MIT/LCS/TM-267

PROPOSAL FOR A SMALL SCHEME IMPLEMENTATION

Richard Schooler

James W. Stamos

October 1984

Proposal for a Small Scheme Implementation[1]

by
Richard Schooler
and
James W. Stamos

Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square
Cambridge, MA 02139

Abstract

Scheme is a lexically scoped dialect of LISP developed at MIT. In this report we determine the feasibility of implementing a Scheme-based programming/application environment on a contemporary personal computer such as the Apple Macintosh. The absence of virtual memory, coupled with a limitation on the maximum amount of physical memory, means that space is at a premium. We suggest the use of bytecodes and sketch a possible instruction set.

Because of space constraints, tail-recursion optimization and an efficient mechanism for the reclamation of inaccessible contexts are also examined. Using the built-in operating system and user interface of the Macintosh realizes speed, functionality, and friendliness but raises a number of interesting issues. For example, the Pascal and assembler routines make many assumptions about data representation, type checking, and parameter passing. Since an implementation of Scheme is likely to have radically different conventions, the two environments must be interfaced smoothly and efficiently.

In addition to the bytecoded instruction set, we specify the virtual machine informally, discuss the implementation of basic and advanced features, estimate the performance of such an implementation, and finally evaluate the proposed design.

Key Words: Scheme, Macintosh, bytecodes, language implementation, garbage collection, tail recursion, closures.

---

[1]This work was supported by DARPA contract N00014-83-K-0125.

Table of Contents

1. Introduction

Scheme [14] is a lexically scoped dialect of LISP developed at MIT by Steele and Sussman. Closures are first-class objects in Scheme, which was designed to be a small version of LISP that would be easy to implement and easy to learn.

Scheme is used at MIT mainly as a teaching language. A traditional timesharing implementation was originally used, with about 400 students sharing an already overloaded TOPS-20 system. Students currently use Scheme on a large number of Hewlett-Packard 9836 personal computers and thus enjoy the main advantages of personal computing: constant load and greater reliability of the system as a whole.

One remaining problem is the expense of the HP machines, which are known as "Chipmunks." A Chipmunk equipped with the required 4 megabytes of physical memory costs about $30,000.[2] Scheme is ideally suited for use as a teaching language, but a much cheaper machine is required if Scheme is to be more widely used.

This report examines the feasibility of implementing Scheme on smaller, cheaper machines. The ideal target machine appears to be the Apple Macintosh [3] with a 512K memory. It is relatively inexpensive, costing a few thousand dollars, and provides desirable features such as a powerful processor, a bitmapped graphics display, and a user interface using windows, menus, and a mouse.

In the next section, various microcomputer-based language implementations are explored. Based on that discussion, a proposal for a small, bytecoded [15] Scheme implementation is elaborated. Finally, some rough estimates are given as to the performance and functionality of such an implementation.

---

[2] HP donated enough machines for the programming class at MIT.

2. Previous Experience

In order to determine the feasibility of a small Scheme implementation, we surveyed existing and planned implementations of Scheme. The Smalltalk system [5] was also examined because of its similarity to Scheme. Smalltalk is an untyped, heap-based programming language with full closure values. At least five different versions of Smalltalk have existed since 1972. The Smalltalk system has also been implemented a number of times by different research groups on a variety of machine architectures. For these reasons, we consider Smalltalk first.

2.1 Smalltalk

Smalltalk is a good example of a desirable environment: a sophisticated, flexible language designed for use on personal machines. A great deal of effort has been expended on implementing such a language efficiently in both space and time. This section relates some of that experience.

The fastest implementation of Smalltalk is a microcoded bytecode interpreter on the Xerox Dorado, which executes approximately 300,000 bps (bytecodes per second). This machine is a high-performance, rather expensive personal computer. Since the bytecode interpreter is microcoded, all of main memory can be used for Smalltalk. Apparently, a one-megabyte main memory is sufficient for the object table and reference-counted heap. The size of a small *virtual image* is currently about 700K bytes.[3] The Dorado implementation is a 16-bit Smalltalk, meaning that a maximum of 32K objects can be referenced. Excluding device drivers, the virtual machine itself (the bytecode interpreter, memory management, and many primitives) represents about 2000 lines of Dorado microcode. Since an earlier version of Smalltalk required about 2000 lines of 8086 assembly language, we feel this program size characterizes Smalltalk interpreters implemented in low level languages.

Peter Deutsch [2] has implemented a 32-bit Smalltalk on a microcomputer based on the Motorola MC68000 microprocessor [8]. This implementation, which has a one-megabyte image, will need about two megabytes of memory. 68K bytes are taken up by the translator and (machine-code) utility routines, including memory management, screen management, etc. Deutsch's hybrid compilation scheme executes about 100,000 apparent bps on a 68000; the implementation is quite usable.

Several groups outside Xerox have implemented Smalltalk [7]. Some conclusions from this

---

[3] The virtual image is a conventional file containing a (truncated) object table and a compacted heap. No inaccessible objects are contained in such an image.

experience are that timeshared implementations don't work well, that **BitBlt** (bitmap manipulation) must be at least partially supported by the hardware, and that both speed and space are severe problems. In particular, when garbage collection was used, allocating environment frames from the heap and reclaiming them as ordinary objects created an excessive amount of work for the garbage collector.

2.2 Chipmunk Scheme

At MIT, Scheme has been implemented on Chipmunks. These are large, expensive machines with floppy disks for personal program storage. A Chipmunk has a 12.5 MHz 68000 processor and 4 megabytes of physical memory. The Chipmunks at MIT are networked together and share printers and large disks.

The Chipmunk actually runs two LISPs: Scheme and PSL (Portable Standard Lisp). PSL, which is used only for the editor, requires 1.2 megabytes. PSL itself uses about 600K bytes; the editor code and the heap require another 600K bytes.

The copying garbage collector uses two one-megabyte heaps.[4] Additionally, Scheme has a 600 kilobyte constant space for pure code and data, of which about 450K bytes are currently used. The machine language portion, which includes the interpreter, memory management (including the garbage collector), and many utilities including string-handling, bignum-handling (infinite-precision integers), graphics, and I/O, is about 125K bytes in size. The interpreter stack uses about 120K bytes.

Chris Hanson has indicated that a 1.5-megabyte implementation is possible by eliminating the PSL portion, rewriting the editor in Scheme, and shrinking the heaps to 256K bytes each. The Scheme group at MIT is also exploring bytecoding. Their feeling is that a bytecode representation is about 3 times as space efficient as their current S-code (parse tree) representation, and will be a little faster as well (less than a factor of two).[5] By separating the system into components, a number of small, tailored systems will be available. Reducing functionality and using bytecoding may enable a one megabyte implementation of Scheme.

---

[4] Actually, the two Scheme heaps and the PSL heap rotate.

[5] Although bytecoding significantly reduces code size, the overall reduction in size is much less than a factor of two when variable names are retained.

2.3 Scheme 312

Will Clinger at Indiana U. has a small, bytecoded implementation of Scheme (known as Scheme 312) that runs on a 68000-based machine. Scheme 312 is based on the Scheme dialect described in [4]. Clinger generally uses a 256K byte heap, although toy programs will run in a 100K byte heap. The bytecode interpreter, which is hand-coded in assembly language, is just under 6K bytes in size. Initialization and I/O code take about 12K, depending on the host operating system. The whole system barely runs in 128K but runs comfortably in 256K. A copying garbage collector is used for the sake of compaction (see Section 3.2).

Scheme 312 is a less ambitious implementation than Chipmunk Scheme. Neither an editor nor a debugger is included. Moreover, there is no support for floating point numbers or arbitrary precision integers. On the other hand, an interactive compiler is included and advanced Scheme features such as dynamic variables and exception handling mechanisms (see Section 3.5) are implemented.

Clinger reports that a typical line of Scheme source code uses 10-20 bytes of heap space when compiled. He does not keep system source code in the heap. His implementation, which uses an 8 MHz 68000 with no wait states, is about 10% faster than Chipmunk Scheme, which uses a 12.5 MHz 68000 with wait states and a large cache.

3. Proposal

We propose a bytecoded implementation of Scheme that uses a non-copying garbage collector. We hope to save a great deal of space over the Chipmunk implementation by using bytecoding, a single heap, and the built-in facilities of the Apple 68000-based machines. Space is the main consideration in this design, although speed is also important. This section introduces the proposal, discusses some optimization techniques, and sketches an instruction set.

3.1 Virtual Machine Specification

The virtual machine is a stack-oriented machine that contains a heap and a few registers. Since object and environment lifetimes are defined by accessibility, they may not be correlated with function invocations. Therefore, all objects and environments exist in the heap. Environment structures must also support Scheme's lexical scoping. In addition to retaining control information, the stack is used for parameter passing (expression evaluation).

The virtual machine contains four registers, the first three of which refer to the currently executing function:[6]

- PC - a pointer to the next instruction;
- LIT - a pointer to the literal area;
- ENV - a pointer to the current environment; and
- SP - a pointer to the top element on the stack.

All the registers except for SP are saved on the stack for a function call and are restored on function return. Figure 3-1 shows what the stack looks like upon function entry, assuming the stack grows downwards.[7] The callee creates a new environment and initializes it with the arguments. Local variables are initialized to appropriate values.[8] Once the result is computed, the callee pops the arguments, pops the control information into the virtual machine's registers, pushes the result, and finally returns to the caller.

---

[6] A user-defined function is a triple consisting of:
1. a pointer to the function's definition-time environment;
2. a pointer to the function's bytecode vector; and
3. a pointer to the function's data area, or literal vector.

As in Smalltalk, the literals are those values, such as strings and lists, that are not directly contained in the code. (See Figure 3-2). An optimization that eliminates LIT is to merge the bytecode vector with the literal vector and to reference literals relative to the PC.

[7] In traditional implementations, the control information is pushed after the arguments. We reverse the order to simplify the implementation of the optimization discussed in Section 3.3.1.

[8] Although all local variables in Scheme must be initialized when they are defined, we assume the general case in which locals are automatically initialized to a default value when an environment is created.

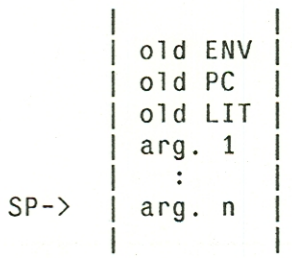

Figure 3-1: The Stack

3.2 Implementation Guidelines

The intended target is the Apple Macintosh with a 512K memory. Of this 512K bytes, about 20K bytes are the screen bitmap. Another 20K bytes are reserved for the system heap and other system areas. Therefore, there are roughly 470K bytes available for the Scheme implementation. This might be divided into 20K bytes for the bytecode interpreter, including memory management and other utilities, 50K bytes for the stack, and 400K bytes for the Scheme heap. We estimate that about half of the heap would be used for Scheme's run-time system.

The Macintosh has a 64K ROM containing highly optimized, hand-coded routines for screen management, device management, etc. In order to achieve a compact system, the Scheme implementation will heavily exploit these ROM routines. Unfortunately, the built-in memory management routines are not suitable for our use, as they assume a reference-counted, explicitly-managed heap.

The bytecode interpreter will have to be hand-coded in assembly language, both to save space and to increase performance. A critical component of a bytecode interpreter is the dispatching and decoding of instructions. One possibility is to use a 256-element jump table. Many bytecodes would dispatch to the same routine, which would take its argument from the bytecode (or from the next bytecode for a two-byte instruction).

A discussion in [5] explores some trade-offs between different garbage collection methods. A mark-and-sweep garbage collector appears to be best for small heap sizes, since it reclaims all garbage. A reference-counting scheme cannot reclaim cyclic structures, but is cheaper for large heaps or virtual memory systems, since the entire memory does not have to be scanned. A copying garbage collector reclaims all garbage and compacts memory, but requires double the heap storage. On the other hand, the copying garbage collector scans only the accessible cells, while the mark-and-sweep collector scans the entire heap. Since our emphasis is on space efficiency, the heap size is

relatively small, and no virtual memory is assumed, a mark-and-sweep garbage collector seems the best choice. An in-place algorithm that avoids the need for a stack, such as the Deutsch-Schorr-Waite algorithm [6, 10], could be used.

In order to reduce the garbage collection work load, the run-time system should reside in a *static* heap that is off limits to the garbage collector. This heap would consist of code and constants and be completely self-contained. Objects in the *dynamic* heap and pointers on the stack could reference objects in the static heap, but objects in the static heap could not reference anything outside the static heap. The garbage collector would neither mark nor sweep the static heap.

The main disadvantage of mark-and-sweep garbage collection is the lack of compaction. After several garbage collection cycles, memory would become excessively fragmented. One possible solution is a dual-mode garbage collector. Mark-and-sweep would be used until excess fragmentation occurred, at which point a copying garbage collection would be done. The floppy disk would be used as the second heap.[9]

An alternative method for reducing external fragmentation is to split the heap into several areas. Each area would contain objects of a single size. For example, an object size could be contrained to be a power of two. Since experience with Smalltalk and Scheme indicates that most objects are quite small, an appropriate collection of object sizes should not be too difficult to determine.

The proposed interpreter uses a stack and a separate heap. The (dynamic) heap is used for general memory allocation. The stack, which is used for passing parameters, also retains control information. To call a function, the stack pointer is advanced by a fixed amount to leave room for the control information. Next, the parameters are computed and pushed on the stack. The return address, current literal area, and current environment are then inserted into the reserved stack locations. Finally, control is passed to the called function. The function, whether built-in or user-defined, must eventually pop the parameters and control information from the stack, push a return value, and return control to the caller. Functions with a variable number of parameters, optional parameters, etc., must be handled specially.

---

[9] A hard disk would be helpful if this solution is adopted. To avoid thrashing, marking could be done as usual in the memory heap. The sweep phase would be replaced by a 3-step compaction phase:

1. copy all accessible objects to the disk to form a compact image, overwriting the first field of each accessible object in memory with its new address;
2. update all pointers in the disk image; and finally
3. swap the entire disk image into memory.

Since only sequential disk access will be done, performance should be acceptable.

Registers other than those in the virtual machine specification may be needed in the implementation. For example, a free memory pointer, display registers, and scratch registers may be useful. The Macintosh operating system routines preserve five of the 68000's seven address registers, and five of the 68000's eight data registers [3]. Since all the virtual machine registers can reside in machine registers, acceptable performance should be attainable.

For the sake of security, it may be necessary to dynamically type-check the arguments to built-in routines. The garbage collector also needs to distinguish pointers from immediate data. To simplify garbage collection and permit type checking, we suggest tagged pointers. The tag bits could reside in the upper byte of a 32-bit pointer, since only the lower 24 bits of an address are used by the 68000.

In order to minimize data conversions, we suggest making the concrete representations of the built-in Scheme types be as similar as possible to those expected by the Macintosh's built-in routines. Unfortunately, tag bits and compatibility with the built-in representations seem mutually exclusive. Type checking and conversion will have to take place in interface routines. An application program would call such a routine instead of calling the built-in routine directly. Any interface routine that converts representations and can trigger garbage collection must ensure that:

1. none of the converted values are visible to the garbage collector; and
2. all accessible objects are accessible via valid pointers.

3.3 Optimizations

Two useful optimizations applicable to Scheme are context optimization [1], in which stack allocation of environment frames is used as much as possible, and tail-recursion optimization [13], in which certain calls are recognized as jumps and implemented accordingly. The following sections discuss each optimization in turn.

3.3.1 Context Optimization

Stacks are ordinarily used for three distinct purposes. They retain control information, facilitate expression evaluation, and maintain environment structures. Control information and subexpression values have lifetimes correlated with that of a LIFO stack. On the other hand, a Scheme implementation can not use a simple stack discipline for environment frames, since the language has full closure values. For instance, if a function creates and returns a function value that references its definition-time environment, the lifetime of the associated environment frame can be greater than that of the function invocation. Luckily, the compiler can detect the need for heap-allocated contexts, since they are necessary only when a function with free variables is created and is accessible after the creating function has returned [9].

An advantage of lexical scoping is that local variables are inaccessible outside their defining environment. A compiler can easily determine all the procedures that access the variables contained in an environment frame. Therefore, it is free to use nonstandard representations, such as the stack, when implementing environments [11]. We capitalize on this observation by permitting environment frames to reside either on the stack or in the heap. Functions whose contexts obey a stack discipline can be optimized to keep their parameters and locals on the stack, thus saving both space and time. On the other hand, if the compiler can not guarantee that the environment frame for a function invocation may reside on the stack, the function must pop the arguments from the stack into a heap-allocated environment frame. Such a frame would also contain the function's local variables. Nevertheless, the stack would still be used for parameter passing. Since returning a newly-created closure is uncommon, heap-allocated frames will probably be the exception rather than the rule.

Section 3.2 presented a calling convention in which control information was pushed "before" the arguments were even computed. This method was chosen so that heap frames and stack frames could have identical representations. The first slot in an environment frame, whether it exists on the stack or in the heap, contains the *static link*, which is a reference to the environment that encloses ENV. The arguments follow the static link in the environment frame; the local variables come last. Since the control information fits in three stack elements, the caller must reserve four elements to accommodate the static link. If the callee's environment is stack-allocated, the static link is filled with the ENV field of the function. If the callee's environment is heap-allocated, the field is unused but remains on the stack when the arguments are moved to the heap frame.[10] ENV may refer to a heap object or a stack element, depending on whether the current environment frame resides in the heap or on the stack.

A *display* is a vector of pointers to frequently accessed environment frames. Displays could be used to optimize variable references further, as was done for some Algol implementations. Using the above representations, a display could reference either stack- or heap-allocated frames. Displays and other relevant optimizations are discussed in the description of Liar, an Algol-like compiler for Scheme [9].

3.3.2 Tail-Recursion Optimization

Scheme, which has no iteration construct, uses tail recursion to achieve the same semantic effect.

---

[10] Although this convention wastes a little space, it permits an efficient implementation of the tail-recursive optimization discussed in the next section.

Tail-recursive[11] calls should be optimized to avoid stack growth and superfluous context manipulation. Steele [12] observed that a return address need be pushed only if control must be returned to the caller, which is *not* the case for a tail-recursive call.

The compiler can special-case a tail-recursive call. No return address is pushed, since the called function can use the return address already on the stack. The callee's parameters simply replace the caller's parameters on the stack.[12] The execution of the called function thus *replaces* the execution of the calling function. Whether the caller had a heap frame or a stack frame does not matter. The new parameters are passed on the stack just as in an ordinary procedure call. Whenever the environment frames are stack allocated, tail recursion is as space efficient and almost as fast as conventionally implemented iteration.

Care must be taken when a *local* function is called tail-recursively. For example, consider the following definition:

```
(define (foo a)
   (define (bar b)
      (* a b))
   (bar 3))
```

`Foo`'s arguments can not be overwritten by the call to `bar`, since they are needed by `bar`. One solution is to omit the tail-recursion optimization and push another frame on the stack. `Foo` would then have to pop its frame after `bar` returns. When indirect recursion does not occur on the tail-recursive call, the additional space required is proportional to the lexical nesting depth. Hence the space overhead is not too important.

However, pushing and popping unnecessary control information wastes time. In simple cases, such as the above example, the caller's frame could be extended with the arguments of the callee. The local function's arguments would then appear to be local variables in the calling function. For instance, `foo` would have argument `a` and local variable `b`. `B` would be assigned just before the call to `bar`. On the other hand, the arguments and locals of the calling function appear as arguments to the local function. For example, the compiler would pretend that `bar` had two arguments: `a` and `b`.

---

[11] A tail-recursive call is the last call in a function body. Traditionally, tail recursion refers to the case where the last function called is the same as the calling function. The usage is extended here to eliminate this restriction.

[12] If the current environment frame is in the heap, no further optimization can be done. The control information is in place, the placeholder for the static link exists, and the new parameters are exactly where the old parameters were. On the other hand, if the current environment frame is on the stack, the new parameters can be slid to overwrite the old parameters, *assuming the caller's context is not visible to the callee*. A block-move instruction could be used to some advantage. A further enhancement is possible if the number of new parameters is at most the number of old parameters plus the number of old locals. Instead of computing all the new parameters and sliding them along the stack as a group, an extremely good compiler could generate code to overwrite only *changed* parameters and local variables. As always, the callee would then have the responsibility for initializing its locals if its environment frame were stack allocated. If the current environment frame is on the stack and the tail-recursive call is actually recursive, local variables that are defined before used would not need to be initialized.

3.3.3 An Example

Figure 3-2 represents a snapshot of the system just after a function with a heap-allocated frame (**func 1** with frame **frame 1**) has called a function with a stack-allocated frame (**func 2**). Both functions were defined in the global environment. Hence their ENV components point to the global environment. The **code** component of a function points to its bytecode vector, and the **LIT** component points to the function's data area.

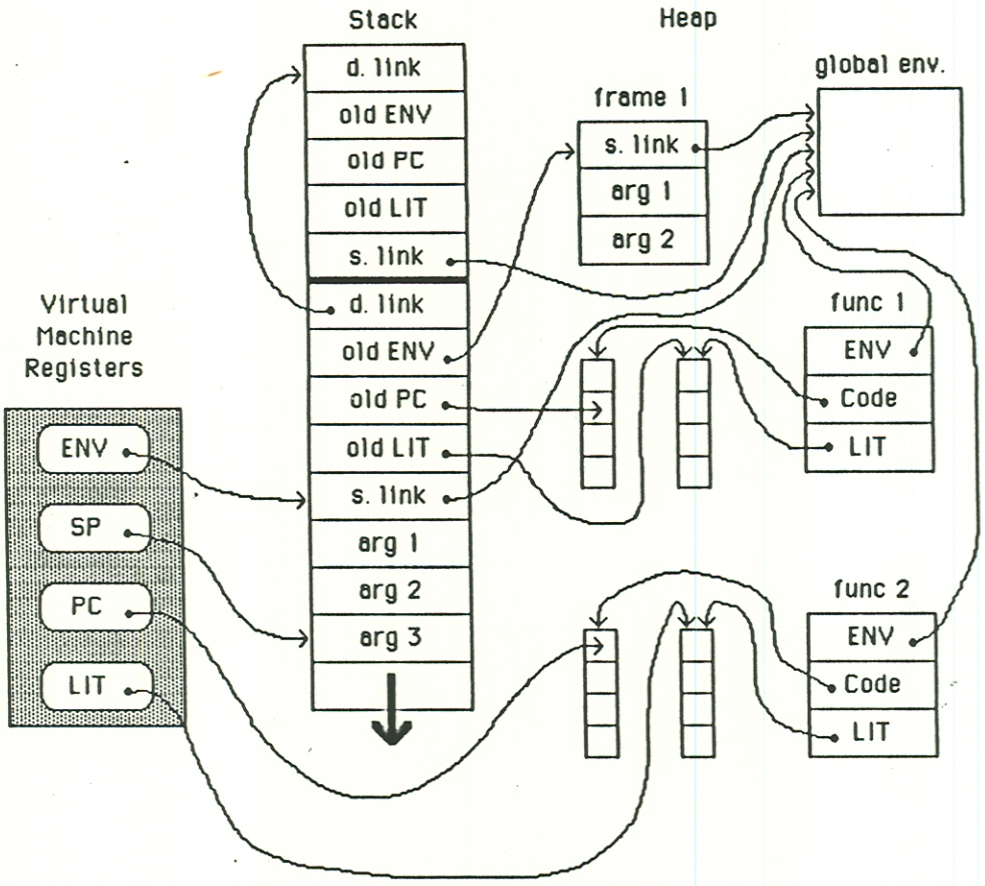

Figure 3-2: A System Snapshot

As mentioned in Section 3.1, the virtual machine register LIT points to the currently-running function's literal vector. The PC register points to the currently executing bytecode, which is an element of the current function's code vector. SP points to the current top-of-stack element. ENV points to the base of the current frame, regardless of whether the frame is on the stack or in the heap. Since heap- and stack-allocated frames have the same format (static link, then arguments), the same variable-reference instructions work in either case.

The dynamic link (**d. link**) is used by the debugger. An alternative implementation for dynamic links is to mark the stack with tag bits. A special sequence of tag bits could be reserved and used only for a particular component of the control block, such as the OldEnv component. The debugger would interpret the stack by scanning it and looking for occurrences of this sequence.

3.4 The Instruction Set

3.4.1 Bytecodes

The instruction set is quite similar to other bytecode sets (for instance, see [5]), since the basic stack architecture is similar. The suitability of the unconventional instructions we propose will depend on the characteristics of the Scheme system.[13] Although static and dynamic measurements will be needed to tune the instruction set, we offer a starting point. The proposed set of bytecodes includes the following conventional instructions:

- PushArg *n* - push argument *n* on the stack by indexing off ENV.
- PushConst *n* - push a constant (True, False, Nil, small integers) on the stack.
- PushLit *n* - Push literal *n* on the stack by indexing off LIT.
- PushVar *m n* - Push the *m*-th component of the *n*-th environment frame on the stack. Note that PushArg *m* is a shorthand for PushVar *m 0*.
- PushGlobal *n* - Push the *n*-th component of the global environment on the stack. PushGlobal is probably a 2-byte instruction.
- PushEnv - Push ENV on the stack.
- Duplicate - Push the top of the stack.
- Store *n* - Overwrite the *n*-th argument with the stack top and pop the stack.
- StoreVar *m n* - Overwrite the *m*-th component of the *n*-th environment frame with the stack top and pop the stack. Note that Store *m* is a shorthand for StoreVar *m 0*.
- StoreGlobal *n* - Overwrite the *n*-th component of the global environment with the stack top and pop the stack. StoreGlobal is probably a 2-byte instruction.
- Call *n* - Call a function with *n* arguments: pop the function from the stack, insert PC, LIT, and ENV in the stack, load LIT, PC, and StaticLink from the called function, compute ENV from the stack pointer and the number of arguments, and start executing the called function. Call is used for all invocations except primitives and tail-recursive calls.

---

[13] For example, the instruction set does not include the capabilities needed to support the full Liar system [9].

- Branch *condition offset* - If *condition* is met, jump to *PC + offset*.

- ShortAllocate *n* - Allocate a block of *n* elements from the heap and push a pointer.

- Allocate - Pop *n* from the stack and proceed as in ShortAllocate.

- GetWord - Pop offset *i* and pointer *p* from the stack; push the contents of memory location *p + i*. GetWord may be used to access vector elements.

- ShortGetWord *i* - Pop pointer *p* and proceed as in GetWord. Note that CAR is ShortGetWord *0*, CDR is ShortGetWord *1*, etc.

- SetWord - Pop value *v*, offset *i*, and pointer *p* from the stack. Write *v* in memory location *p + i* and push v. GetWord may be used to update vector elements.

- ShortSetWord *i* - Pop value *v* and pointer *p*. Proceed as in SetWord. Note that RPLACA is ShortSetWord *0*, RPLACD is ShortSetWord *1*, etc.

- ITimes, etc. - Primitive routines: pop the requisite number of arguments and push the result on the stack.[14] The primitives will provide arithmetic and logic functions, I/O support, and other miscellaneous capabilities by invoking the Macintosh's built-in routines and other hand-coded utilities.

Scheme also requires a number of new instructions. To enhance modularity, the caller need assume nothing about the callee when calling a function.[15] The callee likewise need assume nothing about the caller when returning. Most of the following instructions are connected with the transfer of control:

- ReserveControl *n* - Compute and push the dynamic link. Then reserve a fixed block on the stack for control information. The size of this block is currently four elements. The dynamic link is computed from the original value of SP and *n*, which is the number of stack elements *between* the two control blocks. ReserveControl must appear before any arguments are pushed for a function call that is neither primitive nor tail-recursive.

- HeapContext *n* - Create a heap context with *n* slots. Pop the *m* arguments into the appropriate slots in the new context. Copy (do not pop) the StaticLink into the zeroth slot. Set ENV to refer to the new heap context. HeapContext must be the first bytecode in every function with a heap-allocated frame. *m* is determined by the difference between SP and the original value of ENV.

- StackTRCall *n* - Call a tail-recursive function when the current environment is on the stack. The callee's environment *replaces* the caller's environment on the stack. First,

---

[14] For compatibility with user-defined functions, primitive routines would normally be called from inside interface routines that would pop control information as well. The compiler could optimize in certain cases by not pushing control information and calling the primitive routines directly.

[15] However, the compiler may wish to optimize function invocation when the function is known to be a primitive routine.

pop the function to be called from the stack. Then slide the next *n* stack elements, which are the arguments to the function, to their appropriate locations. Load LIT, PC, and StaticLink from the function saved in the first step. Finally, start executing the called function. Note that ENV does not change; it is used to determine the appropriate locations for the arguments. SP is set so that it appears *n* arguments have been pushed.

- HeapTRCall *n* - Call a tail-recursive function when the current environment is in the heap. First, pop the function to be called from the stack. Load LIT, PC, and StaticLink from this function. Compute ENV from SP and *n* arguments, and start executing the called function. Since no space is reserved for control information, the arguments are already in the appropriate location. HeapTRCall differs from Call in that the control information on the stack is not overwritten.

- StackSlide *n* - Slide *n* stack elements into the argument slots and reset the stack pointer so that it appears as if *n* arguments were pushed. StackSlide is used for tail-recursive calls that are actually recursive. StackSlide, which applies only to stack-allocated environments, would normally be followed by an unconditional branch to an appropriate starting location in the function. Note there is no need for a HeapSlide instruction, since the parameters are in the correct location on the stack and will be copied to the heap immediately by HeapContext. ENV, which does not change, is used to determine the locations for the arguments.

- HeapReturn - Return from a function when the current environment is on the heap. Pop the stack and save the value. This value is the value that will be returned by the function. Pop and discard the static link, pop the control information into the registers, push the saved value, and resume execution of the caller.

- StackReturn - Return from a function when the current environment is on the stack. Pop the stack and save the value. This value is the value that will be returned by the function. Pop and discard the arguments, locals, and the static link by setting SP to refer to OldLit, the stack element "above" the one referenced by ENV. Pop the control information into the registers, push the saved value, and resume execution of the caller.

- CreateFunction - Combine ENV with code vector and literal area pointers popped from the stack to create a closure. Push the result.

3.4.2 Examples

Some typical Scheme functions are shown below, along with their (hand-)compilations:

```
(define (fact n m)    ; tail recursive, call (fact n 1)
  (if (< n 1) m
      (fact (- n 1) (* n m))))

fact:   PushArg 1       ; push n
        PushConst 1     ; push 1
        ILess?          ; (< n 1)
        BranchFalse %1
        PushArg 2       ; push m
        StackReturn     ; return m (pop 2 args)
%1:     PushArg 1       ; push n
        PushConst 1     ; push 1
        IMinus          ; (- n 1)
        PushArg 1       ; push n
        PushArg 2       ; push m
        ITimes          ; (* n m)
        StackSlide 2    ; over-write m and n
        Branch fact     ; iterate

(define (curry func arg1)
  (lambda (arg2)
    (func arg1 arg2)))

curry:  HeapContext 2   ; move args to a new heap frame of
                        ; size 2 and let this frame be the
                        ; current environment
        PushLit 1       ; push the code vector
        PushLit 2       ; push the literal area
        CreateFunction  ; create the local function
        HeapReturn      ; return the closure

(define (foo a)
   (bar (baz a) 1))

foo:    ReserveControl 1; reserve space for call to baz
        PushArg 1       ; push a
        PushGlobal 17   ; push baz, which is a variable
                        ; defined in the global environment
        Call 1          ; call baz with 1 argument
        PushConst 1     ; push 1
        PushGlobal 3    ; push bar
        StackTRCall 2   ; call bar tail recursively
```

The compilation of **fact** requires only 14 lines. Depending on the particular encoding used, **fact** could be represented in 14-17 bytes. The list representation of **fact** requires at least 16 cons cells. Depending on cdr-coding, each cons cell requires four to eight bytes. The space savings of the bytecoded implementation should be evident from this simple example.

3.5 Implementing Advanced Features

Scheme has a number of features that will require special attention. We have already alluded to some of these features, such as optional arguments; other advanced features include symbolic variable access and run-time code construction.

The **access** special form allows one to access a variable inside any closure's environment. This feature does not interfere with context optimization, since any closure with a lifetime that may exceed

that of the corresponding function invocation will be heap-allocated. Since name lookup may be done at run time, variable names must be retained.[16] One implementation strategy is to keep variable names in a function's literal area and let each environment frame reference the literals.[17] Since a function's variable names can easily use more space than the code vector, it may be necessary to store all names on the disk.

We introduce two bytecodes for symbolic variable access:

- Access - Find the binding of a symbol in a specified environment. Pop a symbol and an environment from the stack. If the symbol is defined in the environment or any outer environments, push its value. Otherwise, push NIL.

- ShortAccess - Find the binding of a symbol in the current environment. Pop a symbol from the stack and use ENV as the first environment frame. Proceed as in Access.

Since local environments tend to be small, a linear-search implementation for **access** is justified. The global environment, on the other hand, is large and deserves special treatment. We suggest an auxiliary hash table that maps symbols to slots in the global environment. If the symbol to be accessed is known at compile time, its lexical level and offset may be computed then. The hash table will be consulted at compile time only if the symbol does not appear in any of the local environments. If the symbol to be accessed is not known at compile time, the search must be done at run time. The hash table will be consulted at run time only if the search progresses to the global environment.

Another advanced feature deserving mention is the **eval** function, which allows one to create and evaluate code at run time. To implement this function, the compiler must be part of the run-time system. Otherwise, the use of **eval** in compiled code must be prohibited. The compiler will assign an unused slot in the global environment for each free variable in dynamically contructed code that does not already exist in the global environment. The auxiliary hash table for the global environment will also be updated to reflect the new global variables.

Scheme provides an exception-handling mechanism that uses **catch** and **throw** to implement non-local exits. We have not dealt with the issues of implementing this mechanism.

---

[16] The debugger will also require access to variable names.

[17] For instance, the second slot of each environment frame could reference the literals. This field would be initialized when the function was called. LIT would not need to be saved with the other control information each time a conventional CALL instruction was executed. Although it would no longer be necessary to keep LIT in a register, LIT could be cached for performance. The LIT register would then be restored from the second slot in the environment frame instead of from the control block on the stack.

4. Discussion

4.1 Programming Environment

The entire system will contain not only the bytecode interpreter, but also a compiler, editor, debugger, etc. Debugging can be handled in various fashions: one can debug at the bytecode level, which is not desirable; one can maintain pointers from the compiled representation to the source representation, which is the Smalltalk approach, and requires some amount of disk space; or one can de-compile the bytecodes back to a source representation, which is the Chipmunk approach. The latter approach has the advantage of minimal space requirements, assuming a sufficiently small de-compiler. As long as one is careful not to over-optimize the bytecode representation, it appears that de-compilation is feasible, and is the most space efficient approach.

If all the desired Scheme features can not fit comfortably in a 512K memory, the system should be structured as follows. There should be a small, stand-alone kernel containing the bytecode interpreter and device drivers. Additional features would then be provided by independent packages. For instance, one package could contain the compiler and a read-eval-print loop. Another could contain the editor, while a third could contain a debugger. The kernel, in combination with the appropriate system packages, would then form a customizable base for applications written in Scheme.

Maximal advantage will be taken of the existing user interface primitives on the Apple machines to provide a good environment as well as good performance.

4.2 Estimated Performance

A Macintosh has a 8-MHz 68000 processor, but the main memory is multiplexed between the processor and the display refresh hardware. While the ROM code may execute at 8 MHz, code in RAM executes at only 5 MHz. An average instruction takes 15 cycles to execute, so code in RAM executes at about 1/3 MIPS (Million Instructions Per Second). If the execution of each bytecode takes 10 instructions, we can execute 30,000 bps, which is a bit slow, but tolerable, especially as many device handlers and user interface routines are implemented by highly optimized ROM code.

In order to estimate the garbage collection performance, we must make some assumptions. With a 200K dynamic heap, there are 50K fields and at most 25K objects. In the worst case, marking and sweeping will read or write each field about 10 times. Hence 500K memory references will be needed. Assume a total of 20 instructions are executed per field. For instance, 15 could be for marking and 5

for sweeping. With a 1/3-MIP processor, garbage collection will take 20 * 50K * 3 microsec = 3 seconds using the above worst-case numbers. In terms of both memory references and instructions executed, a worst-case garbage collection will be noticeable but tolerable. The Smalltalk experience indicates that one byte is allocated every 4 or 5 bytecodes. Using the above performance estimates, about 7 kilobytes will be allocated per second. If a typical garbage collection takes 2 seconds and reclaims half of the dynamic heap (100K bytes), then garbage collection will require about 14% of the processing time. Since many of the above numbers are guesses, some experimentation will be needed to refine our estimates and guide the design of the garbage collector.

4.3 Disadvantages of the Proposed Design

By taking advantage of the Macintosh's ROM routines, portability has been compromised. In effect, the virtual machine specification now includes most of the Macintosh ROM. Any other implementation would have to emulate this Apple-provided functionality.

A related disadvantage is that the user interface is frozen. Since much of the implementation is fixed (and hidden) in ROM, users can alter the environment in rather limited ways. The environment is therefore not suited to experimentation and research with two-dimensional images on bitmap displays.

4.4 Conclusions

Based on existing implementations and the proposed space-saving techniques, we feel that a 512K Macintosh implementation will be useful for teaching purposes. Such an implementation will have an environment capable of supporting the development of small programs. A Lisa with 1 or 1.5 megabytes of memory and a hard disk should be suitable for more ambitious programming projects.

Acknowledgments. The authors wish to thank Will Clinger, L. Peter Deutsch, Chris Hanson, and Jim Miller for discussing existing and planned implementations of Smalltalk and Scheme. Steve Berlin and Bill Rozas deserve thanks for contributing a number of suggestions.

References

[1] Bobrow, Daniel G. and Wegbreit, Ben.
A Model and Stack Implementation of Multiple Environments.
Communications of the ACM 16(10):591-603, October, 1973.

[2] Deutsch, L.P. and Schiffman, A.M.
Efficient Implementation of the Smalltalk-80 System.
In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297-302. ACM, 1984.

[3] Espinosa, et al.
Inside Macintosh.
Apple Computer, Inc., 1984.

[4] Fessenden, C., Clinger, W., Freidman, D.P., and Haynes, C.
Scheme 311 Version 4 Reference Manual.
Computer Science Technical Report 137, Indiana University, February, 1983.

[5] Goldberg, A. and Robson, D.
Smalltalk-80: The Language and its Implementation.
Addison-Wesley, Reading, MA, 1983.

[6] Knuth, Donald E.
The Art of Computer Programming. Volume 1: *Fundamental Algorithms.*
Addison-Wesley, Reading, MA, 1968.

[7] Krasner, Glenn, ed.
Smalltalk-80: Bits of History, Words of Advice.
Addison-Wesley, Reading, MA, 1983.

[8] Motorola Inc.
M68000 16/32-bit Microprocessor: Programmer's Reference Manual.
Prentice-Hall, 1984.

[9] Rozas, G. J.
Liar, an Algol-like Compiler for Scheme.
MIT Bachelor's Thesis.
January, 1984

[10] Schorr, H. and Waite, W.M.
An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures.
Communications of the ACM 10(8):501-506, August, 1967.

[11] Steele, G.
Lambda: The Ultimate Declarative.
AI Memo 379, Massachusetts Institute of Technology, 1976.

[12] Steele, G.
Debunking the Expensive Procedure Call Myth or, Procedure Call Implementations Considered Harmful or, Lambda: The Ultimate Goto.
In *Proceedings of the 1977 ACM National Conference*, pages 153-162. ACM, 1977.

[13] Steele, Guy L.
Rabbit: A Compiler for Scheme (A Study in Compiler Optimization).
Technical Report AI-474, Massachusetts Institute of Technology, March, 1978.

[14] Steele, G. and Sussman, G.
The Revised Report on Scheme, A Dialect of LISP.
AI Memo 452, Massachusetts Institute of Technology, 1978.

[15] Tanenbaum, Andrew S.
Implications of Structured Programming for Machine Architecture.
Communications of the ACM 21(3):237-246, March, 1978.