Chapter 6 Overall System Architecture

The SELF compiler does not operate in isolation. It is an integral part of the whole SELF system, depending on the facilities provided by the rest of the system and constrained to satisfy requirements imposed by the rest of the system. To place the compiler in context, this chapter describes the overall architecture of the SELF system, focusing on the impact these other parts of the system have on the design of the compiler.* The following several chapters describe the compiler itself.

6.1 Object Storage System

The *object storage system* (also called the *memory system*) represents SELF objects and their relationships. It provides facilities for creating new objects and automatically reclaims the resources consumed by inaccessible objects. It supports modifying objects via programming and for scanning objects to locate all occurrences of certain kinds of references.

Much of the memory system design exploits technology proven in existing high-performance Smalltalk and Lisp systems. For minimal overhead in the common case, the SELF system represents object references using direct tagged pointers, rather than indirectly through an object table as do some Smalltalk systems. An early version of the SELF memory system was documented by Elgin Lee [Lee88]; a more recent version was described in [CUL89].

The following two subsections describe techniques for efficient object storage systems pioneered by the SELF implementation. Subsection 6.1.3 describes constraints placed on the compiler by SELF's garbage collection algorithm. Appendix A describes object formats in detail.

6.1.1 Maps

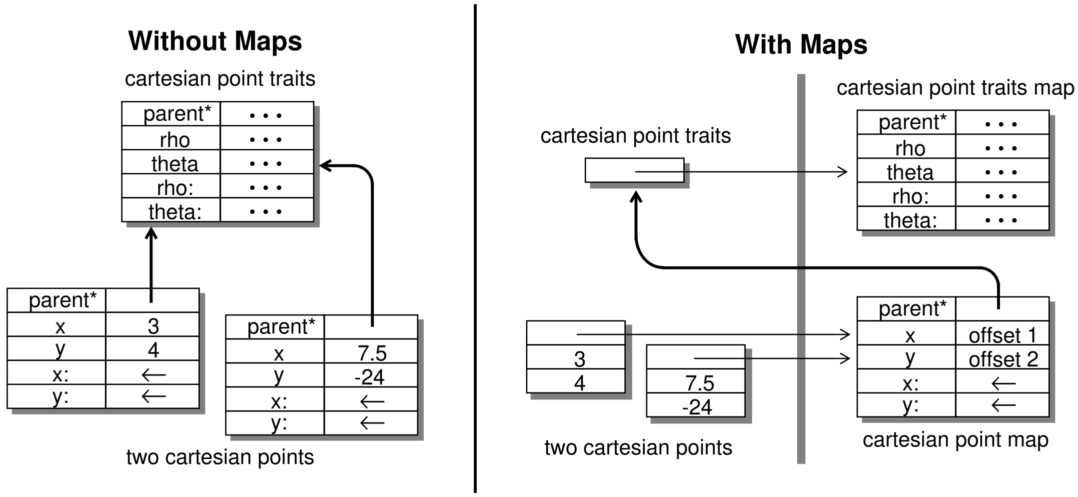
In traditional class-based languages, a class object contains the format (names and locations of the instance variables), methods, and superclass information for all its instances; the instances contain only the values of their instance variables and a pointer to the shared class object. Since SELF uses a prototype model, each object is self-sufficient, defining its own format, behavior, and inheritance. A straightforward implementation of SELF would therefore represent both the class-like format, method, and inheritance information and the instance-like state information in each SELF object. This representation would consume at least twice as much space as for a traditional class-based language.

Fortunately, the storage efficiency of classes can be regained even in SELF's prototype object model by observing that few SELF objects have totally unique format and behavior. Almost all objects are created by cloning some other object and then possibly modifying the values of the assignable slots. Wholesale changes in the format or inheritance of an object, such as those induced by the programmer, can be accomplished only by invoking special primitives. Therefore, a prototype and the objects cloned from it, identical in every way except for the values of their assignable slots, form what we call a *clone family.*

The SELF implementation uses *maps* to represent members of a clone family efficiently. In the SELF object storage system, objects are represented by the values of their assignable slots, if any, and a pointer to the object's map; the map is shared by all members of the object's clone family. For each slot in the object, the map contains the name of the slot, whether the slot is a parent slot, and either the offset within the object of the slot's contents (if it is an assignable slot) or the slot's contents itself (if it is a constant slot, such as a non-assignable parent slot). If the object has code (i.e., is

* Many of the techniques described in this chapter were designed by the SELF group as a whole and implemented by various members of the SELF group, including Elgin Lee, Urs Hölzle, David Ungar, and the author, and so should not be viewed as contributions solely attributable to the author.

a method), the map stores a pointer to a SELF byte code object representing the source code of the method (byte code objects are described further in section 6.2).



Maps are immutable so that they may be freely shared by objects in the same clone family. However, when the user changes the format of an object or the value of one of an object's constant slots, the map no longer applies to the object. In this case, a new map is created for the changed object, thus starting a new clone family. The old map still applies to any other members of the original clone family. If no other members exist (i.e., the modified object was the only member of its clone family), the old map will be garbage-collected later automatically.

From the implementation point of view, maps look much like classes, and achieve the same sorts of space savings for shared data. In addition, the map of an object conveys its static properties to the SELF compiler, much as does an instance's class in a class-based language. Nevertheless, maps are completely invisible to the SELF programmer. Programmers still operate in a world populated by self-sufficient objects that in principle could each be unique. The implementation simply is optimizing representation and execution based on existing usage patterns, i.e., the presence of clone families.

6.1.2 Segregation

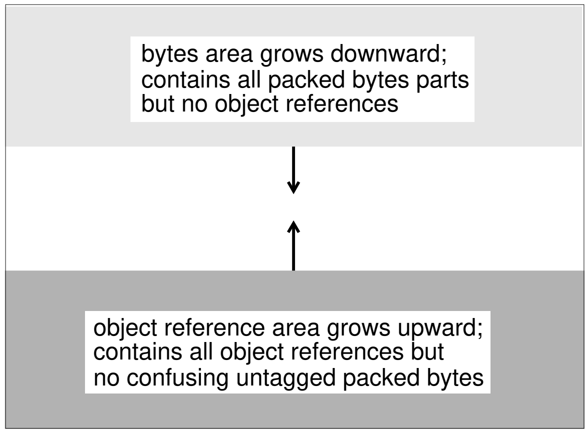
The memory system frequently scans all object references for those that meet some criterion:

- The scavenger scans all objects for references to objects in from-space as part of garbage collection.
- The programming primitives have to find and redirect all references to an object if its size changes and it has to be moved.
- The browser may need to search all objects for those that contain a reference to a particular object that interests the SELF user (i.e., following "backpointers").

To support these and other functions, the SELF implementation has been designed for rapid scanning of object references.

Ideally the system could just sweep through all of memory word-by-word to find object references matching the desired criterion. Unfortunately, since the elements of byte arrays are represented using packed bytes rather than tagged words (see Appendix A), byte array elements may *masquerade* as object references if byte arrays are scanned blindly. Most systems handle this problem by scanning the heap object-by-object rather than word-by-word. To scan an object, the system examines the header of the object to locate the part of the object containing object references and skip the part containing packed bytes (or other non-pointer data that could masquerade as a pointer). The pointer parts of the object are scanned, while the other parts are ignored. The scanner then proceeds to the next object. This procedure avoids the problems associated with scanning byte arrays, but slows down the scan with the overhead to parse object headers and compute object lengths.

The SELF system avoids the problems associated with scanning byte arrays without degrading the object reference scanning speed by *segregating* the packed untagged bytes from the other SELF objects. Each Generation Scavenging memory space (described more in the next section) is divided into two areas, one for the bytes parts of byte arrays and one for the rest of the data (including the object-reference part of byte arrays).* To scan all object references, only the object reference area of each space needs to be scanned (ignoring any scans for references to integers in the range [0..255], which require special support but almost never occur). This optimization speeds scans by eliminating the need to parse object headers.



A Segregated SELF Memory Space

To avoid slowing the tight scanning loop with an explicit end-of-space check, the word after the end of the space is temporarily replaced with a *sentinel* reference that matches the scanning criterion. This enables the scanner to check for the end of the space only on a matching reference, instead of on every word. Early measurements on a 68020-based Sun-3/50 showed that the SELF system scanned memory at the rate of approximately 3 megabytes per second. Measurements of the fastest Smalltalk-80 implementation on the same machine, ParcPlace Smalltalk-80, indicated a scanning speed for non-segregated memory spaces of only 1.6 megabytes per second. Current measurements indicate a scanning speed for SELF of 12 megabytes per second on a SPARC-based Sun-4/260 workstation.

For some kinds of scans, such as finding all objects that refer to a particular object (following backpointers), the scanner needs to find the objects that *contain* a matching reference, rather than the reference itself. In a system that scans object-by-object, this task is no more difficult than searching for references. However, in a system that scans word-by-word it may be difficult to locate the beginning of the object containing the matching reference. To support these kinds of scans without resorting to object-by-object scanning the SELF system specially tags the first header word of every object (called the *mark* word) to identify the beginning of the object. Thus, to find all objects containing a particular reference, the scanner proceeds normally, searching for matching references. Once a reference is found, the scanner locates the object containing the reference by scanning backwards to the object's mark word and then converting the mark word's address into an object reference by adding the right tag bits to the address.

6.1.3 Garbage Collection

The SELF implementation reclaims inaccessible objects using a version of *Generation Scavenging* [Ung84, Ung87] with *demographic feedback-mediated tenuring* [UJ88], augmented with a traditional mark/sweep collector to reclaim tenured garbage. The SELF heap is currently configured using a 200KB *eden* memory space for newly-allocated objects, a pair of 200KB *survivor* memory spaces for objects that have survived at least one scavenge but have not yet been tenured, and a 5MB *old* space for tenured objects.

The implementation of this algorithm imposes certain constraints on the compiler. The run-time system must be able to locate all object references embedded in compiled instructions whenever a scavenge or garbage collection occurs. Conversely, the garbage collector must be protected from examining any data values which could falsely masquerade as an object reference. In particular, the SELF compiler does not produce *derived pointers* to the interior of an object. This restriction allows the garbage collector to assume that all data tagged as an object reference really points to the

* This design is slightly different from segregation as described in [Lee88] and [CUL89], in which byte arrays were stored completely in the bytes area. The design was changed so that a byte array could have user-defined references to other objects in addition to its array of bytes.

beginning of an object in the heap, thus speeding the collector at some hopefully small cost in run-time efficiency for certain kinds of programs (notably those that iterate through arrays).

Generation Scavenging requires the compiler to generate *store checks*. Each store to a data slot in the heap needs to be checked to see whether it is creating a reference from an object in old-space to an object in new-space; all such references need to be recorded in a special table for use at scavenges. The current SELF implementation uses a *card marking* scheme similar to that used by some other systems [WM89].* Each card corresponds to a region of a SELF memory space (currently 128 bytes long), and records whether any of the words in the corresponding region contain pointers to new space. Whenever the compiler generates a store into an object that might be in old-space to an object that might be in new-space, the compiler also must generate code to mark the appropriate card for the modified data word.

Since stores into objects need to be fast, the SELF compiler attempts to generate code that is as fast as possible. If the compiler can prove that the target of the stored reference is an integer or a floating point immediate value (i.e., not a pointer), then no store check needs to be generated, since such a store does not create a reference from old- to new-space. Otherwise the compiler generates the following sequence (given in SPARC assembly syntax):

```
st   %dest, [%source + offset] ; do the store
add  %source, offset, %temp ; compute address of modified word
sra  %temp, log_base_2(card_size), %temp ; compute card index
stb  %g0, [%card_base + %temp] ; mark card by zeroing
```

The compiler generates code to shift the address of the modified data slot right by a number of bits equal to the $\log_2$ of the card size, adds it to the contents of a dedicated global register on the SPARC (a global variable on the Motorola 680x0), and zeros the byte at that address. Our system uses a whole byte per card, even though only a single bit is required to record whether the card is marked, since the store checking code would be slowed by the bit manipulation operations. Space for cards is allocated even for objects in new space so that the store checking code doesn't have to check to see whether or not the object being modified is in old space; the scavenger simply ignores the cards for objects in new space. With 128-byte cards, the space required to store the card mark bytes is less than 1% of the total space of the SELF heap, adding less than 45KB for our standard heap size of 5.6MB. The amount of space overhead can be varied against the cost of scanning a card at scavenging time by changing the size of the cards.

The dedicated global register named **`%card_base`** in the code above represents the base address of the array of bytes for the cards. It is initialized so that the address of the lowest memory word in the heap, shifted right the appropriate number of bits, and added to the global register contents yields the address of the first byte in the array of cards:

```
%card_base = &cards[0] - (&heap[0] >> log_base_2(card_size))
```

This store checking design imposes a relatively small overhead of 3 instructions for each store into memory to support generation scavenging; we are not aware of any other store checking designs that impose less overhead.

6.2 The Parser

To minimize parsing overhead, textual SELF programs are parsed once when entered into the system, generating SELF-level *byte code* objects, much like Smalltalk-80 **`CompiledMethod`** instances [GR83]. Each method object represents its source code by storing a reference to the pre-parsed byte code object in the method's map; all cloned invocations of the method thus share the same byte code object since they share the same map. A byte code object contains a byte array holding the byte codes for the source and an object array holding the message names and object literals used in the source; the byte code object also records the original unparsed source and the file name and line number where the method was defined for user-interface purposes.

* Earlier SELF implementations including that described in [Lee88] used a more traditional *remembered set* to record old objects containing pointers to new objects.

Each byte code in the byte array represents a single byte-sized virtual machine instruction and is divided into two parts: a 3-bit opcode and a 5-bit object array index. The opcodes used to represent SELF programs are the following:

0: INDEX-EXTENSION <index extension>
    extend the next index by prepending this index extension

1: SELF
    push **`self`** onto the execution stack

2: LITERAL <value index>
    push a literal value onto the execution stack

3: NON-LOCAL RETURN
    execute a non-local return from the lexically-enclosing method activation

4: DIRECTEE <parent name index>
    direct the next message send (which must be a resend) to the named parent

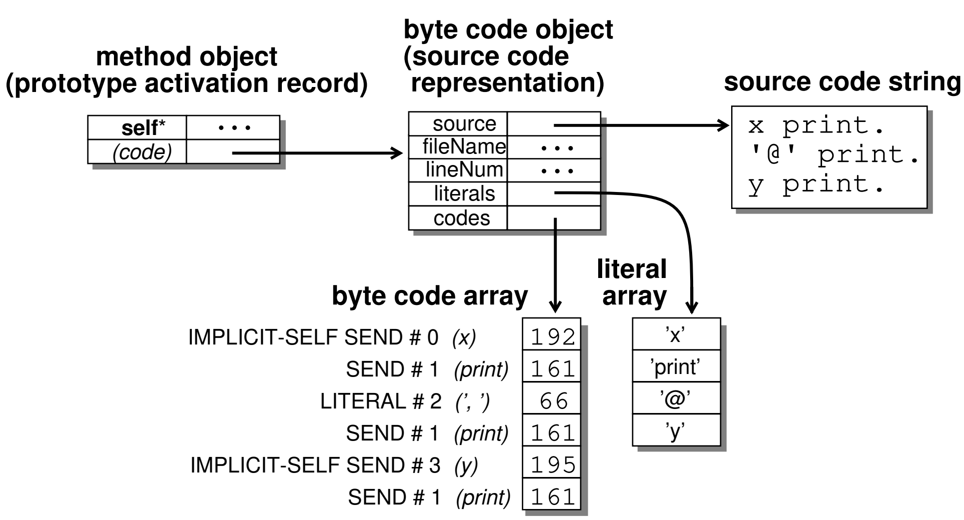
5: SEND <message name index>
    send a message, popping the receiver and arguments off the execution stack and pushing the result

6: IMPLICIT SELF SEND <message name index>
    send a message to (implicit) **`self`**, popping the arguments off the execution stack and pushing the result; begin the message lookup with the current activation record

7: RESEND <message name index>
    send a message to **`self`** but with the lookup beginning with the parents of the object containing the sending method, popping the arguments off the execution stack and pushing the result; like a **`super`** send in Smalltalk*

These opcodes are specified as if for direct evaluation by a stack-oriented interpreter; in reality, the SELF system dynamically compiles machine code that simulates such an interpreter. The index specified by several of the opcodes is an index into the byte code object's accompanying object array. The 5-bit index allows the first 32 message names and literals to be referenced directly; indices larger than 32 are constructed using extra INDEX-EXTENSION instructions. Since primitive operations are invoked just like normal messages, albeit with a leading underscore on the message name, the normal SEND byte codes can be used to represent all primitive operation invocations, simplifying the byte codes and facilitating extensions to the set of available primitive operations.

For example, the following diagram depicts the method object and associated byte code object for the point **`print`** method originally presented in section 4.1. The top-left object is the prototype activation record, containing placeholders for the local slots of the method (in this case, just the **`self`** slot) plus a reference to the byte code object representing the source code (actually stored in the method's map). The byte code object contains a byte array for the byte codes themselves, and a separate object array for the constants and message names used in the source code.



* Resends and directed resends are described in detail in [CUCH91] and [HCC+91].

6.3 The Run-Time System

6.3.1 Stacks

A running SELF program is a collection of lightweight processes, each process sharing the SELF heap address space but with its own set of activation records. As with most traditional language implementations, these activation records are implemented as a stack of frames, linked by stack pointers and frame pointers. The machine hardware provides support for efficiently managing stack frames. For example, the Motorola 680x0 architectures provide special instructions such as **`link`**, **`jsr`**, and **`movem`** for managing linked stacks of activation records and stack pointers [Mot85], and the Sun SPARC architecture provides hardware register windows to support fast procedure calls and returns with little register saving and restoring overhead [Sun91].

Garbage collection places some requirements on the design and implementation of the run-time system and the compiler. The garbage collector must be able to locate all object references stored in registers or on the stack. In the current SELF implementation, the compiler places a *saved locations mask* word at a fixed offset from each call instruction that could trigger a scavenge (i.e., all message sends and many primitives), identifying to the scavenger which registers and stack locations may contain tagged heap object references and should be scanned. On the SPARC, any of the 8 incoming and 8 local registers can contain valid object references (the 8 outgoing registers are handled by the next frame, which calls them its incoming registers, and none of the 8 global registers contain valid object references), thus requiring 16 bits of the saved locations mask word to mark which registers to scan. The remaining 16 bits of the 32-bit mask word indicate which of the first 16 stack temporary locations need to be scanned. Any additional stack temporaries are assumed to always need scanning; the compiler zeros these excess temporaries upon entering the method so that their contents will always be acceptable to the garbage collector.* This mask-word-based design allows the compiler more freedom in allocating data to registers than the alternative approach of fixing which registers can contain only object references and which can contain only non-pointer data. It does, however, slow scavenging with the overhead to extract and interpret the mask word for every stack frame; fortunately, we have not noticed this potential problem to be a performance bottleneck in practice.

In keeping with SELF's robust implementation, stack overflow is detected and reported via a signal to the process-controlling SELF code. In the current implementation, stack overflow is detected through an explicit check at the beginning of each compiled method that requires a new stack frame. On the SPARC, a dedicated global register maintains the current stack limit. On entrance to a compiled method, this global register is compared against the current stack pointer (also a register), and if the current stack pointer is past the stack limit, the stack overflow code is invoked. The 680x0 system is similar, except that the current stack limit is stored in a global variable in memory rather than a dedicated register. This stack overflow detection imposes a small run-time overhead to check for overflow on every method invocation.**

This polling for stack overflows at the beginning of methods actually is used for much more in the SELF system: it also is used for signal handling, keyboard interrupt handling, and memory scavenging requests. Whenever a running SELF process needs to be interrupted, either because a signal has arrived or a scavenge needs to be performed, the current stack limit is reset back to the base of the stack. This causes execution to be interrupted at the next message send point. When the stack overflow handler is invoked, it first checks to see what caused the "overflow": a pending signal, a scavenge request, or a real stack overflow, and branches to the appropriate handler.

Unfortunately, this polling approach to handling interrupts does not work for loops in which all message sends have been inlined away. To support interrupts and scavenges in these loops, the compiler generates extra code to check the stack limit value against the current stack pointer at the end of each loop body (at the **`_Restart`** primitive call, described in section 4.1). This ensures that interrupt handlers are always invoked relatively quickly after an interrupt is posted.

* Earlier memory system implementations only used the saved locations mask word for registers but not for stack locations. This forced methods with stack temporaries to execute a lengthy prologue to zero out all the stack locations. The new design avoids this overhead in nearly all cases arising in practice.

** An alternate implementation could avoid this run-time overhead by using hardware page protection to protect the memory page at the upper limit of the stack. If this page were accessed, the run-time system would interpret the subsequent memory access trap as a stack overflow error.

Interrupting SELF programs only at message send boundaries allows the compiler more freedom in generating code. The execution environment (the stack and registers) needs to be in a consistent state only when an interrupt could be caught (such as at message send boundaries) rather than at each instruction boundary. Debugging information to describe the state of execution, such as the saved locations mask word and the mapping from variable names to register assignments (described in section 13.1), need only be generated for interruption points such as message sends rather than for every machine instruction.

6.3.2 Blocks

In the current SELF implementation, blocks cannot outlive their lexically-enclosing scope. A block may only be passed down to the called routine, such as when the block is part of a user-defined control structure or is an exception handler.* This restriction is included so that activation records may be stack-allocated without additional special implementation techniques. Since blocks cannot outlive their lexically-enclosing scope, the contents of the implicit lexical parent slot of the block can be represented as a simple untagged pointer to the stack frame representing the lexically-enclosing activation record. Since all stack frames are aligned on a double-word (8-byte) boundary on the SPARC and at least a half-word (2-byte) boundary on the 680x0, the untagged address of the stack frame can be used in the representation of a block without fear of unfortunate interactions with the garbage collector.

The SELF implementation does not prevent a block object from being returned by its lexically-enclosing scope or stored in a long-lived heap data structure, but instead disallows the block's **`value`** method from being invoked after the lexically-enclosing scope has returned; such a "zombie" block is termed a *non-LIFO block*, since its lifetime does not follow the normal last-in-first-out (LIFO) stack discipline. To enforce this restriction, the compiler generates code to *zap* each block when its lexically-enclosing scope returns by zeroing out the block's frame pointer. Subsequent up-level accesses through a zapped block's null frame pointer cause segmentation faults which are caught by the SELF implementation, interpreted as non-LIFO block invocation errors, and signalled back to the SELF program.

6.4 Summary

The SELF memory system provides several important facilities for the compiler. Maps capture essential similarities among clone families, embodying the representation-level type information crucial to the compiler's optimizations. However, the garbage collector and the run-time system place constraints on the design and implementation of the compiler. Some of these constraints impose extra overhead on the run-time execution of programs (such as the current design of polling for interrupts) while others restrict the possible optimizations included in the compiler (such as disallowing derived pointers that could confuse the garbage collector). Fortunately, these restrictions are not too severe, and some parts of the overall system architecture ease the burden on the compiler, such as limiting interrupts to well-defined points in the compiled code and placing no restrictions on the allocation of pointer and non-pointer data to registers.

---

* Because they cannot be returned upwards, SELF blocks are not as powerful as blocks in Smalltalk or closures in Scheme. We do not miss this power in our SELF programming, since we can use heap-allocated objects created from in-line object literals to hold the long-lived state shared by the lexically-enclosing method and the nested blocks, and allow this in-line object to be returned upwards. Even so, this restriction on the lifetime of blocks in some ways reduces the elegance of the language, and some future SELF implementation may relax this restriction; we believe this can be accomplished without seriously degrading performance.