LISP AND SYMBOLIC COMPUTATION: An International Journal, ?, ??–??, 1994

A Tractable Scheme Implementation

RICHARD A. KELSEY (*kelsey@ccs.neu.edu*)
Northeastern University
JONATHAN A. REES (*jar@cs.cornell.edu*)
MIT and Cornell University

Abstract. Scheme 48 is an implementation of the Scheme programming language constructed with tractability and reliability as its primary design goals. It has the structural properties of large, compiler-based Lisp implementations: it is written entirely in Scheme, is bootstrapped via its compiler, and provides numerous language extensions. It controls the complexity that ordinarily attends such large Lisp implementations through clear articulation of internal modularity and by the exclusion of features, optimizations, and generalizations that are of only marginal value.

1. Introduction

Scheme 48 is an implementation of the Scheme programming language constructed with tractability and reliability as its primary design goals. By tractability we mean the ease with which the system can be understood and changed. Although Lisp dialects, including Scheme, are relatively simple languages, implementation tractability is often threatened by the demands of providing high performance and extended functionality. The Scheme 48 project was initiated in order to experiment with techniques for maintaining implementation tractability in the face of countervailing pressures and to find out what tradeoffs were involved in doing so.

Small Lisp implementations are usually tractable merely by virtue of being small; it is usually possible for an experienced programmer to read and understand the entire source program in a few days. However, Scheme 48 specifically attempts to tackle the structural problems posed by large Lisp implementations. Although it is impossible to define the terms small or large precisely in this context, by a small Lisp system we generally mean one that is less than about 15,000 lines of source code, is based on an interpreter or simple translator not written in Lisp, and provides few, if any, features beyond a small core language such as IEEE standard Scheme [1]. By a large Lisp system we mean one that is more than about 15,000 lines and implements a large language such as Common Lisp [10] or a substantially extended dialect of Scheme.

We arrived at the number 15,000 by examining the source code for a collection of Lisp implementations. We found seven small Lisp systems in public file archives on the Internet, none of which contained more than 14,000 lines of source code, and three large systems of 25,000 to 120,000 lines. All of the large systems and two of the small ones included native code compilers. The line counts are those for the interpreters and run-time code and do not include the compilers.

Size is not the only reason that large Lisps tend towards incomprehensibility. Large implementation size is a result of additional functionality. Usually difficulties of comprehension are compounded by the authors using the additional features within the implementation itself. Even worse, once a system has been bootstrapped, features are often re-implemented in terms of themselves, and the original, non-circular code is deleted from the sources. Circular definitions are hard to understand and even harder to modify.

Scheme 48 is meant to be a substantial yet comprehensible system. The current Scheme 48 system includes many extensions to standard Scheme and has the basic structural characteristics of the large systems we surveyed. Tractability is difficult to measure, but our claim that Scheme 48 is tractable is based on it having the following features:

- explicit interfaces are used to divide the code into modules;
- these modules are organized in a hierarchical fashion: the major components of the system can be understood independently;
- the modules are largely independent: Scheme 48 can be built from the available sources, in a mix and match fashion, as anything ranging from a minimal implementation of a Scheme subset to an extended development environment;
- there are multiple implementations of many of the interfaces;
- all of the code is written in Scheme;
- major parts of the system, including the code to build a working image from the source files, can be run stand-alone using any IEEE Scheme implementation.

In addition, when writing the code for Scheme 48 we took the unusual step of giving simplicity priority over execution speed, unless measurements demonstrated that the increase in speed was significant. Implemented features not meeting this criterion were removed. The system's overall simplicity helped in performing such experiments by making it easy to isolate and evaluate different implementation choices.

Dialect	Use within Scheme 48
Pre-Scheme	Virtual machine implementation
Primitive Scheme	Minimal compiled subset
Scheme	General, standard
Big Scheme	General
Configuration language	Module descriptions

Table 1: Scheme Dialects

2. System Organization

We used two main strategies to keep Scheme 48 comprehensible: the system is written entirely in Scheme, using a variety of dialects, and is divided into modules with explicit interfaces and dependencies. For each module we used the simplest adequate dialect and tried to minimize the number of dependencies on other modules. The Scheme dialects used are listed in table 1. Primitive Scheme and Big Scheme have the same semantics as Scheme, with the deletion or addition of procedures and syntax but without significant change to the underlying semantics. Pre-Scheme is more of a departure, as it is designed to allow compilation to idomatic C code (integers are `int`s and so forth). This requirement results in some significant restrictions on Pre-Scheme programs, although they can still be run as Scheme programs. Pre-Scheme is discussed in section 4.

The Scheme 48 system is based on a virtual machine. Using a virtual machine raised many of the same organizational issues as using a native code compiler, such as how bootstrapping is accomplished, without forcing us to deal with actually generating machine code or with other details of hardware architecture. The virtual machine architecture gives a well-defined interface to the execution hardware. The run-time code can use that interface and ignore the question of whether the underlying machine is real or virtual. Using a virtual machine also has the immediate and practical benefit of making Scheme 48 easy to port to new platforms.

The Scheme 48 implementation has four main parts: a realization of the virtual machine, a Scheme compiler that produces code for the virtual machine, a static linker used to build executable images, and implementations of both standard and non-standard Scheme library routines. The virtual machine is written in Pre-Scheme, so that it can be compiled to efficient C or native code, and then run as a stand-alone program. The byte-code compiler provides an implementation of Primitive Scheme, and the rest of Scheme is implemented using code written in Primitive Scheme. The compiler, linker, utilities, and the extensions that make up Big Scheme are written in Scheme. Some of the extensions, such as multitasking, make use of functionality provided by the virtual machine that is not part of standard Scheme, and these modules cannot be run using other Scheme implementations. The virtual machine, compiler, and linker can be (and are) run using Scheme implementations other than Scheme 48.

Scheme 48's module system is designed to support both static linking and rapid turnaround during program development. The design was influenced by Standard ML modules [8] and the module system for Scheme Xerox [4]. Each module has its own isolated namespace, with visibility of bindings controlled by module descriptions written in a module configuration language. The module system bears some similarity to Common Lisp's package system [10], although it controls the mapping of names to denotations instead of the mapping of strings to symbols.

3. The Virtual Machine

The Scheme 48 virtual machine is a stack based machine similar to that described in [2]. It is written in Pre-Scheme, a Scheme subset that is described in section 4. The implementation of the virtual machine is organized according to the internal interfaces listed in Table 2. The following sections describe the interfaces and their implementations.

3.1. Architecture

The Scheme 48 architecture description is the interface between the virtual machine and the rest of the Scheme 48 system. It contains descriptions of the virtual machine's instruction set and data structures, and the kinds of possible interrupts.

Three kinds of data structures are described in the interface: fixed size objects containing other Scheme objects, vectors of Scheme objects, and vectors containing untagged data. For objects of fixed size, such as pairs, the architecture description lists the names of the various components (`car`, `cdr`) in the order in which they are stored in memory. Vectors of Scheme objects include Scheme vectors and records, and the templates described below. Vectors of untagged data include strings and vectors of byte codes.

3.2. Data Representations

The basic data structure representing Scheme objects is a descriptor. A descriptor is the size of a pointer and contains a tag in the low-order two bits. (This particular representation is tuned for use on byte-addressed machines where a pointer is four bytes.) The tag is one of:

External interface	
`architecture`	Instruction set description
Data structures	
`memory`	Pointers and pointer arithmetic
`descriptors`	Typed descriptors for Scheme objects
`fixnum-arithmetic`	Small integer arithmetic with overflow checks
`stored-objects`	Manipulating heap objects
`data-types`	Particular stored object types
`ports`	Scheme I/O ports
Storage management	
`heap`	Heap (including garbage collector)
`environments`	Lexical environments
`stacks`	Operand stack and continuations
Byte-code interpreter	
`interpreter`	Instruction dispatch; error detection
`resume`	Initialize, read a heap image, and start interpreting

Table 2: Virtual machine interfaces

- Fixnum: the non-tag portion of the descriptor represents a small integer. We use zero as the fixnum tag to simplify arithmetic operations.
- Immediate: the low-order byte gives the type of the object; the rest contains any other necessary information. Characters, the empty list, `#t` and `#f` are all represented as immediate values. Another use of immediate values is as illegal data values, allowing the virtual machine to detect and report attempts to refer to the value of uninitialized variables and other errors.
- Stored object: a pointer to a stored object. The first descriptor in the object is a header describing the object. The pointer actually points to the first non-header descriptor in the object. Most Scheme objects, such as strings, pairs, vectors, and so on are represented as stored objects. Stored objects either contain descriptors or untagged data depending on their type.
- Header: description of a stored object. The descriptor includes the type of the object, its size, and an immutability flag. If the immutability flag is set, attempts to modify the object result in an exception being raised.

The garbage collector and other heap manipulation routines are written to allow headers within stored objects. An object that normally contains only tagged data, such as a record, can also contain untagged data, as long as an appropriate untagged data header is placed before the untagged data. The system currently makes no use of this facility.

Having headers on all stored objects allows the contents of memory to be scanned. Starting at the head of any stored object, it is possible to scan through memory determing the type and contents of every subsequent stored object. The virtual machine makes use of this in a number of ways.

There are number of alternatives to this particular data layout, some of which have been used in other implementations [7]. Headers can use either a fixnum or an immediate tag, freeing up one tag value. This tag can be used to denote a pointer to a commonly used stored object type, typically pairs, eliminating the need for headers on objects of that type. Or the fourth tag can be unused, allowing some tag tests to be done using bit-test instructions. We chose the current data representations because they are simple and contain a certain amount of redundancy, and because saving one or two machine instructions per type check or `cons` is not likely to be important for an interpreted system. The modular nature of the entire system would make it easy to change the data representations, should the need arise.

There are five aggregate data structures used by the virtual machine: templates, closures, continuations, locations, and run-time environments. These are all stored objects. A template is a piece executable code. Each contains a vector of instructions and a vector of values that includes constants, locations, and other templates. A closure is the representation of a procedure; it contains a template and an environment. A continuation contains the information needed to restart the machine when a procedure call returns, specifically the operand stack and contents of some of the machine's registers. A location contains the value of a top-level variable. Locations are distinct from symbols to allow for multiple top-level environments.

Run-time environments contain the values of lexically bound variables. These are implemented as Scheme vectors, with the first value in each vector being the superior lexical environment. The address of a lexical variable is given as two indices: the depth of the containing environment, with the current environment being depth zero, and the index of the variable within the containing environment. A common alternative is to use a single vector for each environment, copying any necessary values from other environments whenever a new environment is made. This eliminates the need to chain back through environments when fetching the values of variables. We used nested environments to keep from having to add free variable analysis to the byte-code compiler.

3.3. Storage Management

The storage management system provides for the creation and reclamation of stored objects, including continuations and environments. In addition, a stack interface includes operations for maintaining the operand stack. Most objects are allocated from a heap. Heap storage is reclaimed using a simple two-space copying garbage collector. The design of the virtual machine puts few restrictions on the garbage collector and an early version of the VM has been used for extensive experiments in this area [13].

The contents of the heap may be written to and read from files. Heap image files contain a pointer to a distinguished entry procedure in that heap. Heap images are machine independent; they contain the information needed to relocate the heap in memory and to correct byte order if necessary. When the virtual machine is started it reads a heap image from a file and begins execution by calling the entry procedure. Execution ceases when that procedure returns. Heap image files can be created either using the `write-image` instruction or by using the static linker. A heap image file can be quite small, since it only needs to contain the information required to execute a call to the entry procedure.

Continuations and environments are treated specially, since they can often be reclaimed more efficiently than most objects. They are ordinarily not needed by the program after the return from the call for which they were created. The only exception to this is when `call-with-current-continuation` is used.

There are four operations involving continuations: create a continuation that contains the current argument stack and the machine's internal state; invoke a continuation, restoring the argument stack and the machine's state; preserve the current continuation for later use; and make a preserved continuation become the current one. The simplest implementation of this interface is to create all continuations in the heap. Creating and invoking continuations then requires copying the operand stack to and from the heap. Preserving and selecting continuations are no-ops, since continuations are always in the heap and all heap objects have indefinite extent.

An alternative implementation is to create continuations on the operand stack by pushing the machine's state on top of the current operands. This avoids the need to copy the operand stack back and forth. It also improves data locality by efficiently reusing stack space. Preserving a continuation becomes more expensive because it requires copying the continuation into the heap. The continuation is copied back to the stack when it is invoked. Continuations may be freely copied back and forth, because they are never modified, only created and invoked.

Environments can also be allocated either in the heap or on the stack. An environment is created by making a vector that contains a pointer to the current environment and the contents of the operand stack. This can be done by allocating a vector in the heap and moving the values into it, or by adding a vector header to the operand stack, at which point the stack pointer points to the new environment. As with continuations, preserving an environment that is on the stack requires moving it to the heap. Environments can be modified through the use of `set!`, so there must be only one copy of each environment. Once an environment has migrated to the heap it remains there.

Stack storage is reclaimed in two ways. When a continuation is invoked the stack pointer is set to point to the top of the restored operand stack, freeing up any stack space that was allocated since the continuation's creation. If the stack overflows, the current continuation and environment are copied to the heap, allowing the entire stack to be reused. All environments and continuations pointed to by these values are also copied if they are on the stack. This makes stack allocation compatible with Scheme's requirement for proper tail recursion, and also allows for recursions that are deeper than would fit in the stack. Tail-recursive calls cause the calling procedure's environment to become inaccessible, although it remains on the stack until the program returns past that point or the live portions of the stack are moved to the heap.

The storage allocation interface has been implemented both with and without stack allocation of environments and continuations. Using stack allocation is faster than using the heap, with some cost in increased complexity of the `stack` and `environment` modules. For more information see [5].

Another method of implementing proper tail recursion is to ensure that at every procedure call the arguments to the call are on the stack directly above the continuation for the call. For tail-recursive calls this requires moving the arguments just before jumping to the code for the called procedure. We added this form of tail call to the VM by adding a special call instruction that did the argument copying, and found that it was only slightly slower than using the stack garbage collector. However, the stack copying logic is still required for implementing `call-with-current-continuation`, so the argument copying instruction is not used. For more details see [5].

3.4. Interpreter

The section describes the more important virtual machine instructions. These instructions make use of the interpreter's registers, which are listed in Table 3. To start with a simple example, here are the instructions for

`Value`	the most recent instruction result
`PC`	byte-code program counter
`Template`	instruction vector and a vector of literal values
`Cont`	continuation
`Env`	environment
`Nargs`	number of arguments
`Dynamic`	dynamic state
`Enabled-Interrupts`	which interrupts are currently enabled
`Interrupt-Handlers`	vector of procedures for processing interrupts
`Exception-Handler`	procedure for processing exceptions

Table 3: The virtual machine's registers

the procedure `(lambda (x) (+ 10 x))`.

```
(check-nargs= 1)
(make-env 1)
(literal '10)
(push)
(local 0 1)
(+)
(return)
```

When control is transferred to the code for the procedure, `Nargs` contains the number of arguments being passed and `Env` holds the lexical environment in which the procedure was created. The procedure first checks that it was called with exactly one argument, and creates a new environment containing the argument and the current value of `Env`. An exception is raised by `check-nargs=` if the procedure was passed the wrong number of arguments.

The body of the procedure begins by loading the literal value `10` from `Template` and pushing it onto the stack. The `local` instruction is used to obtain the value of `x`, using two operands that give the depth of `x`'s environment in the chain of lexical environments and the offset of `x` in that environment. `+` then pops `10` off of the stack, adds it to the value of `x` (which is in `Value`), and leaves the result in `Value`. Finally, the procedure returns. The `return` instruction restores the operand stack and sets the `PC`, `Template`, `Env`, and `Cont` registers using values from the current contents of `Cont`.

For a procedure that takes an arbitrary number of arguments, such as `(lambda (x y . more-args) ...)`, the first few instructions would be

```
(check-nargs>= 2)
```

```
(make-rest-list 2)
(push)
(make-env 3)
```

The instruction `(check-nargs>= 2)` verifies that there are least two arguments present. `(make-rest-list 2)` puts all but the first two arguments into a list, which is left in `Value`. This list is then pushed onto the stack to become part of the environment.

The value of a `lambda` expression is a closure, containing a template and an environment. A closure is made using the `closure` instruction. The new closure contains the current environment and a template obtained from the current template.

A procedure call is performed by creating a new continuation, pushing the arguments onto the stack and using the `call` instruction. The code for `(g 'a)` is:

```
    (make-cont L1 0)
    (literal 'a)
    (push)
    (global g)
    (call 1)
L1: ...
```

The label operand to `make-cont` specifies the program counter to be used when the continuation is resumed. The second operand to `make-cont` is the number of values that are currently on the operand stack. These values must be saved in the continuation. The `make-cont` instruction is omitted for tail-recursive calls. The value of a top-level variable is obtained from a location stored in the template. The number of arguments in the call is itself an operand supplied to the `call` instruction.

A variable's value is set using `set-local!` or `set-global!`, which are identical to `local` and `global` except that they set the variable's value to be the contents of `Value` instead of vice-versa.

The architecture contains both jump and conditional jump instructions for executing conditionals. `(if a 1 2)` compiles to:

```
    (global a)
    (jump-if-false L1)
    (literal '1)
    (jump L2)
L1: (literal '2)
L2: ...
```

For both jump instructions the offset to the label must be positive; backwards jumps are not allowed. This ensures that every loop contains a procedure call, which in turn makes handling interrupts more uniform. There could be a backward jump instruction that included a check for interrupts, but making use of it would require a byte-code compiler somewhat more sophisticated than the current one.

The following is a description of some of the less frequently used instructions that manipulate the interpreter's internal state. In addition to the instructions described, there are a number of others that deal with numeric operations, allocating, accessing, and modifying storage, various input/output operations, and so on. There are a set of generic instructions for dealing with stored objects such as pairs, symbols, vectors and strings. These instructions take the type of the object as an additional operand from the instruction stream. Adding new types of stored objects does not require changing the instruction set.

`current-cont`
`with-continuation`
The first instruction sets `Value` to be the contents of `Cont`. The second takes two operands, a continuation obtained using `current-cont` and a procedure of no arguments. It sets `Cont` to be the continuation and then calls the procedure. Together these are used to implement Scheme's `call-with-current-continuation`.

`get-dynamic-state`
`set-dynamic-state!`
These move the contents of `Value` to `Dynamic-State` and vice versa. The machine makes no other use of `Dynamic-State`. An alternative design would be to keep the dynamic state in a top-level variable instead of in a VM register. Using a machine register allows the VM to be used in a multiprocessor environment, where each processor has its own registers but all share a single heap.

`set-exception-handler!`
This sets `Exception-Handler` to be the procedure in `Value`. Whenever an exceptional situation is detected the VM calls this procedure, passing to it the current instruction and its operands. If the exception handler returns, execution proceeds with the instruction following the one that caused the exception. Most causes of exceptions are type errors. Others include arithmetic overflows, unbound and unititialized variables, passing the wrong number of arguments to procedures, and errors when calling operating system routines. Appropriate exception handlers can be used to extend and augment the virtual machine.

`set-interrupt-handlers!`
`return-from-interrupt`

0	`(check-nargs= 2)`	
2	`(make-env 2)`	
4	`(local 0 1)`	value of `j`
7	`(push)`	push `j` as the first argument to `+`
8	`(make-cont (=> 20) 1)`	`20` is the code, stack has one value
12	`(local 0 2)`	
15	`(push)`	push `i` as the first argument to `h`
16	`(global h)`	
18	`(call 1)`	call `h` with one argument
20	`(+)`	code resumes here after the call to `h`
21	`(push)`	push second argument to `g`
22	`(global g)`	
24	`(call 1)`	

Figure 1: Byte codes for `(lambda (i j) (g (+ j (h i))))`

`set-enabled-interrupts!`
The VM checks for interrupts whenever a `call` instruction is executed. If an enabled interrupt has occurred since the previous call, the state of the machine is saved in a continuation and the procedure appropriate for that interrupt is obtained from the vector `Interrupt-Handlers` and called. The `return-from-interrupt` instruction can then be used to restore the state of the machine and continue execution. `set-enabled-interrupts!` sets `Enabled-Interrupts` and places its old value in `Value`.

`write-image`
This takes two operands, a procedure and the name of a file, and writes the procedure into the file as a restartable heap image. The writing process is similar to a garbage collection in that it only writes objects reachable from the procedure into the file. See section 3.3 for more information.

There are several instructions used to allow calls to return multiple values. They are more complex than strictly necessary, due to our desire not to require any additional overhead when returning exactly one value, and are not described here for reasons of space.

4. Pre-Scheme Implementations

The virtual machine is written in Pre-Scheme, a subset of Scheme chosen to allow compilation into fairly natural C code without losing too much of Scheme's expressive power. The virtual machine and the rest of the system are written in overlapping subsets of the same language, allowing some modules (such as the architecture description code) to be used in both. Because Pre-Scheme is a subset of Scheme, the virtual machine can be run and debugged using any Scheme implementation. This fact has greatly assisted development, in particular because it permitted us to work on the Scheme 48 virtual machine prior to the existence of a direct Pre-Scheme compiler or working Scheme 48 system. The VM is quite slow when run as a Scheme program. At one point a hand translation of the VM into C was done by Bob Brown, an MIT graduate student.

Except for modification and debugging, the virtual machine is compiled from Pre-Scheme into C, using a compiler based on the transformational compiler described in [6]. The Pre-Scheme compiler evaluates top-level forms, performs static type checking and a great deal of partial evaluation, and translates the resulting program into C. The result is a C version of the VM which is as efficient and as portable as a hand-coded C version would be.

Pre-Scheme differs from Scheme in the following ways:

- Every top-level form is evaluated at compile time and may make unrestricted use of Scheme.
- The Pre-Scheme compiler determines all types at compile time. Type reconstruction is done after the top-level forms have been evaluated. There is no run-time type discrimination.
- There is no automatic storage reclamation. Unused storage must be freed explicitly.
- Proper tail recursion is guaranteed only for calls to `lambda` forms bound by `let` and `letrec` or when explicitly declared for a particular call.

The main restrictions imposed by Pre-Scheme are that the VM must be accepted by Pre-Scheme's type-checking algorithm and that there is no automatic storage reclamation. The latter restriction is reasonable, as Pre-Scheme is intended for writing such things as garbage collectors. The Pre-Scheme compiler uses a Hindley-Milner polymorphic type reconstruction algorithm modified to allow overloaded arithmetic operators and to take into account the compiler's use of partial evaluation. If a procedure is to be in-lined for all uses, then it can be fully polymorphic, as every use will have a distinct implementation.

Using Pre-Scheme has been very successful. The code for the VM makes use of higher-order procedures, macros that operate on parse trees, and other features of Scheme that would not have been available had we used

C. Procedures are also used to implement data abstractions within the VM. This results in a large number of small procedures. The virtual machine's 570 top-level forms, consisting of 2314 lines of Scheme code (excluding commments), are compiled to 13 C procedures containing 8467 lines of code. Once compiled into C it runs as quickly as similar hand-coded C programs.

Figure 2 illustrates the coding style used in the VM. The example consists of the code implementing the addition instruction. The first two forms are from the `fixnum-arithmetic` module, which exports `add-carefully`, and the last two are from the `interpreter` module. The procedure `carefully` takes an arithmetic operator and returns a procedure that performs that operation on two arguments, either passing the tagged result to a success continuation, or passing the original arguments to a failure continuation if the operation overflows. `extract-fixnum` and `enter-fixnum` remove and add type tags to small integers. The function `overflows?` checks that its argument has enough unused bits for a type tag. `goto` indicates that a tail-recursive call should be compiled using proper tail recursion. `carefully` can then be used to define `add-carefully` which performs addition on integers.

`define-primitive` is a macro that expands into a call to the procedure `define-opcode` which actually defines the instruction. The three arguments to the macro are the instruction to define, input argument specifications, and the body of the instruction. The expanded code retrieves arguments from the stack, performs type checks and coercions, and executes the body of the instruction. This is a simple Scheme macro that would be painful to write using C's limited macro facility.

Figure 3 shows the C code produced for the addition instruction. This is part of a large `switch` statement which performs instruction dispatch.

(A note on identifiers: many Scheme identifiers are not legal C identifiers, while on the other hand C is case-sensitive and Scheme is not. The compiler uses upper-case letters for the characters that are legal in identifiers in Scheme but not in C; for example `*val*` becomes `SvalS`. The compiler also introduces local variables to shadow global variables to improve register usage (similar to [11]). These introduced variables begin with `R`; thus `RSvalS` is a local variable shadowing the global variable `SvalS`.)

This code is not what we would have written if we had used C in the first place, but it is at least as efficient. The use of Pre-Scheme makes the program relatively more comprehensible and easier to modify without incurring any run-time cost.

```
(define (carefully op)
  (lambda (x y succ fail)
    (let ((z (op (extract-fixnum x) (extract-fixnum y))))
      (if (overflows? z)
          (goto fail x y)
          (goto succ (enter-fixnum z))))))

(define add-carefully (carefully +))

(define (arith op)
  (lambda (x y)
    (op x y return arithmetic-overflow)))

(define-primitive op/+ (number-> number->) (arith add-carefully))
```

Figure 2: Implementation of the addition instruction

`module-system`	Signatures, modules, and name lookup
`syntactic`	Denotations; hygienic macros
`compilation`	Byte-code compiler
`analysis`	Optional optimization phase
`linker`	Static heap image linker
`reification`	Support for doing `eval` relative to a statically linked image
`assembler`	Optional byte-code assembler

Table 4: Byte-code compiler interfaces

5. Byte Code Compiler

The byte-code compiler compiles Scheme expressions into appropriate sequences of VM instructions. The compiler is as simple as we could make it and still get acceptable performance. It does very few optimizations. We use a simple compiler because it was easier to write, is easier to read, is more likely to be correct, and improves the quality of debugging information provided to users. The less processing the compiler does the easier it is to relate the state of the machine to the source program when errors are encountered.

The compiler takes as arguments an expression, an environment, the number of values currently pushed on the stack, and information about the expression's continuation. The stack value count is needed to generate

```
case 46 : {
  long arg2_267X;
  /* pop an operand from the stack */
  RSstackS = (4 + RSstackS);
  arg2_267X = *((long*)((unsigned char*)RSstackS));
  /* check operand tags */
  if ((0 == (3 & (arg2_267X | RSvalS)))) {
    long x_268X;
    long z_269X;
    x_268X = RSvalS;
    /* remove tags and add */
    z_269X = (arg2_267X >> 2) + (x_268X >> 2);
    /* overflow check */
    if ((536870911 < z_269X)) {
      goto L20950;}
    else {
      /* underflow check */
      if ((z_269X < -536870912)) {
        goto L20950;}
      else {
        /* add tag and continue */
        RSvalS = (z_269X << 2);
        goto START;}}
   L20950: {
    merged_arg1K0 = 0;
    merged_arg0K1 = arg2_267X;
    merged_arg0K2 = x_268X;
    goto raise_exception2;}}
  else {
    merged_arg1K0 = 0;
    merged_arg0K1 = arg2_267X;
    merged_arg0K2 = RSvalS;
    goto raise_exception2;}}
  break;
```

Figure 3: Compiler output for the addition instruction

`make-cont` instructions. The continuation argument has two parts, a kind and an optional name. The kind is either `return` (the expression's value is returned by the procedure being compiled), `ignore` (the value will not be used), or `fall-through` (there is some following code that uses the value). The name indicates the variable to which the value of the expression will be bound, if any. If the expression is a `lambda` form, the name will be used to provide debugging support.

For each lexically bound variable the compile-time environment has the location of the variable in terms of the number of run-time environments that must be chained through (back) and the index of the variable within the final environment (over). For each top-level variable the environment has either a location, from which the variable's value can be obtained at run time, a special compilation method, or both. For a variable bound to a primitive operator the compilation method is a procedure for generating code for the operator. If the variable is bound to a macro, the compilation method is a code transformation procedure. The macro facility is based on the system described in [3]. Many of the special forms in Scheme are implemented using macros.

The compiler itself is quite simple. Each procedure is compiled into a separate template. The code for each checks the number of arguments, makes a new environment and then executes the body.

Literal values are put in the template, and loaded at run time using the `literal` instruction. Each variable is looked up in the environment. The instruction generated for accessing or setting the variable's value depends on the denotation returned. `if` is compiled using `jump-if-false` and then `jump` to reach the following piece of code. A `lambda` form is compiled into a template, which is placed in the current template as an operand to `closure`. `begin` concatenates the code generated for each expression.

From the point of view of the compiler there are three kinds of procedure calls: calls to primitives, calls to `lambda` forms, and all other calls. Calls to primitives are compiled using the procedure obtained from the environment. The code for other calls starts with `make-cont` if the call is not tail-recursive (if the `cont` argument to the compiler is not `return`.) Then the arguments are compiled in order followed by `push` instructions. If the call is to a `lambda` form the code for the `lambda` is generated in-line to save the cost of creating a closure. For all other calls code to evaluate the procedure is compiled followed by a `call` instruction.

The compiler also contains code to compile procedural versions of the primitives. For many primitives, such as `+`, it is necessary to also have a procedure of the same name that invokes the primitive.

The output of the compiler is passed to an internal assembler which calculates jump offsets, builds the vector of values for the template, and makes the template itself. This assembler also produces a structure containing any debugging information emitted by the compiler. Some of this information is indexed by the sections of the assembled code to which it pertains. Debugging information includes the names of bound variables and their locations in the environment, the source code for procedure calls and their continuations, and names of procedures. There is a separate assembler, not used by the compiler, that can assemble hand-written byte-code programs.

5.1. Optimizations

We have added a few optimizations to the compiler and the instruction set since the original implementation. There are quite a number of other optimizations that could have been done but were not, usually because the increase in execution speed was not felt to be worth the added complexity. Among the optimizations that are in the system are special instructions for accessing local variables in the current environment or its immediate ancestors, not creating closures for `lambda` forms found in call position, not creating empty environments, and a few others.

The instruction set has not been optimized for execution speed or code size to any great extent. A few instructions, such as the `string=?` instruction, could be replaced by signifigantly slower Scheme code in the run-time system. There are a number of common sequences of instructions that could be merged into single instructions. Experiments have determined that the resulting speed up is not large; the VM does not appear to be spending a great deal of its time in fetching instructions.

Other possibilities for increased optimization abound. Some examples: calls to known procedures, such as those for simple loops, could be compiled as jumps; a `jump-if-null` instruction would speed up list processing code; boolean expressions could be compiled for effect; leaf procedures (those that do not create other procedures) don't need to create environment vectors.

The VM instructions needed for many such optimizations already exist. However, the compiler would have to be made considerably more complex in order to make use of them. Because the compiler is itself usually run using Scheme 48, additional compiler complexity carries with it the penalty of signifigantly slower compilation speed. A system was built that did not create leaf environments. In our judgement, the increase in speed did not make up for the additional complexity and increased compilation time, so this optimization was removed.

One change that might or might not improve the simplicity or speed of the system would be to eliminate the value register and the `push` instruction. The value register is equivalent to a cache for the top of the stack. Removing

`primitive-scheme`	Special forms and procedures that are necessarily implemented directly by a compiler or interpreter: `lambda`, `if`, `car`, `read-char`, etc.
`usual-macros`	Scheme's usual derived expression types: `let`, `cond`, `do`, etc.
`scheme-level-1`	Procedures that are easily implemented in terms of `primitive-scheme`: `not`, `memq`, `equal?`, etc.
`numeric-tower`	Number types other than small integers (bignum, ratnum, flonum, recnum)
`winding`	`call-with-current-continuation` and `dynamic-wind`
`number-i/o`	`number->string` and `string->number`
`reading`	The `read` procedure
`writing`	The `write` procedure
`evaluation`	`eval` and `load`
`high-level-macros`	The `syntax-rules` macro
`scheme`	Union of the above interfaces

Table 5: Scheme language interfaces

it would make the architecture description a little shorter. The size of the code for the system would be essentially unchanged. The number of VM instructions executed would decrease, as there would not be any `push` instructions, but many of the instructions would have to do a little more work.

6. Run-time library

The byte-code compiler and virtual machine implement a core Scheme dialect that we call Primitive Scheme. Scheme 48's run-time library builds additional functionality on top of Primitive Scheme.

6.1. Scheme in terms of Primitive Scheme

The `scheme` interface specifies the entire Scheme language as defined by the Revised^4 Report. `scheme` is organized as the union of a number of smaller interfaces for two purposes: (1) configuring small systems that use only particular subsets of the language, and (2) structuring implementations of the full language.

The modules that provide implementations of `scheme`'s sub-interfaces

(Table 5) are straightforward, for the most part. The `primitive-scheme` interface is implemented directly by the byte-code compiler and virtual machine, while all the other interfaces have ordinary implementations as Scheme programs.

The module for `numeric-tower` is more closely tied to the VM architecture than most of the other library modules. It consists of a set of exception handlers for the VM's arithmetic instructions. If the VM interpreter encounters such an instruction for which the arguments are numbers that are not implemented directly by the VM, then it raises an exception. The exception is handled by a procedure that dispatches to an appropriate specialist function: for example, if the two arguments are large integers, then the large integer addition routine is invoked.

Following the philosophy of minimizing internal dependencies, the numeric tower is carefully constructed so that the numeric types do not depend on one another. It is possible to make use of any combination of number types; for example, rational numbers can be used in the absence of large integers.

The `evaluation` interface has two implementations. The first is a simple meta-circular interpreter that does not rely on the virtual machine architecture. The second implements `eval` as a three-step process:

1. Invoke the byte-code compiler on the form, producing a template.
2. Make a closure from the given template and a null lexical environment. (Global variables are accessed via the template, not via the environment component of the closure.)
3. Invoke the closure as an ordinary procedure.

Step 3 is the "reflective" step that starts the virtual machine running on the newly compiled form.

Many of these interfaces include entry points that do not become part of Scheme. For example, the `numeric-tower` interface includes the procedures necessary to add additional types of numbers.

6.2. Big Scheme in terms of Scheme

"Big Scheme" consists of a diverse collection of utilities and language extensions of general interest. A few of these modules are used in the implementation of the basic Scheme library. For example, the definitions of `current-input-port` and `current-output-port` are in terms of dynamic variables, which come from the `fluids` interface. Also, the byte-code compiler makes heavy internal use of records, tables, and enumerated types.

`bitwise`	Bitwise logical operations on integers
`records`	Record package
`fluids`	Dynamically bound variables
`enumerated`	Enumerated types
`tables`	Hash tables
`byte-vectors`	Blocks of memory accessible by byte, word, or half-word
`conditions`	Condition system
`exceptions`	Particular conditions for errors at the level of `primitive-scheme`
`interrupts`	Handling asynchronous interrupts; critical sections
`queues`	FIFO queues
`random`	Random number generator
`sort`	Sorting lists
`pp`	Pretty-printer
`format`	Formatted output
`extended-ports`	I/O from/to strings, etc.
`external-calls`	External function call
`threads`	Multitasking
`weak-pointers`	Weak pointers
`big-scheme`	Union of the above interfaces

Table 6: Utilities and extensions interfaces

Other modules, such as conditions and interrupts are used in the development environment (see next section). Still other modules, such as multitasking and external calls, are useful for general applications programming.

Many of the interfaces in this group have two implementations: one written in portable Scheme, and another that exploits special features of the Scheme 48 virtual machine architecture. For example, the record package has a portable implementation in which records are represented using vectors, and a Scheme 48-specific implementation in which records are a distinct primitive type. The portable version can be used when running the linker or other applications on a substrate other than Scheme 48, while the Scheme 48-specific version provides increased functionality, performance, or debuggability.

6.3. Scheme development environment

A complete development environment, comprising byte-code compiler, run-time library, a command processor, and debugging utilities, can be packaged as a single heap image for execution by the VM. This image can be

`command-processor`	Command-oriented user interface
`package-commands`	Commands for manipulating modules
`application-builder`	Application image builder
`disclosers`	Extensions to `write` and `display-condition` to assist in debugging
`debugging`	Trace, backtrace, time, and similar commands
`inspector`	Data structure, stack, and environment inspector
`disassembler`	Byte-code disassembler

Table 7: Development environment interfaces

generated by the byte code compiler and linker using any implementation of standard Scheme. Scheme 48 is "bootstrapped" only in the straightforward sense that it includes an implementation of standard Scheme, which is a sufficient basis for executing the code necessary for it to build itself.

The command processor is part of the Scheme development environment. It is similar to an ordinary Lisp read-evaluate-print loop in that it repeatedly reads and executes forms (expressions and definitions), but differs in that it also accepts requests for meta-level operations, and these requests (called commands) are syntactically distinguished from forms to be executed. Meta-level operations include obtaining backtraces or trace output, specifying compiler directives, measuring execution time, and exiting the development environment. The purpose for this distinction is to leave the program namespace uncluttered with bindings of operators that are only meaningful during program development. For example, in Common Lisp, one would say `(trace foo)` to request tracing output for calls to `foo`, but in the Scheme 48 command processor one says `,trace foo`. When debugging programs that make use of the module system, the entire command set remains available regardless of the current expression evaluation context.

7. Discussion

Our claim is that by writing Scheme 48 as a collection of largely independent modules we have produced a sophisticated and extended Scheme implementation that is reliable, tractable, and easy to modify. Although we have not stressed implementation efficiency in this paper, Scheme 48 is a reasonably fast system, and is in use as a turnkey Scheme programming system. It runs at approximately the speed of the fastest available interpreters. (The widely differing implementation methodologies make it difficult to obtain meaningful numbers.)

The Scheme 48 system has been used for research in memory management, embedded systems, multiprocessing, language design, and computer system verification. Scheme 48 was chosen as the platform for these projects because of its internal tractability and flexibility.

Paul Wilson has done extensive research on memory management techniques using an early version of Scheme 48 [12, 13]. Because of its virtual machine architecture, Scheme 48's memory usage patterns are similar to those of more complex systems, and is easier to modify and experiment with.

Scheme 48 is currently being used to program mobile robots [9]. Multi-threaded user programs are executed by the virtual machine on an embedded processor on board the robot, while the byte-code compiler and development environment run on a work station using a Scheme implementation built on top of Common Lisp. Programs are compiled on the work station and then downloaded to the robot via a detachable tether. The on-board component of the system uses the standard virtual machine (with some additional hardware operations) and a stripped down version of the run-time library. There is no need for a read-eval-print loop on board, for example, and the system fits easily within the limited memory available on the robot (0.5 megabytes of RAM and 0.25 megabytes of EPROM). The virtual machine and the initial heap image take up 21 and 80 kbytes, respectively, of the EPROM.

Olin Shivers used Scheme 48 as a substrate for `scsh`, a Unix shell programming language based on Scheme that gives the user full access to existing Unix programs. Scheme 48 was chosen because it provides a good programming environment, a general exception handling mechanism, and the ability to make small stand-alone programs, all of which are desirable for a shell programming language.

The VLISP project at MITRE Corporation and Northeastern University used Scheme 48 as the basis for a fully verified Scheme implementation. The existence of the virtual machine interface, the simplicity of the byte-code compiler, and the fact that the entire system is written in Scheme combined to greatly simplify the verification process. The fact that the VLISP project was able to adopt much of Scheme 48's design unchanged corroborates our claim that Scheme 48 is tractable and reliable.

References

1. IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY (1991).

2. Clinger, William. The scheme 311 compiler: An exercise in denotational

semantics. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming* (August 1984) 356–364.

3. Clinger, William and Rees, Jonathan. Macros that work. In *Conf. Rec. 18 ACM Symposium on Principles of Programming Languages* (1991).

4. Curtis, Pavel and Rauen, James. A module system for scheme. In *Proc. 1990 ACM Symposium on Lisp and Functional Programming* (1990) 13–19.

5. Kelsey, Richard. *Tail-Recursive Stack Disciplines for an Interpreter.* Technical Report NU-CCS-93-03, Northeastern University College of Computer Science, Boston, MA (1992).

6. Kelsey, Richard and Hudak, Paul. Realistic compilation by program transformation. In *Conf. Rec. 16 ACM Symposium on Principles of Programming Languages* (1989) 281–292.

7. Lee, Peter. *Topics in Advanced Language Implementation.* MIT Press, Cambridge, MA (1991).

8. MacQueen, David B. Modules for standard ML. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming* (1984) 198–207.

9. Rees, Jonathan and Donald, Bruce. Program mobile robots in scheme. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation* (1992) 2681–2688.

10. Steele, Guy L. *Common Lisp: the Language.* Digital Press, Burlington MA, second edition (1990).

11. Tarditi, D., Acharya, A., and Lee., P. *No Assembly Required: Compiling Standard ML to C.* Technical Report, School of Computer Science, Carnegie Mellon University (1991).

12. Wilson, Paul R. and Moher, Thomas G. Design of the opportunistic garbage collector. In *Proceedings of the Sigplan 1989 Conference on Object-Oriented Programming: Systems, Languages, and Applications* (1989).

13. Wilson, Paul R., Lam, Micheal S, and Moher, Thomas G. Effective "static-graph" reorganization to improve locality in garbage collected systems. In *Proceedings of the Sigplan 1988 Conference on Programming Language Design and Implementation* (1991).