part of the logical byte array. An integral number of words are allocated so that all bytes objects begin on word boundaries, allowing all bytes pointers may be tagged in the two least significant bits.

Floating point numbers are presently encoded in 32-bit Sun (IEEE) floating point format.

4.11.6. Map format

Maps contain format information, read-only slot contents, and dependency information for members of its clone family. A map has the following format:



For an object which has a bytes pointer, oops length, or oops array, the offsets for those fields are stored in the map, so that the fields may be accessed by adding the appropriate offset to the object pointer (after tag removal). If one of the oop fields does not exist, the corresponding offset is zero. The offsets are in bytes; since all oop fields are word aligned, the offsets may also be considered to be word offsets tagged as integers: a byte offset of eight (binary 1000), indicating the ninth byte in the object, is equivalent to a word offset of two (binary 10), indicating the third 32-bit word in the object; the number two represented as a SELF oop has a 00 tag appended at the end of its binary representation, yielding binary 1000, the same value as the original byte offset. The fact that the offsets can be regarded as SELF integers is important for storage reclamation because the garbage collector and generation scavenger will recognize that they are not objects stored in memory and ignore them.

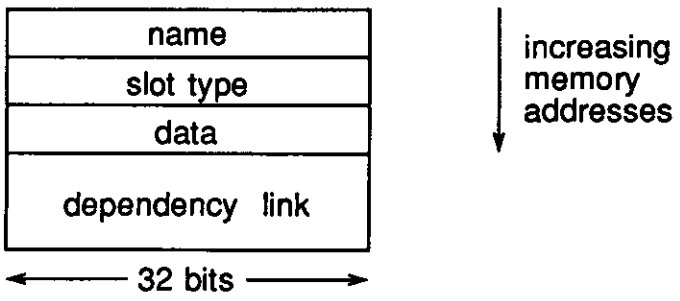
The flags field is used to store boolean flags. The field is tagged as an integer, and the 30 non-tag bits are used for flags. Currently, the only flag stored in the flags field is the inherited flag, which indicates whether a successful method search has found a matching slot through one of the parents stored in the map. If so, then whenever a parent is changed or a new slot added, all compiled native code is flushed because they could rely on a now obsolete inheritance view. Although that behavior flushes much more code than is necessary, the present system does not maintain enough dependency information to flush only those methods affected in such a situation. The inherited flag provides a highly conservative dependency mechanism that should eventually be replaced when more dependency information is maintained.

If an object has code, its map contains a tagged pointer to a byte-code SELF object representing the code. (The byte-code reference is stored in the map rather than the object because it is read-only.) A byte-code pointer of zero indicates that the map's objects are static objects without code.

The next new map field is used by the generation scavenger and will be explained in Chapter 5, which describes storage reclamation.

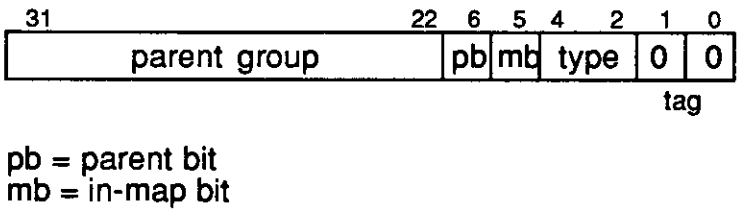
4.11.6.1. Slot descriptors

Every slot in an object is described by a slot descriptor in the object's map. A slot descriptor is fixed-sized and has the following format:



The name is stored as an oop to a SELF string object. The SELF memory system canonicalizes slot names so that all slots with the same names store the same SELF string object in their name fields. Because the compiler canonicalizes message names in the same way, message lookups can check whether a message matches a slot by performing an identity comparison on their names, rather than by comparing the slot and message names character by character.

The slot type is represented in the following fashion:



The parent bit indicates whether the slot is a parent. If so, the parent group field stores the numeric priority for a parent slot, otherwise the parent group field is zero. The in-map bit indicates whether the slot data is shared and stored in the map rather than in each individual object. The type field is encoded as follows:

TYPE	MEANING	COMMENTS
0	slack	reserves space for slots to be defined later
1	data	normal user-level slot
2	argument	—
3	assignment	conceptually contains assignment primitive
4	parent assignment	assignment slot for a parent slot
5	VM slot	used by the implementation; invisible to SELF programmer

The data field in a slot descriptor is interpreted differently for different slot types:

SLOT TYPE	DATA FIELD	COMMENTS
slack	unused	—
data (read-only)	slot contents	shared by all objects with this map
data (assignable)	slot offset	byte offset from beginning of object
argument	argument index	first argument has index zero
assignment	name of assignable slot	name of corresponding data slot
parent assignment	name of assignable slot	used to detect dynamic inheritance
VM (read-only)	slot contents	shared by all objects with this map
VM (assignable)	slot offset	byte offset from beginning of object

For historical reasons, assignment slots store the name of the corresponding data slot rather than the slot descriptor offset; an assignment slot originally was not required to reside in the same object as its corresponding assignable data slot. Assignment slots for parent slots are given a separate slot type to allow easy detection of cases where dynamic inheritance is used. The current implementation, however, treats parent and normal assignment slots identically.

In the current implementation, read-only and VM slots are the only slots that may contain dynamic objects (objects with code). This restriction is imposed because the compiler generates code based on the map of the receiver rather than its identity. If objects with code could be stored in slots whose contents reside in the individual objects rather than the map, the compiler could not determine whether to generate a state access or a function call for a message send.. Rather than requiring a run-time check for every message send, the compiler assumes that data slots stored in the object can only contain static objects (objects without code) and so will always be accessed as state. Some VM slots may contain dynamic objects; the compiler recognizes their names and generates native code appropriately.

The restriction on slots containing dynamic objects is normally never perceptible to the user. In order to obtain a value to be stored into an assignable slot or to be

passed as an argument, a SELF program must either invoke a primitive, send a message, or use an object literal. In the present system, no primitives can return a dynamic object; that behavior is unlikely to change in the future. No message send can return a dynamic object either, because a message matching a slot containing a dynamic object will cause that object to be evaluated, producing further message sends. The value returned from the message send can therefore only be a static object, either stored in a slot matching the last message, returned from a primitive, or produced by a static object literal. Finally, no object literal can return a dynamic object; object literals specifying dynamic objects are inner methods, which are evaluated when they are encountered. As a result, no SELF program can normally obtain a dynamic object to store into an assignable slot or to pass as an argument, so the restriction can never normally be encountered.†

4.12. Object allocation and cloning

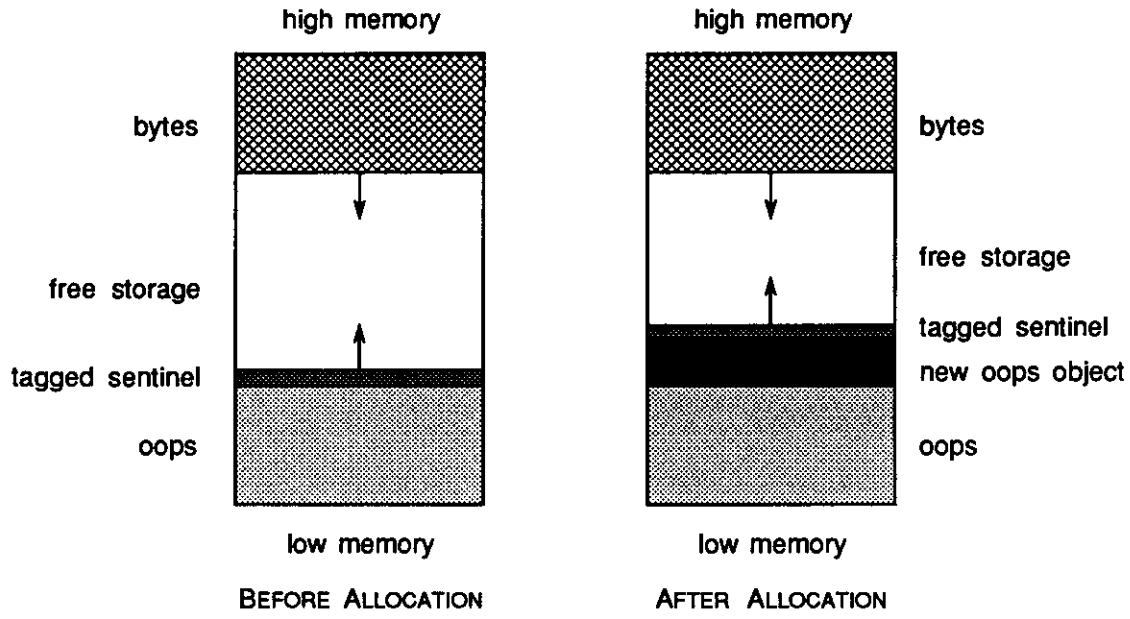
The memory system implements two basic forms of object allocation: allocating objects whose sizes are initially known and those whose sizes are initially unknown. Allocation of oops or bytes objects of known size is simple: the allocation routines move the bounds of the oops or bytes portion of an object space by the specified object size and return a pointer to the newly allocated storage. If the target object space does not have room for the new object, the routines attempt to allocate the object in another space.‡ Failure to allocate the object in any space is a fatal error.

---

† The only exception is that SELF syntax permits a dynamic object as the initializer for an assignable slot. Although legal SELF syntax, the present implementation disallows that case. The utility of that case is limited, since only a static object can be subsequently assigned to the slot.

‡ Spaces are tried in the following order: *eden*, *from*, *to*, and *old*. Chapter 5 describes the four SELF object spaces in more detail.

The following diagram shows the situation before and after an oops object allocation.

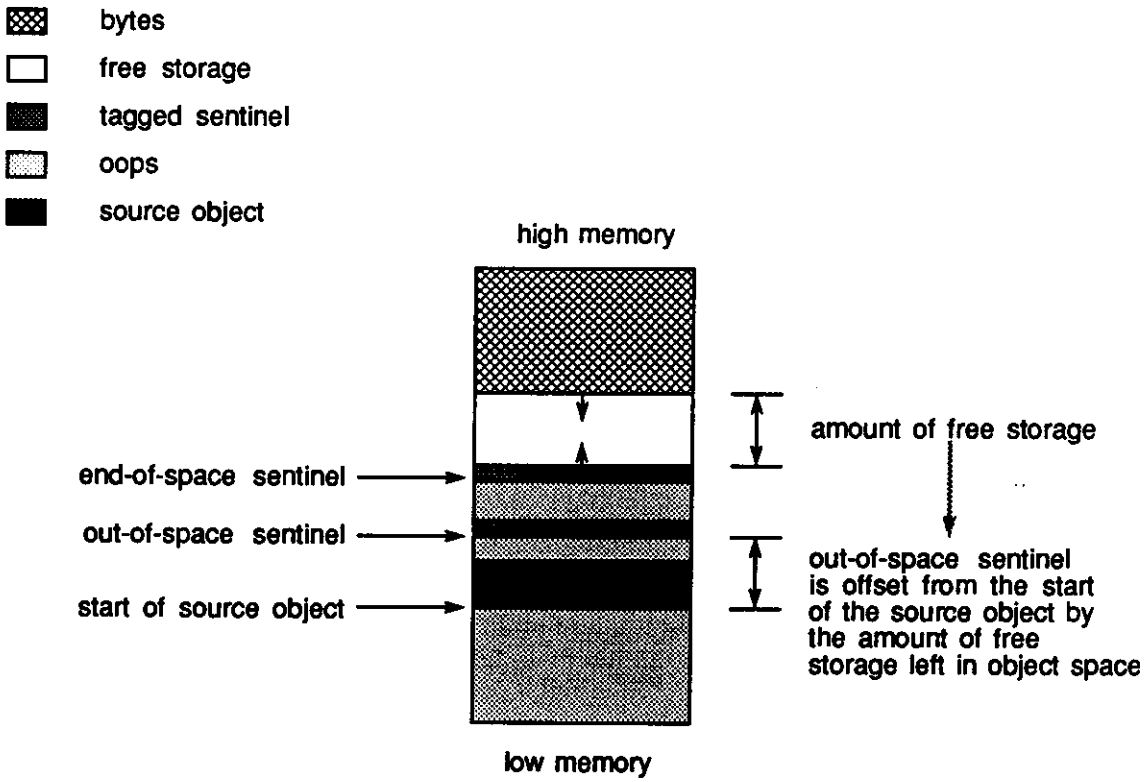


Such quick and simple object allocation is one of the advantages of a compacting storage collector.

The simple known-size oops object allocation routine cannot be used to allocate space for general cloning operations. Because an oops object does not carry a size field, its entire contents must be scanned in order to determine its size. Such a scan is expensive and undesirable, especially because the original object must be scanned again after the allocation in order to copy the contents of the original into the clone. For that reason, general cloning operations for oops objects do not use the known-size allocation routines. They instead combine allocation and copying into one scan by copying the original (source) oops object word by word to the end of the oops region until the scan reaches a word tagged as an oop, indicating the end of the original object.†

† Measurements (presented in section 4.18) show that this approach is significantly slower than a known-size clone. For that reason, the most recent version of SELF (implemented subsequent to the work in this thesis) uses size fields in maps to allow known-size cloning.

Such a general clone is more complex and more expensive than a known-size clone, which allocates the new object using the simple known-size allocation routine and then copies the contents of the original oops object into the clone. The extra complexity and expense principally arises from the need to handle out-of-space conditions when the object size is unknown. The clone routine detects an out-of-space condition through the use of a sentinel mark word. First, it calculates the amount of free storage remaining in the object space; that amount is the size of the largest object that can be allocated in the space. It then stores an out-of-space sentinel word (tagged as a mark) into the memory location that is offset by that amount from the beginning of the source object, preserving the old contents of the calculated location to be restored after the clone is completed.



Since the copying operation will stop when the scan reaches a mark, it will never overflow the object space. When the scan reaches a mark, the clone routine tests

whether the mark is the out-of-space sentinel. If it is, then the object space does not contain enough free space to hold the new object, so the routine tries to clone the object to another space. If the terminating mark is a normal object header or the end-of-space sentinel separating the oops region from the free space region, then the clone is successful. Such a general cloning procedure is more computationally expensive than the fixed-size procedure because of the extra overhead involved in setting up and testing the sentinel values; that overhead is especially significant when the object to be cloned is small. In addition, the lack of an initially known size precludes the use of loop unrolling to speed up the copying operation.

The most general clone routine is even more complex because it allows the new object to be a different size than the original. If the new object is larger than the original, the clone routine will insert the new fields where required, possibly even in the middle of the new object. To implement those semantics, the clone routine inserts size-change sentinels into the proper places in the original object. If such a sentinel is encountered, the clone routine inserts the new fields into the new object, then continues the copy. Shrinking operations are implemented similarly.

4.13. Code

SELF programs are compiled in two stages, first to compact byte codes and then at run time to native code. The memory system accordingly deals with two types of code objects: byte-code objects and native-code objects.

Byte-code objects are user-level SELF objects stored in the heap. A byte-code object has slots and both an oop and a byte array. Only one slot is currently used; it contains a string indicating the source file and line number for the SELF text producing the byte code. The byte code itself is stored in the byte array. The oop array contains the oop literals used by the byte-code method.

Native-code objects are stored in a separate object space. The space is managed as a cache for native code in order to bound the amount of storage used by native code. A native-code object that is removed from the cache for space reasons is simply regenerated from the corresponding byte code the next time that it is needed.

4.14. Dependencies

The SELF implementation maintains dependencies to enable compiled native-code to be updated when the assumptions that it was compiled under become invalid. When the contents of a slot change, for example, compiled native code that depends on the slot contents could become out of date. To keep track of what compiled code would need to be invalidated, each slot descriptor has a link pointing to a chain of native-code objects depending on its contents. The dependency could arise either because a native-code object was directly compiled from the contents of the slot or because native code for the method was substituted in-line into the code body generated for another method. Modifying the contents of a normal slot or removing the slot entirely will cause all dependent native-code objects on the slot dependency list to be invalidated. Dependencies operate in the reverse direction as well: native-code objects have links back to slot descriptors in maps to enable the forward links to be updated when the native-code objects are removed, such as when the code cache is full and space is needed for a new native-code object. The existence of back pointers from native-code objects to maps affects the operation of storage reclamation, as will be seen in Chapter 5. More information on dependencies may be found in Chambers's forthcoming dissertation [Cha].

4.15. Internal data structures

The design of the memory system originally followed a philosophy of placing internal data structures of the implementation into the SELF universe whenever possible. For that reason, byte code, native code, and generation scavenging data structures like the remembered set and the age table were SELF objects stored in the heap. The principal advantage of following that philosophy was that the internal data structures were readily accessible from SELF programs, both for inspection and for alteration.

The present implementation has backed away from aggressively following that philosophy, because placing objects like the remembered set in the SELF universe complicated the code in the implementation and led to subtle bugs—a SELF-level remembered set, for example, requires careful handling in the case where it must be resized during a scavenge. The current implementation now treats the remembered set as a purely C++-level data structure. Native code has also been removed from the heap, making the code for automatic storage reclamation simpler and faster. (Previously, native code was allocated in the heap and then migrated to a code cache during scavenging.)

Access to implementation-level data structures from SELF programs could still be possible through primitive routines providing such access. Unlike the scheme of placing implementation-level data structures directly in the SELF object universe, accessing such data structures with primitives requires the implementation to decide in advance exactly what data structures should be accessible and which primitives should be written. For objects like the remembered set, such a price seems rather small in comparison with the reduction achieved in code complexity.

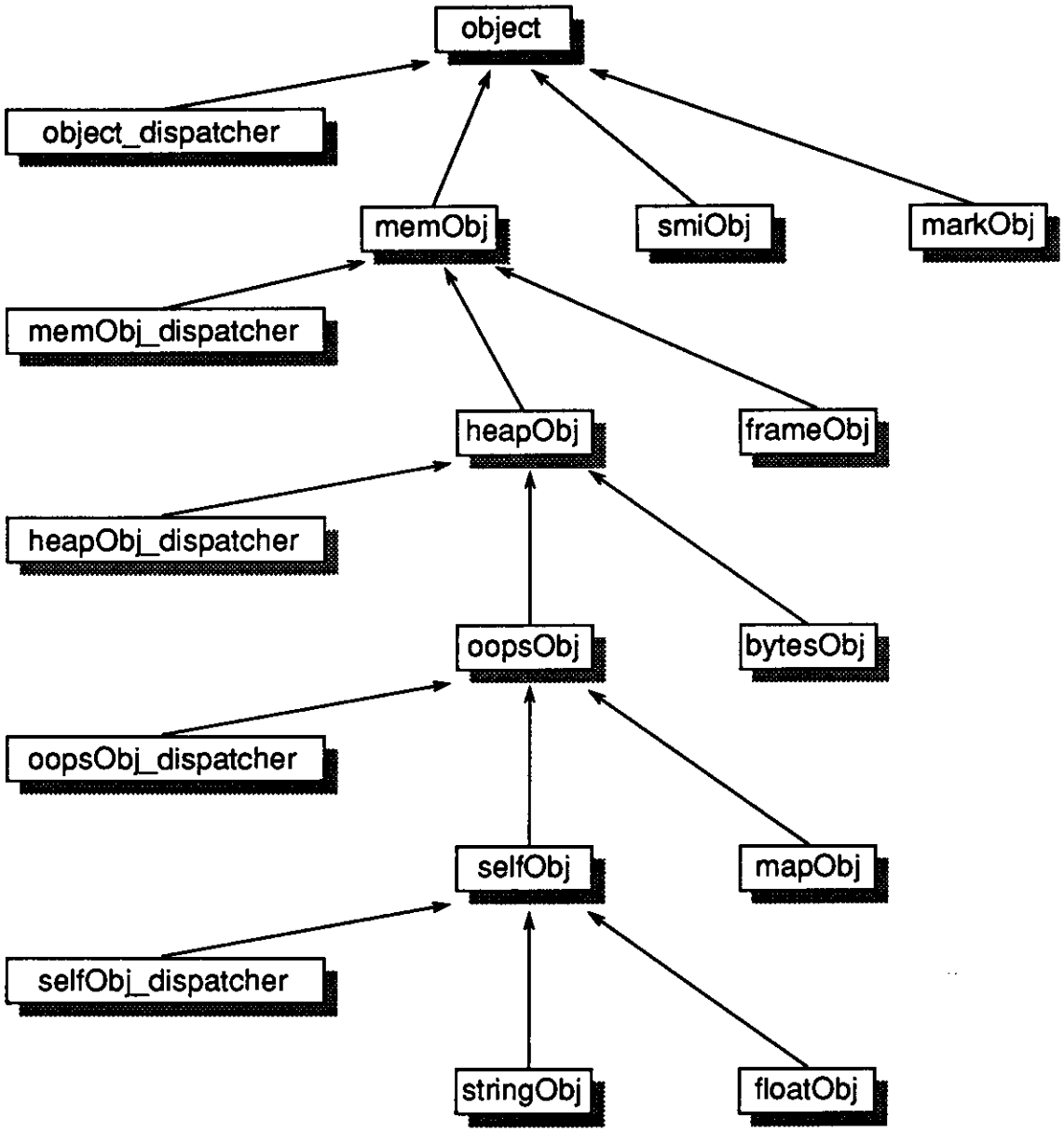
4.16. Dispatching on oops

A desire to avoid storing C++ virtual table pointers in VM objects complicated the task of writing the memory system in an object-oriented fashion. Three points motivated that desire:

- Storing a virtual table entry in each object would greatly increase the space costs.
- Objects like integers and marks are represented directly by their object pointers and do not have individual object storage that could hold virtual table pointers.
- It could produce a dependency on the C++ implementation. The dependency could arise because to be absolutely safe, the storage reclamation routines would need to know where the virtual table pointer is placed in each object in order to ignore it. (At greater run-time cost per oop, the storage reclamation routines could also perform a bounds check on every heap oop to check that it does indeed point into a SELF object space.)

Without the use of virtual tables, however, C++ provides no mechanism for dynamically choosing which object-specific procedure to apply to an oop at run time, which is necessary to take advantage of the full power of object-oriented programming. Our solution was to create a system of dispatching classes in conjunction with the normal VM object classes. Such dispatching classes inherit most operations from their associated object classes; when subclasses override a procedure in an object class, the dispatcher for that class overrides the object operation to dispatch to the appropriate procedure, either the original object procedure or an overriding subclass procedure. By declaring an oop to be a pointer to one of the dispatching classes rather than to one of the normal object classes, the C++ code can

invoke a dispatching version of an object operation using the standard C++ syntax for calling member functions. The following is an abbreviated diagram of the C++ inheritance hierarchy involving the dispatching classes and the normal object classes:



Most of the class names should be self-explanatory. The possible exceptions are `smiObj`, the class of small 30-bit tagged integers, and `frameObj`, the class of method activations stored as the frames.

The dispatching system was implemented using macros in combination with C++ inline procedures; it is written in 1002 lines of source code. The dispatching system distinguishes classes of objects (integers, bytes objects, oops objects, and so forth) by checking the tag of the oop, fields in the object header, and fields in the object's map. The dispatchers perform only as many checks as necessary to decide on the proper procedure. If, for example, an operation binds to the same procedure for all bytes objects, the dispatcher for that operation would not bother to distinguish between string oops and floating point oops. The dispatchers also take advantage of the C++ type system by assuming that an oop can only belong to a subclass of its declared class, or to the declared class itself. Given such partial type information, the dispatchers will only perform the checks necessary to further narrow down the class.

There are a number of disadvantages to the dispatching system. It performs redundant checks on oops because it has no way of caching information from one statement to the next. If the dispatchers decide that an oop is a string in one statement, they may have to perform a redundant check in the very next statement. In addition, the dispatching system adds a significant amount of complexity to the memory system. For those reasons, future implementations may store C++ virtual tables in the map, with dispatching used only to decide how to obtain the map for a given oop.†

4.17. Implementation notes

The memory system is written in 10,785 lines of C++ source code, distributed over 134 files (including header files and files of included inline procedures). In terms of object code, the memory system takes up 96,200 bytes of code, 9672 bytes of initialized data, and 24 bytes of uninitialized data on the Sun-4 (SPARC).

† This has been implemented in the most recent version of SELF, subsequent to the work reported in this thesis.

4.18. Measurements of object allocation and cloning times

One of the design goals for the memory system was to support fast object allocation and cloning. This section presents measurements of the time required to allocate and clone objects in the current SELF implementation. The object allocation measurements show the amount of time spent in the basic allocation routines `universe::alloc_oops()` and `universe::alloc_bytes()`, which respectively allocate an oopsObj or bytesObj given a target object space and the new object size as parameters. The cloning measurements include time spent initializing the fields of the new object as well as the time spent in allocation. In addition, the cloning measurements include time spent allocating objects whose size is not immediately known.

The measurements were collected for a set of small SELF benchmarks. One is a translation of the Smalltalk-80 version of the Richards benchmark, originally written by Martin Richards for BCPL and translated to Smalltalk by L. Peter Deutsch. The remainder are translations of the Stanford compiler benchmarks, collected by John Hennessy and modified by Peter Nye. David Ungar and Craig Chambers translated the benchmarks to SELF. We do not consider these benchmarks to be representative of expected system activity, but the SELF environment is not yet sufficiently mature to support the long interactive programming sessions that would provide more representative data. Of all the benchmarks, the Richards benchmark is probably the most relevant because it was adapted from a real program and performs some polymorphic message sends; the others are translated from a conventional, non-object-oriented programming language (C). Although unrepresentative, the measurements do indicate rough bounds for allocation and cloning times.

4.18.1. Methodology

I measured the amount of user CPU time required for each object allocation and cloning operation using the Berkeley UNIX™ *gprof* profiling tool [GKM82]. Because the profiler has a resolution of 10 milliseconds, each measurement reflects the profiling of multiple iterations of the measured benchmark. The measurements were performed on a Sun-4/260 (SPARC) with eight megabytes of memory; all instructions except memory references and untaken conditional branches execute in 62.5 nanoseconds (one 16 MHz clock cycle).

To enable calculation of object allocation and cloning overhead as a percentage of running time, I also measured the total (user and system) CPU time required to run each benchmark run, where a benchmark run consists of the same number of iterations used in the profiling measurements. I performed the running time measurements using SELF's timing facility, which measures the running time of a user-specified SELF block. The timing facility is currently implemented using the UNIX™ `getrusage` system call. The measurements of running time were performed separately from the profiling measurements in order to exclude profiling overhead.

All benchmark measurements were performed twice, once with the desired survivor size at 10 kilobytes and once at 80 kilobytes. The desired survivor size is a scavenging parameter explained in more detail in section 5.2.5.

4.18.2. Results of the measurements

This section summarizes the results of the object allocation and cloning measurements. The measurements are presented in more detail in Appendix B. Few of the benchmarks performed significant amounts of cloning or object allocation. Eight

of the eleven benchmarks spent less than 3% of total run time in cloning and allocation, with six under 1%. The only exceptions were *tree* (a tree-sort program that allocates many tree nodes), with a cloning overhead of 9% to 11% (depending on the desired survivor size), and the floating-point benchmarks *mm* and *fft*, with cloning overheads ranging from 61% to 72%.

The floating-point measurements point out the inefficiency of floating-point numbers in the current implementation of SELF. The current memory system creates new floating-point numbers by cloning the prototype floating-point object and modifying the contents of the clone's byte array. The memory system does not attempt to optimize such cloning operations—it uses the general unknown-size cloning routine for the oops part of a floating-point number. Since bytes objects do carry size fields, the bytes part is cloned using a fixed-size clone routine. The resulting overhead is high. The following table indicates the amount of time spent creating floating-point numbers. These times differ from the previous figures (61% to 72%) in measuring only creation of floating-point numbers (typically through floating-point arithmetic); it does not include time spent in cloning other types of objects or time spent in scavenging floating-point numbers.

BENCHMARK	ITERATIONS	FLOAT CREATION OVERHEAD (%)	
		10KB SURVIVORS	80KB SURVIVORS
mm	50	65.	59.
fft	10	56.	53.

The high cost of floating-point numbers is not a surprise; when we designed the SELF memory architecture, we recognized that floating-point operations would be expensive but did not optimize them because we did not feel that floating-point performance was a high priority for SELF applications. Part of the reason for the high

overhead is that the floating-point cloning and allocation operations are not optimized—for example, they invoke the general unknown-size oops object cloning routine even though both the oop and byte sizes of floating-point objects could be known. Another source of inefficiency is the representation of a floating-point number by an oops object as well as a bytes object; if bytes objects had maps, the oops part of floating-point numbers could be entirely eliminated, since the only slots contained in floating-point objects are read-only and could reside in their maps. As an indication of the possible time savings, the allocation of the floating-point bytes part takes roughly 16% of the floating-point cloning time. Even if the cloning and allocation time for floating point objects were reduced to the time to allocate the bytes, however, the cloning and allocation overhead for floating point numbers would still be 11% for *mm* and 10% for *fft*. If floating-point performance is important, the representation of floating-point numbers should be changed to represent their values immediately in the object pointers, similarly to small integers. We are considering such an immediate representation for future implementations of SELF.

The measurements also suggest that the lack of object size fields results in significant cloning overhead. The measurements for the *tree* benchmark indicate that the average cloning times for a five-word tree node are 29 μs (10 KB survivors) and 24 μs (80 KB survivors) for the general unknown-size cloning routine used by the scavenger, as opposed to 13 μs (10 KB survivors) and 12 μs (80 KB survivors) for the known-size cloning routine used when the compiler can determine the size of the cloned tree nodes. Those times indicate that the unknown-size clone routine is twice as slow as the known-size routine in the *tree* benchmark. Using a known-size routine in all cases would have had reduced the cloning overhead in *tree* from 11% (10 KB survivors) and 9% (80KB survivors) to 7% (for both survivor sizes). Although

the amount of impact would differ depending on the sizes of cloned objects, these figures suggest that using object size fields would improve the performance of SELF.

Finally, mean time measurements indicate that the current memory system could be more highly optimized. The average known-size allocation time, for example, ranged from 5 μs to 9 μs over the four benchmarks with meaningful data. That corresponds to spending from 80 to 140 SPARC clock cycles per allocation, which would appear to be much more than warranted by the amount of required work. Tuning the cloning and allocation routines could provide significant benefits for allocation-intensive programs like *tree*.

4.19. Second thoughts: what we would do differently

In retrospect, a number of design decisions in the current memory system architecture were probably mistakes.

Avoiding object size fields has increased complexity in the implementation, especially in the cloning routines, which contain elaborate sentinel-based code to perform efficiently while detecting space overflow conditions. The lack of size fields also prevents the use of loop unrolling to speed up a number of basic object operations. Future implementations will probably store the size of an object in its map. Placing the size fields in maps rather than in each object is beneficial both for storage efficiency and for reducing the amount of data that must be moved during a clone. The cost is that determining an object's size would take slightly longer if the object contains both slots and an oop array.

A related problem is that the clone routines allowing size changes are too complex and inefficient. They attempt to combine the copying of data with the allocation of object storage, requiring elaborate use of sentinels to mark the places

where new data should be inserted or old data omitted. A cleaner alternative would be to separate allocation from copying, so that the resized object storage would be allocated first and the contents of the new object written afterwards. Such a solution would be efficient if object size fields are supported, either in maps or in each object.

The format of objects with oop arrays results in inefficient array resizing. Since the array resides in the main body of the SELF object, resizing the array requires allocation of a new SELF object with the proper array length followed by an inefficient scan of memory to change references to the original object to point to the newly created one. If array resizing is to be supported, storing an oop array as an extension object—similarly to the current scheme for byte arrays—would be a better solution.

The current scheme of allowing objects to contain slots, oop arrays, and byte arrays may suffer from an excess of generality. Future language semantics may make slots, oop arrays, and byte arrays mutually exclusive—a SELF object would contain only one of those types of storage. More complex objects would be built out of those basic components at the SELF language level, rather than in the implementation.

Removing the bytes type field from bytes objects may be desirable. Though the bytes type field provides a means of type-checking to ensure that primitives operate on the proper arguments, the desirability of such low-level type-checking is unclear. A better solution may be to rely upon inheritance for type-checking, which is more in line with the treatment of other types of objects. Primitives in that case could be defined to operate on byte arrays rather than floats or strings. (The format of small integers would be considered to be an optimized representation for a byte array.) The inheritance hierarchy for the basic run-time system would be charged with ensuring

that the primitives would not operate on the wrong types—for example, no operation invoking a floating point primitive should be inheritable by a string object.

Despite the second thoughts mentioned above, the current memory system provides both storage efficiency and good run-time performance for object storage in a prototype-based object-oriented programming language.

Chapter 5

Automatic Storage Reclamation

5.1. Introduction

SELF removes the burden of storage management from the programmer by providing automatic reclamation of inaccessible objects. Automatic storage reclamation not only eliminates the need to write storage management code but the need to debug it as well. It frees the programmer from having to spend time debugging storage management errors like premature reclamation (dangling references) and failure to reclaim inaccessible objects (storage leaks), errors which are both common and often time-consuming to track down. Automatic storage reclamation increases programming productivity by eliminating a large class of programming errors.

Because SELF programs allocate a large number of objects, object storage can quickly become exhausted, requiring frequent storage reclamation. An efficient automatic storage reclamation facility is therefore essential for a high performance SELF system. For that reason, our SELF implementation uses a variation of the Generation Scavenging algorithm [Ung86] employed by high-performance Smalltalk systems [Ung86, UnJ88], supplemented with a mark-and-sweep garbage collector to reclaim tenured garbage. This chapter describes and evaluates the SELF generation scavenging and garbage collection algorithms.†

† Method activation objects and native code objects are managed differently. Method activation records are allocated and deallocated on a stack, and native code objects are stored in a LIFO cache. Storage management details for those objects are outside the scope of this thesis.

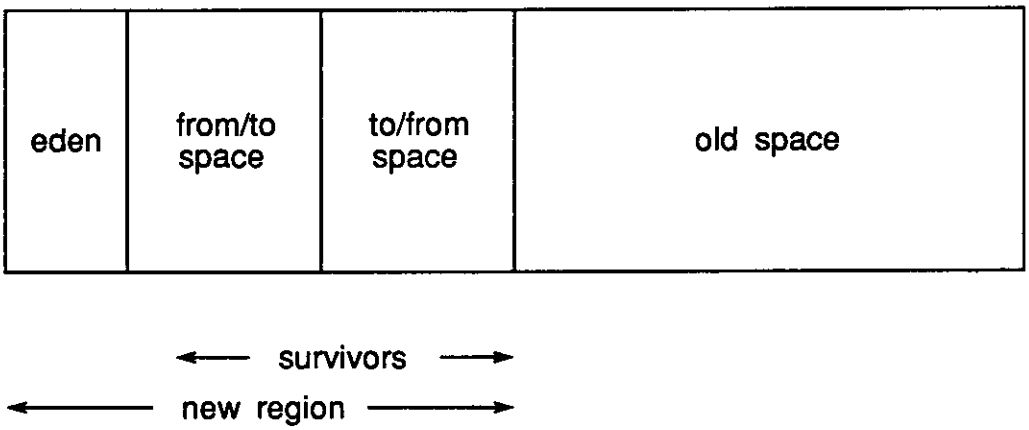
5.2. Generation Scavenging in SELF

SELF uses a version of Generation Scavenging with a demographic feedback-mediated tenuring policy similar to the one described by Ungar and Jackson [UnJ88]. SELF's implementation of generation scavenging differs from previous versions in two principal ways: bytes and oops are stored in separate regions of an object space, and finalization actions are performed for certain objects (maps) when they are reclaimed. This section describes the SELF scavenger in detail.

5.2.1. Arrangement of object spaces

As in the basic Generation Scavenging algorithm, four spaces are used: *eden*, *from* space, *to* space, and *old* space. The *from* and *to* spaces are referred to as survivor spaces. *Eden*, *from* space, and *to* space are collectively termed the *new* region of memory.

The following diagram indicates the layout of the main object spaces in SELF:



Each survivor space may play the role of either the *from* space or the *to* space. The identities of *from* and *to* spaces are interchanged after each scavenge. *Eden*, *from* space, and *to* space are contiguous in memory so that determining whether an object is old or new can be accomplished using only two bounds checks.

In the current system, the default virtual memory allocations are 100 KB for *eden*, 200 KB for each of the survivor spaces, and 2 MB for *old* space. Those allocation are untuned and may be expected to change in the future. For ease of experimentation and easy customizing, users may specify other memory allocations for the object spaces when they enter SELF.

5.2.2. Keeping track of references to new objects

To avoid scanning through the code zone and *old* space (which are typically quite large) to update references to objects moved during scavenging, the memory system keeps track of which objects in the code zone and *old* space contain references to objects in the *new* region. Old objects that contain references to new objects are stored in the remembered set, an array of oops that automatically expands to the length required to hold all the remembered objects. The remembered set is implemented as a linked list of array segments. If the remembered set is full when a new object is remembered, a new array segment is allocated from the C++ heap and added to the end.

During execution of SELF code, all stores to objects stored in the heap must check whether a new reference is being stored into an old object that is not remembered. If so, the old object is added to the remembered set. To avoid having to scan the remembered set in order to tell whether an object is already remembered, a bit in the object header is used to indicate whether the object is remembered. An object's remembered bit is set when the object is added to the remembered set and reset when the object is removed.

Native code objects in code space that contain embedded references to new objects are similarly remembered, though on a pure linked list rather than in an

extendible array like the remembered set. Stores into native code objects must check to see if a reference to a new object is being stored, and if so, must add the native code object to the remembered chain.

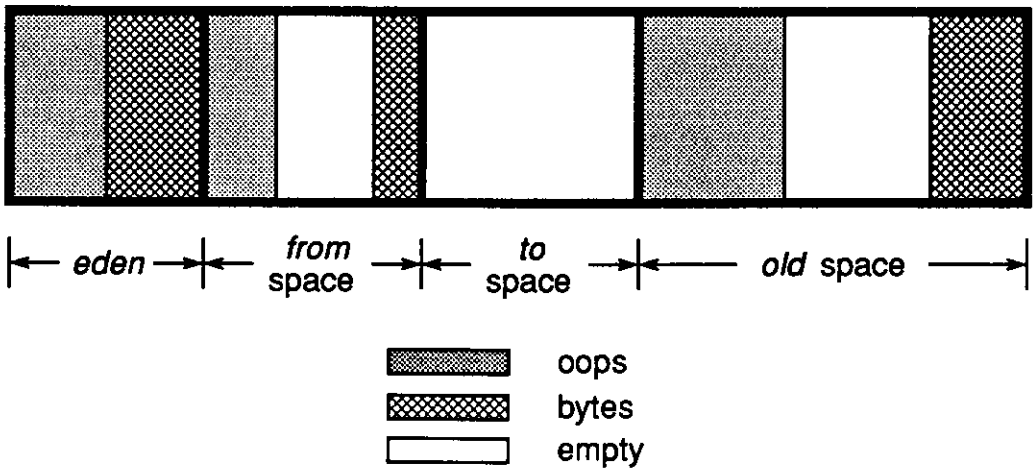
During scavenging, objects that reside in the *new* region may be promoted into *old* space, with the result that some objects in the remembered set or native-code chain may no longer contain new references. That situation is detected during scavenging, and the appropriate objects are removed from the remembered set or native-code chain at that point.

5.2.3. Scavenging algorithm

The basic scavenging algorithm is essentially Ungar's Generation Scavenging algorithm [Ung86] adapted for use with segregated oops and bytes regions and with map finalization. The basic algorithm operates in two phases, one scavenging the base set of accessible objects and the other scavenging all objects reachable from the base set.

5.2.3.1. Phase one: scavenging the base set

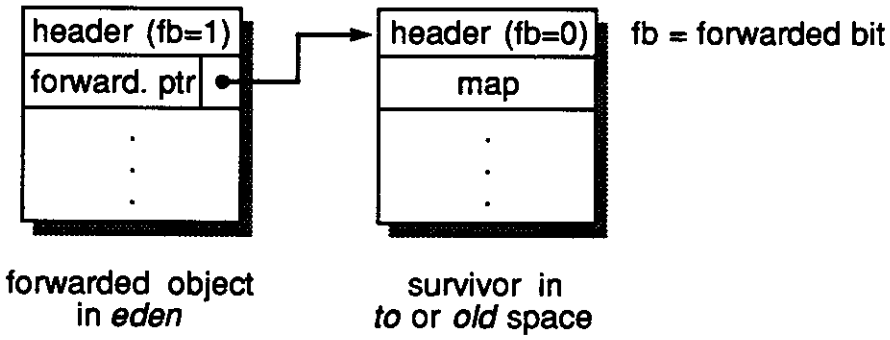
At the beginning of scavenging, *eden* is full of objects allocated since the previous scavenge and *from* space holds the survivors of the last scavenge:



During the first phase of scavenging, the base set of reachable objects is scavenged into *to* or *old* space.† The base set comprises objects in *eden* and *from* space that are embedded in SELF native code or that are directly accessible from machine registers, the SELF stack, or implementation-level (C++) program variables and data structures. Native code objects containing references to new objects are found by following the remembered code object chain.

5.2.3.2. Scavenging an object

When the scavenger moves a surviving object to *to* or *old* space, it leaves behind a forwarding pointer to the survivor at the site of the original object so that subsequently encountered references to the original location can be updated to point to the survivor:



Forwarding an object involves setting the forwarded bit in the object header and replacing the map field with the oop of the object in its new location. Forwarding pointers are only valid during the course of the scavenge. Unlike the case in incremental algorithms like the Baker [Bak78] and Lieberman-Hewitt [LiH83] collectors, no forwarding pointers exist after a scavenge is over.

---

† The diagram shows *to* space as being empty at the beginning of a scavenge. Although that is the usual situation, it is not always the case because objects may be allocated in *to* space if they do not fit in *eden* or *from* space.

To decide where to scavenge an object, the scavenger compares the object's age field against the tenuring threshold. An object's age field is initially zero and is incremented for each scavenge that the object survives. If the age is less than the tenuring threshold, the object is scavenged into *to* space; otherwise the object is tenured into *old* space. Since a newly tenured object may contain references to new objects, it is added to the remembered set upon tenuring. The scavenger does not check whether a newly tenured object actually does contain new references, since the location of any references may change during the course of scavenging—a new reference may become old if the referenced object is subsequently scavenged into *old* space. A newly tenured object that is unnecessarily added to the remembered set will be removed later in the scavenging run, when the remembered set is scanned.

In addition to tenuring, overflow may also cause a surviving object to be promoted to *old* space.† If insufficient space remains in *to* space to hold a scavenged object, the object is scavenged into *old* space instead. If an object containing both oops and bytes is scavenged into *to* space but there is only enough space for the oops part, then the bytes part of the object is moved into *old* space.

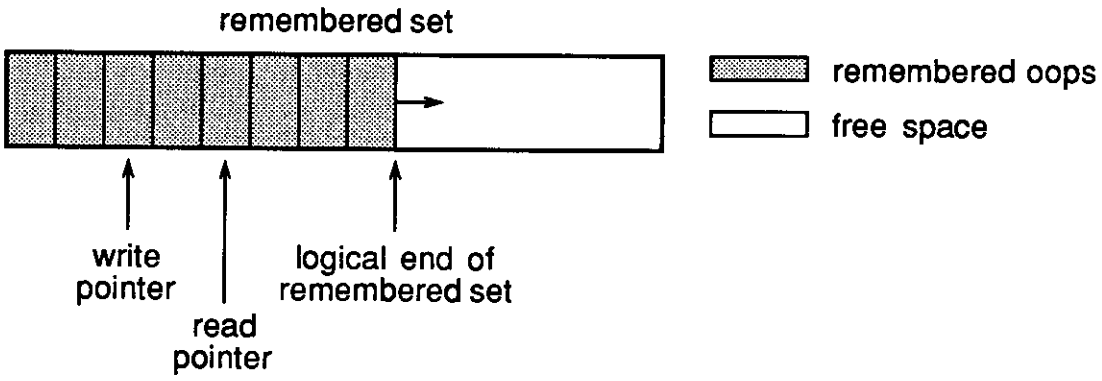
Overflowing *old* space produces an error. Before a scavenge run commences, however, the maximum storage for objects in *from* space that might be tenured is calculated and compared against free storage in *old* space. If insufficient free storage remains in *old* space to handle the worst case of tenuring, then a full garbage collection is performed to reclaim storage in *old* space before scavenging begins.

---

† Overflow is undesirable because it can create tenured garbage: the storage for a tenured object that dies cannot be reclaimed by the scavenger. The problem of tenured garbage is exacerbated by the phenomenon of nepotism [UnJ88]: when a tenured object dies, not only does it become garbage unreclaimable by the scavenger, but so does any object that is only accessible through it.

5.2.3.3. Phase two: scavenging indirectly reachable objects

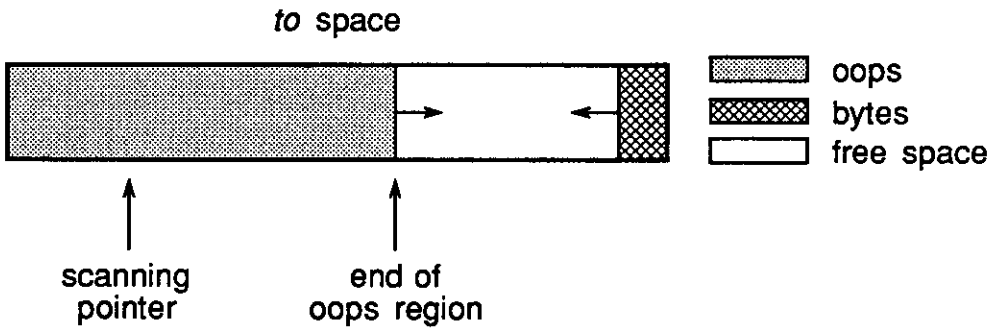
The second phase consists of two alternating scanning operations, one scavenging referents of remembered old objects and the other scavenging referents of previously scavenged objects in *to* space. New objects reachable from remembered old objects are found by scanning through the objects in remembered set:



Two scanning pointers are used, one to write into the remembered set and one to read from it. When scavenging begins, both the read and write pointers refer to the beginning of the remembered set. The read pointer scans through the remembered set to read the oops of the remembered objects. Each time an oop is read, the corresponding remembered object is searched for references to new objects. If any are found, the corresponding new objects are scavenged to either *to* or *old* space. Objects scavenged to *old* space are also remembered, causing the remembered set to be extended at the end and its logical end pointer to be incremented. The scavenger writes oops for remembered objects containing new references back into the remembered set through the write pointer, which is incremented after every write.

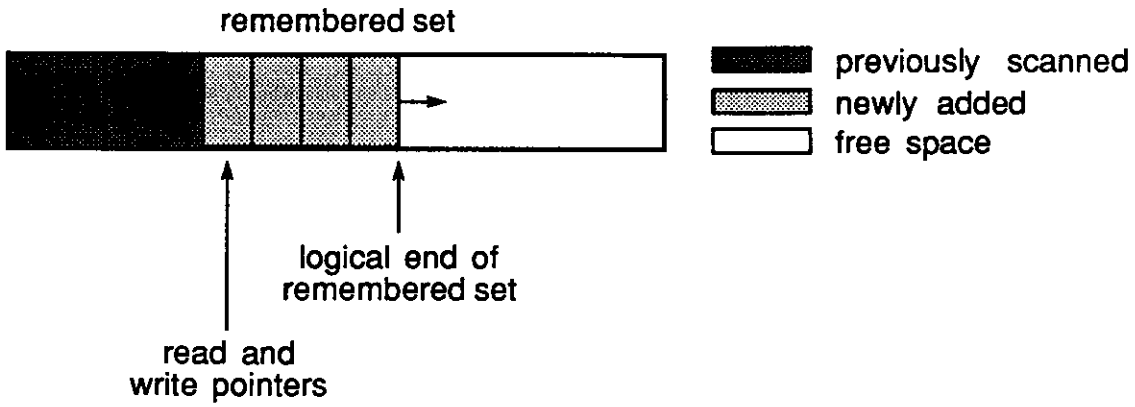
Since the oops of objects not containing new references are not written back, such objects automatically drop out of the remembered set. To completely reverse the remembering process, such objects also have their remembered bits cleared. Because the write pointer is only incremented for every oop written back and not for every read, it may trail the read pointer at the end of the scan, when the read pointer reaches the logical end of the remembered set. That indicates that objects have been removed from the remembered set, so the read and logical end pointers are set to the value of the write pointer to indicate the shrinking of the remembered set.

The contents of *to* space are then scanned to find new objects reachable from objects previously scavenged into *to* space:



At the beginning of the first *to* space scan, the scanning pointer refers to the beginning of the oops region in *to* space. During the scan, the scanning pointer moves oop by oop through the oops region. At each step, the oop under the scanning pointer is read; the corresponding object is scavenged if it resides in *eden* or *from* space. An object scavenged into *to* space is appended at the end, so the end pointer for the oops region may move during the course of the scan. The scan ends when the scanning pointer reaches the end of the oops region.

Since the scan of *to* space may cause objects to be scavenged into *old* space, new objects may have been added to the remembered set:



The read and write pointers were left at the previous end of the remembered set by the previous scan, so the remembered set scan continues from where it left off, scavenging the referents of the newly remembered objects.

The scan of the remembered set may in turn cause objects to be scavenged into *to* space, so the scan of *to* space must likewise be continued following the end of the remembered set scan. The remembered set and *to* space are scanned alternately until no new object is scavenged during a scan. When the alternating scans terminate, all accessible objects have been scavenged. The basic algorithm of Generation Scavenging ends by resetting the logical end pointers in *eden* and *from* space (logically emptying those spaces), then interchanging the *from* and *to* spaces.

5.2.4. Scavenging maps

Do not go gentle into that good night
Old age should burn and rave at close of day;
Rage, rage against the dying of the light.

—Dylan Thomas

Most heap objects die quietly of abandonment, alone and unloved; they are buried en masse without so much as a single eye noticing their individual deaths. Of all

heap objects, only maps receive a better fate. After all of the surviving heap objects have been scavenged but before objects in *eden* and *from* space are swept into oblivion, each expiring map is visited individually so that it may take a final action before perishing.

The reason for the special treatment of maps is that native-code objects contain dependency links to maps; such links must be updated when a map dies. Because dependency links are weak pointers, which do not prevent their referents from being reclaimed, they are not automatically updated at the beginning of scavenging by the traversal of the remembered chain of native-code objects. That traversal updated embedded new oops by causing the corresponding objects to be scavenged to *to* or *old* space and changing the embedded new oops to point to the new locations. Embedded oops updated by that traversal therefore act as strong pointers, which do prevent their referents from being reclaimed by the scavenger. In order to update the weak dependency links, the scavenger visits each dead map to remove it from dependency chains.

A dependency link to a map is a weak pointer because it should not prevent that map from being reclaimed. To see the reason for this point, consider that each native-code object contains a dependency link back to the map defining the slot it was compiled from.† If that back pointer were a strong pointer, it would prevent the indicated map from being reclaimed. That dependency link, however, is necessary only so that the map can be notified if the native-code method has to be removed from the code cache—for example, because the cache is full and storage is needed for a new native-code method. If a map is inaccessible except through dependency links, it may be removed from all the dependency chains and then reclaimed by the scavenger.

† A native-code object that contains inlined code will also have dependency links to the maps containing the slots that hold the inlined methods.

To allow a map to be reclaimed despite the dependency link from the native-code object, the dependency link is represented by a weak pointer, which does not prevent collection. A weak pointer are currently implemented by tagging the pointer as an integer (00 tag). Since integers do not take up any object storage, the scavenger ignores all oops tagged as integers, so dependency links tagged as integers will not cause their referents to be scavenged.

Like other moving collection algorithms, Generation Scavenging touches only live objects, not dead objects. To enable deaths of maps to be detected, all maps in the *new* region are chained together by weak pointers.† After all objects have been scavenged, the new map chain is scanned map by map. If a map has been scavenged, then its map chain link points to a forwarded object, so that link and any back pointers to the map from code space are updated to indicate the map's new location; the map is removed from the new map chain if it has been scavenged to *old* space. If a map chain link points to an unforwarded object in *eden* or *from* space, then the map has died, so the scavenger flushes all dependency links to it and removes it from the new map list.

5.2.5. Demographic feedback-mediated tenuring

As pointed out by Ungar and Jackson [UnJ88], using a fixed tenuring threshold can result in poor performance. If many long-lived objects are born in a clump, they will be scavenged back and forth—producing long pauses—until they finally reach the threshold age. On the other hand, if few objects survive a scavenge, the pause time

† Map deaths could also be detected by chaining together all native-code objects that contain back pointers to new maps. That technique would be slower because the dependency link from a native-code object to a map often consists of more than a single pointer. Because of method inlining (procedure integration), a native-code object depends on all the maps for methods it inlines as well as for its top-level method. Searching through such a dependency chain could involve visiting many more maps than necessary, since some dependent maps could live while others die.

will be short, making tenuring unnecessary. Tenuring the objects anyway when they reach a fixed threshold age may result in tenured garbage if they die soon after being tenured. Experiments by Ungar and Jackson verified the tenuring problem, showing that long-lived objects are indeed produced in clumps during long interactive Smalltalk sessions.

SELF uses essentially the same demographic feedback-mediated tenuring policy advocated by Ungar and Jackson. Rather than being fixed, the tenuring threshold is dynamically changed based upon recent scavenging patterns. When many objects survive a scavenge, the system is presumed to be in a mode of creating many long-lived objects, so the tenuring threshold is be reduced in order to tenure more surviving objects to *old* space; this reduces subsequent pause times because the system will not waste time moving a large clump of long-lived objects back and forth in subsequent scavenging runs. On the other hand, when few objects survive a scavenge, the tenuring threshold is raised to infinity so that no objects are tenured, reducing the possibility of tenured garbage.

The scavenger calculates the proper tenuring threshold based on data stored in the age table, which keeps track of how much data of each age was scavenged during the last scavenge run. The age table is maintained by the scavenger, which updates the appropriate entry whenever an object is scavenged into *to* space. After a scavenging run has been completed, the scavenger examines the age table to determine the tenuring threshold for the next run.

Since pause times are generally proportional to the amount of data scavenged, the tenuring algorithm uses the desired amount of scavenged data as a parameter chosen to produce an acceptable maximum pause time.† The amount of data scavenged into

† The current system uses a desired survivor size of 10 KB. That value is too low, as will be seen in Section 5.6.

to space is determined from the age table. If that amount is less than the desired amount, the tenuring threshold is set to infinity so that the subsequent scavenging run tenures nothing. If more data was scavenged than desired, the scavenger calculates the amount of excess data and consults the age table to determine the tenuring threshold required to tenure the excess data during the next scavenging run. A backwards scan of the age table cumulatively sums the amount of surviving data, starting at the oldest age group and progressing to the youngest. The age group that causes the running sum to exceed the excess data amount indicates the next value for the tenuring threshold.

The age table contains counts of the number of objects for each age, not just the number of bytes as required by the Ungar-Jackson tenuring algorithm. Although unused by the current tenuring algorithm, the object counts were used in a previous version of the memory system, in which the remembered set was stored as an object managed by the scavenger. To avoid resizing the remembered set during a scavenging run, the scavenger used the object age counts along with the tenuring threshold to estimate the amount by which the remembered set to be resized before scavenging commenced. Although the original reason for the age counts is no longer valid because the remembered set has been moved out of the SELF heap, the current system retains the age counts as instrumentation that permits the implementer to see what is going on during scavenging.

5.2.6. Triggering a scavenge

When a memory allocation request cannot be satisfied by free storage in *eden*, the memory system decides to scavenge . Scavenging is not performed immediately, however, because the triggering memory allocation may have occurred deep within a

primitive memory system routine, in which case heap oops may be stored in various C++ local variables. To avoid having to parse and update the C++ stack, the memory system defers scavenging until the primitive routine exits back into SELF code.

To implement the deferred trigger, the memory system finds the return address on the stack where the currently executing primitive returns into SELF code. The address is stored away, and the address of a scavenging wrapper routine replaces it. The primitive then continues execution. Exiting the primitive automatically invokes the scavenging wrapper routine. The wrapper performs scavenging and then returns to SELF code through the original return address. Scavenging is invoked in that manner to avoid the overhead of checking a flag after every primitive invocation.

To enable the primitive to continue execution after storage in *eden* is exhausted, the memory system satisfies the triggering memory allocation and any subsequent allocation requests by trying to allocate storage first from *from* space, then from *to* space, and finally from *old* space. Failure to find enough free storage in any of those spaces currently produces a fatal error.

5.3. Garbage collection

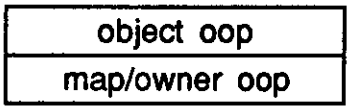
SELF uses a compacting mark-and-sweep garbage collector to reclaim storage in all the object spaces, including *old* space. The mark-and-sweep collector is invoked by the scavenger when insufficient free space remains in *old* space to handle the maximum amount of data that may be tenured during the next scavenging run. The remembered set and new map chain are cleared before the collection; they are reconstructed by the garbage collector during the course of collection.

5.3.1. Object table

The garbage collector uses an indirect object table to relocate object pointers after compaction. The collector changes all pointers to reachable objects into indirect pointers to object table entries which contain the actual object addresses. When an object is moved during compaction, only the single address stored in its object table entry needs to be updated. After all objects have been compacted, the collector scans through the heap to change all references to object table entries back into direct object pointers.

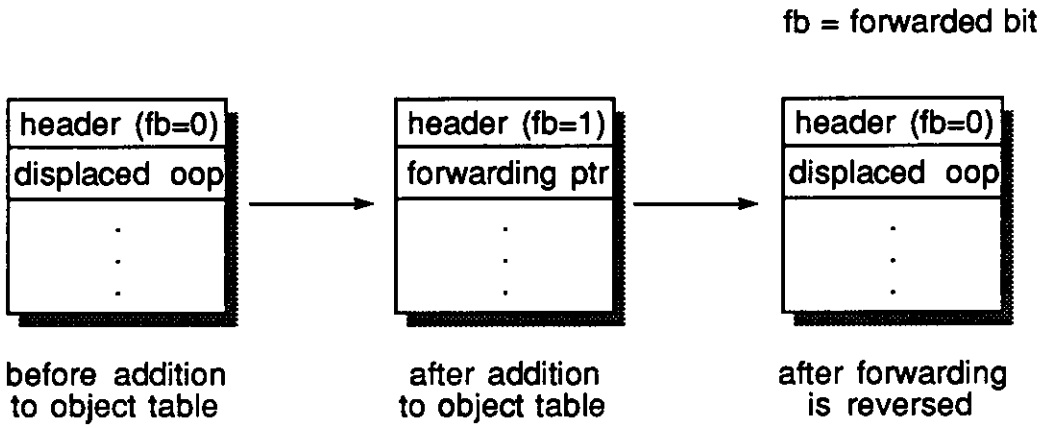
The object table is implemented as an extendible array, allocated from the same pool of storage used by the SELF compiler. It is built anew each time a garbage collection takes place. The garbage collector places each accessible object into the object table and leaves a forwarding pointer to the corresponding object table entry at the original object site.

An object table entry has the following format:



The first data field is the oop of the corresponding object; this field is updated when the object is moved during compaction. The second field contains the oop displaced by the forwarding pointer; the displaced oop is the map for oops objects and the owner for bytes objects. The displaced oop replaces the forwarding pointer when the

forwarding process is reversed after compaction. The layout at the object site during that process is as follows:



Unlike scavenging, in which only oops objects are forwarded, garbage collection causes both oops and bytes objects to be forwarded to the corresponding object table entries. The garbage collector assumes that the object field where the forwarding pointer is stored (currently the second field) must contain an oop. That constraint is indeed satisfied for both oops and bytes objects: the displaced field is the map oop for an oops object and the oop of the owning oops object for a bytes object. If the constraint were not satisfied—for example, because the displaced field coincided with the byte array in a bytes object—then the value of the displaced oop may appear to be an object pointer when it is really random data. In such a case, the current marking algorithm could attempt to access memory indirectly through the displaced oop, resulting in a random memory access and concomitant havoc. Relaxing the constraint would require the marking algorithm to check each object entered into the object table to see whether it contains oops or bytes.

A reference to an object table entry is represented as a direct pointer tagged as a memory oop (01 tag). The garbage collector must be able to distinguish such pointers

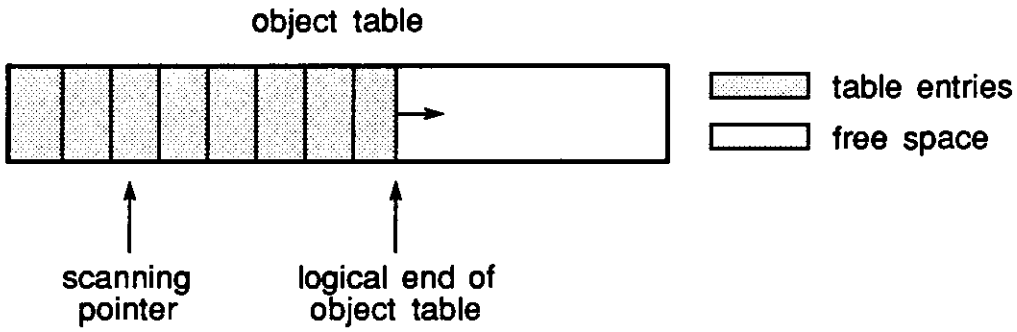
from normal object pointers. Currently, the distinction is that the first word of a normal object pointer is always tagged as a mark, while the first word of an object table entry is never tagged as a mark. An alternate test might be to check the pointer against the bounds of the SELF object spaces; an oop pointing outside of SELF space could be assumed to point to an object table entry. (A check against the object table bounds would be more involved since the object table is currently implemented as a linked list of table pages—each containing multiple table entries—in the C++ heap.)

The simple object table approach to compaction is used rather than a more complex scheme using break tables or Morris-style pointer reversal because the collector is expected to operate in a virtual memory environment, where additional space for the object table is readily available. (For a non-virtual memory system, the object table could be stored in leftover storage in the object spaces.) Additionally, full garbage collection is expected to occur much less frequently than scavenging, so the time required for garbage collection is not critically important. A simple scheme providing reasonable performance was therefore considered preferable to a a more complex scheme providing slightly better performance.

5.3.2. Marking and object table construction phase

The garbage collector uses a single-pass, non-recursive breadth-first algorithm to mark reachable objects and convert embedded object references into object table references. The basic approach is reminiscent of scavenging. Like scavenging, the marking phase uses a scanning algorithm to avoid the overhead of explicit recursion, both in procedure call overhead and in unnecessary control storage. The scanning algorithm also integrates the building of the object table with the marking of reachable objects.

First, the top-level reachable objects are marked and added to the object table. The garbage collector then scans the object table entry by entry:

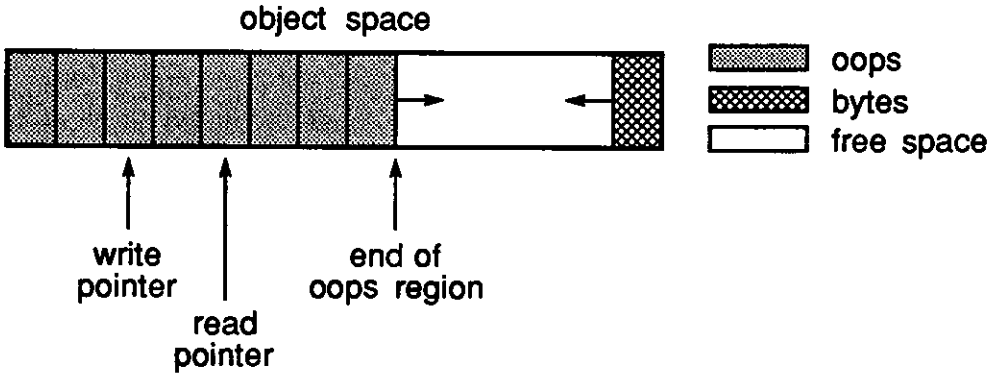


For each entry, the garbage collector scans the oops contained in the corresponding object, including the displaced oop stored in the object table entry. An oop pointing to a forwarded object is updated to indicate the forwarded location, which should be an object table entry. An oop pointing to an unforwarded object denotes a previously unvisited object; the object is marked and added to the end of the object table, and the oop is updated to indicate the corresponding object table entry. The logical end of the object table will therefore move as the scan progresses. When the scanning pointer meets the logical end pointer, all reachable objects will have been marked and added to the object tables, and all oops embedded in those objects will have been converted to point to object table entries.

5.3.3. Compaction

Following the marking phase, the garbage collector compacts each object space using a straightforward two-pointer scanning algorithm; the new-map list is also rebuilt during the course of compaction. The oops and bytes portions of a space are compacted separately, the oops toward the bottom of the space and the bytes toward

the top. The following diagram illustrates the compaction algorithm for oops; compaction for bytes proceeds similarly but in the opposite direction.



The read and write pointers initially are set to the beginning of the space. The read pointer scans object by object through the oops region.

If the read pointer encounters a marked object, the object is copied back to the point indicated by the write pointer and the appropriate object table entry is updated with the new location of the object. If the copied object is a map, its dependency links are adjusted; if the map is new, it is also added to the new map list. Following the copy, the write pointer points to the word after the copy, and the read pointer points to the object after the original.

If the read pointer encounters an unmarked object, that object is dead. For a non-map, no finalization action is necessary so the object can be ignored. As in scavenging, a dead map must be finalized by being removed from all dependency chains. Following any necessary finalization actions, the read pointer advances to the next object.

The process repeats until the read pointer reaches the end of the oops region, at which time the oops region will be completely compacted. The garbage collector sets the end pointer to the value of the write pointer to reclaim the storage between the final read and write pointers.

5.3.4. Relocating oops and reversing forwarding

The final phase of garbage collection scans through the heap, converting object table references back into direct object pointers. The conversion is performed in straightforward fashion by replacing a pointer to an object table entry with the direct object pointer stored in the table entry.

The scan also reverses the forwarding process and rebuilds the remembered set. At each scanned object, the forwarding pointer is replaced with the displaced oop stored in the corresponding object table entry. If the object was previously remembered (detected by checking the remembered bit in the object header), it is added back to the remembered set. The quick remembered bit test is justified because if an object contained referents in the *new* region before garbage collection, it would still contain such referents afterwards—garbage collection moves objects within spaces but not from one space to another.

5.4. Native code

Native code is presently stored in a FIFO cache implemented as a circular buffer. If the code cache is full when a new native-code object is allocated, the cache manager flushes the oldest native-code objects in order until enough space is freed to hold the new native-code object. The cache manager does not flush native-code objects that are currently running.

5.5. Method activation objects

Method activation objects are managed using a LIFO stack discipline. They are stored in machine-oriented form, in register windows on the Sun-4 and on the processor stack on the Sun-3. Non-LIFO method activation objects are not currently supported.

5.6. Performance

This section presents performance figures for automatic storage reclamation in SELF. The performance data was collected for the same set of small SELF benchmarks used for the measurements in Section 4.17. As indicated in that section, we do not consider these benchmarks to be representative of expected system activity. More representative data is unavailable because the SELF environment is not yet sufficiently mature to allow measurements of the long interactive programming sessions that SELF was designed to support. Although unrepresentative, the benchmark data presented here does indicate roughly whether SELF storage reclamation performance falls within the bounds of reasonability. The measurements focus primarily on scavenging because it occurs much more frequently than full mark-and-sweep garbage collection.

5.6.1. Methodology

The time overhead for scavenging has three components: the pause time for each scavenge, the time between scavenges, and the time required to maintain the remembered set. To assess scavenging overhead, all three components were measured for each benchmark. The measurements were performed on a Sun-4/260 with 8 megabytes of memory and a cycle time of 62.5 nanoseconds. (All instructions except memory references and untaken conditional branches execute in a single clock cycle.)

Because of SELF's compilation strategy, the native code for each benchmark is compiled during its first run. To neglect time due to compilation, all measurements were taken after the initial run of each benchmark, when the native code is available in the code cache without need for further compilation.

Most of the data was gathered using timers measuring elapsed user and system CPU time. The timers were implemented using the UNIX™ `getrusage` system call, which has a 10 millisecond time quantum on the Sun-4. The scavenger used two timers, one measuring the pause time for each scavenge and the other measuring the time between scavenges. During the test runs, the scavenger printed out the pause time and interval time for each scavenge, as well as the amount of data scavenged and the amount tenured. A third timer measured the running time for each benchmark.

The time required to maintain the remembered set has three components, each measured separately:

- The code adding an object to the remembered set is written as a C++ procedure, which allows its running time to be measured using the UNIX *gprof* profiling facility.
- The code checking whether an user-level memory store should cause an object to be remembered is compiled in-line for each store. To allow measurement of the time required to perform those in-line checks, the SELF compiler inserted code counting the number of times SELF code executed each check-store instruction. Given the instruction counts, calculating the CPU time for the check-store code is straightforward, since all instructions except loads and stores execute in a single cycle on the SPARC (Sun-4). The calculations assume a cycle time of 62.5 nanoseconds (16 MHz); this time agreed with measured performance from benchmarks designed to measure the cycle time.
- Implementation-level stores are checked using a C++ procedure. The running time of that procedure was measured using the *gprof* profiling facility.

Since both profiling and instruction counting affects the overall running time of each benchmark, those measurements were performed separately from the measurements using CPU timers. In addition, to separate the effects of garbage collection and scavenging, the size of *old* space was raised to 12 megabytes to avoid a garbage collection from taking place during a benchmark run.

Each benchmark measurement encompasses multiple runs of the underlying benchmarks. The use of multiple runs is necessary because most of the benchmarks are quite short and would not scavenge during the course of a single run. In addition, the time quantum for the profiling facility is only 10 ms, so multiple runs are necessary to produce a running time long enough for the meaningful profiling measurements.

To gauge the effects of the desired survivor size (used by the tenuring algorithm) on scavenging, each measurement was performed twice, once with a desired survivor size of 10 kilobytes and once with a desired survivor size of 80 kilobytes.

The running time of an example garbage collection was measured to indicate the performance of mark-and-sweep garbage collection. The garbage collection is invoked after a run of all the SELF benchmarks. This measurement cannot be used to predict normal garbage collection performance since the amount of object storage is small (around 2 megabytes), with the result that the garbage collection can operate without paging. A full SELF programming environment is likely to take more object storage, which would increase the likelihood of paging and resultant longer garbage collection times.

5.6.2. Results

This section summarizes the results of the performance measurements. Appendix B presents the results of the measurements in more detail.

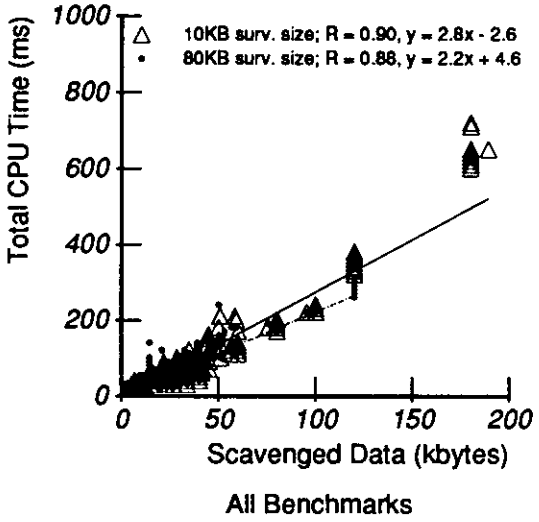
5.6.2.1. Scavenging performance

The following table summarizes the performance of scavenging in SELF: the total scavenging overhead, the length of scavenging pauses, and the amount of time between scavenges. The eleven benchmarks are divided into four groups in the presentation. The *richards* benchmark stands by itself because we regard it to be most representative. The two floating point benchmarks are placed into a single group to indicate the effect of floating-point numbers on scavenging. Of the remaining eight benchmarks, only one (*tree*) stands out as performing significant amounts of object allocation; the other seven are grouped together as integer benchmarks. The performance for a group of benchmarks is expressed as a range of the times for the benchmarks in that group.

BENCHMARK	SCAV. OVERHEAD (%)		SCAVENGING PAUSES (MS)		SCAV. INTERVALS (SEC)	
	10 KB SURV.	80 KB SURV.	10 KB SURV.	80 KB SURV.	10 KB SURV.	80 KB SURV.
richards	3.7	3.7	20-30	30	180-200	180-200
tree	17.	13.	50-720	50-350	1.1-1.7	1.1-1.6
integer	0.7-4.8	0.7-4.8	0-130	20-140	3-240	3-240
float	8.7-19.	16.-19.	0-210	0-240	0.19-0.33	0.19-0.68

5.6.2.2. Correlation of pause times to scavenged data

To keep pause times bounded, the scavenger attempts to scavenge a target amount of data (the desired survivor size) each time it runs. That approach assumes that scavenging pause time depends linearly on the amount of data scavenged. The following scatter plot shows that the relationship of scavenged data to pause times is indeed linear over all the benchmarks (3652 scavenges for each desired survivor size).



5.6.2.3. Garbage collection performance

The time taken to perform an example mark-and-sweep garbage collection is presented in the following table.

user time (sec)	2.9 - 3.1
system time (sec)	0.06 - 0.15
real time	3.1 - 3.2
CPU utilization	99% - 100%
page faults	0 - 1
initial storage (bytes)	1916372
final storage (bytes)	1353448

5.6.3. Discussion of performance measurements

Because the benchmarks are not representative of expected system activity, the benchmark measurements serve only as a very rough estimate of the performance of the SELF storage reclamation facility. Overall, pause times and scavenging overhead appear to be reasonable for the scavenger, and the mark-and-sweep garbage collector seems to perform acceptably. A good part of the credit for the generally low scavenging overhead should go to the SELF compiler, which in many cases can avoid allocating object storage for blocks by compiling them in-line. The performance figures do, however, indicate a number of problems with current memory system.

The floating point benchmarks, *mm* and *fft*, provide two of the three worst scavenging performances, pointing out the serious performance problems posed by the representation of floating point numbers as normal SELF objects. The two floating point benchmarks are almost pathological cases for the scavenger because they involve large arrays of floating point numbers; such numbers must be preserved during a scavenge, but they die soon afterwards as the programs calculate new values for the array elements. The result is high scavenging frequency and a large amount of tenured data, most of which becomes garbage. Increasing the desired survivor size from 10 kilobytes to 80 kilobytes alleviates the tenured garbage problem for *fft* but not for *mm*. The most attractive solution for the floating point problem is to provide an immediate representation for small floating point numbers analogous to the current immediate representation of small integers. That would eliminate almost all the scavenging overhead in the floating point benchmarks.

Another problem is the high store-checking overhead (4% - 5%) in *richards*, *perm*, and *towers*. Part of the reason for the high overhead is that the current store-checking code does not fill delay slots optimally. As a result, the store-checking code

executes three instructions in the common case where the receiver is new, as opposed to two instructions if the delay slots in the store-checking code were filled optimally. With the addition of a peephole optimizer to fill delay slots in the store-checking code, the store-checking overhead for *richards*, *perm*, and *towers* would drop to 2% - 3%, which is comparable to the worst store-checking times reported by Ungar [Ung86] (2.96% for *testCompiler* and 1.76% for *testDecompiler*).

The current desired survivor size of 10 kilobytes should probably be increased. Aside from *mm*, which jumped from 9% to 19% scavenging overhead, none of the benchmarks showed unacceptable increases in pause times when the desired survivor size was increased to 80 kilobytes; in fact, pause times for the second worst scavenging performer (*tree*) decreased from 17% to 13%. Although the increased desired survivor size severely increased scavenging overhead for *mm*, that overhead could be addressed by implementing immediate representations for floating point objects. The benefits of increased desired survivor size can be seen in the *tree* and *fft* benchmarks, which show dramatic reductions in the amount of tenured data (50% and 89% respectively). Such reductions would decrease the overall overhead of storage reclamation because the mark-and-sweep garbage collector would not have to run as frequently.

5.7. Summary

SELF uses a version of Ungar's Generation Scavenging algorithm as the principal mechanism to reclaim storage. Generation Scavenging carries low overhead, small pause times, and good virtual memory performance. A compacting mark-and-sweep garbage collector is used to reclaim tenured garbage.

Chapter 6

Inheritance and Message Lookup

Who, if I shouted, among the hierarchy of angels would hear me?

—Rainer Maria Rilke
(translation by C.F. MacIntyre)

6.1. Introduction

SELF provides a rich and flexible set of inheritance mechanisms: ordered and unordered multiple inheritance, explicit delegation, and dynamic inheritance. Wealth does not come without a price, however—performing a message lookup in SELF can require a substantial amount of computation. To avoid the expense of run-time message lookup while retaining the advantages of dynamic binding, SELF joins recent high-performance Smalltalk systems [DeS84, Ung86] in using an implementation technique called in-line method caching.† This chapter describes the implementation of inheritance mechanisms in SELF and examines how the in-line method cache, originally designed for a class-based language, fits into an object inheritance framework.

6.2. Performing a message lookup

Section 3.3.6 described inheritance semantics on an abstract level. This section examines in more detail how the implementation finds the proper method to invoke when an object receives a message at run time.

---

† The SELF compiler also performs some compile-time optimizations to avoid run-time method searches when possible. Compile-time optimizations are beyond the scope of this thesis; more information on them may be found in Craig Chambers's forthcoming dissertation.

6.2.1. Lookup parameters

When SELF code calls for a message lookup, it passes the lookup entry routine six parameters: the message's receiver, message name, lookup frame (current method activation frame), type of message send (normal, implicit-receiver, super, or delegated), delegate's slot name (for delegated sends only), and method holder (object with a slot holding the method that sent the message). Those six parameters, along with the inheritance hierarchy, completely determine the outcome of the method search. The following table indicates which parameters affect the outcome of the various types of method search†:

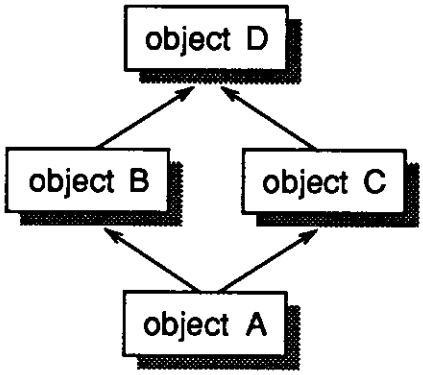
METHOD SEARCH TYPE	LOOKUP PARAMETERS					
	MESSAGE RECEIVER	MESSAGE NAME	LOOKUP FRAME	MESSAGE TYPE	DELEGATE'S SLOT NAME	METHOD HOLDER
normal	•	•		•		
implicit-receiver		•	•	•		•
delegated		•		•	•	•
super		•		•		•

Only the receive and lookup frame are passed dynamically to the lookup entry routine. The other four parameters are known at compile time and so are compiled statically into the native-code object that invoked the message lookup; the lookup entry routine will look back at the call site to pick them up. As will be seen later, passing the lookup parameters in this fashion reduces the cost of an in-line method cache hit.

† Although a parameter may not directly affect the outcome of a particular method search, it may still be needed by the lookup routine. The receiver, for example, is passed to the compiler by the lookup routine when a looked-up method is compiled into native code; and the lookup frame is needed to retrieve the values of the message name, message type, delegate's slot name, and method holder.

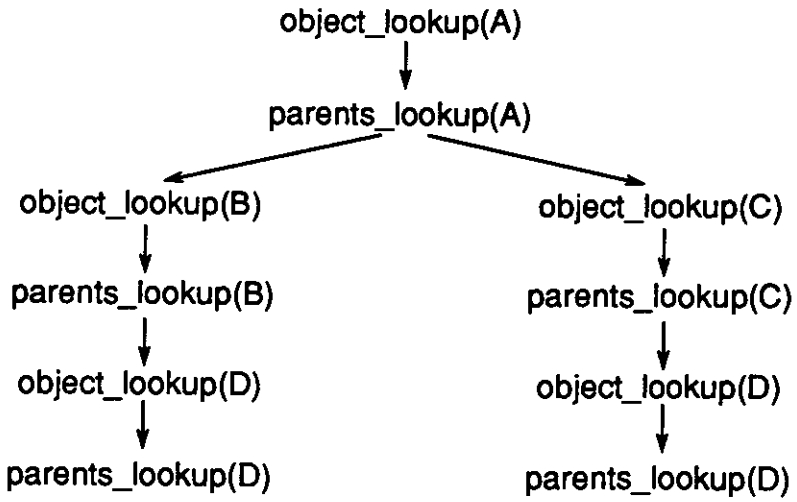
6.2.2. Main lookup cycle

The mutually recursive routines `object_lookup()` and `parents_lookup()` form the body of the main lookup cycle. `object_lookup()` searches the lookup start object for a slot matching the message name. If no matching slot is found, it invokes `parents_lookup()` to search the parents of the lookup start. `parents_lookup()` operates by calling `object_lookup()` on the parents of the lookup start and by resolving any ensuing conflicts if more than one matching slot is found. As an example, consider the following inheritance hierarchy:



inheritance hierarchy

The following diagram illustrates the call graph for an unsuccessful method search beginning at object A:



As may be seen from the call graph, the object D is searched twice, once along each parent path. The present implementation of message lookup does not maintain a record of previously searched nodes, so objects found along more than one parent path may be redundantly searched. Although such redundant searches may appear to be undesirable, they are not a performance problem in practice.

In addition to the lookup start argument, both `object_lookup()` and `parents_lookup()` take a sender path flag, which is used to implement the sender path constraint for unordered inheritance. If two matching slots are found along different parent paths, where one slot lies on a sender path (an inheritance path containing the method holder) and the other does not, then the slot on a sender path overrides the one not on a sender path. To avoid redundant searching, the sender path flag indicates only whether the search path up to the current search point lies on a sender path. When the search reaches the method holder, the sender path flag is set to true so that all objects above the method holder would be considered on a sender path. If a matching slot is found before the method holder has been visited on the matching path, the lookup determines whether the slot is on a sender path by searching above the match for the method holder.

The sender path constraint is only used for implicit-receiver, delegated, and super message sends. The message lookup for a normal message send is performed by starting the lookup cycle with the sender path flag already set to true, thereby treating all matching slots as lying on a sender path.

6.2.3. Effects of the message type on a lookup

SELF supports four types of message sends: normal, implicit-receiver, super, and delegated. As far as the method search algorithm is concerned, the different types of message sends matter only in determining the entry point and initial arguments

(lookup starting point and sender path flag) for the main lookup cycle. Once the main lookup cycle is entered, the message types are never consulted again during the course of the method search.

The following table summarizes the effect of the message type on a lookup.

MESSAGE SEND TYPE	ENTRY POINT	SENDER PATH FLAG	LOOKUP START
normal	object_lookup()	true	receiver
implicit-receiver	object_lookup()	false	current activation object (lookup frame)
delegated	object_lookup()	false	contents of delegate slot in method holder
super	parents_lookup()	false	method holder

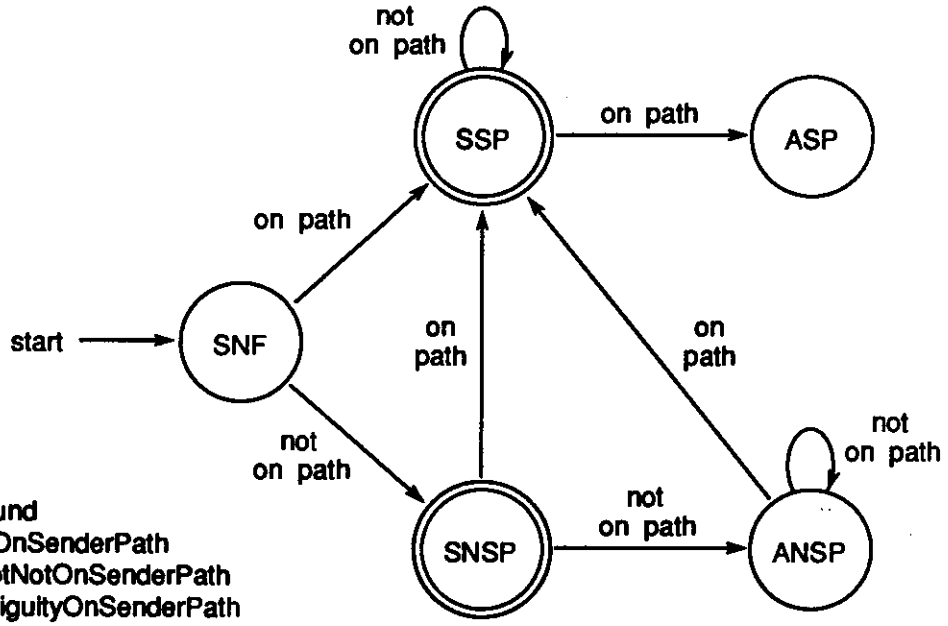
As may be seen from the table entries, not every combination of entry point, sender path flag, and lookup start represents a legal SELF message send. For example, an entry point of `parents_lookup()`, a false sender path flag, and a lookup starting at the receiver would correspond to a `super` send directed towards an explicit receiver. Such a send is illegal in SELF because it would reduce the control an object has over its message protocol: it would allow an external object to bypass an overriding method defined in the receiver, directly invoking the overridden method in the receiver's ancestors.

6.2.4. Resolving unordered inheritance conflicts

In a message lookup involving unordered multiple inheritance, finding a matching slot along one parent path does not automatically terminate the lookup because searching an alternate parent path may produce another matching slot. The second match can either override the first match, be overridden by it, or produce an ambiguity; which of those actions should be taken is determined by the previous state of the lookup and by whether the matches lie on sender paths.

The lookup algorithm includes a finite state machine to resolve unordered inheritance conflicts. A state transition is triggered each time a matching slot is found. The state machine takes as input a boolean value indicating whether the newly found slot lies on a sender path; output actions keep track of the final result of the lookup. Five lookup states are used: slotNotFound, foundSlotOnSenderPath, foundSlotNotOnSenderPath, foundAmbiguityOnSenderPath, and foundAmbiguityNotOnSenderPath.† The lookup states express not only the number of slots found so far but whether those slots were found along a sender path. The sender path distinctions allow the lookup algorithm to resolve slot conflicts using SELF's sender path constraint for unordered inheritance.

The following diagram illustrates the state transitions:



No outward state transitions are specified for the ASP state because the message is

† The lookup routines actually use two additional states for error handling; those two states are neglected because they do not directly affect the main lookup algorithm.

known to be ambiguous at that point; the lookup algorithm will terminate and indicate the ambiguity without searching for any more matching slots. The states enclosed in double circles are accepting states; a lookup ending in one of those states will successfully return a single matching slot for the message send.

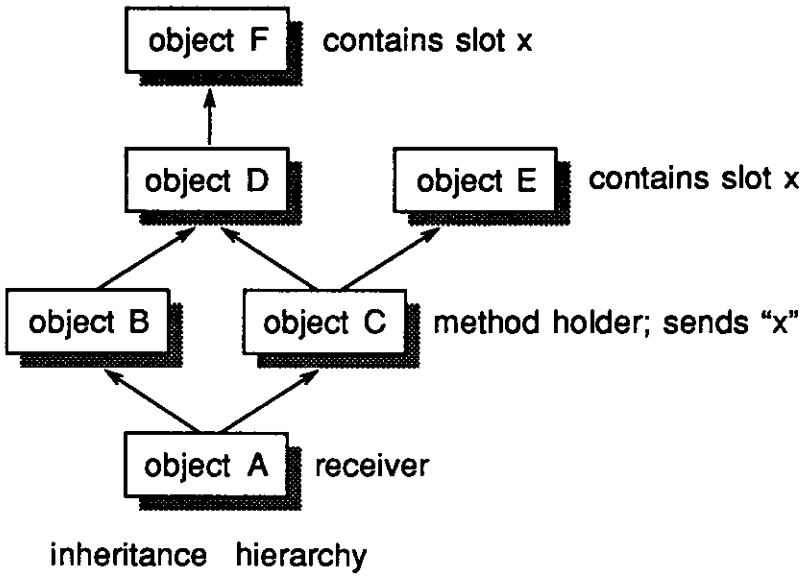
Output actions for the state transitions keep track of the result for the message lookup. On any transition entering the SSP or SNSP states, the lookup algorithm sets the lookup result to be the slot causing the transition. On all other transitions, the lookup result becomes the null slot. The state machine does not express the rule that finding the same slot through two different inheritance paths does not produce an ambiguity. That rule is implemented by comparing a newfound slot against the previous lookup result. The state machine is triggered only if the new slot is not the same as the previous result.

One subtlety in the state machine is the need for an ANSP state separate from the ASP state. Because the sender path constraint allows a slot on a sender path to override slots not on a sender path, the lookup algorithm cannot simply terminate when it detects an unresolvable conflict between two non-sender-path slots. On the other hand, the conflict cannot be ignored because it will cause the lookup to fail if no sender-path slot is subsequently found. The ANSP state flags an ambiguity that may be subsequently ignored if a matching sender-path slot is eventually found.

The sender path constraint is also responsible for a subtlety involving redundant object searches. As previously noted, the current lookup algorithm does not maintain a record of previously visited objects and so may search some objects many times during the course of a single lookup. One possible optimization might be to eliminate redundant searches by marking a bit for each visited object and only searching unmarked objects; such marking could be integrated with cycle detection. Although

such a scheme might be infeasible because of performance problems (such as the need to reset all the bits at the end), a more serious and less obvious problem is that it would result in a incorrect implementation of SELF's unordered inheritance semantics.

Consider the following scenario:



In this example, object C is the method holder when the implicit-receiver message "x" is sent with object A as the receiver. According to the SELF inheritance semantics, that message is ambiguous because both object F and object E contain matching slot, both on sender paths (A-C-D-F and A-C-E). If the simple marking optimization were in effect, however, that ambiguity would not be detected. Object D would be marked during the search A-B-D-F, which would have found the x slot in F as a non-sender-path match. The A-C-D-F match would not be found because the lookup would not search D a second time, thinking it redundant. The x slot would then be found on the A-C-E path, which is a sender path and which therefore overrides the A-B-D-F match. The x slot in E would then be incorrectly returned as the result of the message lookup.

Redundant searches could be correctly eliminated if the algorithm uses an additional bit for each object to indicate whether a match lies within or above it. We did not implement such a scheme because the complexity does not seem justified by possible performance gains, even if there were a way to quickly reset the bits at the end of the lookup. Redundant searches do not appear to be a performance problem in the current implementation.

6.2.5. Resolving ordered inheritance conflicts

Resolving conflicts for ordered inheritance is a simpler task than for the unordered case. Since the priority for a parent slot is contained in its slot descriptor, the lookup algorithm can quickly determine parent priorities and then search parent paths in order of decreasing priority. After finding a matching slot, the lookup does not need to search parent paths of lesser priority, since the slot would override any slots subsequently found.

To make ordered inheritance simple to implement, the memory system stores all parent slot descriptors for an object consecutively in the map and in decreasing order of priority. The lookup algorithm operates by searching each parent in order. When searching consecutive parents of the same priority, the lookup uses the previously mentioned state machine approach to resolve conflicts using the unordered inheritance rules. The search only continues to a parent group of lower priority if the unordered inheritance search of the previous group failed to find either a matching slot or an ambiguity.

6.2.6. Searching an object for a matching slot

The lookup algorithm searches an object for a matching slot by scanning linearly through the slot descriptors in the object's map. More sophisticated schemes were

not adopted for reasons of simplicity. In addition, the current ordering of slot descriptors (parent slots stored consecutively in decreasing order of priority) precludes algorithms like hashing that require different slot ordering.

6.2.7. Detecting inheritance cycles

SELF inheritance semantics permit the inheritance graph to contain cycles. To detect and break cycles during a message lookup, the lookup algorithm sets the cycle-detection bit in every object on the current search path. If the lookup reaches an object with the cycle-detection bit already set, it breaks the cycle by ignoring the object and returning to the preceding object in the search path. When an object leaves the current search path, its cycle-detection bit is cleared.

The cycle-detection bit only marks objects on the current lookup path, not all objects visited during the course of the lookup. For that reason, it does not prevent the lookup from redundantly searching objects that are accessible through multiple parent paths. An alternative would be to mark all objects on the whole search tree, rather than just the current search path. That would allow the cycle-detection algorithm to also prevent redundant searches; as discussed earlier, however, such an algorithm would also require an additional bit and possibly greater expense in revisiting searched objects to clear all the bits at the end of lookup.

6.2.8. Ending a lookup

If a message lookup ends successfully with a single matching slot, the contents of that slot are retrieved, and the corresponding native-code is generated if necessary and then evaluated. Otherwise, an error is indicating by sending an error message (`messageNotUnderstood:`, `messageAmbiguous`, or `delegateeNotFound:`) to the root; the name of the erroneous message is passed as the sole argument to the

error message. If the error message is itself unsuccessfully looked up, an internal C++-level error routine is called.

6.2.9. Discussion

Because of the sender path constraint, a message lookup involving unordered multiple inheritance can be more expensive than in other languages, which typically regard multiple matching slots as ambiguous. In SELF, the message lookup cannot abort after finding two conflicting non-sender-path slots, since further searching may produce a matching sender-path slot that overrides the previous conflict.

Additionally, the sender path constraint may require a method search to continue above an object containing a matching slot, since if the sender path flag is false at the point of the match, the message lookup must visit the ancestors of that object in order to tell if the matching slot lies on a sender path. In the current implementation, the message lookup performs such a sender-path check whether or not one is actually necessary. The lookup algorithm could eliminate such unnecessary checks, at the cost of somewhat greater complexity, by deferring the sender-path check until a second matching slot is found and the sender-path information is needed to resolve the conflict.

6.3. In-line method cache

Performing a full message lookup for every message send is both expensive and unnecessary. To reduce the number of full message lookups, SELF follows the PS [DeS84] and SOAR [Ung86] Smalltalk systems in using global and in-line method caches.

The global method cache is a hash table containing native code objects. The lookup routine checks the global method cache before doing a full message lookup. If

a native-code object in the global cache matches the lookup parameters, then the lookup routine returns it as the result of the lookup, avoiding an expensive method search. The lookup routine maintains the global method cache by entering lookup results into it.

The in-line method cache is based on the observation that a message send will almost always bind to the same method from one execution to the next. To take advantage of that observation, the message lookup caches the result of the lookup in-line at the call site, overwriting the call to the lookup routine with a direct call to the method. The next time that particular message send is encountered, the method is immediately invoked without a message lookup. The prologue of each method must check the validity of the in-line cache before proceeding to the body of the method. If the cache is not valid, the prologue will invoke the lookup routine, which first checks the global method cache and, if that fails, then performs a full message lookup to rebind the message send.

6.3.1. Handling state access

To prime the in-line method cache in PS and SOAR, the message lookup overwrites the address of the lookup routine at the call site with the entry address for the native code routine implementing the newly looked-up method. The SELF in-line method cache is primed similarly when the lookup result is a dynamic object, which carries explicit SELF code.

Unlike Smalltalk, SELF uses message sends to access object state as well as to invoke dynamic methods, so a message lookup may result in a static data object or an assignment primitive, neither of which is directly associated with a native-code method. To handle a data access, the message lookup generates a native-code data

access routine and caches the address of that routine in the in-line and global method caches. When the cache entries are valid, the original and present receivers should share the same format, so the data access code for the original receiver will also operate properly for the current one. (The SELF compiler also performs compile-time optimizations to efficiently implement many state accesses without resorting to either method cache.)

6.3.2. Checking cache validity

In Smalltalk, an object's class holds all its local methods and completely determines its inheritance characteristics, so a Smalltalk method prologue can check the validity of the in-line method cache by comparing the class of the current receiver with the class of the last receiver, stored at the call site for the message send. If the classes are the same, then the message send will bind to the same method for both the past and present receivers, and so the in-line method cache is valid. If the classes are not the same, then the message send could bind differently, so a full message lookup is invoked. A probe of the global method cache can similarly compare the original class, message name, and lookup type with the present ones.

Since SELF does not have classes, the validity check must have a different basis. Although SELF objects do not belong to classes, they do belong to clone families, whose members typically have similar inheritance characteristics. A SELF method prologue therefore compares the maps of the past and present receivers. If the maps are the same, the objects are assumed to have identical inheritance characteristics, and the in-line cache is considered to be valid. (All integers are considered to share the same map.) If the past and present receiver maps are not the same, the global method cache is consulting, keying on the receiver map, message name, lookup type,

and delegate's slot name (if any). A full message lookup is performed if the no matching native-code object is found in the global method cache.

Unlike a class, a map does not completely determine the inheritance characteristics for its associated objects, so a cache validity check based on map identity is not always correct. Only read-only data shared by all members of the clone family are stored in the map; the contents of assignable slots are individually stored in each object. A purely map-based cache validity check may fail if the receiver or any of its ancestors have assignable parent slots, since the maps of the past and present receivers may be the same even though the values of their corresponding assignable parent slots differ.

The current implementation uses a purely map-based validity check but does not cache the lookup result in either the in-line or global method cache if the any object on the method search path includes an assignable parent. Full message lookups are therefore performed for all messages whose bindings depend on assignable parent slots. That approach is unacceptably expensive. One possible solution might be to check the values of assignable parents as well as the map in the cache validation code of the method prologue. Another solution might be to ensure that objects with different inheritance characteristics have different maps. Further research needs to be done on efficient implementation of dynamic inheritance.

The current map-based validity check also requires all dynamic objects (procedures) to be stored in maps. If that requirement were not in effect, two objects could share the same map but contain different dynamic methods in corresponding slots. The same message sent to both objects could then bind to different dynamic methods even though the maps would be the same. Static data objects can be stored

into individual objects without that difficulty because the same data access code would handle the message for both receivers, even though the slot contents differ.

To ensure that the above condition is satisfied, SELF requires dynamic objects to be stored only in read-only slots. That requirement is not normally a difficulty, since a SELF program can never pass a dynamic object as an argument to an assignment or any other kind of message—aside from literals, the only way to refer to an object is to access it through a slot, and a dynamic object is immediately executed when a message unambiguously matches a slot containing it. (Reflection primitives will allow programs such as a browser to manipulate dynamic objects.)

6.3.3. Performance of the in-line method cache

Although the in-line method cache has performed well in Smalltalk systems, its success in a class-based system may not carry over to a prototype-based system like SELF. To gauge its applicability to SELF, I measured its performance for the *richards* benchmark. The *richards* benchmark was chosen for measurement because it was the only benchmark that uses polymorphism to any significant extent, necessary to adequately exercise the in-line method cache. To measure the cache performance, I instrumented the compiler to insert instructions counting the occurrences of instructions in the cache validation code in method prologues. Such instruction counts allow the calculation of the time spent probing the in-line method cache. To measure probes of the global method cache, which invoke C++ procedures, I used the *gprof* profiler [GKM82] to determine the amount of time spent in those procedures.

The *richards* benchmark averaged a message send every 300 nanoseconds on the Sun-4/280 (ignoring references and assignments to local variables and arguments). The SELF compiler was able to statically bind 92% of those message sends; most of

those sends do not incur any procedure overhead. Of the 8% not statically bound, only 4% of those message sends (0.4% of all sends) missed the in-line method cache. All of the message sends missing the in-line method cache produced cache hits in the global method cache, so no run-time lookups were performed for the *richards* benchmark. Although infrequent, cache misses are expensive. Probes of the in-line method cache accounted for 14% of total run time and probes of the global method cache (on in-line cache misses) accounted for 11%. The total overhead for polymorphism in *richards* was therefore 25%, which is comparable to the 23% overhead reported for SOAR [Ung86].

6.4. Pseudo-code description of the lookup algorithm

The core of the lookup algorithm is presented below in pseudo C++ code. The bulk of the algorithm rests in the two mutually recursive routines: `object_lookup` (which searches an object for a matching slot) and `parents_lookup` (which searches an object's ancestors for a matching slot). Depending on the message type, either one of those two routines could be the entry point to the main lookup cycle.

In the following code, the parameters for the method search are assumed to have been stored into instance variables of a `lookup` instance; the lookup routines operate as member functions of the `lookup` class. The `status` instance variable is initially `slotNotFound`, and the `found_assignable_parent` flag is initially false.

For clarity, distinctions between objects stored in the heap and method activation objects stored on the stack are omitted. C++ storage management statements (e.g., statements freeing slotRef storage) are also omitted.

```
class lookup {
    oop receiver;
    oop delegate;
    oop message_name;
    lookupType message_type;
    oop method_holder;
    slotRef* result;
    boolean found_assignable_parent;
    lookupStatus status;
};
class object;
typedef object *oop;
boolean lookup::object_lookup(oop lookup_start, boolean on_sender_path) {
    if lookup_start->in_cycle() then return result.  // break cycle
    if lookup_start equals method_holder then:
        on_sender_path ← true.  // on_sender_path is passed by value
    find a slot s matching message_name in lookup_start.
    if a matching slot s exists then:
        if not on_sender_path then:
            if arg1_is_ancestor_of_arg2(method_holder, lookup_start) then:
                    on_sender_path ← true.
        update_lookup_state(s, on_sender_path).
        found_slot ← true.
    else:
        found_slot ← parents_lookup(lookup_start, on_sender_path).
    lookup_start->clear_cycle_mark().
    return found_slot.
}
boolean lookup::parents_lookup(oop lookup_start, boolean on_sender_path) {
    set inherited bit in p.     // for dependencies; see Section 4.11.6
    found_slot ← false.
    for each parentPriorityGroup g in lookup_start do:
        for each parent p of priority g in lookup_start do:
            if p is assignable then:
                found_assignable_parent ← true.
            found_slot ← object_lookup(p, on_sender_path) or found_slot.
            if status equals foundAmbiguityOnSenderPath then:
                return found_slot.  // message could not be resolved.
        if found_slot then:
            break out of for loop.
    return found_slot.
}
```

```
void lookup::update_lookup_state(slotRef* s, boolean on_sender_path) {
    switch (status) {
        case slotNotFound:
            if on_sender_path then:
                status ← foundSlotOnSenderPath.
            else:
                status ← foundSlotNotOnSenderPath.
            result ← s.
            break out of switch statement.
        case foundSlotNotOnSenderPath:
            if on_sender_path then:
                status ← foundSlotOnSenderPath.
                result ← s.
            else if s is not equal to result then:
                status ← foundAmbiguityNotOnSenderPath.
                result ← null slotRef.
            break out of switch statement.
        case foundSlotOnSenderPath:
            if on_sender_path and s is not equal to result then:
                status ← foundAmbiguityOnSenderPath.
                result ← null slotRef.
            break out of switch statement.
        case foundAmbiguityNotOnSenderPath:
            if on_sender_path then:
                status ← foundSlotOnSenderPath.
                result ← s.
            break out of switch statement.
    }
}
```

```
boolean lookup::arg1_is_ancestor_of_arg2(oop desired_ancestor, oop base_object) {
    if desired_ancestor equals base_object then:
        return true.
    for each parent p of base_object do:
        if p is an assignable parent then:
            found_assignable_parent ← true.
        if not p->in_cycle() then:
            if arg1_is_ancestor_of_arg2(desired_ancestor, p) then:
                p->clear_cycle_mark().
                return true.
        p->clear_cycle_mark().
    return false.
}
boolean object::in_cycle() {
    if cycle bit is set for this object then:
        return true.
    set cycle bit for this object.
    return false.
}
void object::clear_cycle_mark() {
    clear cycle bit for this object.
}
```

6.5. Summary

This chapter has described a message lookup algorithm implementing the SELF inheritance semantics. A finite state machine is used to resolve unordered inheritance conflicts, and a priority-based parent search order handles conflicts involving ordered inheritance.

Because a full message lookup can require a substantial amount of computation, SELF uses an in-line method cache to avoid repeating message lookups unnecessarily. The current implementation relies on a simple map comparison to check the validity of the in-line cache. Since that scheme does not correctly handle methods inherited through an assignable parent, full message lookups are performed for all message sends whose binding depends on assignable parents.

Chapter 7

Conclusions

We have implemented object storage and inheritance mechanisms for SELF, a prototype-based programming language. By creating a high-performance implementation of SELF, we hoped to answer two questions regarding the implementation of a prototype-based language:

- Can a prototype-based object-oriented programming language be as space- and time-efficient as a class-based language?
- How well do implementation techniques originally designed for class-based languages carry over to a prototype-based language?

Our implementation experiences have shown that a prototype-based language can be implemented as efficiently as a class-based language, both in terms of time and space. Many of the same implementation techniques apply equally well to class-based and prototype-based languages because both types of languages face many of the same implementation issues. Particular examples in our system were the use of generation scavenging to keep storage reclamation costs low and the use of in-line and global method caches to reduce the costs of dynamic binding of polymorphic messages.

Efficient implementation of prototypes required some new techniques as well. One innovation in our system was the introduction of maps to hold information shared by objects related through cloning. We observed that objects related through cloning form a clone family, whose members share read-only information like object format and the contents of read-only slots. Our implementation factors out all read-only information for a clone family into a map, which determines the shared characteristics of the clone family. In particular, an object's static inheritance characteristics, which are stored in read-only slots, are shared by

all members of its clone family. Maps differ from classes in being solely an implementation optimization that is not visible to the SELF programmer and that does not restrict the flexibility of the prototype model.

Maps are a key element of our design because they play many of the same implementation roles as classes in a class-based system. By storing shared format information for a clone family, maps provide our prototype-based object memory with comparable storage efficiency to a class-based system. Maps also enable use of method caching. Because maps represent the static inheritance characteristics for members of their clone families, identical messages sent to objects with the same maps should bind to the same routine if dynamic inheritance is not in use. Maps, like classes, therefore serve as the lookup keys for probes of the in-line method cache. Although transparent to the user, maps factor object attributes similarly to classes, allowing our implementation to apply class-based techniques to prototypes.

Prototype-based languages present an implementation challenge not found in class-based languages: dynamic inheritance. When objects can change their inheritance at run time, maps cannot completely specify the inheritance characteristics of their clone families and so cannot act as simple lookup keys for the in-line method cache. The result is that our simple adaptation of in-line method caching does not work for objects whose inheritance characteristics can dynamically. Efficient implementation of dynamic inheritance is still an open issue.

SELF's multiple inheritance semantics require a more complex lookup algorithm than in other language. Much of the extra complexity is due to the sender path constraint, which causes the lookup to maintain a sender path flag and at times to search more objects than necessary under other inheritance semantics. Despite the possibility of more expensive message lookups, in practice we have not found SELF's inheritance semantics to carry

significant run-time overhead, in large measure due to the efficiency of the compiler and the method caches in eliminating run-time method searches.

Overall, our implementation has been successful. We have produced a high-performance implementation of a prototype-based object-oriented programming that is comparable in efficiency to high-performance implementations of class-based languages like Smalltalk. Potential users of prototype-based programming languages need not fear that the greater flexibility of prototypes over classes carries a higher cost at run time.

Future work

A number of implementation issues remain to be addressed. The largest issue is the efficient implementation of dynamic inheritance. Dynamic inheritance is a powerful but new language mechanism. An efficient implementation of dynamic inheritance is essential to enable us to explore its uses.

An open question for systems employing generation scavenging is how objects with intermediate lifetimes should be managed. That issue is likely to become more important as the SELF programming environment becomes larger and more mature.

There are a few holes in our implementation. The current implementation of SELF does not support non-LIFO blocks. Although future implementations will implement some form of non-LIFO blocks, the exact semantics of such blocks are still under discussion. Possibilities include Smalltalk-style blocks and blocks as full continuations. The lack of multitasking in the current system is another deficiency that must be addressed before the full SELF programming environment can be implemented.

Appendix A

SELF Syntax

for life's not a paragraph
And death i think is no parenthesis

—e. e. cummings

A.1. Syntax introduction

This appendix contains a description of SELF syntax. An early version of the syntax was specified in the original SELF paper by Ungar and Smith [UnS87]. This appendix incorporates subsequent changes to the language designed by the SELF research group (David Ungar, Craig Chambers, and Elgin Lee) and implemented by Craig Chambers in the SELF parser. The syntax will be described by both informal textual explanation and by formal production rules. The presentation assumes a basic understanding of the SELF semantics described in Chapter 3.

A.2. Notation

The syntax will be presented in Extended Backus-Naur Form (EBNF). Terminal symbols appear in a fixed font and are enclosed in single quotes; they should appear in code as written (not including the single quotes). Non-terminal symbols are italicized. The following table describes the meta-symbols:

META-SYMBOL	FUNCTION	DESCRIPTION
(and)	grouping	used to group syntactic constructions
[and]	option	encloses an optional construction
{ and }	repetition	encloses a construction that may be repeated zero or more times
|	alternative	separates alternative constructions
→	production	separates the left and right hand sides of a production

A.3. Lexical elements

This section describes the lexical structure of SELF programs—how sequences of characters in SELF source code are grouped into lexical tokens.

A.3.1. Character set

SELF programs are written using the following characters:

- *Letters*. The fifty-two upper and lower case letters:
  `ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz`
- *Digits*. The ten numeric digits: `0123456789`
- *Whitespace*. The formatting characters: space, horizontal tab (ASCII HT), newline, carriage return (CR), vertical tab (VT), backspace (BS), and form feed (FF). (Comments are also treated as whitespace.)
- *Graphic characters*. The 32 non-alphanumeric characters:
  `!@#$%^&*()_-+=|\~'{}[]:;"'<>,.?/`

A.3.2. Identifiers

An identifier is a sequence of letters, digits, and underscores ('_') beginning with a letter or an underscore. Case is significant: `self` is not the same as `Self`.

Production:

identifier → (*letter* | '_') {*letter* | *digit* | '_'}

Examples: `i` `_IntAdd` `M` `cloud9` `a_point`

A few identifiers are reserved. Section A.3.7 provides a list of the reserved identifiers.

A.3.3. Operators

An operator consists of a sequence of one or more of the following characters:

```
!@#$%^&*-+=~/?<>,;|`\
```

A few such sequences are reserved and are not operators. Section A.3.7 provides a list of reserved tokens.

Productions:

op-char → '!' | '@' | '#' | '$' | '%' | '^' | '&' | '*' | '-' | '+' | '=' | '~' | '/' | '?' | '<' | '>' | ',' | ';' | '|' | '`' | '\'

operator → *op-char* {*op-char*}

Examples: + - && || <-> %#@^

A.3.4. Numbers

Integer literals are written as a sequence of digits, optionally prefixed with a minus sign and/or a radix. No whitespace is allowed between a minus sign and the digit sequence. Real constants may be either written in fixed point or exponential form.

Numbers may be written in radices ranging from 2 to 36. For radices greater than ten, the characters 'a' through 'z' (case insensitive) represent digit values 10 through 35. The default radix is decimal. A non-decimal number is prefixed by its radix value, specified as a decimal number followed by either a 'r' or an 'R.' If the number is negative, the minus sign must precede the radix specification. Both integers and real numbers may be specified in non-decimal radices, but only numbers in bases less than eleven may be written in exponential form. The exponent of a real number in exponential form is always interpreted in decimal, regardless of the

number's radix. The exponent specifies the power of the base (not always ten) by which the remainder of the number should be multiplied.

Hexadecimal and octal constants may also be written as in C [KeR78], by prefixing the number with either a '`0x`' or '`0X`' for hexadecimal or with a '`0`' for octal. Unlike C, real numbers can also be written in that fashion.

A number with a digit that is not appropriate for the base will cause a lexical error.

Productions:

number → *integer* | *real*

general-digit → *digit* | *letter*

radix → *r-radix* | *C-radix*

r-radix → (*digit* {*digit*}) ('`r`' | '`R`')

C-radix → '`0`' | '`0x`'

integer → ['`-`'] [*radix*] *general-digit* {*general-digit*}

real → *fixed-point* | *float*

fixed-point → ['`-`'] [*radix*] *general-digit* {*general-digit*} '`.`' *general-digit* {*general-digit*}

float → ['`-`'] [*radix*] *digit* {*digit*} ['`.`' *digit* {*digit*}] ('`e`' | '`E`') ['`+`' | '`-`'] *digit* {*digit*}

Examples: `123` `-16ra2.d5` `-01272.34e+15`

A.3.5. Strings

String constants are enclosed in single quotes ('' '). With the exception of single quotes and escape sequences introduced by a backslash ('\'), all characters (including formatting characters like newline and carriage return) lying between the delimiting single quotes are included in the string.

To allow single quotes to appear in a string and to allow non-printing control characters in a string to be indicated more visibly, SELF provides C-like escape sequences:

`\t`	tab	`\b`	backspace	`\n`	newline
`\f`	form feed	`\r`	carriage return	`\v`	vertical tab
`\'`	single quote	`\"`	double quote	`\\`	backslash

A backslash followed by one to three octal digits specifies the character with the specified numeric encoding in the character set. A backslash followed by an 'x' and one to three hexadecimal digits allows a character to be specified in hexadecimal form. For example, the following forms all denote the carriage return character in the ASCII character set:

```
\r    \015    \xd    \x0d
```

A long string may be broken into multiple lines by escaping each newline with a backslash. Such escaped newlines are ignored during formation of the string constant.

A.3.6. Comments

As in Smalltalk, comments are delimited by double quotes ('"'). Double quotes may not themselves be embedded in the body of a comment.

All characters (including formatting characters like newline and carriage return) are ignored in the body of a comment.

Example: `"this is a comment"`

A.3.7. Reserved tokens

Only four reserved tokens exist in SELF:

```
self    super    |    ^
```

These tokens may not be used as slot or message names.

A.4. Keywords

Keywords are used as slot names and as message names.

A simple keyword consists of an identifier followed by a colon (':').

A compound keyword is a sequence of two or more simple keywords. The first simple keyword must begin with a lower case letter or underscore; subsequent simple keywords must be capitalized. Compound keywords beginning with an underscore are used only for primitives; keywords used as slot names cannot begin with an underscore.

Productions:

lc-letter → 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'

uc-letter → 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'

letter → *lc-letter* | *uc-letter*

lc-keyword → *lc-letter* {*letter* | *digit* | '_'} ':'

first-prim-keyword → '_' {*letter* | *digit* | '_'} ':'

cap-keyword → *uc-letter* {*letter* | *digit* | '_'} ':'

primitive-keyword → *first-prim-keyword* {*cap-keyword*}

slot-keyword → *lc-keyword* {*cap-keyword*}

keyword → *primitive-keyword* | *slot-keyword*

Examples: `at:` `at:Put:` `_IntAdd:IfFail:`

A.5. Object literals

Object literals are delimited by parentheses ('(' and ')'). Within the parentheses, an object description consists of a list of slots, delimited by vertical bars

(' | '), followed by the code to be executed when the object is evaluated. Both the slot list and code are optional. If both are omitted, the description denotes an empty object. There is no syntactic way to specify the contents or even the presence of a byte or oop array.

Blocks are written like other objects, except that square brackets (' [' and '] ') are used in place of parentheses.

A slot list consists of a (possibly empty) sequence of slot descriptors, separated by periods. To avoid confusion with the syntax for delegated messages, each separating period must be followed by whitespace. A period at the end of the slot list is optional.

Productions:

object → '(' *object-body* ')'

block → '[' *object-body* ']'

object-body → [*slot-list*] [*code*]

slot-list → ' | ' *slot-descriptor-list* [' . '] ' | '

slot-descriptor-list → *slot-descriptor* | *slot-descriptor* ' . ' *slot-descriptor-list* |

slot-descriptor → *(slot-name* | *slot-name-with-interspersed-arguments)*
[*slot-initializer*]

slot-name → *identifier* | *parent-name* | *argument-name* | *operator* |
slot-keyword

slot-name-with-interspersed-arguments → *operator identifier* |
lc-keyword identifier {*cap-keyword identifier*}

parent-name → *identifier* '*' {'*'}

argument-name → ' : ' *identifier*

slot-initializer → ('=' | '<−') *expression*

A.5.1. Slot descriptors

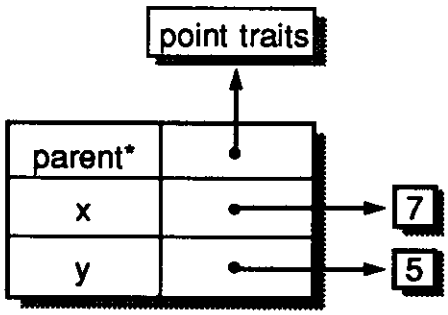
The various forms of slot descriptors have the following meanings:

- *Initialized read-only slots.* A slot name followed by an equals sign and an expression represents a read-only variable initialized to the result of evaluating the expression in the root context. (The root context is also used to evaluate expressions typed at the SELF shell.) As a special case, if the initializing expression is a single object literal (enclosed by parentheses), that object is stored into the slot without evaluation. This allows a slot to be initialized to contain a dynamic object, since the object itself is stored into the slot rather than being evaluated. Such dynamic objects are outer methods, which will be discussed further later in this section.

  For example, an initialized constant point may be defined as

```
(| parent* = traits point.
   x = 3 + 4.
   y = 5 |)
```

  The resultant point would be:

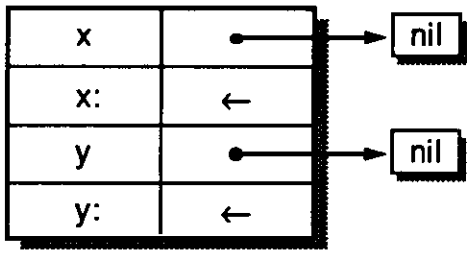


- *Read/write slots.* A simple slot name by itself specifies the equivalent of a read/write variable. The object will contain both a data slot of that name and a corresponding assignment slot. The data slot is initialized to nil.

For example, a simple point might be defined as

```
(| x. y |)
```

producing an object looking like:

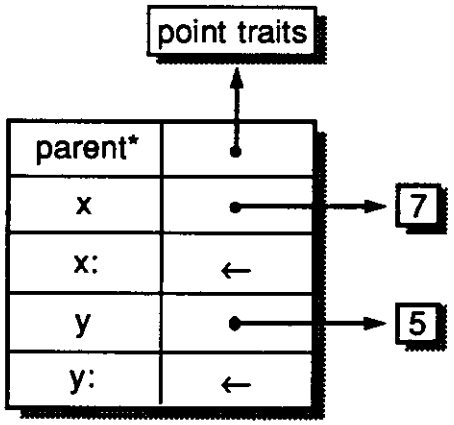


- *Initialized read/write slots.* A simple slot name followed by a left arrow (less-than minus) and a SELF expression represents an initialized read/write variable. The expression is evaluated in the root context and the result stored into the slot at compile time.

  For example, an initialized point might be defined as

```
(|  parent* = traits point.
    x <- 3 + 4.
    y <- 5 |)
```

  producing



- *Parent slots.* As indicated in the preceding examples, a slot name postfixed with one or more asterisks denotes a parent slot. The number of asterisks indicates the priority level of the parent. The trailing asterisks

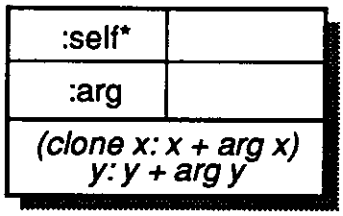
are not considered part of the slot name and are not significant when matching the name against a message.

- *Argument slots*. A slot name beginning with a colon indicates an argument slot. The prepended colon is not considered as part of the slot name and is not significant when matching the name against a message. No initializer may be specified. Unlike the normal case of a slot name by itself, an argument slot name by itself does not signify the existence of an assignment slot. The value of the argument slot is not meaningful until the slot is initialized during the invocation of its containing method.
- *Outer methods*. A dynamic object specified as a slot initializer automatically receives a parent argument slot named `self`. That is the only way the `self` slot may be obtained, since SELF syntax does not currently allow an explicitly defined slot to be both a parent and an argument. An dynamic object used as a slot initializer is termed an outer method.

For example, a point addition method might be written as:

```
(|
     + = ( | :arg |
                (clone  x: x + arg x)
                        y: y + arg y )
|)
```

The resultant object would be:



As in Smalltalk, the argument name can also be written immediately after the operator name (without need of the prepended colon):

```
(|
     + arg = ( (clone  x: x + arg x)
                       y: y + arg y )
|)
```

- *Keyword methods*. Each colon-terminated identifier in a keyword slot name requires a corresponding argument slot in the keyword method object.

  For example:

```
(|
     ifTrue:False: = (
          | :trueBlock. :falseBlock |
          trueBlock value
     )
|)
```

As in Smalltalk, the argument names may be written after the corresponding keyword identifiers. Argument slot names interspersed among keyword identifiers in this fashion do not take a colon prefix.

The example above may therefore also be written as:

```
(|
   ifTrue: trueBlock False: falseBlock = (
      trueBlock value
   )
|)
```

A.5.2. Code

The code body for an object is written as a sequence of statements, each separated by a period. A trailing period is optional. To avoid confusion with delegated messages, each separating period must be followed with whitespace.

Each statement consists of a series of message sends and literals.

The last statement in a code body may be preceded with a '^' operator. A dangling '^' at the end of a code body is equivalent to the statement '^ `self`.'†

Syntax:

code → [*statement-list*] *last-statement* ['.']

statement-list → *statement* '.' [*statement-list*]

last-statement → ['^'] *statement* | '^'

statement → *expression*

expression → *non-self-expression* | '`self`'

non-self-expression → *literal* | *message-sequence*

literal → *string* | *number* | *block* | *object*

† Support for a dangling '^' is a quirk of the current parser. Future implementations may make dangling '^' statements illegal.

A.6. Message syntax

SELF message syntax is similar to Smalltalk's. SELF provides three basic kinds of messages: unary messages, binary operators, and keyword messages. Each has its own syntax, associativity, and precedence. Each type of message can be sent either to an explicit or implicit receiver.

Productions:

message-sequence → *explicit-receiver-message* | *implicit-receiver-message*

explicit-receiver-message → *receiver message-body*

implicit-receiver-message: see Section A.6.4.

receiver → *non-self-expression*

message-body → *unary-message-body* | *binary-message-body* | *keyword-message-body*

A.6.1. Unary messages

An unary message does not specify any arguments other than its receiver. It is written as an identifier following the receiver.

Production:

unary-message-body → *identifier*

Examples of unary messages sent to explicit receivers:

```
17 print
5 factorial
```

Composition. Because unary message sends follow their receivers, unary messages compose from left to right. An expression to print 5 factorial, for example, would be written:

```
5 factorial print
```

which would be interpreted as:

```
(5 factorial) print
```

Precedence. Unary messages have higher precedence than binary operators and keyword messages.

A.6.2. Binary operators

A message with a binary operator involves the receiver, a binary operator, and a single argument. The exact syntax for binary operator names is specified in section A.3.3.

Production:

binary-message-body → *operator argument*

argument → *expression*

Examples of binary messages sent to explicit receivers:

```
3 + 4
7 <-> 8
```

Associativity. Binary operators associate from left to right. All binary operators have the same precedence. For example,

```
3 + 4 * 7
```

is interpreted as

```
(3 + 4) * 7
```

Precedence. The precedence of binary operators is lower than unary messages but higher than keyword messages. For example,

```
3 factorial + pi sine
```

is interpreted as

```
(3 factorial) + (pi sine)
```

A.6.3. Keyword messages

A keyword message corresponds to a procedure call with two or more arguments, including the receiver. Besides the receiver, a keyword message consists of a sequence of one or more keywords, each followed by a message argument. The first keyword must begin with a lower case letter or underscore; subsequent identifiers must be capitalized.

Production:

keyword-message-body → *(first-prim-keyword* | *lc-keyword) argument*
{*cap-keyword argument*}

Example:

```
min: 4 Max: 7
```

is the single message `min:Max:` with arguments 4 and 7, whereas

```
min: 4 max: 7
```

involves two messages: the message `max:` taking 7 as its argument, and the message `min:` taking the result of '`4 max: 7`' as its argument.

Associativity. Keyword messages associate from right to left, so

```
5 min: 4 max: 7
```

is interpreted as

```
5 min: (4 max: 7)
```

Both the association order and capitalization requirements are intended to reduce the number of parentheses necessary in SELF code. For example, taking the minimum of two slots `m` and `n`, and storing the result into a data slot `i` may be simply written as

```
i: m min: n
```

Precedence. Keyword messages normally have the lowest precedence. For example,

```
i: 5 factorial + pi sine
```

is interpreted as

```
i: ((5 factorial) + (pi sine))
```

As a special case, an implicit-receiver keyword message following a binary operator has higher precedence than the operator. For example,

```
1 + power: 3
```

is interpreted as

```
1 + (power: 3)
```

A.6.4. Evaluation of arguments.

The actual arguments in a message send are evaluated from left to right before the matching method is invoked. Each argument is evaluated fully before turning to the next—argument evaluations may not be interleaved. For instance, in the message send

```
1 to: 5 * i By: 2 * j Do: [|:k | k print ]
```

the "`5 * i`" is evaluated first, then "`2 * j`," and finally "`[|:k | k print ]`" (which is a static object literal evaluating to a block). After those arguments are evaluated, the `to:By:Do:` method is invoked.

A.6.5. Implicit-receiver messages

Unary, binary operator, and keyword messages may be written without an explicit receiver. Such messages use the current receiver (`self`) as the implicit receiver. The method search, however, begins at the current activation object rather than the current receiver.

Productions:

implicit-receiver-message → *simple-implicit-receiver-message* |
delegated-message

simple-implicit-receiver-message → *message-body*

Examples:

`factorial`	(implicit-receiver unary message)
`+ 3`	(implicit-receiver binary operator message)
`max: 5`	(implicit-receiver keyword message)

A.6.6. Delegated messages

Any implicit-receiver message may be delegated to another object. The effect is to start the method search at the delegate but to retain the current receiver (`self`). A delegated message is written as an implicit-receiver message prefixed by the delegate's name and a period. No whitespace may separate the delegate's name, period, and message name.

Productions:

delegated-message → *delegate* '.' *message-body*

delegate → *delegate-slot* | *super-delegate*

delegate-slot → *identifier*

Examples:

```
listParent.height
intParent.min: 17
```

Only implicit-receiver messages may be delegated. Delegated messages with explicit receivers are illegal.

Illegal: `point x.print`

A.6.7. Super messages

Instead of being delegated to a single object, a message may be delegated to all the parents of the method holder. Such a message is called a super message send by analogy to the super sends of Smalltalk. A super send is written as a delegated send with the reserved word `super` as the delegate name. As a shortcut, a super send may also be written by omitting the delegate name.

Syntax:

super-delegate → ['super']

Examples:

```
super.display
.display
.+ 5
.min: 17
```

A.6.8. Summary of message syntax

There are three basic types of messages: unary, binary, and keyword. The following table summarizes their characteristics:

MESSAGE	ARGUMENTS	PRECEDENCE	ASSOCIATIVITY	SYNTAX
unary	0	highest	none	*receiver identifier*
binary	1	medium	left-to-right	*receiver operator argument*
keyword	≥ 1	lowest	right-to-left	*receiver* (*first-prim-keyword* | *lc-keyword*) *argument* { *cap-keyword argument* }

All three types may be sent to either explicit or implicit receivers. An implicit-receiver message may additionally specify a delegate (specific or `super`) to handle

the message. The following table summarizes the syntax of explicit-receiver, implicit-receiver, delegated, and super message sends:

MESSAGE TYPE	SYNTAX
explicit-receiver	*receiver message-body*
implicit-receiver	*message-body*
delegated	*delegate-slot* '.' *message-body*
super	`['super']`'.' *message-body*

Appendix B

Memory System Measurements

B.1. Introduction

This appendix presents the performance data used for the evaluation of allocation and cloning time in Chapter 4 and of automatic storage reclamation in Chapter 5. The performance data was gathered from a small suite of benchmarks composed of the the Richards benchmark (translated from Smalltalk) and the Stanford compiler benchmarks (translated from C†). Because all the running times for the base benchmarks are short, each measurement encompassed multiple iterations of the base benchmark in order to provide accurate measurements under the profiler and to induce scavenging in all benchmarks. To gauge the effect of the desired survivor size on scavenging, each measurement was performed twice, once with the desired survivor size at 10 kilobytes and the other time at 80 kilobytes. Sections 4.18.1 and 5.6.1 provide more details on the measurement methodology.

The first measurement section shows the running times of the benchmarks. The next section presents the measurements of object allocation and cloning times used in Chapter 4. The next few sections present measurements for a number of aspects of scavenging. The first scavenging section focuses on time costs: pause times, scavenging intervals, and time spent maintaining the remembered set. The next scavenging section examines the degree of correlation between scavenging pause times and the amount of data scavenged, testing the assumption that scavenging pause time is proportional to the amount of data scavenged; that assumption is used

† The Stanford benchmarks were originally written in Pascal. Our SELF versions were translated from the C versions of the original Pascal benchmarks.

by SELF's demographic feedback-mediated tenuring algorithm to keep scavenging pause times bounded by controlling the amount of data scavenged. The final scavenging section looks at the amount of storage allocated and tenured for each of the benchmarks. The appendix concludes with measurements of a mark-and-sweep garbage collection.

B.2. Benchmark descriptions

The table below describes the composition of the benchmark suite.

BENCHMARK	ITERATIONS	DESCRIPTION
richards	150	Task scheduling simulation.
perm	1000	Heavily recursive permutation program, originally written by Denny Brown.
towers	1000	Towers of Hanoi, with three stacks and eighteen disks.
queens	1000	Eight queens problem. Each iteration solves the problem 50 times.
intmm	500	Integer matrix multiplication of two 40x40 matrices.
puzzle	100	A compute-bound program from Forest Baskett, using array manipulations to solve a puzzle.
quick	1000	Program sorting a 5000-element integer array using the Quicksort algorithm in recursive form.
bubble	500	Program sorting a 500-element integer array using the bubble sort algorithm.
tree	150	Program sorting 5000 integers using the tree sort algorithm. Allocates tree nodes from the heap.
mm	50	Floating-point matrix multiplication of two 40x40 matrices.
fft	10	Fast Fourier Transform, using floating-point numbers.

The Richards benchmark was originally written by Martin Richards in BCPL and was subsequently translated into Smalltalk by L. Peter Deutsch. The Stanford benchmarks were collected by John Hennessy and modified by Peter Nye. Although the original versions of the Stanford benchmarks were written in Pascal, the SELF

translations were based on the C versions, which were translated from the original Pascal. David Ungar and Craig Chambers translated the benchmarks to SELF.

B.3. Benchmark running times

The following table describes the running times for the benchmarks. Running times were measured in terms of total CPU time (sum of user and system times).

Benchmark	Iterations	Total Cpu Time (ms)	
		10KB Survivors	80KB Survivors
richards	150	434230	432680
perm	1000	662930	662440
towers	1000	909560	905790
queens	1000	527630	529500
intmm	500	488620	488220
puzzle	100	530740	533680
quick	1000	693000	693730
bubble	500	827840	833800
tree	150	276360	264840
mm	50	565850	633620
fft	10	225830	233600

B.4. Allocation and cloning time

This section looks at the time spent in object allocation and cloning during the benchmark runs. The object allocation measurements cover low-level allocation of known amounts of data; those measurements indicate the lower bound for object allocation in our system. Because objects currently do not carry size fields, the low-level allocation routines are used only for objects whose size is known at compile time. In addition to the low-level object allocation time, cloning time includes time for field initialization and for cloning objects whose sizes are not known at compile time.

The cloning measurements encompass all the cloning time overhead. The measurements were performed using the *gprof* profiling facility [GKM82].

The measurements are meaningful only in giving a rough indication of the costs of the various object cloning and allocation operations. Because the benchmarks are not representative of expected system operation, they cannot be used to predict the distribution of cloning operations or the average cloning overhead. In particular, the Stanford compiler benchmarks make extensive use of arrays, which biases the cloning and allocation measurements. The cloning overhead in *perm*, *queens*, *intmm*, and *bubble*, for example, is almost entirely due to the creation and initialization of arrays at the beginning of those benchmarks. Similarly, the large mean cloning time for *quick* is due to the creation and initialization of a 5000-element array at the beginning of each benchmark iteration.

The following table shows the allocation and cloning times for the benchmarks in terms of user CPU time. Benchmarks using floating-point numbers are indicated with an asterisk.

Benchmark	Iterations	Allocation Time (ms)		Cloning Time (ms)	
		10KB Survivors	80KB Survivors	10KB Survivors	80KB Survivors
richards	150	20	10	80	90
perm	1000	0	0	430	590
towers	1000	120	90	320	350
queens	1000	10	30	15440	15050
intmm	500	10	0	6650	6530
puzzle	100	0	0	1120	1080
quick	1000	10	0	5540	5880
bubble	500	0	0	270	310
tree	150	6520	6700	31360	25010
mm*	50	65430	73200	402120	454650
fft*	10	22320	23480	146630	142580

The next table expresses the same data as a percentage of total running time.

Benchmark	Iterations	Allocation Time (%)		Cloning Time (%)	
		10kb Survivors	80kb Survivors	10kb Survivors	80kb Survivors
richards	150	0.005%	0.002%	0.018%	0.021%
perm	1000	0.0%	0.0%	0.065%	0.089%
towers	1000	0.013%	0.010%	0.035%	0.039%
queens	1000	0.002%	0.006%	2.9%	2.8%
intmm	500	0.002%	0.0%	1.4%	1.3%
puzzle	100	0.0%	0.0%	0.21%	0.20%
quick	1000	0.001%	0.0%	0.80%	0.85%
bubble	500	0.0%	0.0%	0.033%	0.037%
tree	150	2.4%	2.2%	11.%	9.4%
mm*	50	12.%	12.%	71.%	72.%
fft*	10	9.9%	10.%	65.%	61.%

* uses floating-point numbers

The next table indicates the average time required for a known-size allocation.

Benchmark	Iterations	Mean Allocation Time (µs)	
		10kb Survivors	80kb Survivors
richards	150	†	†
perm	1000	†	†
towers	1000	6.	5.
queens	1000	†	†
intmm	500	†	†
puzzle	100	†	†
quick	1000	†	†
bubble	500	†	†
tree	150	8.3	8.9
mm*	50	7.4	7.4
fft*	10	7.3	7.7

† not meaningful because of *gprof* time quantum
* uses floating-point numbers

The following table shows the average time required for a cloning operation.

Benchmark	Mean Cloning Time (µs)					
	All		Known-Size Only		General Only	
	10KB Survivors	80KB Survivors	10KB Survivors	80KB Survivors	10KB Survivors	80KB Survivors
richards	8.3	9.3	5.6	4.2	16.	24.
perm	84.	106.	-	-	84.	106.
towers	15.	16.	9.5	6.8	60.	67.
queens	71.	71.	-	-	71.	71.
intmm	82.	80.	-	-	82.	80.
puzzle	400.	390.	-	-	400.	390.
quick	3500.	3800.	-	-	3500.	3800.
bubble	390.	190.	-	-	390.	190.
tree	22.	19.	13.	12.	31.	26.
mm*	45.	46.	7.9	7.9	48.	48.
fft*	48.	46.	12.	7.4	48.	47.

* uses floating-point numbers

In addition to the overall average, the table shows the average time for known-size cloning as opposed to general sentinel-based cloning. The data is not adjusted for the amount of data cloned. It should be noted that the known-size cloning routines are only for small amounts of data (< 12 oops per clone) and that the general cloning time includes resizing operations as well as simple cloning.

The following tables break the allocation and cloning times down by individual routines:

- The `cloneN` routines perform known-size clones of objects with N oops of data in addition to the two words of object header (mark and map); a `cloneN` routine would therefore clone objects of size N + 2 words.

- The `cloneO` routine clones the source object's slots (without foreknowledge of their number) but creates the clone's oops array with a different size than the source's.
- The `clone` routine clones an object whose size is initially unknown.
- The `blockClone` routine clones a SELF block; as all blocks have the same size, this routine performs a known-size clone.
- The `new_floatObj` routine is used to create a new floating-point object. Although the bytes part of a float is created with a known size, I consider the `new_floatObj` routine as a whole to be an unknown-size clone because the oops part is unknown.
- The `survivor` routines are called by the scavenger to perform an unknown-size clone; the table further distinguishes between the routines used to scavenge normal SELF objects and maps.

ALLOCATION AND CLONING BREAKDOWNS FOR RICHARDS

Routine	Invocations		User Cpu Time (ms)	
	10KB Survivors	80KB Survivors	10KB Survivors	80KB Survivors
alloc_oops	7205	7205	10	20
alloc_bytes	10	10	0	0
clone1	300	300	0	0
clone2	4650	4650	20	30
clone3	900	900	10	10
clone5	1200	1200	0	0
clone8	150	150	0	0
clone	2250	2250	50	40
selfObj survivor	140	140	0	0
mapObj survivor	66	66	0	10

ALLOCATION AND CLONING BREAKDOWNS FOR PERM				
ROUTINE	INVOCATIONS		USER CPU TIME (MS)	
	10KB SURVIVORS	80KB SURVIVORS	10KB SURVIVORS	80KB SURVIVORS
alloc_oops	4	4	0	0
alloc_bytes	10	38	0	0
cloneO	5000	5000	420	530
selfObj survivor	74	302	10	30
mapObj survivor	66	258	0	30

ALLOCATION AND CLONING BREAKDOWNS FOR TOWERS				
ROUTINE	INVOCATIONS		USER CPU TIME (MS)	
	10KB SURVIVORS	80KB SURVIVORS	10KB SURVIVORS	80KB SURVIVORS
alloc_oops	19010	19010	120	90
alloc_bytes	10	68	0	0
clone2	19000	19000	180	130
cloneO	2000	2000	130	170
selfObj survivor	257	717	10	30
mapObj survivor	82	582	0	20

ALLOCATION AND CLONING BREAKDOWNS FOR QUEENS				
ROUTINE	INVOCATIONS		USER CPU TIME (MS)	
	10KB SURVIVORS	80KB SURVIVORS	10KB SURVIVORS	80KB SURVIVORS
alloc_oops	246	246	10	10
alloc_bytes	10	639	0	20
cloneO	200000	200000	14240	14630
selfObj survivor	1307	6302	40	190
mapObj survivor	1058	5420	40	230

Allocation and Cloning Breakdowns for Intmm				
Routine	Invocations		User Cpu Time (ms)	
	10kb Survivors	80kb Survivors	10kb Survivors	80kb Survivors
alloc_oops	224	224	10	0
alloc_bytes	10	10	0	0
cloneO	63000	63000	5390	5380
selfObj survivor	18501	18501	1260	1150
mapObj survivor	82	82	0	0

Allocation and Cloning Breakdowns for Puzzle				
Routine	Invocations		User Cpu Time (ms)	
	10kb Survivors	80kb Survivors	10kb Survivors	80kb Survivors
alloc_oops	56	56	0	0
alloc_bytes	12	12	0	0
cloneO	1800	1800	670	700
selfObj survivor	880	880	420	350
mapObj survivor	84	84	30	30

Allocation and Cloning Breakdowns for Quick				
Routine	Invocations		User Cpu Time (ms)	
	10kb Survivors	80kb Survivors	10kb Survivors	80kb Survivors
alloc_oops	412	412	10	0
alloc_bytes	10	10	0	0
cloneO	1000	1000	3570	3720
selfObj survivor	486	486	1830	2010
mapObj survivor	82	82	140	150

Allocation and Cloning Breakdowns for Bubble				
Routine	Invocations		User Cpu Time (ms)	
	10kb Survivors	80kb Survivors	10kb Survivors	80kb Survivors
alloc_oops	20	20	0	0
alloc_bytes	10	76	0	0
cloneO	500	500	220	190
selfObj survivor	106	486	10	20
mapObj survivor	82	82	40	100

Allocation and Cloning Breakdowns for Tree				
Routine	Invocations		User Cpu Time (ms)	
	10kb Survivors	80kb Survivors	10kb Survivors	80kb Survivors
alloc_oops	750534	750522	6520	6700
alloc_bytes	10	10	0	0
clone3	750000	750000	9500	9220
cloneO	150	150	580	540
selfObj survivor	707314	599048	21180	15170
mapObj survivor	48	48	100	80

Allocation and Cloning Breakdowns for Mm				
Routine	Invocations		User Cpu Time (ms)	
	10kb Survivors	80kb Survivors	10kb Survivors	80kb Survivors
alloc_oops	564611	564611	2680	2750
alloc_bytes	8278841	9388021	62750	70450
blockClone	560000	560000	4420	4430
new_floatObj	7760000	7760000	370350	372740
cloneO	6300	6300	640	520
selfObj survivor	532502	1645766	26710	76950
mapObj survivor	66	124	0	10

Allocation and Cloning Breakdowns for Fft				
Routine	Invocations		User Cpu Time (ms)	
	10kb Survivors	80kb Survivors	10kb Survivors	80kb Survivors
alloc_oops	15011	14995	110	70
alloc_bytes	3033588	3038196	22210	23410
blockClone	7070	7070	100	70
clone2	6440	6440	60	30
new_floatObj	2644220	2644220	126010	123440
cloneO	40	40	20	10
selfObj survivor	401476	413982	20430	19020
mapObj survivor	50	122	10	10

The distribution of cloning routines in the measurements is not representative of normal system operation. The seeming pervasiveness of `cloneO` invocations, for example, is due to the heavy use of arrays in the Stanford compiler benchmarks. Each such benchmark begins by invoking `cloneO` on the prototype vector in order to create a new array to work with. For example, the 200,000 calls to `cloneO` for *queens* is entirely due to the creation and initialization of four one-dimensional arrays: each iteration solves the queens problem 50 times, and the queens benchmark run comprises 1000 iterations, resulting in $4 \times 50 \times 1000 = 200{,}000$ array creations. The prominence of `cloneO` in the measurements is an artifact both of the array-based benchmarks, which dynamically allocate new arrays each time instead of reusing statically allocated arrays, and of the measurement strategy, which repeats each benchmark many times.

The *tree* benchmark provides the best evidence of the overhead for general unknown-size cloning because almost all of the clones—both known-size and unknown-size—operate on objects of the same size (five-word tree nodes).

CLONING BREAKDOWNS FOR TREE				
ROUTINE	INVOCATIONS		USER CPU TIME (MS)	
	10KB SURVIVORS	80KB SURVIVORS	10KB SURVIVORS	80KB SURVIVORS
clone3	750000	750000	9500	9220
cloneO	150	150	580	540
selfObj survivor	707314	599048	21180	15170
mapObj survivor	48	48	100	80

Since the selfObj `survivor` routine can only clone tree nodes produced by `clone3` and the sort array produced by `cloneO`, we can calculate the unknown-size cloning costs by comparing the mean time for a `clone3` operation on a tree node with the mean time for a selfObj `survivor` operation on a tree node. To adjust for the time spent invoking `survivor` on the sort array rather than on a tree node, assume that the array is cloned on every scavenge; since the benchmark scavenged 180 times (for each desired survivor values), the amount of time spent in `survivor` for the sort array would then be roughly $580 \times (180 \div 150) = 700$ ms for a desired survivor size of 10 KB and 650 ms for a desired survivor size of 80 KB. With that adjustment, the mean times for a `survivor` operation on a tree node are 29 µs (10 KB survivors) and 24 µs (80 KB survivors), as opposed to 13 µs (10 KB survivors) and 12 µs (80 KB survivors) for `clone3`. Those times indicate that the unknown-size clone routine is twice as slow as the known-size routine in the *tree* benchmark.

B.5. Scavenging time costs

This section present measurements of four aspects of scavenging time overhead: pause times, scavenging intervals, store-checking overhead, and the time required to place objects into the remember set.

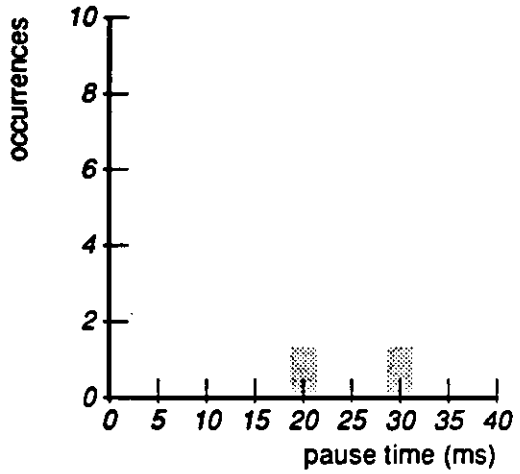
B.5.1. Pause times

The following table describes the length of scavenging pauses for each benchmark run, where each run comprised the number of iterations indicated in section B.2.2. Pause times were measured in terms of total CPU time (sum of the user and system times).

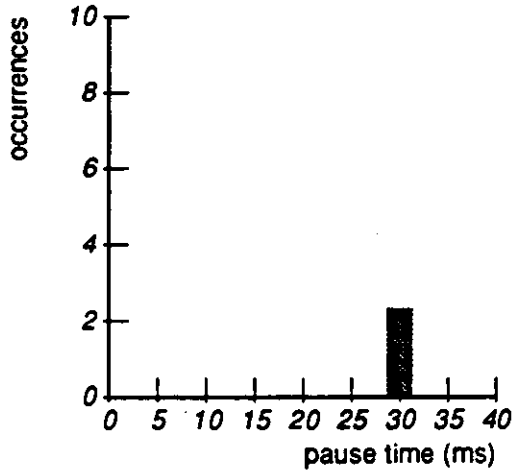
Benchmark	Iterations	Surv. Size	Number of Scavenges	Pause Times (ms)			
				Min	50th Percentile	90th Percentile	Max
richards	150	10	2	20	20	30	30
		80	2	30	30	30	30
perm	1000	10	2	30	30	30	30
		80	2	60	60	70	70
towers	1000	10	4	10	10	30	30
		80	4	70	70	70	70
queens	1000	10	124	10	10	20	40
		80	124	30	30	40	80
intmm	500	10	110	30	40	50	80
		80	110	20	40	50	90
puzzle	100	10	28	60	80	130	130
		80	28	60	90	130	140
quick	1000	10	200	40	60	70	90
		80	200	40	60	70	80
bubble	500	10	10	0	10	30	40
		80	10	20	30	80	90
tree	150	10	180	50	200	620	720
		80	180	50	190	300	350
mm*	50	10	2250	0	10	50	110
		80	2250	0	50	90	180
fft*	10	10	742	30	50	60	210
		80	742	30	40	50	240

* Uses floating-point numbers

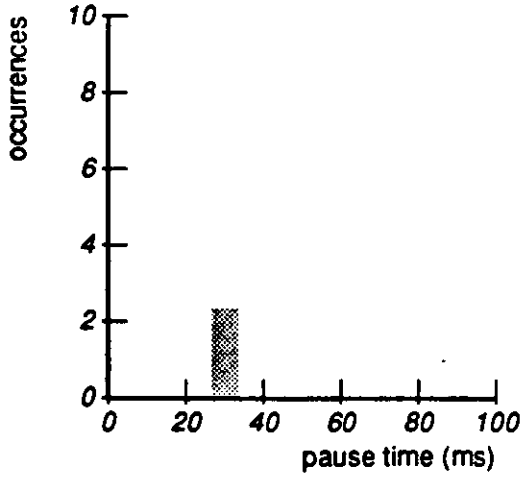
The following histograms show the distribution of pause times in more detail. Two plots are given for each benchmark, one with the desired survivor size set at 10 kilobytes and the other with the survivor size at 80 kilobytes.



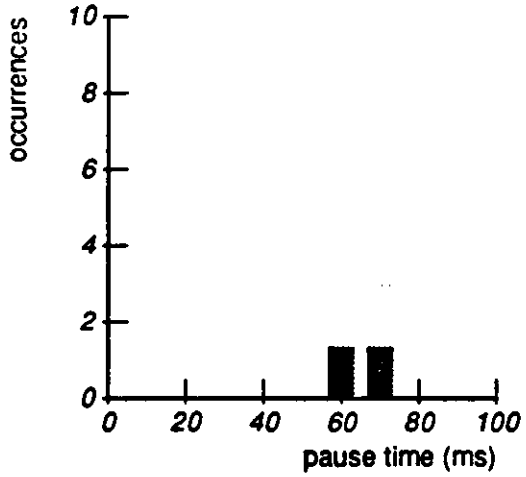
richards (150 iter.; 10kb surv.)



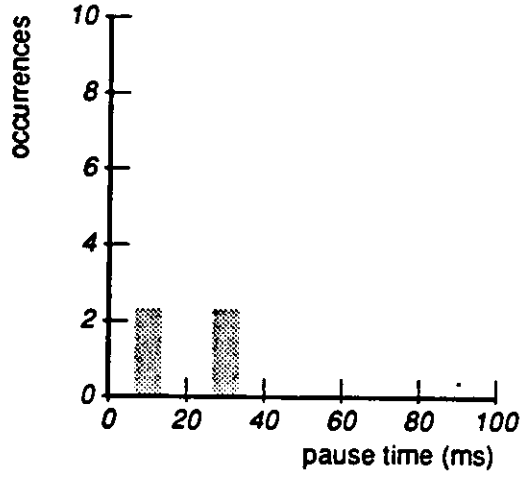
richards (150 iter.; 80kb surv.)



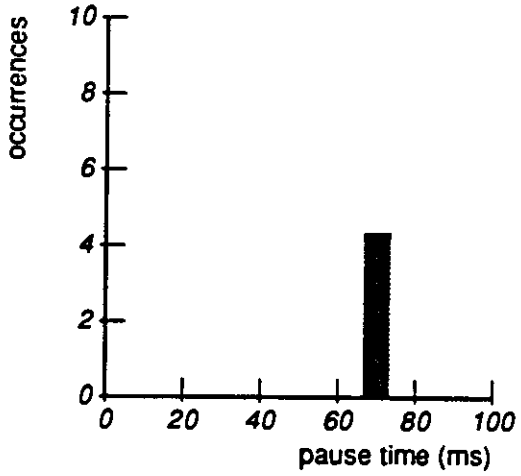
perm (1000 iter.; 10kb surv.)



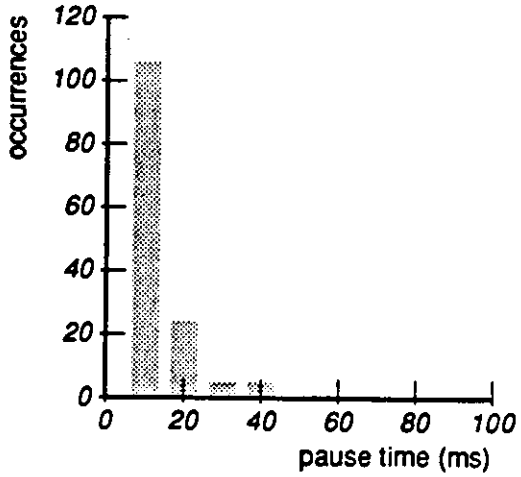
perm (1000 iter.; 80kb surv.)



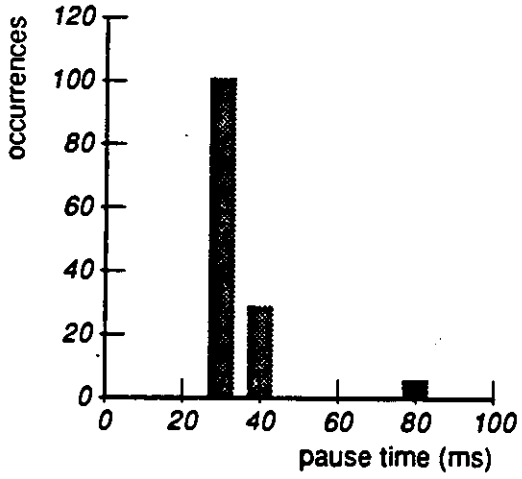
towers (1000 iter.; 10kb surv.)



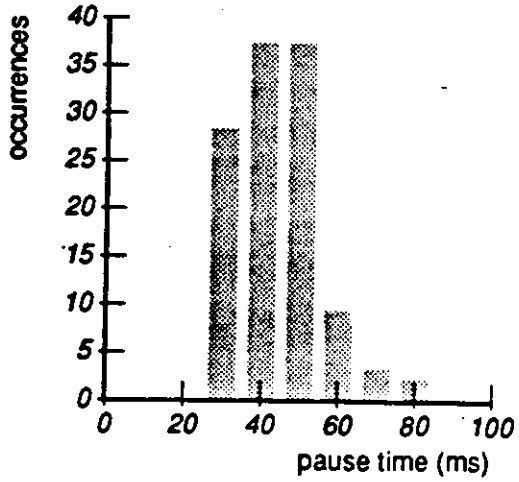
towers (1000 iter.; 80kb surv.)



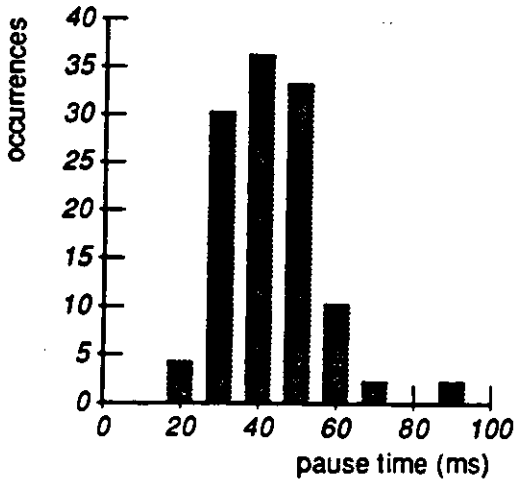
queens (1000 iter.; 10kb surv.)



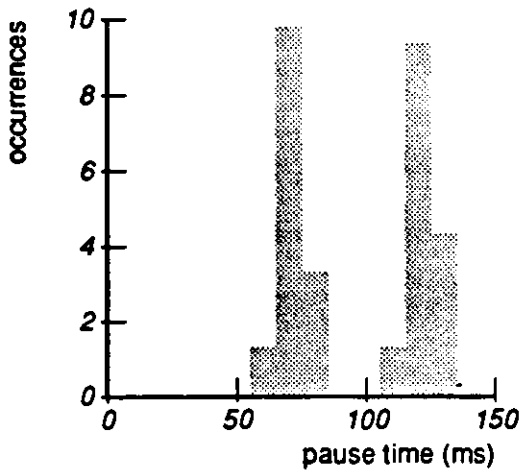
queens (1000 iter.; 80kb surv.)



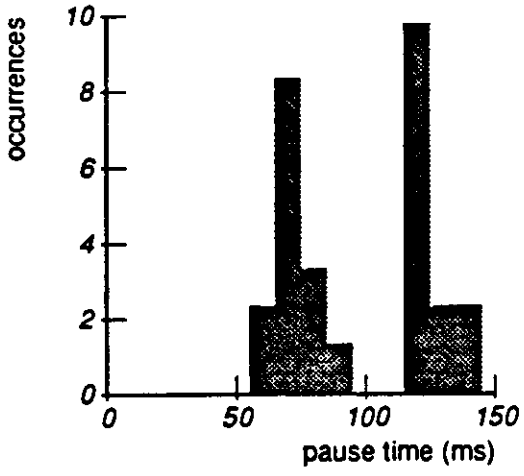
intmm (500 iter.; 10kb surv.)



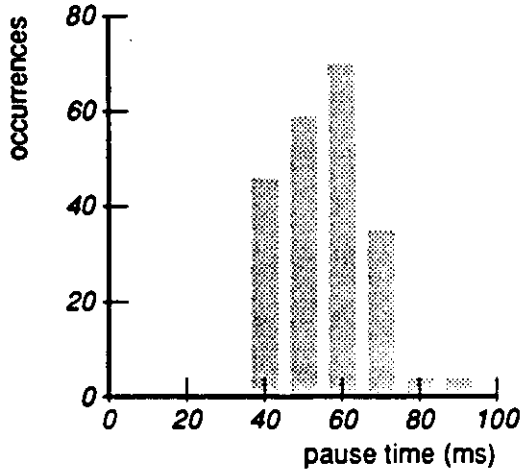
intmm (500 iter.; 80kb surv.)



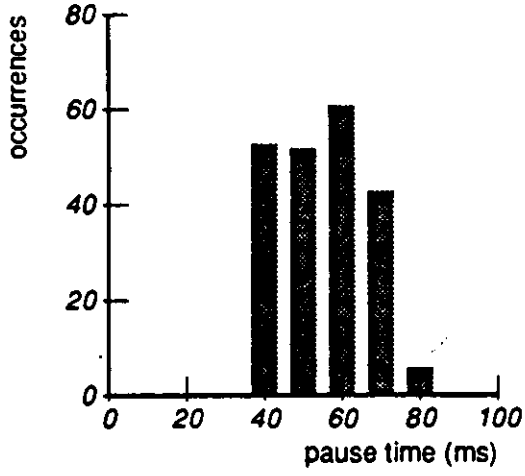
puzzle (100 iter.; 10kb surv.)



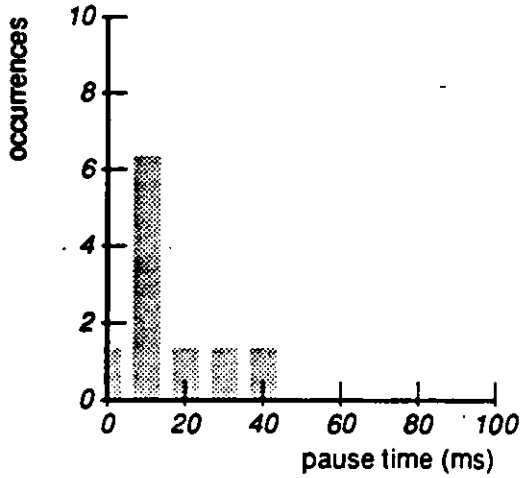
puzzle (100 iter.; 80kb surv.)



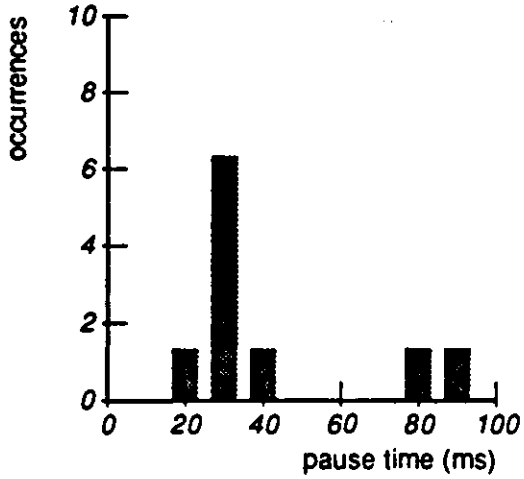
quick (1000 iter.; 10kb surv.)



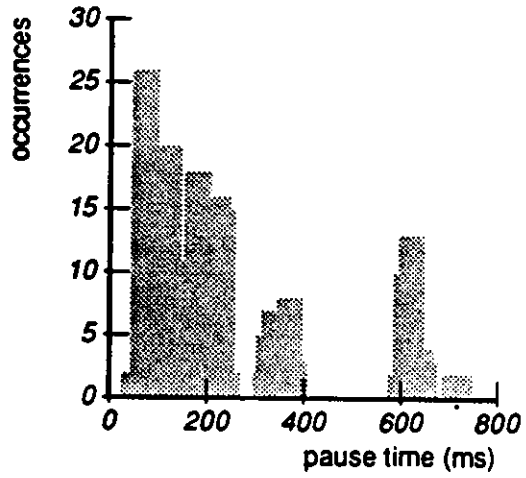
quick (1000 iter.; 80kb surv.)



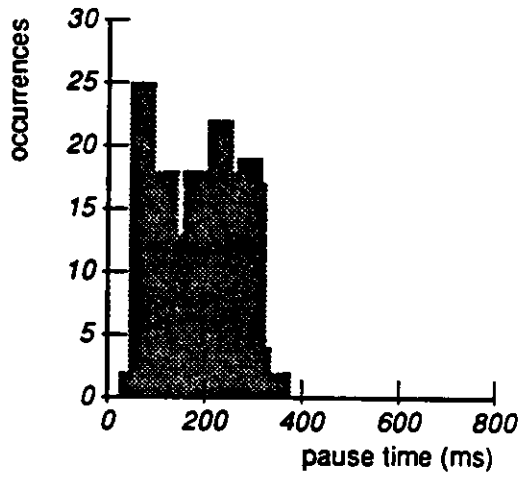
bubble (500 iter.; 10kb surv.)



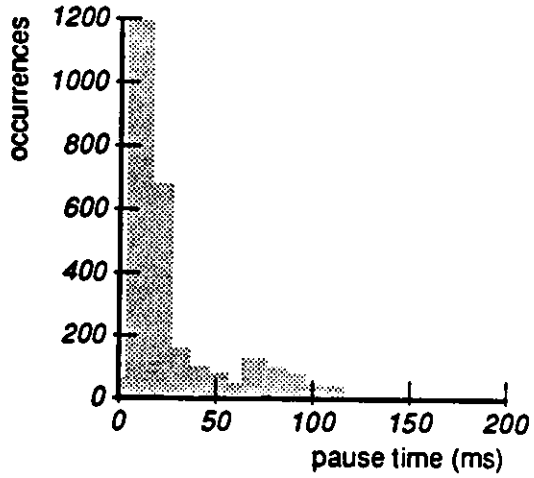
bubble (500 iter.; 80kb surv.)



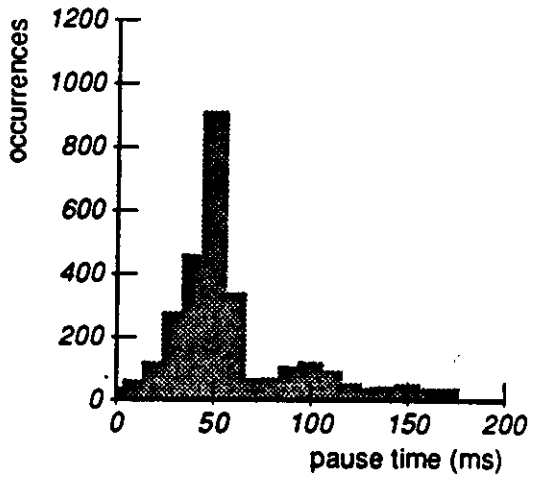
tree (150 iter.; 10kb surv.)



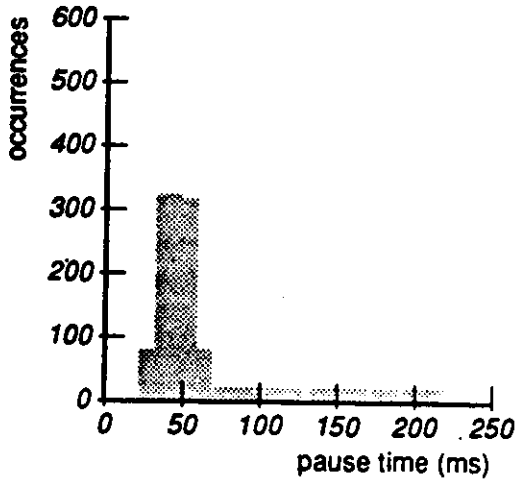
tree (150 iter.; 80kb surv.)



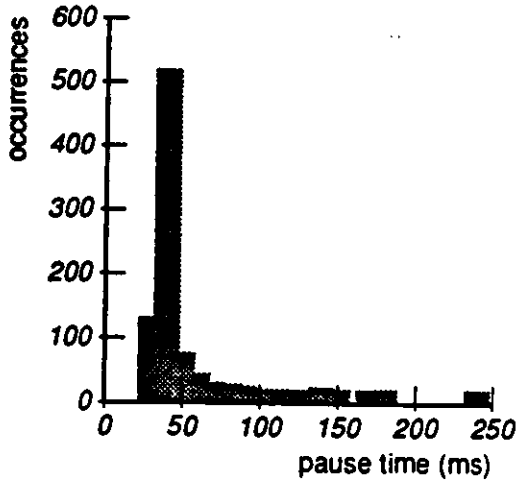
mm (50 iter.; 10kb surv.)



mm (50 iter.; 80kb surv.)



fft (150 iter.; 10kb surv.)



fft (150 iter.; 80kb surv.)

B.5.2. Scavenging intervals

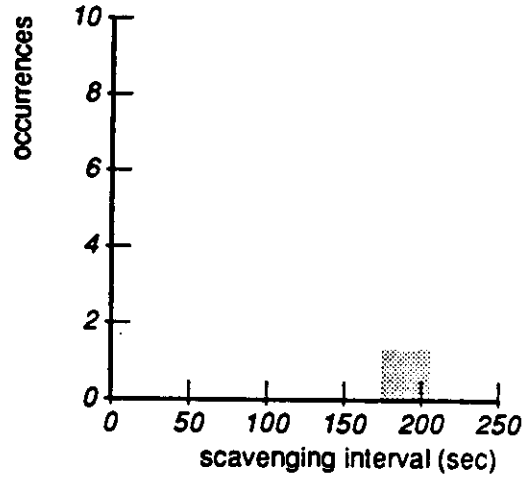
The following table describes the distribution of scavenging intervals for each benchmark run. The scavenging intervals are measured in terms of total (user and system) CPU time.

Benchmark	Iterations	Surv. Size	Number of Scavenges	Scavenging Intervals (sec)			
				Min	10th Percentile	50th Percentile	Max
richards	150	10	2	180.	180.	180.	200.
		80	2	180.	180.	180.	200.
perm	1000	10	2	220.	220.	220.	240.
		80	2	220.	220.	220.	240.
towers	1000	10	4	200.	200.	220.	220.
		80	4	200.	200.	220.	220.
queens	1000	10	124	4.0	4.2	4.2	4.5
		80	124	4.0	4.2	4.2	4.5
intmm	500	10	110	3.9	3.9	4.7	5.2
		80	110	3.8	3.9	4.7	5.1
puzzle	100	10	28	16.	16.	16.	21.
		80	28	16.	16.	16.	22.
quick	1000	10	200	2.9	3.4	3.4	3.7
		80	200	2.9	3.4	3.4	3.7
bubble	500	10	10	75.	75.	83.	83.
		80	10	76.	76.	83.	86.
tree	150	10	180	1.1	1.2	1.3	1.7
		80	180	1.1	1.2	1.3	1.6
mm*	50	10	2250	0.19	0.22	0.23	0.33
		80	2250	0.19	0.22	0.23	0.41
fft*	10	10	742	0.22	0.25	0.25	0.33
		80	742	0.23	0.25	0.25	0.68

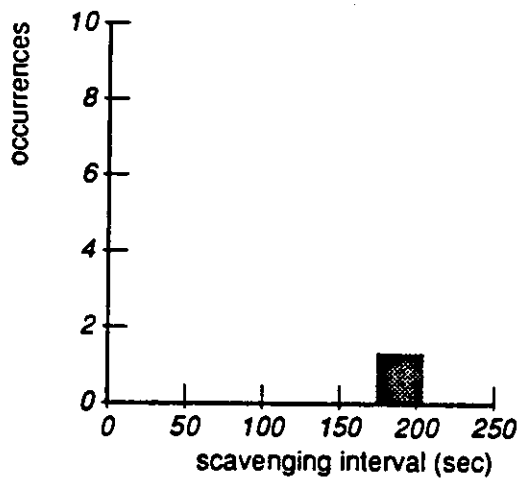
* Uses floating-point numbers

All scavenging intervals are specified to two significant figures.

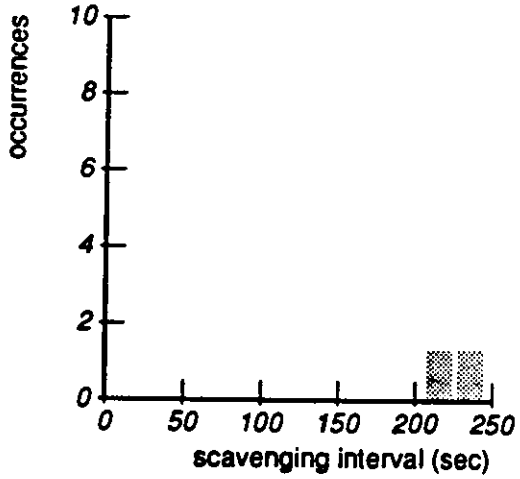
The following histograms show the distribution of scavenging intervals in more detail. The scavenging intervals are measured in terms of total (user and system) CPU time.



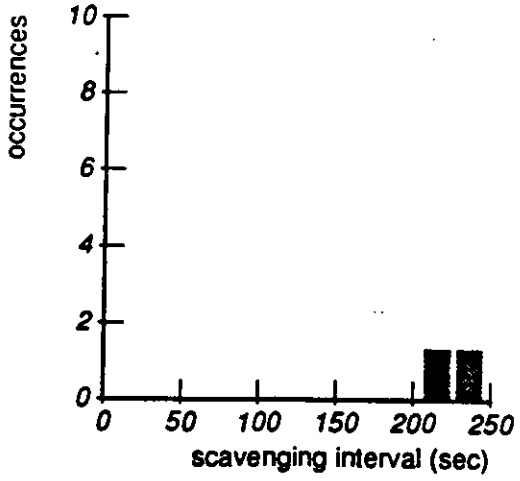
richards (150 iter.; 10kb surv.)



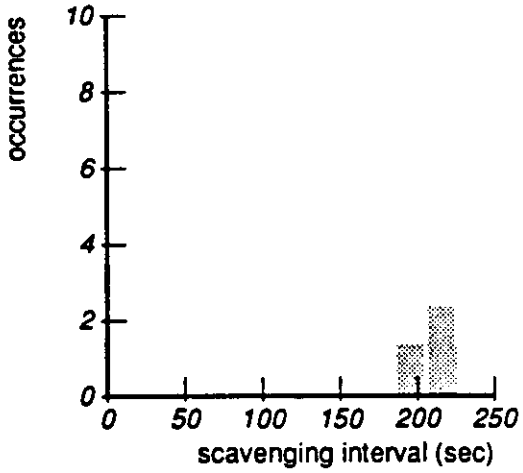
richards (150 iter.; 80kb surv.)



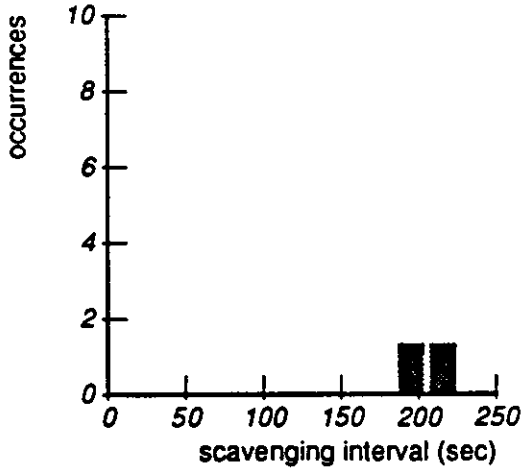
perm (1000 iter.; 10kb surv.)



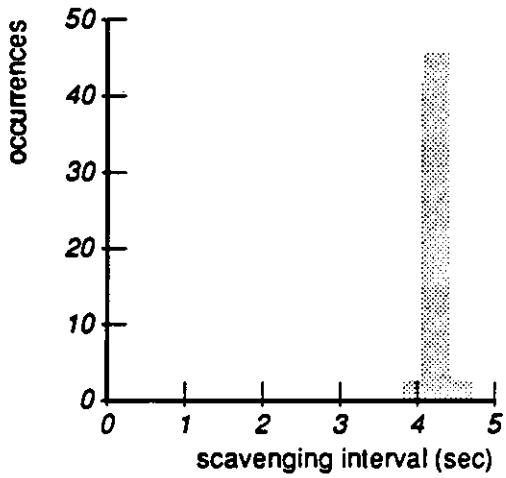
perm (1000 iter.; 80kb surv.)



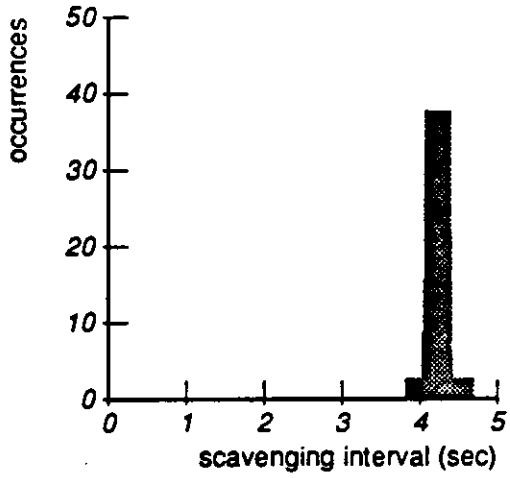
towers (1000 iter.; 10kb surv.)



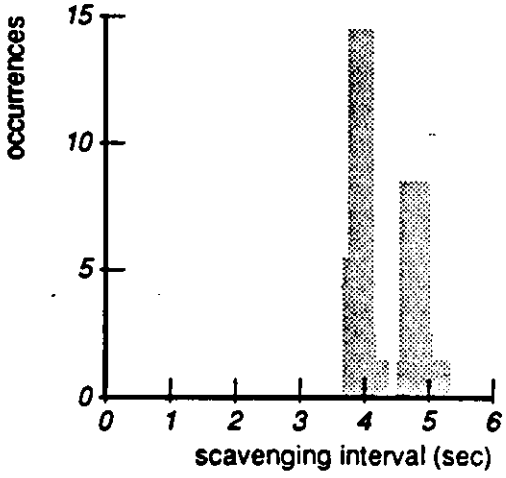
towers (1000 iter.; 80kb surv.)



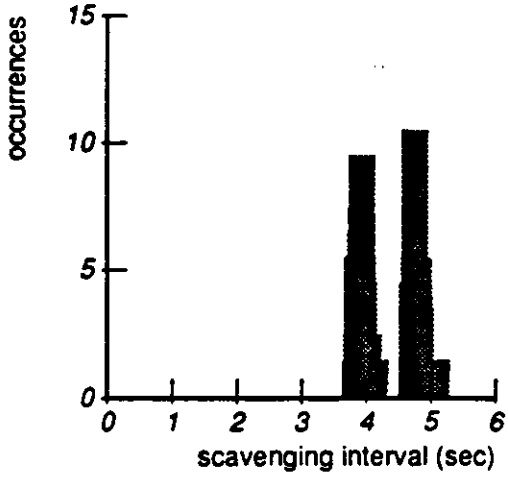
queens (1000 iter.; 10kb surv.)



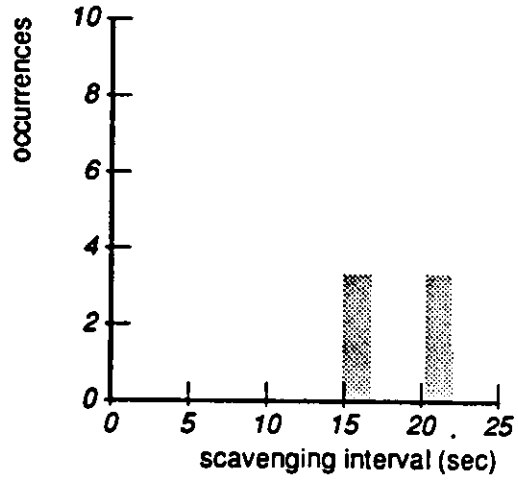
queens (1000 iter.; 80kb surv.)



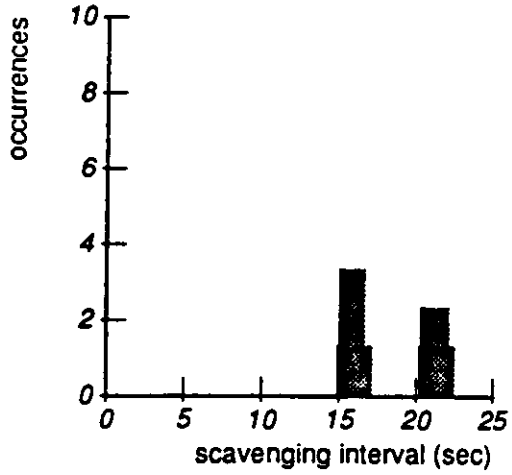
intmm (500 iter.; 10kb surv.)



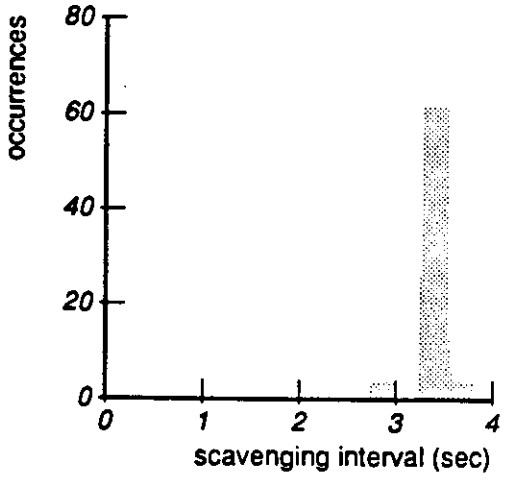
intmm (500 iter.; 80kb surv.)



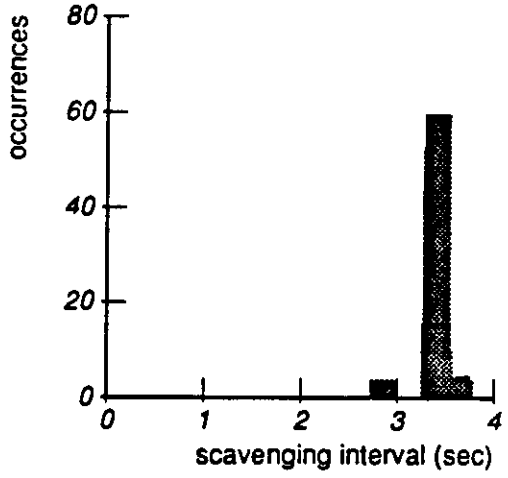
puzzle (100 iter.; 10kb surv.)



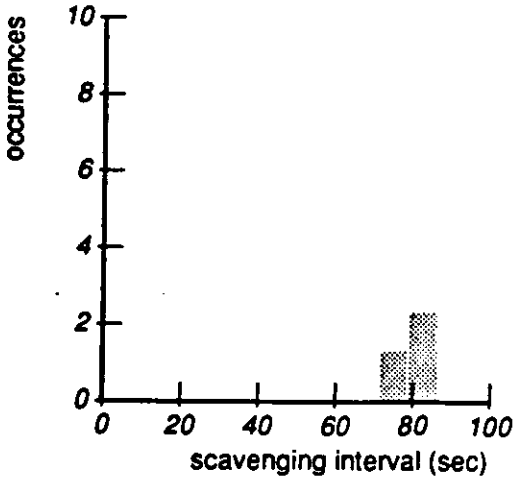
puzzle (100 iter.; 80kb surv.)



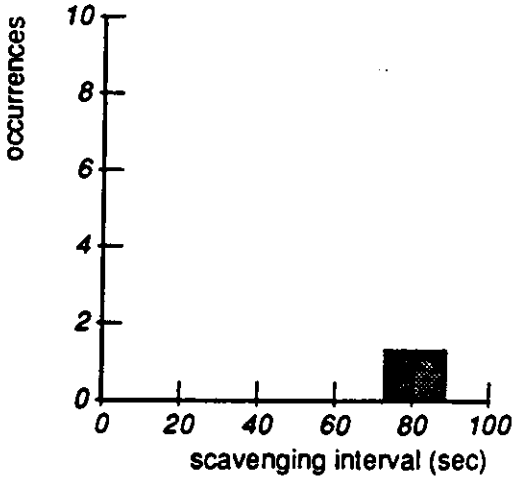
quick (1000 iter.; 10kb surv.)



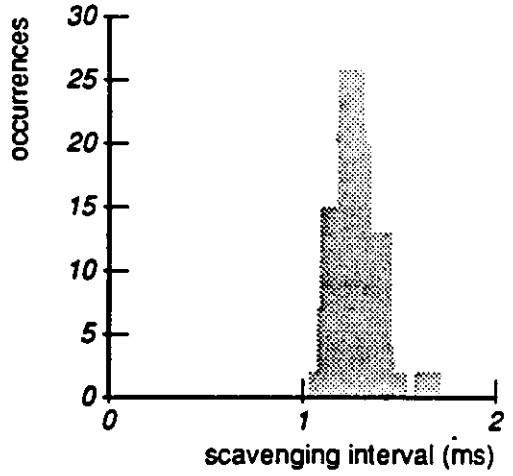
quick (1000 iter.; 80kb surv.)



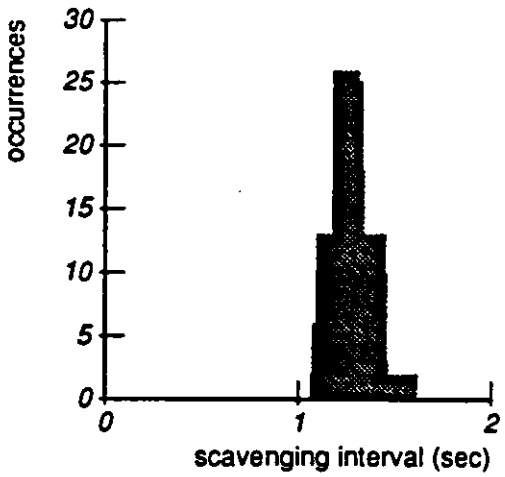
bubble (500 iter.; 10kb surv.)



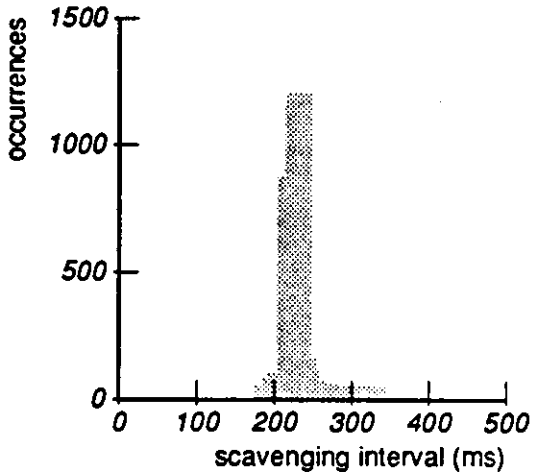
bubble (500 iter.; 80kb surv.)



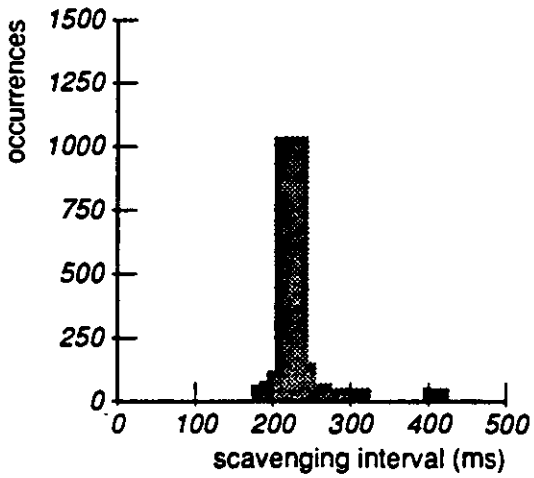
tree (150 iter.; 10kb surv.)



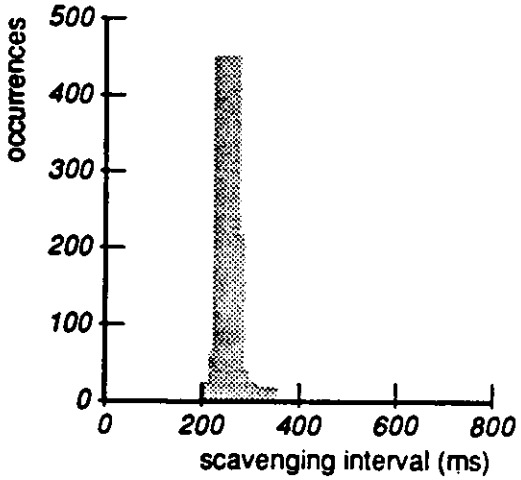
tree (150 iter.; 80kb surv.)



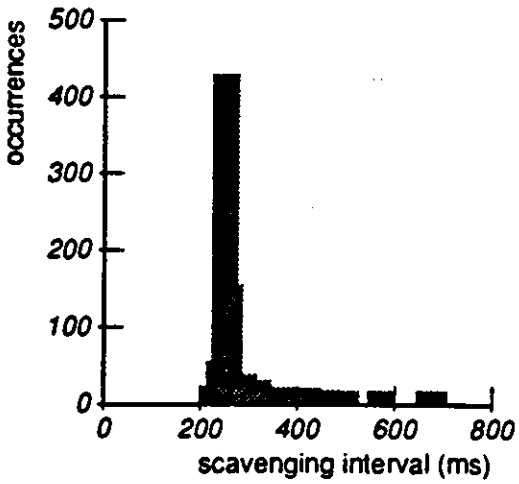
mm (50 iter.; 10kb surv.)



mm (50 iter.; 80kb surv.)



fft (150 iter.; 10kb surv.)



fft (150 iter.; 80kb surv.)

B.5.3. Store-checking overhead

This section presents measurements of the amount of time spend checking memory stores to see if the targets of the stores should be added to the remembered set. Store-checking takes two forms: in-line store-checking instructions embedded in native code generated by the SELF compiler, and a C++ routine called to check stores performed in by the memory system. In-line store-checking overhead is measured by causing the compiler to generate code counting occurrences of the store-checking instructions. Memory system store-checking overhead is measured using the UNIX *gprof* profiling facility.

B.5.3.1. In-line store-checking overhead

The in-line code checking each SELF-level memory store has the following form on the Sun-4 (SPARC):

```
subcc %receiver, new_old_boundary, %g0
blt _done                  // is receiver old?
andcc %arg, #Mem_Tag, %g0
beq _done                  // is arg a memOop?
subcc %arg_reg, new_old_boundary, %g0
bgt _done                  // is argument old?
mov %receiver, %o0         // pass rcvr to Remember
or %g0, #false, %o1        // pass false arg to Remember
call <Remember>
add %o7, #oopSize, %o7     //skip register mask on return
<in-line register mask>
```

The compiler inserted code counting the number of times each store-checking instruction was executed. The following tables indicate the number of times each instruction was executed for each benchmark run. (Note that an instruction in the delay slot of a branch is executed whether the branch is taken or not.)

Store-Checking Overhead for 10 KB Survivor Size

INSTRUCTION	RICHARDS	PERM	TOWERS	QUEENS	INTMM
subcc %receiver ...	84210912	144140012	131162014	40250012	4063012
blt _done	84210912	144140012	131162014	40250012	4063012
andcc %arg ...	84210912	144140012	131162014	40250012	4063012
beq _done	106983	43325167	49165015	233	1697155
subcc %arg_reg ...	106983	43325167	49165015	233	1697155
bgt _done	97673	5000	2015	233	2755
mov %receiver ...	97673	5000	2015	233	2755
or %g0 ...	13956	4999	2015	0	2648
call <Remember>	13956	4999	2015	0	2648
add %o7 ...	13956	4999	2015	0	2648
Total (counts)	253083916	519095367	491826147	120750968	15596800
Calculated time (ms)	16000	32000	31000	7500	970

INSTRUCTION	PUZZLE	QUICK	BUBBLE	TREE	MM	FFT
subcc %receiver ...	3688112	41273012	64967012	3001962	406312	1355872
blt _done	3688112	41273012	64967012	3001962	406312	1355872
andcc %arg ...	3688112	41273012	64967012	3001962	406312	1355872
beq _done	660216	8639000	1149620	902100	352660	1334645
subcc %arg_reg ...	660216	8639000	1149620	902100	352660	1334645
bgt _done	403308	1000	500	300	192660	1329525
mov %receiver ...	403308	1000	500	300	192660	1329525
or %g0 ...	542	900	495	270	192534	921340
call <Remember>	542	900	495	270	192534	921340
add %o7 ...	542	900	495	270	192534	921340
Total (counts)	13193010	141101736	197202761	10811496	2887178	12159976
Calculated time (ms)	820	8800	12000	680	180	760

Store-Checking Overhead for 80 KB Survivor Size

INSTRUCTION	RICHARDS	PERM	TOWERS	QUEENS	INTMM
subcc %receiver ...	84210912	144140012	131162014	40250012	4063012
blt _done	84210912	144140012	131162014	40250012	4063012
andcc %arg ...	84210912	144140012	131162014	40250012	4063012
beq _done	0	43305000	49165000	0	1697155
subcc %arg_reg ...	0	43305000	49165000	0	1697155
bgt _done	0	5000	2000	0	2755
mov %receiver ...	0	5000	2000	0	2755
or %g0 ...	0	5000	2000	0	2648
call <Remember>	0	5000	2000	0	2648
add %o7 ...	0	5000	2000	0	2648
Total (counts)	252632736	519055036	491826042	120750036	15596800
Calculated time (ms)	16000	32000	31000	7500	970

INSTRUCTION	PUZZLE	QUICK	BUBBLE	TREE	MM	FFT
subcc %receiver ...	3688112	41273012	64967012	3001962	406312	1355872
blt _done	3688112	41273012	64967012	3001962	406312	1355872
andcc %arg ...	3688112	41273012	64967012	3001962	406312	1355872
beq _done	660216	8639000	505000	752100	326501	1310319
subcc %arg_reg ...	660216	8639000	505000	752100	326501	1310319
bgt _done	403308	1000	500	300	166501	1305199
mov %receiver ...	403308	1000	500	300	166501	1305199
or %g0 ...	542	900	500	300	166399	908910
call <Remember>	542	900	500	300	166399	908910
add %o7 ...	542	900	500	300	166399	908910
Total (counts)	13193010	141101736	195913536	10511586	2704137	12025382
Calculated time (ms)	820	8800	12000	660	170	750

The time overhead for store-checking in each benchmark was calculated by multiplying the total number of instructions executed by the SPARC cycle time of 62.5 nanoseconds. All calculated times are specified to two significant digits.

B.5.3.2. Memory-system store-checking overhead

The user CPU time spent in the memory-system C++ store-checking routine is given in the following table. The measurements were performed using the UNIX *gprof* profiler, which uses a time quantum of 10 ms. The times given in the table does not include store-checking time on behalf of the scavenger because that time was already included in the measurements of scavenging pause time; with that exclusion, only two benchmarks, *mm* and *fft*, had any memory-system store-checking overhead:

Benchmark	Store-Check CPU Time (ms)	
	10 KB Surv.	80 KB Surv.
mm	470	400
fft	2420	2230

B.5.3.3. Remembering time

The following table indicates the amount of user CPU time spent in adding objects to the remembered set. The given times do not include time spent remembering objects on behalf of the scavenger, since that time was already counted in the pause times. The measurements were performed using the *gprof* profiler.

Benchmark	Remembering CPU Time (ms)	
	10 kb Surv.	80 kb Surv.
richards	180	0
perm	20	10
towers	10	20
queens	0	30
intmm	10	10
puzzle	0	0
quick	0	20
bubble	20	0
tree	0	0
mm	470	400
fft	2420	2230

B.5.4. Page faults

Scavenging only caused page faults in one case: the *fft* benchmark with the desired survivor size set at 10 kilobytes. The following table shows the distribution of page faults for that case:

FFT Benchmark (10kb Survivors) Scavenging Page Faults	
Page Faults	Occurrences
0	738
1	3
2	0
3	1

B.5.5. Total scavenging overhead

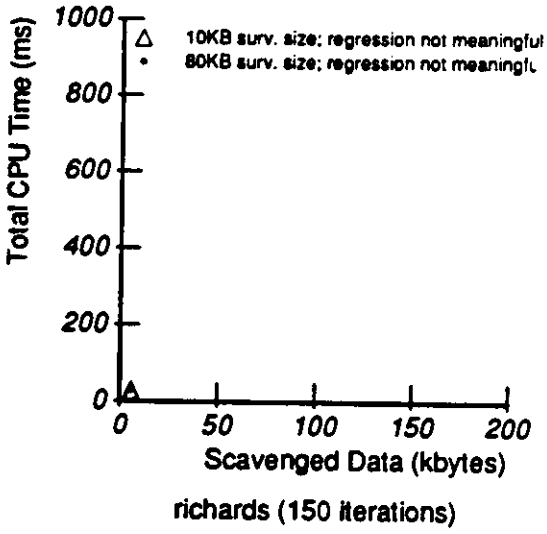
The following table indicates the total scavenging overhead for the benchmarks, calculated from the measurements of pause time, store-checking time, and remembering time in previous sections.

Bench-mark	Itera-tions	Surv. Size (KB)	Scavenging Pause Overhead (%)	Store-Checking Overhead (%)	Remembering Overhead (%)	Total Scavenging Overhead (%)
richards	150	10	0.01	3.7	0.04	3.7
		80	0.01	3.7	0.0	3.7
perm	1000	10	0.009	4.8	0.004	4.8
		80	0.020	4.8	0.002	4.8
towers	1000	10	0.009	3.4	0.001	3.4
		80	0.031	3.4	0.002	3.4
queens	1000	10	0.28	1.4	0.0	1.7
		80	0.77	1.4	0.006	2.2
inumm	500	10	0.97	0.20	0.002	1.2
		80	0.95	0.20	0.002	1.2
puzzle	100	10	0.51	0.15	0.0	0.66
		80	0.52	0.15	0.0	0.67
quick	1000	10	1.6	1.3	0.0	2.9
		80	1.6	1.3	0.003	2.9
bubble	500	10	0.018	1.4	0.002	1.4
		80	0.049	1.4	0.0	1.4
tree	150	10	17.	0.25	0.0	17.
		80	13.	0.25	0.0	13.
mm*	50	10	8.5	0.11	0.080	8.7
		80	19.	0.090	0.063	19.
fft*	10	10	15.	1.4	1.1	18.
		80	14.	1.3	0.95	16.

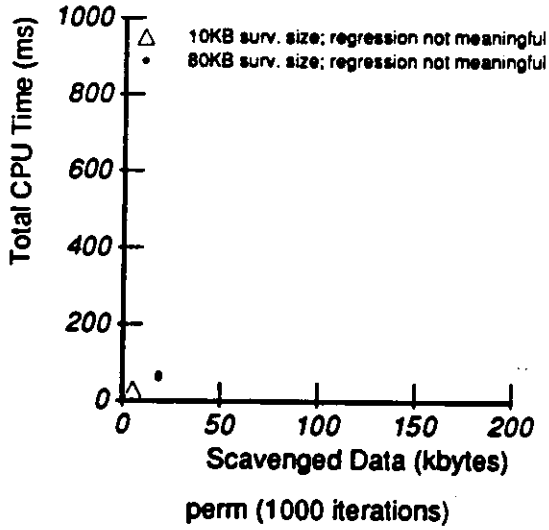
* Uses floating-point numbers

B.6. Bytes scavenged versus pause times

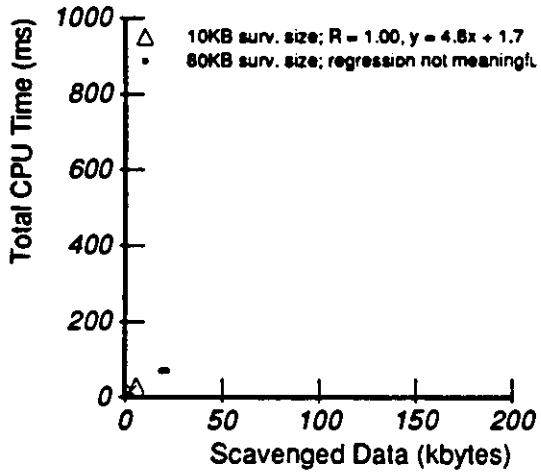
This section presents measurements testing one of the assumptions behind the demographic feedback-mediated tenuring—namely, that scavenging pause times are proportional to the amount of data scavenged. If that is true, then by keeping the amount of scavenged data below a constant ceiling, the tenuring algorithm would also keep the pause time bounded by a desired maximum pause time. To test that assumption, the scavenger was instrumented to record the amount of scavenged data and pause times for every scavenge in the benchmark runs. The following scatter plots show the results.



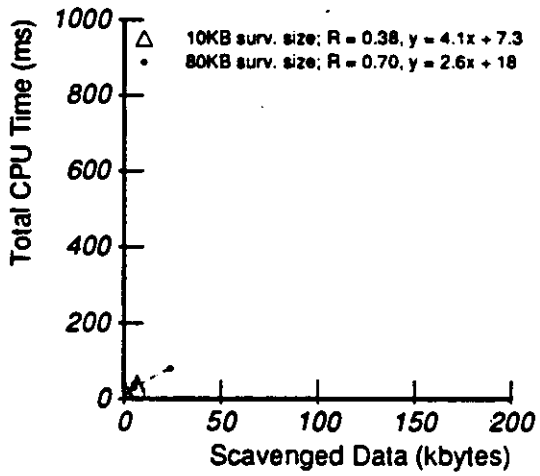
richards (150 iterations)



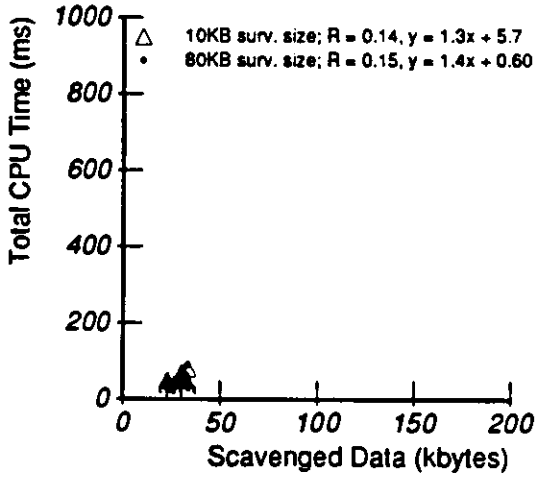
perm (1000 iterations)



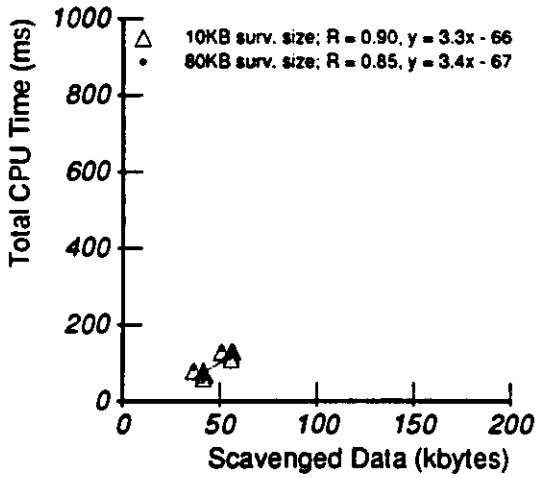
towers (1000 iterations)



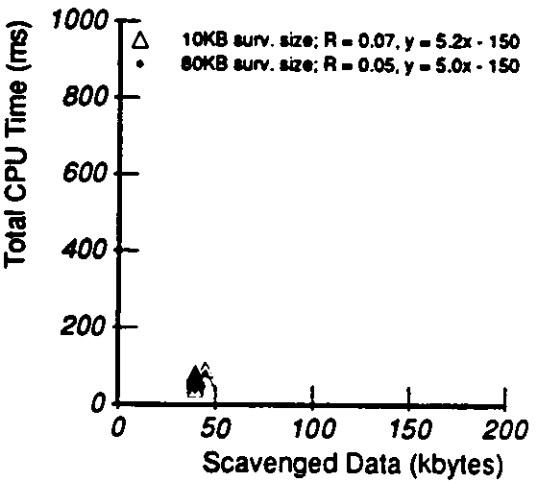
queens (1000 iterations)



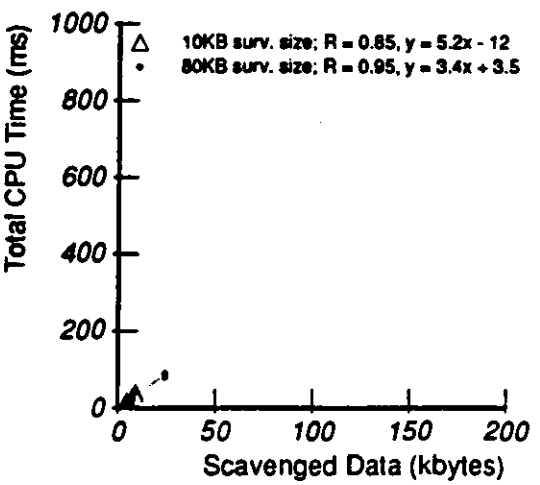
intmm (500 iterations)



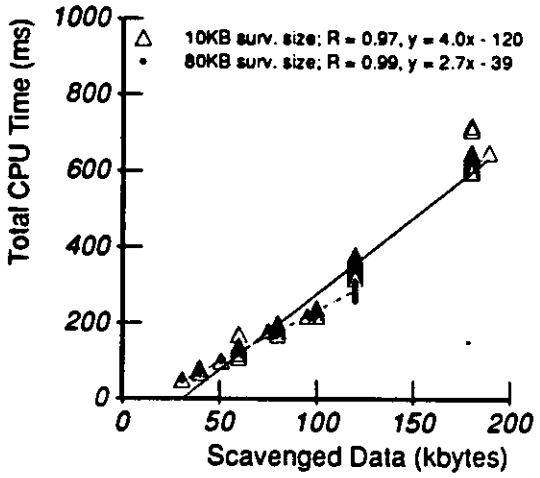
puzzle (100 iterations)



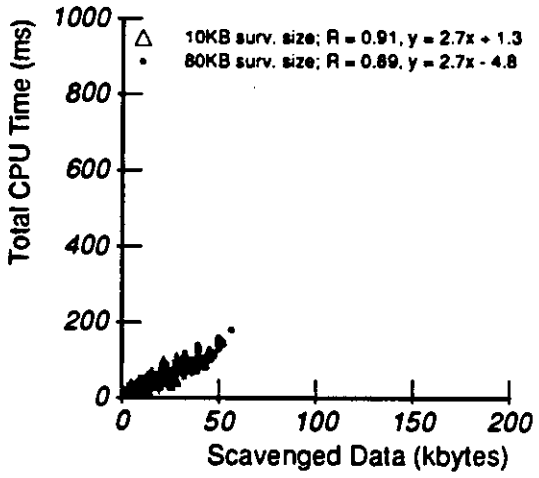
quick (1000 iterations)



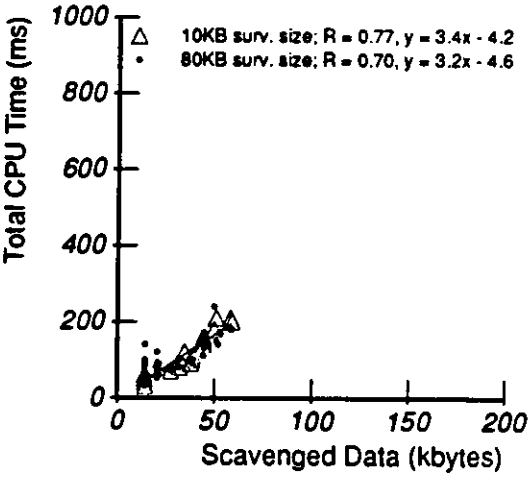
bubble (500 iterations)



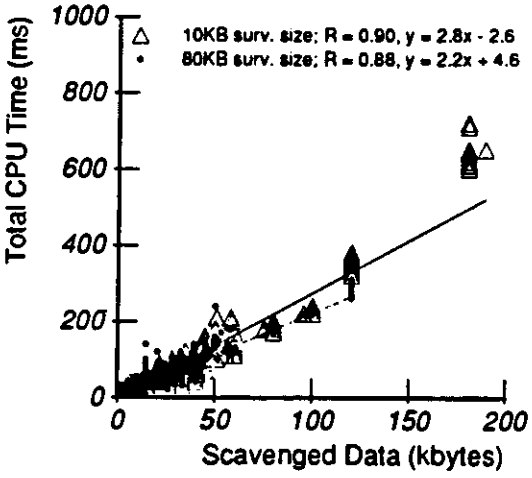
tree (150 iterations)



mm (50 iterations)



fft (10 iterations)



All Benchmarks

B.7. Amount of surviving and tenured data

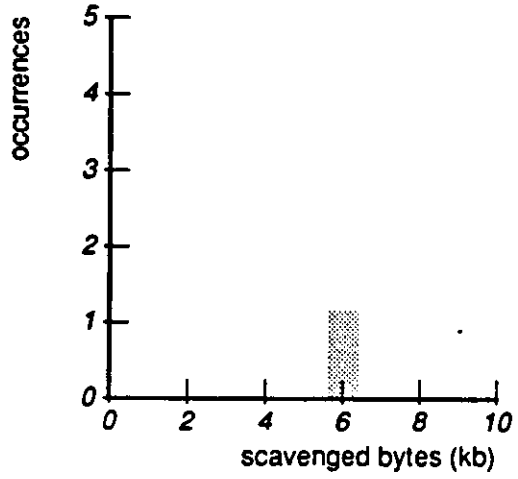
This section examines the amount of data that survives and the amount tenured during a scavenge. The following table shows the total amount of data scavenged and tenured for desired survivor sizes of 10 and 80 kilobytes over the benchmark runs.

Benchmark	Iterations	Total Scavenged Data (bytes)		Total Tenured Data (bytes)	
		10 KB Surv.	80 KB Surv.	10 KB Surv.	80 KB Surv.
richards	150	12032	12032	4676	0
perm	1000	10416	37496	4288	0
towers	1000	15160	81032	4684	2304
queens	1000	142864	717592	5912	18292
intmm	500	3256592	3256592	1624120	1624120
puzzle	100	1355456	1355456	777748	777748
quick	1000	8018432	8018432	4008296	4008296
bubble	500	52352	111080	25256	16300
tree	150	17384188	15575768	5986628	2986628
mm*	50	16636008	48404476	7853776	7831128
fft*	10	11193584	11405116	5684752	622748

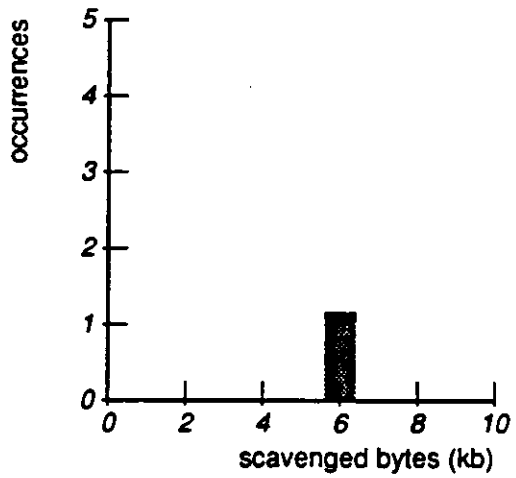
* Uses floating-point numbers

The following histograms show the distributions of surviving and tenured data per scavenge during the benchmark runs.

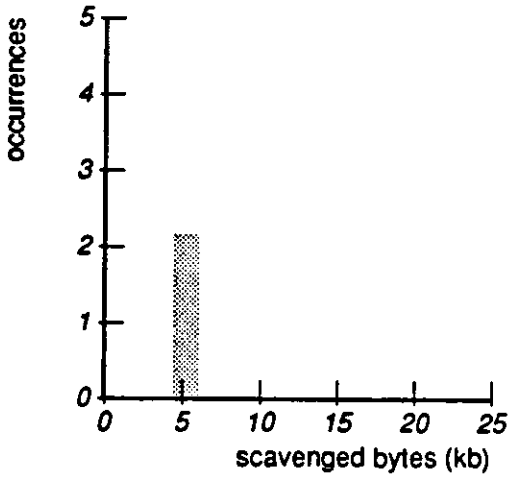
B.7.1. Surviving data



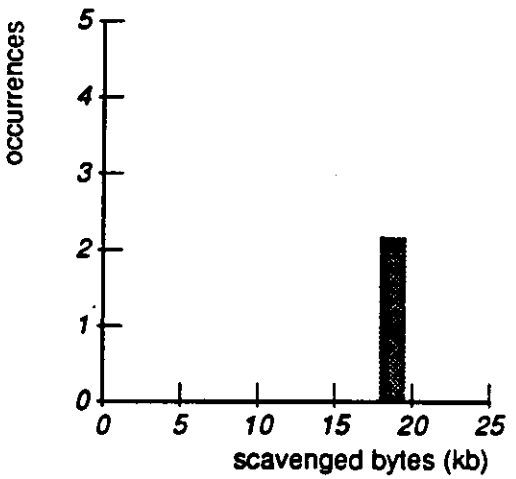
richards (150 iter.; 10kb surv.)



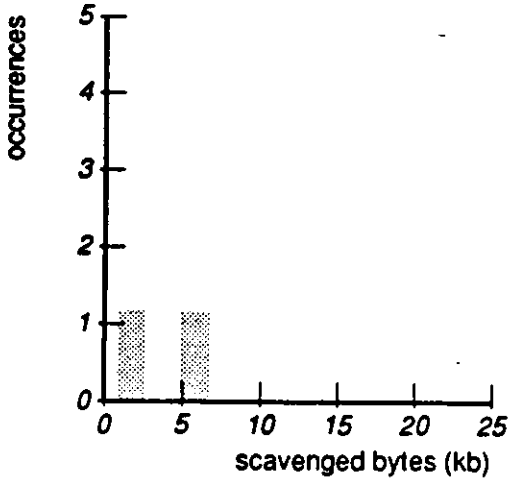
richards (150 iter.; 80kb surv.)



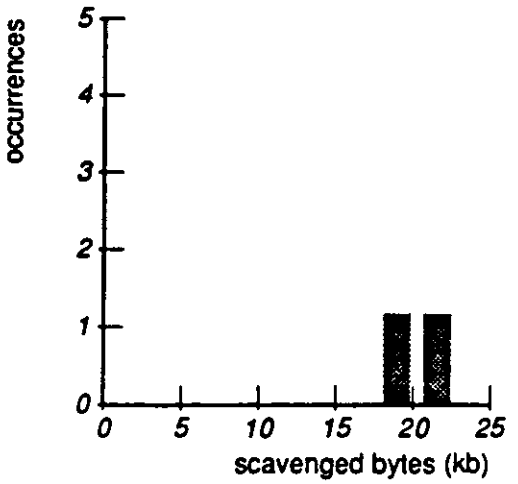
perm (1000 iter.; 10kb surv.)



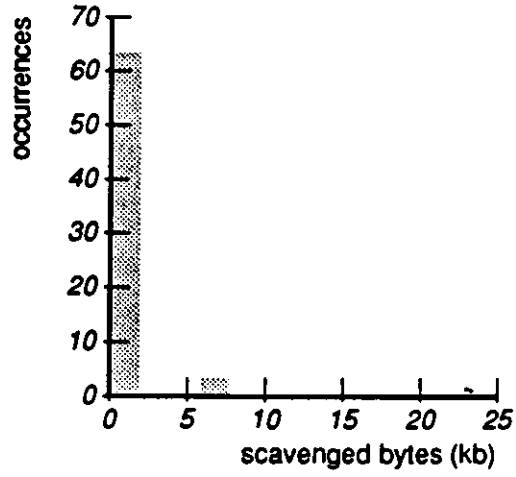
perm (1000 iter.; 80kb surv.)



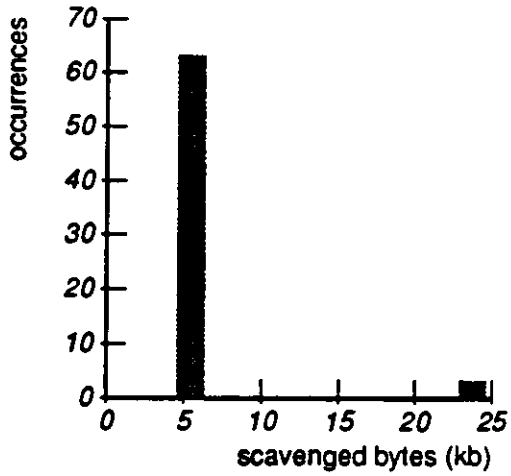
towers (1000 iter.; 10kb surv.)



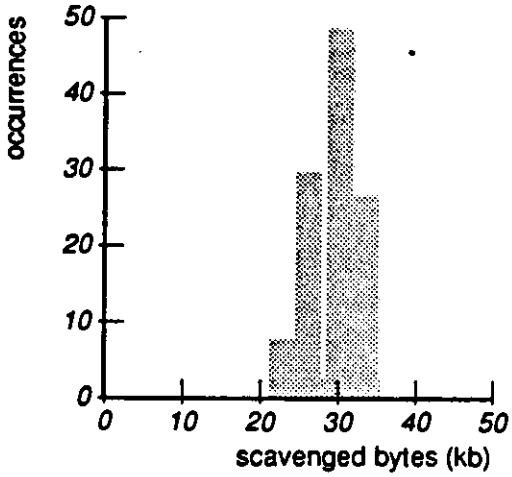
towers (1000 iter.; 80kb surv.)



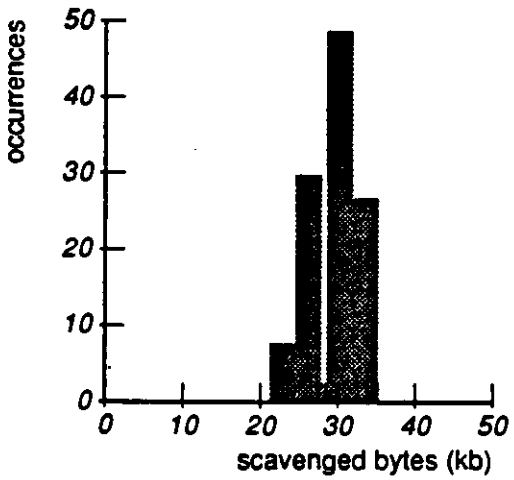
queens (1000 iter.; 10kb surv.)



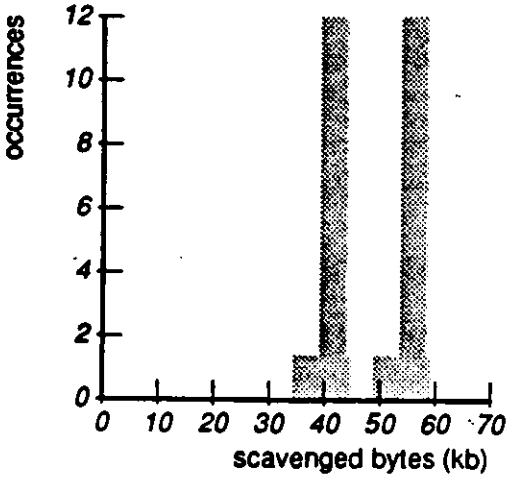
queens (1000 iter.; 80kb surv.)



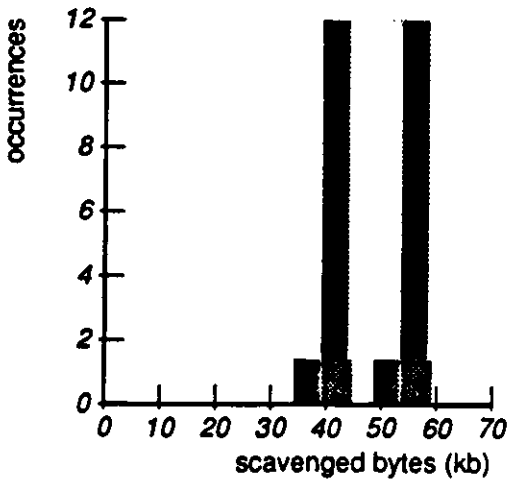
intmm (500 iter.; 10kb surv.)



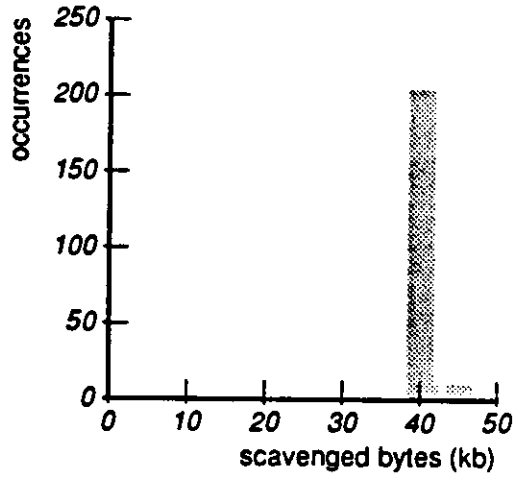
intmm (500 iter.; 80kb surv.)



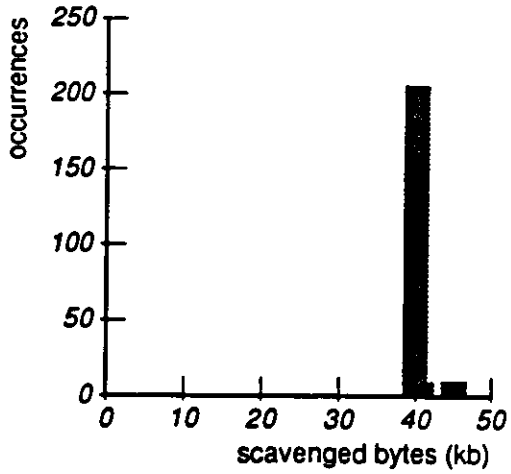
puzzle (100 iter.; 10kb surv.)



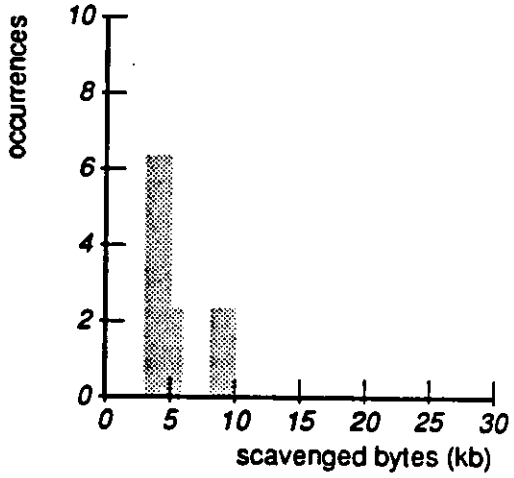
puzzle (100 iter.; 80kb surv.)



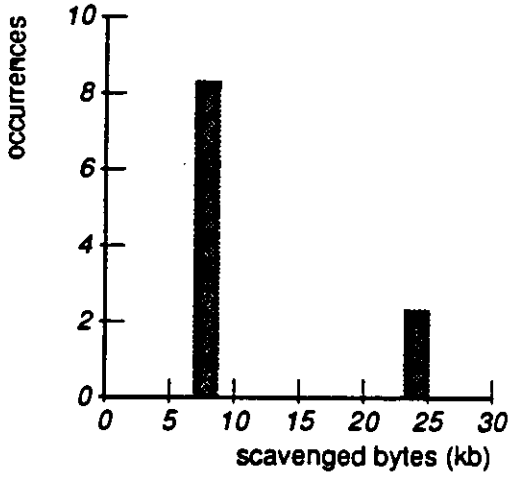
quick (1000 iter.; 10kb surv.)



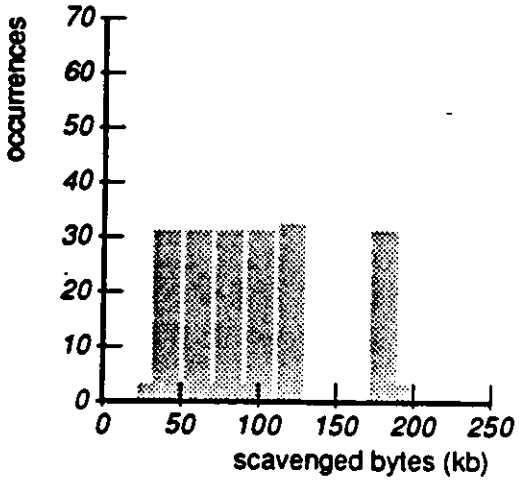
quick (1000 iter.; 80kb surv.)



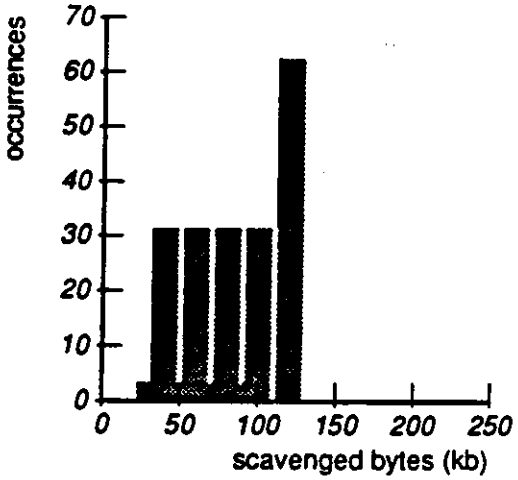
bubble (500 iter.; 10kb surv.)



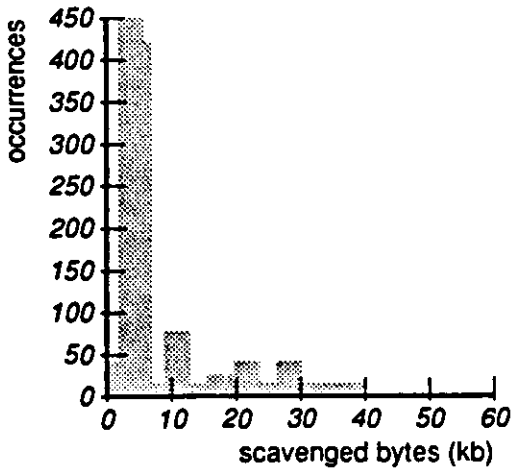
bubble (500 iter.; 80kb surv.)



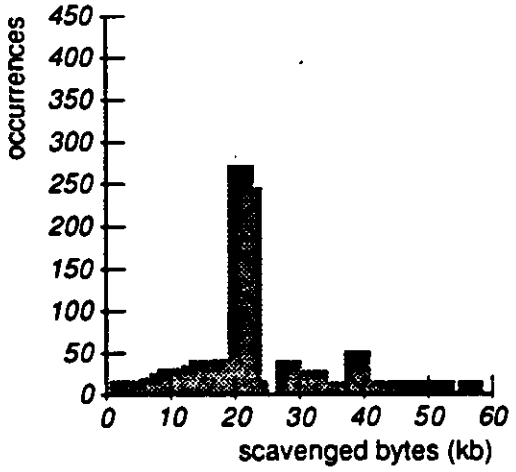
tree (150 iter.; 10kb surv.)



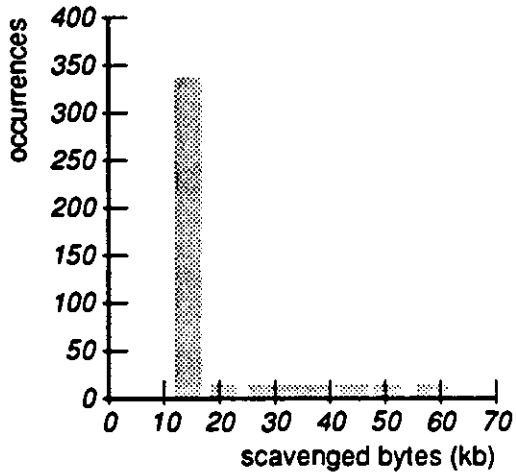
tree (150 iter.; 80kb surv.)



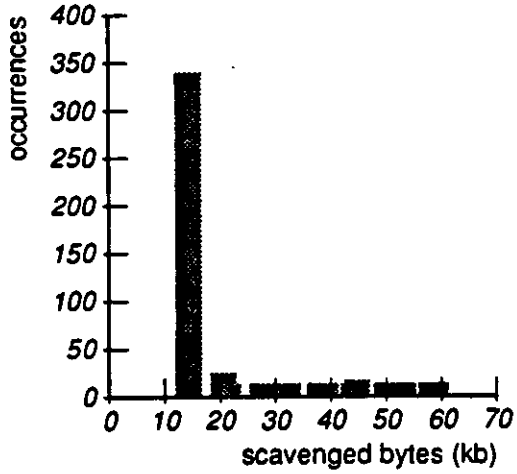
mm (50 iter.; 10kb surv.)



mm (50 iter.; 80kb surv.)

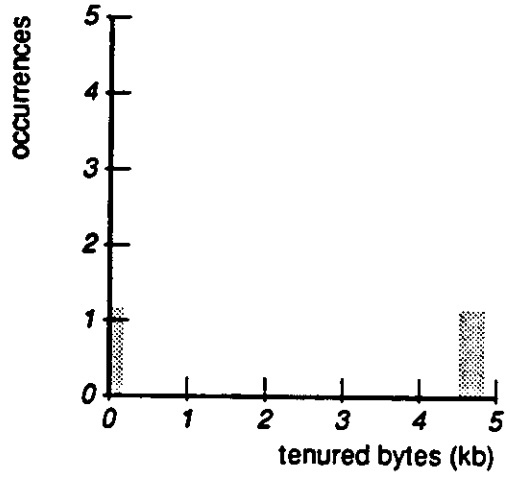


fft (150 iter.; 10kb surv.)

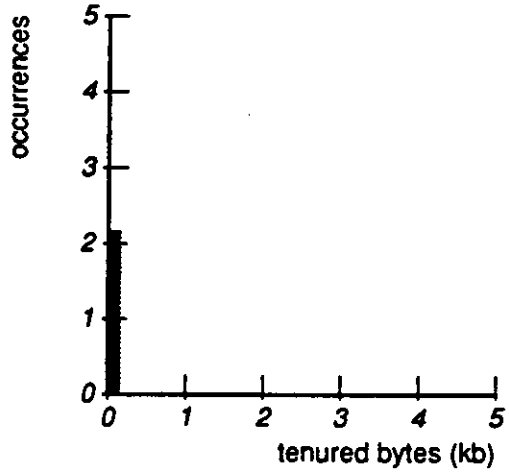


fft (150 iter.; 80kb surv.)

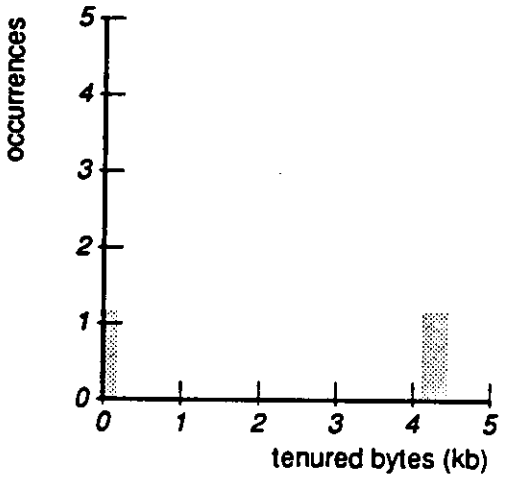
B.7.2. Tenured data



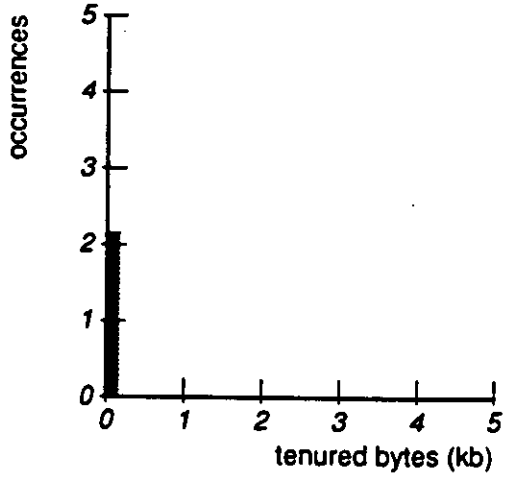
richards (150 iter.; 10kb surv.)



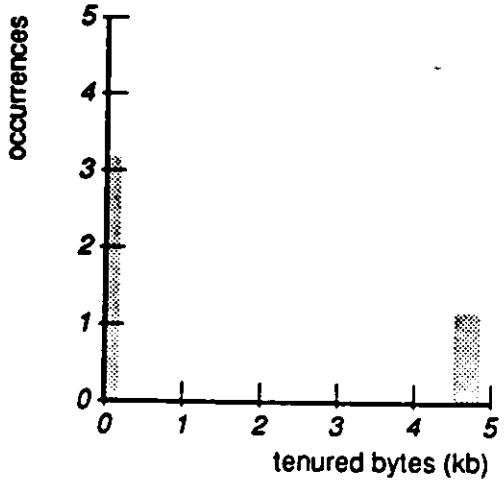
richards (150 iter.; 80kb surv.)



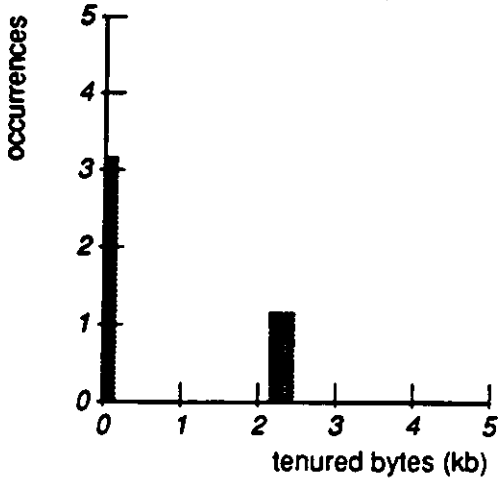
perm (1000 iter.; 10kb surv.)



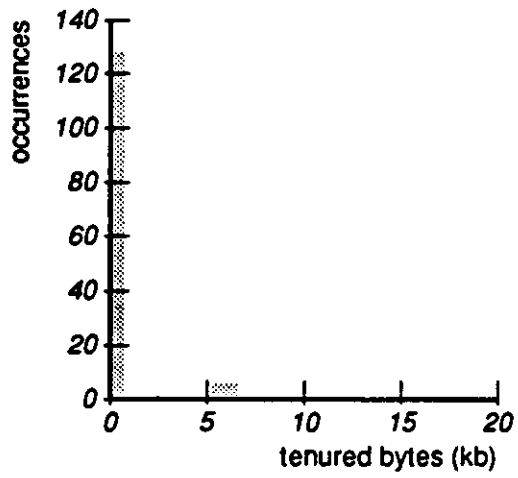
perm (1000 iter.; 80kb surv.)



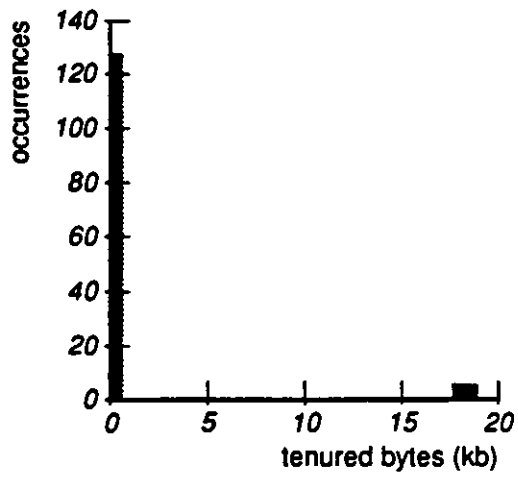
towers (1000 iter.; 10kb surv.)



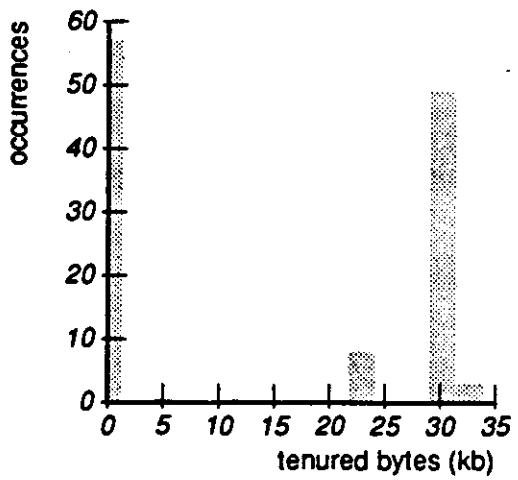
towers (1000 iter.; 80kb surv.)



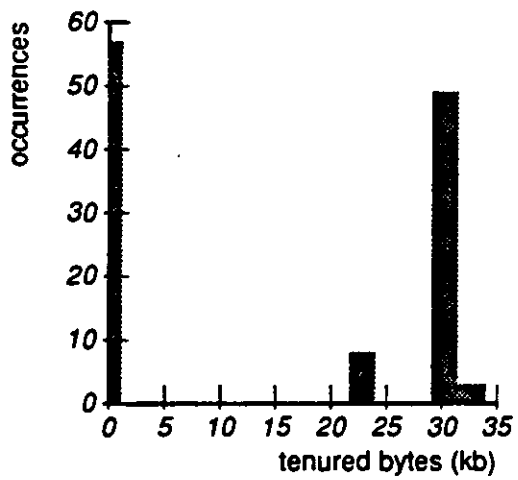
queens (1000 iter.; 10kb surv.)



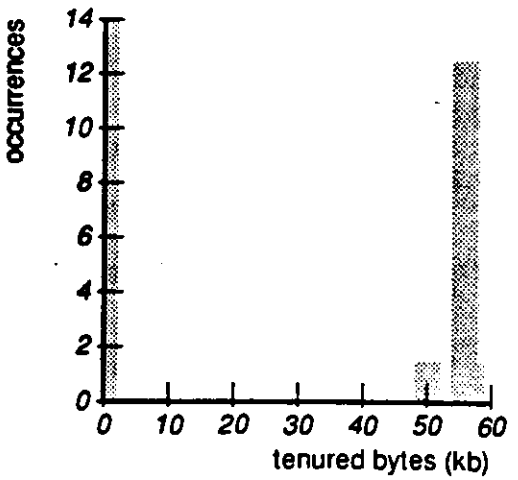
queens (1000 iter.; 80kb surv.)



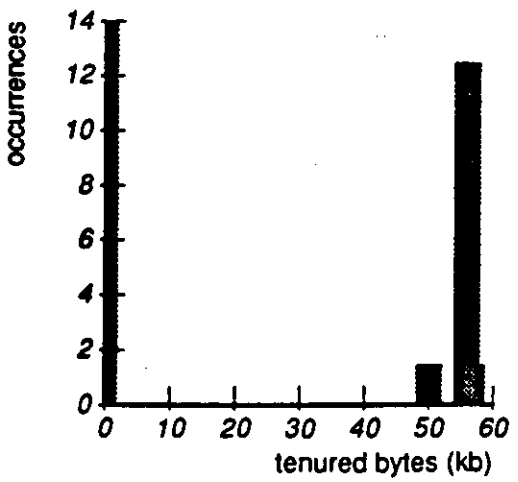
intmm (500 iter.; 10kb surv.)



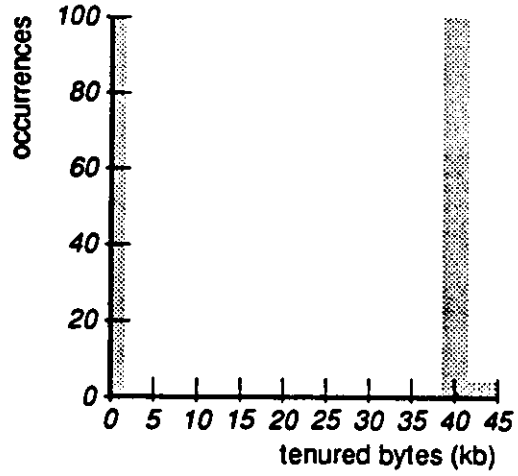
intmm (500 iter.; 80kb surv.)



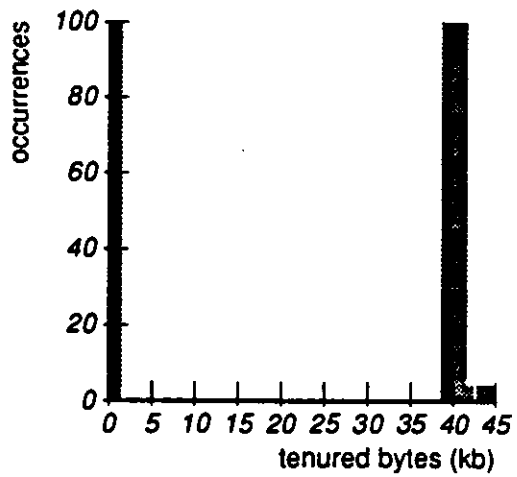
puzzle (100 iter.; 10kb surv.)



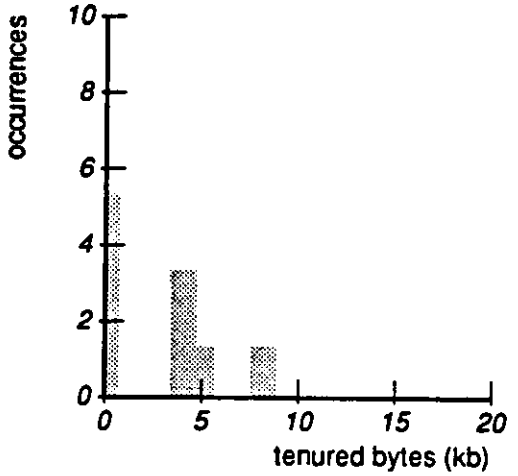
puzzle (100 iter.; 80kb surv.)



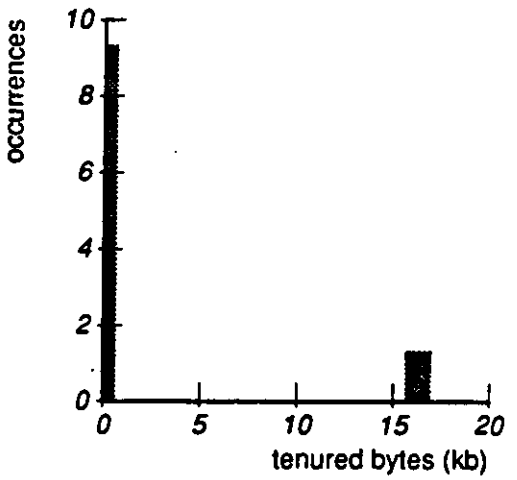
quick (1000 iter.; 10kb surv.)



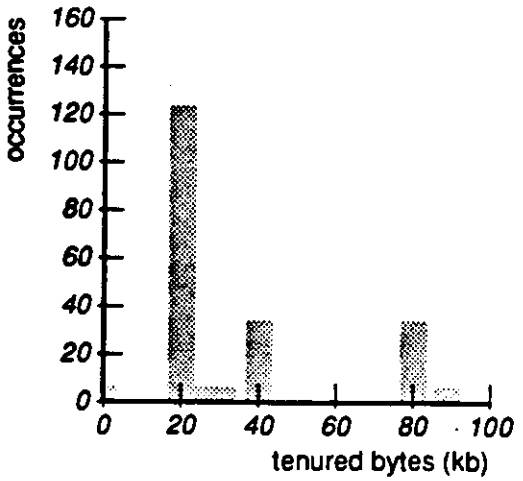
quick (1000 iter.; 80kb surv.)



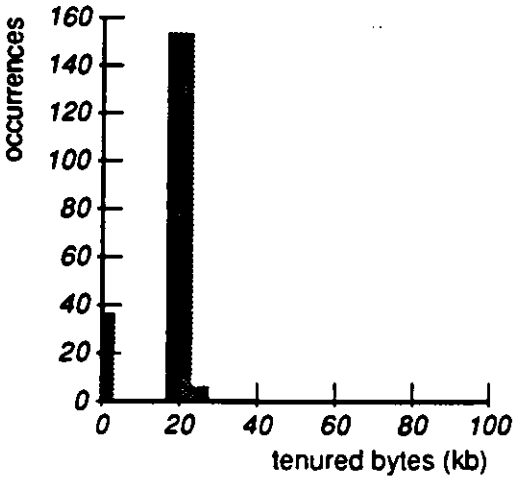
bubble (500 iter.; 10kb surv.)



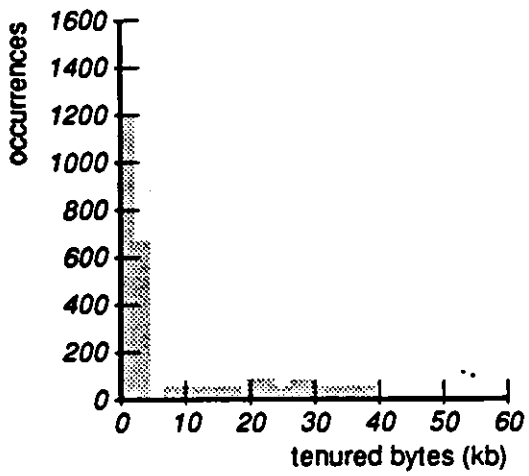
bubble (500 iter.; 80kb surv.)



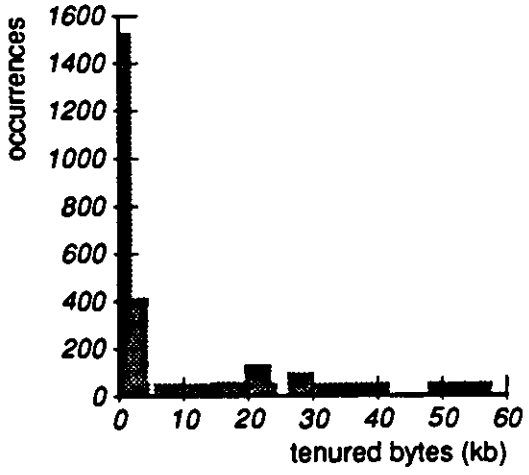
tree (150 iter.; 10kb surv.)



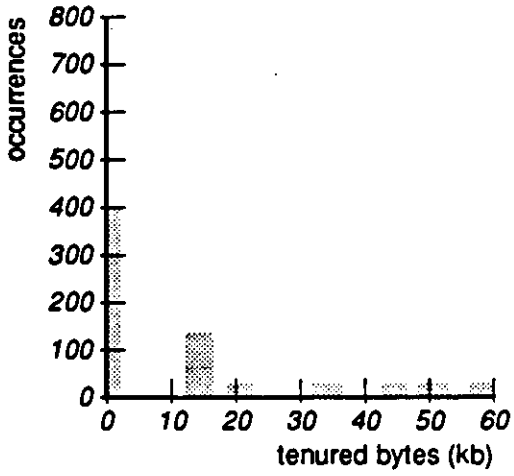
tree (150 iter.; 80kb surv.)



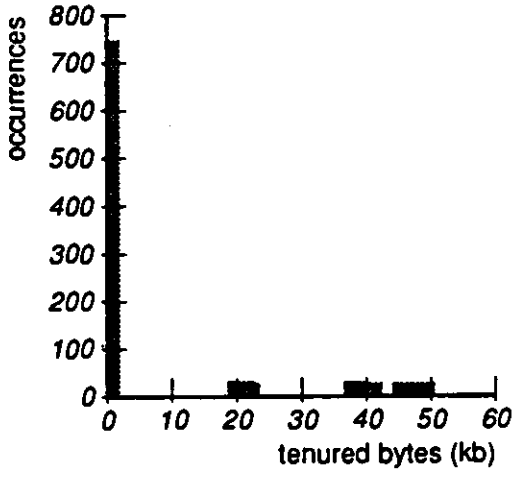
mm (50 iter.; 10kb surv.)



mm (50 iter.; 80kb surv.)



fft (150 iter.; 10kb surv.)



fft (150 iter.; 80kb surv.)

B.8. Garbage collection measurements

To gain a rough idea of the performance of the garbage collector, I measured the time required to perform a full mark-and-sweep garbage collection following a single iteration of each of the benchmarks. I performed the measurement ten times to gauge the range of results. The following tables present the raw measurements.

Collection Time (ms)				Page Faults
Total	User	System	Real	
3150	3020	130	3190	1
3200	3060	140	3210	0
3090	2940	150	3100	0
3150	3010	140	3180	0
3190	3070	120	3210	0
3150	3030	120	3170	0
3080	3020	60	3100	0
3150	3020	130	3170	0
3130	3050	80	3130	0
3140	2990	150	3170	0

	Space Usage (bytes)		
	Eden	To	Old
Before	9584	14392	1892396
After	7596	8904	1336948

References

[AgH87] Gul Agha and Carl Hewitt. *Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming*. In [ShW87], pp. 49-74.

[Bak78] Henry G. Baker, Jr. List Processing in Real Time on a Serial Computer. *Communications of the ACM* 21(4), April 1978.

[BaS83] Stoney Ballard and Stephen Shirron. The Design and Implementation of VAX/Smalltalk-80. In Krasner [Kra83], pp. 127-150.

[Bay84] Duane Bay. Dorado Benchmarks. *Smalltalk-80 Newletter*, September 1984. Palo Alto, CA.

[Bay85] Duane Bay. New Implementations Unveiled. *Smalltalk-80 Newletter*, October 1985. Xerox PARC/SCL, Palo Alto, CA.

[BDG88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System Specification. X3J13 Document 88-002R, June 1988. Also published as *SIGPLAN Notices* 23 (Special Issue), September 1988.

[BoS83] Daniel G. Bobrow and Mark Stefik. The LOOPS manual. Xerox Corporation, 1983.

[BoO87] Alan Borning and Tim O'Shea. Deltatalk: An Empirically and Aesthetically Motivated Simplification of the Smalltalk-80 Language. In *Proceedings of ECOOP '87: European Conference on Object-Oriented Programming*, pp. 1-10, Paris, France, 1987. Published as Springer-Verlag Lecture Notes in Computer Science 276, 1987.

[Bor81] Alan Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems* 3(4): 353-387, October 1981.

[Bor86] A. H. Borning. Classes Versus Prototypes in Object-Oriented Languages. In *Proceedings of the Fall Joint Computer Conference* (ACM and IEEE Computer Society), pp. 36-40, Dallas, Texas, 1986.

[Bro85] D. R. Brownbridge. Cyclic Reference Counting for Combinator Machines. In *Functional Programming Languages and Computer Architecture*, Jean-Pierre Jouannaud (editor), Springer-Verlag Lecture Notes in Computer Science 201, 1985, pp. 273-288.

[CaW86] Patrick J. Caudill and Allen Wirfs-Brock. A Third Generation Smalltalk-80™ implementation. In *OOPSLA '86 Conference Proceedings*, pp. 119-130, Portland, Oregon, 1986. Published as *SIGPLAN Notices* 21(11), November 1986.

[Cha] Craig D. Chambers. Ph.D. dissertation, Stanford University. Forthcoming.

[ChU88] Craig Chambers and David Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. Computer Systems Laboratory, Stanford University. In preparation, November 1988.

[Che70] C. J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM* 13(11): 677-678, November 1970.

[Coh81] Jacques Cohen. Garbage Collection of Linked Data Structures. *Computing Surveys* 13(3), 341-367, September 1981.

[Col60] George E. Collins. A Method for Overlapping and Erasure of Lists. *Communications of the ACM* 3(12): 655-657, December 1960.

[DaN66] Ole-Johan Dahl and Kristen Nygaard. SIMULA—an ALGOL-Based Simulation Language. *Communications of the ACM* 9(9): 671-678, September 1966.

[DaK85] William J. Dally and James T. Kajiya. An Object Oriented Architecture. In *Conference Proceedings of the 12th Annual International Symposium on Computer Architecture*, pp. 154-161, Boston, Massachusetts, 1985. Published as *SIGARCH Newsletter* 13(3), June 1985.

[DAm83] Bruce D'Ambrosio. Smalltalk-80 Language Measurement—Dynamic Use of Compiled Methods. In David A. Patterson, editor, *Smalltalk on a RISC: Architectural Investigations*, pages 110-125. Proceedings of CS292R. Computer Science Division, University of California, Berkeley, California, April 1983.

[DaT88] Scott Danforth and Chris Tomlinson. Type Theories and Object-Oriented Programming. *ACM Computing Surveys* 20(1): 29-72, March 1988.

[Deu83] L. Peter Deutsch. The Dorado Smalltalk-80 Implementation: Hardware Architecture's Impact on Software Architecture. In Krasner [Kra83], pp. 113-126.

[DeB76] L. Peter Deutsch and Daniel G. Bobrow. An Efficient, Incremental, Automatic Garbage Collector. *Communications of the ACM* 19(9): 522-526, September 1976.

[DeS84] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages*, pp. 297-302, Salt Lake City, Utah, 1984.

[Dre85] Gary L. Drescher. ObjectLISP for Experienced LISP Programmers. LMI, 1000 Massachusetts Avenue, Cambridge, MA 02138, 1985.

[Fal83] Joseph R. Falcone. The Analysis of the Smalltalk-80 System at Hewlett-Packard. In Krasner [Kra83], pp. 207-237.

[FeY69] Robert R. Fenichel and Jerome C. Yochelson. A LISP Garbage-Collector for Virtual-Memory Computer Systems. *Communications of the ACM* 12(11): 611-612, November 1969.

[GoR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.

[Gol84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, Massachusetts, 1984.

[GKM82] S.L. Graham, P.B. Kessler, and M.K. McKusick. gprof: A Call Graph Execution Profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pp. 120-126, 1982. Published as *SIGPLAN Notices* 17(6), June 1982.

[HaN87] Brent Hailpern and Van Nguyen. A Model for Object-Based Inheritance. In Shriver and Wegner [ShW87], pp. 147-164.

[HaW67] B. K. Haddon and W. M. Waite. A Compaction Procedure for Variable Length Storage Elements. *Computer Journal* 10: 162-165, August 1967.

[HaO88] Daniel C. Halbert and Patrick D. O'Brien. Using Types and Inheritance in Object-Oriented Languages. Get ref.

[HaF87] Christopher T. Haynes and Daniel P. Friedman. Embedding Continuations in Procedural Objects. *ACM Transactions on Programming Languages and Systems* 9(4): 582-598, October 1987.

[KeR78] Brian. W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Eaglewood Cliffs, New Jersey, 1978.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, Reading, Massachusetts, second edition 1973.

[Kra83] Glenn Krasner, editor. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, Reading, Massachusetts, 1983.

[LaL86] Wilf R. LaLonde. Why Exemplars Are Better Than Classes. Technical Report SCS-TR-93, School of Computer Science, Carleton University, May 1986.

[LTP86] Wilf R. LaLonde, Dave A. Thomas, and John R. Pugh. An Exemplar Based Smalltalk. In *OOPSLA '86 Conference Proceedings*, pp. 322-330, Portland, Oregon, 1986. Published as *SIGPLAN Notices* 21(11), November 1986.

[LGF86] David M. Lewis, David R. Galloway, Robert J. Francis, and Brian W. Thomson. Swamp: A Fast Processor for Smalltalk-80. In *OOPSLA '86 Conference Proceedings*, pp. 131-139, Portland, Oregon, 1986. Published as *SIGPLAN Notices* 21(11), November 1986.

[Lie86] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *OOPSLA '86 Conference Proceedings*, pp. 214-223, Portland, Oregon, 1986. Published as *SIGPLAN Notices* 21(11), November 1986.

[LiH83] Henry Lieberman and Carl Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM* 26(6), June 1983.

[McC83a] Kim McCall. The Smalltalk-80 Benchmarks. In Krasner [Kra83], pp. 153-174.

[McC60] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM* 3(4), 184-195, April 1960.

[McC83b] Paul L. McCullough. Implementing the Smalltalk-80 System: The Tektronix Experience. In Krasner [Kra83], pp. 59-78.

[Mey87] Bertrand Meyer. Eiffel: Programming for Reusability and Extendibility. *SIGPLAN Notices* 22(2): 85-94, February 1987.

[Min63] M. L. Minsky. A LISP Garbage Collector Algorithm Using Serial Secondary Storage. Memo 58, Artificial Intelligence Project, Massachusetts Institute of Technology, October 1963.

[Moo84] David A. Moon. Garbage Collection in a Large Lisp System. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pp. 235-246, Austin, Texas, 1984.

[Moo86] David A. Moon. Object-Oriented Programming with Flavors. In *OOPSLA '86 Conference Proceedings*, pp. 1-8, Portland, Oregon, 1986. Published as *SIGPLAN Notices* 21(11), November 1986.

[Mor78] F. Lockwood Morris. A Time- and Space-Efficient Garbage Compaction Algorithm. *Communications of the ACM* 21(8): 662-665, August 1978.

[OSh86] Tim O'Shea. The Learnability of Object-Oriented Programming Systems. In *OOPSLA '86 Conference Proceedings*, p. 502, Portland, Oregon, 1986. Published as *SIGPLAN Notices* 21(11), November 1986.

[SCW85] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. *Trellis Object-Based Environment: Language Reference Manual.* Digital Equipment Corporation Technical Report DC TR-372, November 1985.

[ScW67] H. Schorr and W. M. Waite. An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures. *Communications of the ACM* 10(8): 501-506, August 1967.

[Sha88] Robert A. Shaw. *Empirical Analysis of a LISP System.* Ph.D. dissertation, Stanford University, February 1988. Published as Technical Report CSL-TR-88-351, Computer Systems Laboratory, February 1988.

[ShW87] Bruce Shriver and Peter Wegner, editors. *Research Directions in Object-Oriented Programming*. The MIT Press, Cambridge, Massachusetts, 1987.

[Smi86] Randall B. Smith. The Alternate Reality Kit: An Animated Environment for Creating Interactive Simulations. In *Proceedings of the 1986 IEEE Computer Society Workshop on Visual Languages*, pp 99-106, Dallas, Texas, 1986.

[Sta80] Thomas A. Standish. *Data Structure Techniques*. Addison-Wesley, Reading, Massachusetts, 1980.

[Ste87] Lynn Andrea Stein. Delegation Is Inheritance. In *OOPSLA '87 Conference Proceedings*, pp. 138-146, Orlando, Florida, 1987. Published as *SIGPLAN Notices* 22(12), December 1987.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 1986.

[SKA84] Norihisa Suzuki, Koichi Kubota, and Takashi Aoki. Sword32: A Bytecode Emulating Microprocessor for Object-Oriented Languages. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*, pp. 389-397, Tokyo, Japan.

[SuT84] Norihisa Suzuki and Minoru Terada. Creating Efficient Systems for Object-Oriented Languages. In *Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages*, pp. 290-296, Salt Lake City, Utah, 1984

[UBF84] David Ungar, Ricki Blau, Peter Foley, Dain Samples, and David Patterson. Architecture of SOAR: Smalltalk on a RISC. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pp. 188-197, Ann Arbor, Michigan, 1984. Published as *SIGARCH Newsletter* 12(3), June 1984.

[Ung86] David Michael Ungar. *The Design and Evaluation of a High Performance Smalltalk System.* Ph.D. dissertation, the University of California at Berkeley, February 1986. Published by the MIT Press, Cambridge, Massachusetts, 1987.

[UnJ88] David Ungar and Frank Jackson. Tenuring Policies for Generation-Based Storage Reclamation. In *OOPSLA '88 Conference Proceedings*, pp. 1-17, San Diego, California, 1988. Published as *SIGPLAN Notices* 23(11), November 1988.

[UnP83] David M. Ungar and David A. Patterson. Berkeley Smalltalk: Who Knows Where the Time Goes. In Krasner [Kra83], pp. 189-206.

[UnS87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*, pp. 227-241, Orlando, Florida, 1987. Published as *SIGPLAN Notices* 22(12), December 1987.

[Weg87a] Peter Wegner. Dimensions of Object-Based Language Design. In *OOPSLA '87 Conference Proceedings*, pp. 168-182, Orlando, Florida, 1987. Published as *SIGPLAN Notices* 22(12), December 1987.

[Weg87b] Peter Wegner. The Object-Oriented Classification Paradigm. In [ShW87], pp. 479-560.

[Wei63] J. Weizenbaum. Symmetric List Processor. *Communications of the ACM* 6(9):524-544, September 1963.

[Wir83] Allen Wirfs-Brock. Design Decisions for Smalltalk-80 Implementors. In Krasner [Kra83], pp. 41-56.