A Parallel Virtual Machine for Efficient Scheme Compilation

Marc Feeley and James S. Miller*
Brandeis University
Waltham, MA 02254-9110

Abstract

Programs compiled by Gambit, our Scheme compiler, achieve performance as much as twice that of the fastest available Scheme compilers. Gambit is easily ported, while retaining its high performance, through the use of a simple virtual machine (**PVM**). PVM allows a wide variety of machine-independent optimizations and it supports parallel computation based on the `future` construct. PVM conveys high-level information bidirectionally between the machine-independent front end of the compiler and the machine-dependent back end, making it easy to implement a number of common back end optimizations that are difficult to achieve for other virtual machines.

PVM is similar to many real computer architectures and has an option to efficiently gather dynamic measurements of virtual machine usage. These measurements can be used in performance prediction for ports to other architectures as well as design decisions related to proposed optimizations and object representations.

1 Introduction

Our primary interest is in efficient mechanisms for implementing `future`-based symbolic computation on currently available MIMD machines. Having already done work in this area using the Scheme language (augmented with the `future` mechanism[8][13]) we are now extending our interpreter-based results into the realm of compiled Scheme code. For this purpose we undertook the implementation of a new Scheme compiler, **Gambit**, with the intention of creating a simple environment for experiments across a wide range of hardware platforms and over a range of implementation techniques. The major design goals for Gambit, from the outset, were:

1. Code generation for multiple target machines, spanning both common CISC computers (DEC Vax, Motorola MC68000) and RISC computers (HP Precision Architecture (HPPA), MIPS R2000, Motorola 88K, BBN Monarch). For our purposes it was important that retargetting the compiler be simple and yet still yield a high performance system. We rejected existing compiler-based Scheme systems (T with Orbit[11], CScheme with Liar[12]) mainly because of the difficulty of retargetting and modifying the compilation strategy of these large systems.

2. High performance of output programs. We are not concerned with program development features. For example we do not allow the user to interrupt execution of a program other than by aborting it.

3. Support for task creation and synchronization through implicit data operations, possibly augmented by control constructs. The `future` construct provides these features compatibly with most other features of the Scheme language and was, therefore, our initial focus. We are also interested in exploring other parallel control and data constructs.

While the first and second goals are somewhat at odds with one another, we believe that architectural comparisons and architecture independent implementation techniques will be among the important results from our research. We have therefore chosen to build a compiler based on an easily retargetted virtual machine even though it may result in less efficient compiled code. Fortunately, our experience with Gambit indicates that a well chosen virtual machine does not result in any noticeable performance penalties.

2 PVM: A Parallel Virtual Machine

In designing our virtual machine we tried to avoid a pair of twin hazards that we have seen in other virtual machines used for compilation. On the one hand, there are virtual machines (like MIT's `scode`[3], or the code objects of UMB Scheme[4]) that are so close to the source language that the machine independent front end of the compiler is unable to express important optimizations in the virtual machine's instruction set. This places a major burden on the back end, which becomes responsible for analysis of the virtual machine code – a task very nearly as difficult as the original compilation task. On the other hand, there are virtual machines (like Multilisp's `mcode`[8] or Scheme 311's byte code) that match neither the actual target machine nor the source language. The result is either a complex back end that again attempts to recover data and control flow information from

*This research was supported in part by the Open Software Foundation, the Hewlett–Packard Corporation, and NSF equipment grant CDA–8715228. Marc Feeley is on study leave from the Université de Montréal.

the virtual machine, or a simple back end that produces poor code.

Our Parallel Virtual Machine, or **PVM**, is intended to fall in between these kinds of machines. We allow each back end to specify a wide range of "architectural details" of the virtual machine, including a description of primitive procedures available on the target machine and the number of general purpose registers. As a result, we can think of PVM as a *set* of virtual machines, depending on the back end description that is used. Each specific virtual machine is close to its target machine, yet the common abstraction hides the precise details from the front end. PVM also remains close to the source language since its small instruction set closely matches the Scheme language itself.

PVM can be viewed as a *bidirectional* communication medium between the front and back ends of the compiler. The traditional role of a virtual machine, of course, is to convey information from the front end to the back end. PVM, however, conveys information in the reverse direction as well:

- The number of general registers.
- The procedure calling convention.
- The format of closures.
- Enumeration and description of primitive procedures.
- Machine-specific declarations.

We view this bidirectional communication as an important component of Gambit's organization. The communication is supported by a language for describing implementation-level objects, which is the basis of the PVM abstraction. Four types of objects are manipulated using this language: primitive procedures, data objects, stack frames, and argument/parameter blocks. Corresponding to each of these is a means of reference: the name of the primitive procedure, slots within a data structure, slots within a stack frame, and argument/parameter number. This particular level of abstraction is convenient for both the front and back ends. For example, both the back and front ends agree to discuss stack slots as positive integers, in units of Scheme objects, increasing as objects are pushed on the stack. This is clearly convenient for the front end, and the back end can easily translate this into appropriate offsets from a base register, taking into account the number of bytes per argument, the direction of stack growth, and the choice of stack discipline on the target machine.

2.1 Operands

PVM has seven classes of operands, as shown in Figure 1, which naturally divide storage into disjoint areas: registers, current stack frame, global variables, heap storage, constant area, and code area. This makes it easy to track values and (with the exception of `mem` operands) removes the traditional aliasing problem.

Neither the stack nor the heap pointer is directly visible. Instead, the stack is accessible by indexing off of a virtual frame base pointer that is modified as part of the procedure call mechanism. The heap is accessed implicitly when allocating objects and explicitly by indexing relative to existing heap-allocated objects. By making the stack pointer and heap pointer invisible, we allow the back end to make a number of optimizations based on the target architecture.

The `mem` operand, which gives access to heap storage, allows nesting of other operands in its *base* component. Our front end, however, uses it only for access to closed variables; we leave other data structure accesses to the more general `APPLY` instruction. As a result, the ability to nest other operands within `mem` is not actually in use, although the back ends support it.

Operand	Meaning
`reg(`n`)`	General purpose register n
`stk(`n`)`	N^{th} slot of the current stack frame
`glob(`*name*`)`	Global variable
`mem(`*base*`,` *offset*`)`	Indexed reference (*base* is an operand, *offset* is a constant)
`obj(`*object*`)`	Constant
`lbl(`n`)`	Program label
`?`*loc*	Parallelism support, see Section 2.7

Figure 1: PVM Operands

Finally, we note that all operands can be used as the *source* of values. However values cannot be stored into `obj`, `lbl`, or `?` operands.

2.2 Instructions for Sequential Computation

The PVM instruction set provides a small set of general instructions to efficiently encode the operation of Scheme programs. Like many compilers, Gambit represents the program as a set of basic blocks. This representation is apparent in the PVM code. Each basic block is headed by a code label, followed by the code for the data operations in the block, and ends with a branch instruction. Our current instruction set for sequential computation consists of four kinds of code labels, three data manipulating instructions, and three branch instructions.

An important part of Gambit's communication mechanism is the description of a set of procedures, known as *primitives*, that are supported by the back end. All primitives are available through the general procedure call mechanism, but some can also be open coded by the `APPLY` and `COND` instructions. The front end requires the back end to supply a specific minimal set of (about 35) primitive operations; but the back end can, in fact, specify *any* procedure as a primitive. The description of each primitive indicates its arity and strictness. It also indicates whether it can be open coded, and whether it can return a placeholder as a value. Thus, `list` has unbounded arity, is not strict in any argument, and never returns a placeholder, while `set-car!` has arity two, is strict in its first argument but not its second, and never returns a placeholder.

PVM's handling of stack frames is unusual, and is described in Section 2.3. The *size* parameter to the label and branch instructions is used to support this mechanism, and is described in detail in that section.

The description of the sequential PMV instructions follows. Figure 2 shows a simple program (iterative factorial), along with its PVM code and the code generated for the MC68020. A comparison to code from other compilers is included in Appendix A.

`LABEL(`n`,` *size*`)`

A **simple label**, n, which may appear only in `JUMP` and `COND` instructions.

```
(##declare (standard-bindings) (fixnum))

(define (fact n) ; Iterative factorial
  (let loop ((i n) (ans 1))
     (if (= i 0)
         ans
         (loop (- i 1) (* ans i)))))
```

Virtual Machine Code	MC68020 Code
`LABEL(1,0,PROC,1)`	`L1:  bmi     L7      ; Entry point`
	`     jsr     20(a6)  ; Arity error`
	`L7:`
`  COPY(obj(1),reg(2))`	`     moveq   #8,d2`
`  COND(##fixnum.=,reg(1),obj(0),5,6,0)`	`     move.l  d1,d0`
	`     beq     L5`
`LABEL(6,0)`	`L6:`
`  APPLY(##fixnum.*,reg(2),reg(1),reg(2))`	`     asr.l   #3,d2`
	`     muls.l  d1,d2`
`  APPLY(##fixnum.-,reg(1),obj(1),reg(1))`	`     subq.l  #8,d1`
`  COND(##fixnum.=,reg(1),obj(0),5,6,0)`	`     move.l  d1,d0`
	`     bne     L6`
`LABEL(5,0)`	`L5:`
`  COPY(reg(2),reg(1))`	`     move.l  d2,d1`
`  JUMP(reg(0),0)`	`     jmp     (a0)`

Figure 2: Sample Program with Gambit Output

`LABEL(`*n*`, `*size*`, CONT)`
: A **continuation label**, similar to a return address in other architectures. These are *not* the objects returned to a Scheme program that calls `call-with-current-continuation`, but are an architectural feature to support fully tail-recursive behavior[15][18].

`LABEL(`*n*`, `*size*`, PROC, `*desc*`)`
: A **procedure label** with *desc* defining the number (or range) of arguments expected. This instruction modifies the stack and registers to account for any discrepancy between *desc* and the number of arguments actually passed (as specified by the `JUMP` instruction used to arrive here).

`LABEL(`*n*`, `*size*`, PROC, `*desc*`, CLOSED)`
: A **closure label**, similar to a procedure label, but the label *n* may appear only within a `closure` object created by `MAKE_CLOSURES`.

`APPLY(`*prim*`, `$operand_1$`, ..., [`*loc*`])`
: Apply the primitive procedure *prim* to the *operands* and store the result in *loc* (or discard it if *loc* is not specified).

`COPY(`*operand*`, `*loc*`)`
: Copy the value of *operand* to the location specified by *loc*.

`MAKE_CLOSURES(`$description_1$`/.../`$description_n$`)`
: Create *n* **closure objects**. A closure object contains a code label and a number of data slots. Each *description* specifies a location into which a closure object will be stored, the closure label for the code of that closure, and operands to be stored in the closure's data slots.

`COND(`*prim*`, `$operand_1$`, ..., `*t_lbl*`, `*f_lbl*`, `*size*`)`
: A conditional branch based on the value of *prim* applied to the *operands* (on false branch to *f_lbl*, otherwise to *t_lbl*, both of which must specify simple labels).

`JUMP(`*operand*`, `*size*`)`
: Jump to the simple or continuation label specified by the value of *operand*.

`JUMP(`*operand*`, `*size*`, `*nargs*`)`
: Jump to the address specified by *operand*. This instruction also states that *nargs* have been placed in the appropriate argument passing locations. The value of *operand* must be either a procedure label, a closure object, or a primitive procedure object.

2.3 Stack Frames

PVM deals with the stack frame in a novel manner, supplying the current stack frame size in the `LABEL`, `COND`, and `JUMP` instructions. Our approach avoids the problems inherent in using virtual machines either based purely around a top of stack pointer or based purely upon a frame pointer. Using a stack pointer leads to varying offsets for a given stack slot and inefficient code on machines lacking explicit stack instructions. Using only a frame base leaves the top of stack unknown at garbage collection time and requires update instructions on entry and exit of every basic block.

While the actual instruction set of PVM makes use of a frame pointer and frame size information, we prefer to think of the machine as having both a stack pointer and a frame pointer. Since the frame size always specifies the distance between the stack pointer and the frame pointer, either pointer can be recomputed from the other. `JUMP` and `COND` instructions cause the stack pointer to be recalculated, while `LABEL` instructions recalculate the frame pointer. Within a basic block, the stack pointer is updated based on the offsets of `stk` operands encountered so that it always covers the active part of the stack frame.

The choice between stack pointer and frame pointer discipline is specific to the back end (see Section 3.3). We take advantage of the fact that our front end produces the PVM code for an entire basic block before beginning native code generation. For each instruction, the front end calculates a tight bound on the size of the stack frame using knowledge of which slots are referenced between the current instruction and the end of the block. It supplies this information to the back end, which can then easily implement any one of four mechanisms: the two pointer model above, or a single pointer model (frame base, frame top, or stack). The single pointer models are derived from the two pointer model by realizing that:

- Both the frame's current size and its size at entry to the current basic block are known at code generation time.
- These frame sizes along with any single pointer specify the other two pointers.

2.4 Calling Convention

A second novel aspect of our virtual machine design is the implementation of the calling convention. PVM itself imposes no specific mechanism, but allows the back end to choose an appropriate mechanism for general procedure calls. The front end will generate PVM instructions for procedure calls that load argument values into registers or stack locations as specified by the back end. At procedure and closure labels, the back end is responsible for emitting code, if necessary, to move arguments from their placement by the caller to the location required by the callee. This is based on the number of arguments actually passed at runtime compared with the number of parameters required by the procedure.

In cases where the front end can analyze a procedure call sufficiently to avoid the general mechanism, it can produce optimized code by using simple labels (rather than procedure or closure labels) as the target address. Unlike a procedure label, a simple label implies no stack reformatting operations. Thus, the calling convention used for jumps to simple labels is essentially under the control of the front end, while for jumps to procedure or closure labels it is under the control of the back end.

The back end specifies, in particular:

1. Where arguments are passed, based on the number of arguments in the call. This is used by the front end to generate the code prior to a `JUMP` instruction. Our front end restricts the choices to combinations of registers and stack slots.

2. Which register contains the value returned by procedures.

3. Where the parameters are located after a `LABEL` instruction is executed. Since procedure and closure labels modify the stack and registers, the back end specifies where values are located after this reorganization has taken place. For a closure label, the back end also specifies the location of the pointer to the closure object so that the front end can generate PVM instructions to access its data slots. Our front end also restricts these to be registers or stack slots.

The back end also decides how the argument count is passed from the `JUMP` instruction to the destination procedure or closure label. This decision is internal to the back end since it is needed only to accomplish moving arguments from the locations where they are passed by the caller to the locations where they are expected by the destination code. All of this code is included in the back end's expansion of the `JUMP` and `LABEL` instructions.

2.5 First-Class Procedures

Since Scheme programs often make use of first-class procedures, we take a short digression to discuss the mechanism Gambit uses to implement them. In general, procedures carry with them the values of any free variables that they reference, and we use the traditional name, *closure*, to refer to the representation used when a procedure references one or more free variables. Procedures with no references to free variables can be represented simply by the address of the code that implements them: in PVM, either a primitive procedure object or a procedure label.

Gambit allocates closures on the heap. They consist of a back-end dependent header region (typically instructions executed when the closure is invoked) followed by storage for the values of the free variables needed by the procedure. Each entry in the storage area contains either the value of a free variable (if it is known to be immutable) or a pointer to the variable's storage location. Figure 4 shows the closure object created from the code shown in Figure 3 (see Appendix A for further implementation details).

The runtime storage allocation required by closures is expensive compared to other procedure representations, and Gambit attempts to minimize the number of closures that are created. The front end performs a combined data and control flow analysis to discover all procedure calls that invoke a given procedure. If all calls can be located, standard lambda lifting is performed; the net effect is to add the free variables to the parameter list of the procedure and to modify all of the procedure calls to pass these values as arguments. The procedure then has no free variable references and is represented as a procedure label.

A second technique used to minimize the size of closures, and possibly eliminate them entirely, is to subdivide the free variables that are referenced. References to global variables do not need to be stored in the closure since their values are directly accessible at runtime (Gambit supports only one top level environment). Similarly, variables that are known to have constant values (either because of a declaration or from data flow analysis) can be eliminated from the list of free variables that must be stored in the closure. Thus, the storage area of a closure contains values for the formal parameters of lexical parents, which are referenced by the body of the procedure, and which the compiler cannot infer to have constant values.

Closures are created by the `MAKE_CLOSURES` instruction. This instruction allows multiple closures to be made "simultaneously" to provide for mutually recursive procedures. Considering the creation of the closures to occur atomically

```
(define (make-adder x)
  (lambda (y) (+ y x)))
```

The following code (PVM and corresponding MC68000) is the body of `make-adder`. It is responsible for creating a closure to represent the value of the `lambda` expression:

```
; PVM -- LABEL(1,0,PROC,1)
L1:     bmi     L5
        jsr     20(a6)
; PVM -- MAKE_CLOSURES(stk(1),3,reg(1))
L5:     lea     -16(a3),a3     ; allocate
        move.l  a3,a2
        move.l  #0x10f8,(a2)+ ; length and type
        addq.l  #2,a2          ; unused word
        move.l  a2,-(sp)       ; store in stk(1)
        move.w  #0x4eb9,(a2)+ ; store JSR opcode
        lea     L3,a1
        move.l  a1,(a2)+       ; destination
        move.l  d1,(a2)+       ; data slot
; PVM -- COPY(stk(1),reg(1))
        move.l  (sp)+,d1
; PVM -- JUMP(reg(0),0)
        cmp.l   64(a5),a3      ; GC check
        bcc     L6
        jsr     32(a6)
L6:     jmp     (a0)
```

The following code is the body of the `lambda` expression:

```
; PVM -- LABEL(3,0,PROC,1,CLOSED)
L3:     move.l  (sp)+,d4       ; reg(4) <--
        subq.l  #6,d4          ;   closure pointer
        move.w  d0,d0          ; arity test
        bmi     L7
        jsr     24(a6)
; PVM -- COPY(mem(reg(4),6),reg(5))
L7:     move.l  d4,a1          ; load x from
        move.l  6(a1),d5       ;   data slot
; PVM -- APPLY(##fixnum.+,reg(1),reg(5),reg(1))
        add.l   d5,d1
; PVM -- JUMP(reg(0),0)
        jmp     (a0)
```

Figure 3: `Make-adder`: A closure generator

length and type	
unused	JSR
`code address`	
slot 1: value of `x`	

Figure 4: Closure for `make-adder` (for the MC68000)

(with respect, in particular, to garbage collection) allows for efficient implementation in some back ends. To make this more concrete, consider the Scheme program `make-adder` shown, with its PVM code, in Figure 3. The PVM code for the body of `make-adder` includes the instruction

`MAKE_CLOSURES(stk(1),3,reg(1))`

which creates one closure object (shown in Figure 4) and stores it on the stack at `stk(1)`. The closure contains space for one value, initialized from the contents of `reg(1)` (where the value of `x` happens to be), and the closure label for the body of the `lambda` expression is label 3.

The second form of the `JUMP` instruction is used for calling procedures and specifies the number of arguments being passed. The back end is responsible for emitting code that stores this argument count and arrives at the appropriate destination address. In the case of a closure, the destination is encoded in the closure object itself in a back-end dependent manner by the `MAKE_CLOSURES` instruction. Thus, the back end must arrange for a jump to a closure to be indirect, whereas a jump to a simple procedure is direct. Furthermore, the address of the closure itself must be made available to the code at the closure label, since it is needed to reference the values of the free variables stored in the closure.

While PVM does not further specify the interface between the `JUMP` instruction and the destination `LABEL`, all of our back ends have made the same implementation decision. As shown in Figure 4, the header of our closure objects is a short instruction sequence that jumps to the destination label and stores the address of the closure's data area into a known register using the target machine's jump-and-link instruction (`JSR` on the MC68000).

2.6 Declarations

Like most other Scheme implementations, Gambit provides a declaration mechanism that allows programmers to tell the compiler that it may violate certain assumptions of the basic language. For example, in Gambit, the declaration `standard-bindings` allows the compiler to assume that references to global variables with the names of the primitive operations are, in fact, references to those primitives. This allows the front end to generate an `APPLY`, `COND`, or `JUMP` instruction that references the primitive directly rather than referencing a global variable as required by the language definition. Similarly, the `fixnum` declaration allows the compiler to generate code for the standard numeric operations that assumes all numbers are small integers and suppresses overflow detection.

Some of these declarations, like `standard-bindings`, are relevant only to the front end, and are available with all back ends. Other declarations, like `fixnum`, are meaningful to only some back ends. In Gambit, we permit the back end to affect the code emitted by the front end based on the current set of declarations as maintained by the front end. For example, the primitive `+` might be usable in an `APPLY` instruction if either the declaration `fixnum` or `flonum` is in effect. In this case, the front end asks the back end what primitive could be used instead of `+`, specifying the declarations that are currently in effect. The back end responds with either `##flonum.+` or `##fixnum.+` (or simply `+` if no other operation is available).

2.7 Parallelism in PVM

We have introduced, so far, the sequential subset of PVM. One of our major goals, however, is to efficiently support the `future` mechanism for parallel computing. In this mechanism, a parent spawns a child task and uses a placeholder[14] to allow the parent task to refer to the value being computed by the child. In earlier systems supporting `futures` ([8], [9], [13]) there is a major cost associated with spawning a task, arising from the need to create a separate thread of control and a placeholder at the time the child task is spawned. PVM has three additional instructions and one operand type to make `future`-based parallel computation efficient. Our model is inspired by conversations with Halstead based on a brief mention in [9].

`LABEL(`t`, `*size*`, TASK, `w`)`
: Define a **task label**, t that marks the beginning of a task. A task label can be used in place of a simple label. A jump to a task label, however, spawns a new (possibly parallel) task to execute the code between the task label and its corresponding `DONE` instruction. The label w is where the parent task continues execution after the new task is spawned.

`LABEL(`w`, `*size*`, WORK)`
: Define a **work label**, w, that specifies where a task should resume execution after it spawns a new task.

`DONE`
: End the current task and deliver the result.

These three instructions can be translated by the back end to provide the same `future` mechanism used by earlier systems or to provide **lazy futures**. Lazy futures treat task spawning as a special kind of procedure call. When a task is "called" it leaves a marker on the stack so that another processor can recreate the parent task (in PVM, this is performed by the task label). The processor is now effectively executing on behalf of the child task and the parent task is suspended.

Should another processor decide to resume the parent task, that processor splits the stack at the marker, allocates a placeholder and begins executing the code of the parent task, using the placeholder to represent the value computed by the child task. PVM provides no direct support for this operation. Instead, a procedure is supplied by the runtime system that understands the format of stack markers and the code supporting task termination.

When control in the child task returns to the stack marker created by a task label, the child will either return as a normal procedure (if no other processor resumed the parent task) or store its result in the placeholder and look for some other processor's parent task to resume. In PVM, this occurs when a `DONE` instruction is executed.

In addition to this support for spawning and terminating tasks, PVM provides support for the underlying placeholder data type through the use of the ? operand annotation. When Gambit compiles code for a parallel back end it places a ? around appropriate operands that are potentially placeholders. "Appropriate operands" includes the strict operands (as specified by the back end) in `APPLY` and `COND` instructions, as well as the destination operand of `JUMP` instructions. By using information supplied by the back end, the front end can determine whether the result of a primitive procedure can be a placeholder; this information is then used to suppress generation of the ? in references to the value.

3 Optimization Techniques

Gambit employs a number of standard optimizations, in both the front and back ends. This section enumerates the current set of optimizations (without further discussion) primarily for completeness. We expect to add additional optimizations in the future.

3.1 Front End Optimizations

- Preferentially allocating temporary values to registers.
- Using a direct `JUMP` to a simple label for calling known procedures.
- Tracking multiple homes for variables.
- Keeping values in registers as long as possible by tracking register contents and saving them on the stack 'lazily.' This entails merging variable home information around conditional branches.
- Lambda-lifting.

3.2 Optimizing the PVM Code

These optimizations are performed on the PVM code itself, and are completely independent of both the source language and the target machine.

- Branch cascade removal by replacing a branch with the instruction at the destination.
- Reordering basic blocks to maximize the number of fall-throughs.
- Dead code removal.
- Common code elimination.

3.3 Back End Optimizations

In addition to the traditional back end optimizations (e.g. branch tensioning), Gambit makes use of its stack discipline abstraction to optimize the allocation and deallocation of stack frames. It is easy for the front end to use `stk` operands in an exclusively stack-like manner (i.e. it only stores into slots into which it has already stored or into the next higher slot). The front end does this, and consequently on machines with "push and pop" instructions (like the MC68000), the back end incrementally allocates the frame by pushing values on the stack as the slot number in `stk` operands increases. Similarly, the frame is incrementally deallocated by popping values when the frame size decreases.

On machines lacking these instructions, such as the MIPS and HPPA, the back end uses a frame top pointer implementation. This allows the frame to be allocated or deallocated with a single instruction at the end of each basic block (conveniently filling the branch delay slot on these machines).

4 Other approaches

We have examined three kinds of virtual machines typically used in implementing Lisp systems: byte codes, syntax trees, and register transfer languages. PVM belongs to this last class, and represents a particular engineering approach to the design of such an intermediate language. This section compares PVM to other virtual machines used by the Scheme community.

4.1 Byte Codes

There are a number of well known Scheme implementations based on a byte code interpreter: Indiana University's Scheme 311 (and its descendants MacScheme and Texas Instruments PC Scheme) and Halstead's Multilisp. In the interpreted systems for which they were developed, byte coding provides two important features: speed of dispatch on most hardware platforms, and code space compression (if the opcodes are based on static instruction frequency statistics).

As an intermediate representation for compilation, however, byte code leaves much to be desired. First, all of the byte code systems mentioned above are based on a pure stack machine. Since many important hardware platforms are not stack based, the process of compiling native code from the byte code requires recovering the higher-level information about intermediate values that was removed in generating the byte code. Furthermore, the creation of the byte code program does not produce information about variable referencing patterns, and this is essential to permit efficient use of hardware registers in the equivalent compiled code.

4.2 Code Trees

In interpreted Scheme systems that implement much of the system code in Scheme, byte coding is problematic since the byte coded programs have no natural representation within Scheme itself (aside, of course, from byte strings). An appealing alternative is to represent a program as a syntax tree, whose components are very similar to the pairs and vectors of standard Scheme. This approach is taken in MIT's CScheme scode[12] (derived from the actual instruction set of the Scheme '79 VLSI chip[10]) and the University Massachusetts Boston UMB Scheme system[4]. The type code of each node in the tree is derived from the syntactic expression (special form or combination) it represents in the program. The leaves of the tree are constants and variable references.

This representation is easier to deal with in a Scheme program than the byte codes, and faster to interpret than the original list structure of the program as provided by `read`. One of the major advantages of a code tree representation is that it can be easily converted back into a program equivalent to the original Scheme program from which it was derived. Systems can, and do, use this equivalence for a variety of debugging tools such as code inspectors and pretty printers. This very fact, however, argues against the syntax tree as a good intermediate code for compilation: the representation provides no information about commonly referenced variables, nor any results of data or control flow analysis.

4.3 Register Transfer Languages

Neither of the earlier representations were envisioned as intermediate representations for compilation, and so it is not surprising that they serve this purpose rather poorly. We are familiar with three intermediate languages designed specifically for this purpose, however: MIT's RTL (register transfer language), LeLisp's LLM3[5], and PSL's c-macros[7].

MIT's RTL is an *ad hoc* language evolved from the machine description language in [1], through a version used in an early compiler[17], and now part of the Liar compiler. Presently, RTL consists primarily of `ASSIGN` commands similar to the `COPY` and `APPLY` commands of PVM, `TEST` commands similar to the `COND` of PVM, special purpose instructions to call compiler support routines in the runtime system, frame adjust commands, and commands to generate procedure headers.

Internally, Liar (like Gambit) has a back end module that provides a description of the target machine to the front end of the compiler. Liar's front end is responsible for more compilation decisions than Gambit's, and consequently the description is at a much lower level of detail. It consists of information about the addressing granularity of the machine, the number of bits used for type codes and data, and mappings from the front end's special purpose registers to the target machine's physical registers. The front end of Liar relies on and directly manipulates four virtual machine registers: the dynamic link register (for return addresses), a stack pointer, free pointer into the heap, and a pointer to a set of memory locations (`C` variables) shared with the interpreter. In addition, the front end supports the notion of register sets by providing the back end with general purpose procedures for allocating, deallocating, and liveness tracking for groups of registers specified by the back end. This is used, for example, to allow the back end to separate the use of general purpose and floating point (co-processor) registers.

The primary interface between the front and back ends of Liar is through a rule-based language. The front end generates RTL instructions that are matched against the rules provided by the back end. This permits the precise nature of Liar's virtual machine to remain undefined while still enabling a variety of back ends to be written. Unfortunately, as the front end changes, the RTL instructions it emits change and the rule sets of each back end must be examined and modified individually. PVM's regular structure, on the other hand, allows the construction of a back end that handles the complete virtual machine and is thus isolated from many changes to Gambit's front end. PVM implementations (i.e. back ends for Gambit) have similar structures, since they are case dispatches on the PVM instruction and operand types; thus, updating a back end to accommodate changes in PVM itself is straightforward.

The register transfer language LLM3, developed at INRIA for the language LeLisp, is a much larger language than either PVM or RTL. It has over 100 instructions (including a number of redundant ones), providing control over aspects as diverse as garbage collection and file system operations. Implementing such a machine is a major undertaking and not suitable for our environment where a quick port to a new architecture or operating system is essential. Furthermore, the low level of control specified by LLM3 requires the front end to be more elaborate than we would like, and leaves little room for optimizations by the back end.

5 Current Status

At the time of writing, we have completed the Gambit implementation for the MC68000 and are in the final debugging stages of a port to the MIPS machines. A port to the HPPA is also nearing completion. Preliminary performance figures comparing Gambit to T's Orbit and MIT's Liar compiler are shown in Figure 5. As that table indicates, the MC68000 implementation achieves very good performance over a wide range of benchmarks. This implementation also includes an option for efficiently gathering dynamic usage statistics, as discussed in Appendix B.

In addition, we have a preliminary version of a Scheme to C compiler, inspired by the work of Bartlett[2]. This back end generates portable C code with good performance characteristics, but is not yet capable of producing separately compilable modules. As a result, it can currently only be used for compiling rather small Scheme programs.

6 Future Plans

Our next major goal is to create a back end for a stock MIMD parallel machine. We have made several early prototype versions and are encouraged by the results. Gambit's control and data flow analysis appear to be sufficiently general to allow us to explore a number of mechanisms for reducing the cost of the `touch` and `future` operations that dominate the performance of our own and other parallel Scheme systems[8][9][13].

As part of this work, we plan to complete our work on the Gambit C back end. This involves the implementation of a separate compilation facility that has already been designed. Measurements on the single module system indicate that the performance is about half that of the native code produced by Gambit, and we consider the advantage of having a single back end that supports a number of hosts to outweigh this performance degradation. The separate compilation design, however, has a number of areas in which performance may degrade and we plan to examine these in detail.

Finally, some very preliminary results indicate that it may be interesting to consider compiling imperative languages such as C and Pascal into PVM. We are particularly interested in combining a C to PVM compiler with the PVM to C back end. An early experiment indicated that PVM's optimization of procedure calls generated C code which, when compiled by a C compiler, outperformed the equivalent hand-coded C program compiled by that compiler! We plan to see whether this holds up under closer investigation.

7 Acknowledgements

The authors would like to thank the other Scheme implementors who have helped us understand both their own systems and Gambit: David Kranz, Chris Hanson, Bill Rozas, Joel Bartlett, and Will Clinger. Bert Halstead also contributed useful ideas and comments with respect to the parallel implementation of Gambit. We are especially grateful to Chris Hanson and Bill Rozas of the MIT Scheme Team for their help in comparing the code from various compilers as well as their help in our efforts to port CScheme to the MIPS. Their efforts allowed us to gather the performance figures included in Figure 5.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] Joel Bartlett. `Scheme->C` a portable Scheme-to-C compiler. Technical Report 89/1, Digital Equipment Corp. Western Research Lab., 1989.

[3] John Batali, Edmund Goodhue, Chris Hanson, Howie Shrobe, Richard M. Stallman, and Gerald Jay Sussman. The Scheme-81 architecture – system and chip. In Paul Penfield Jr., editor, *Proc. of the MIT Conference on Advanced Research in VLSI*, Dedham, Mass., 1982. Artech House.

[4] William Campbell. A C interpreter for Scheme — an exercise in object-oriented design. Submitted to the Software Engineering journal of the British Computer Society, 1989.

[5] Jérôme Chailloux. La machine LLM3. Rapport interne du projet vlsi, INRIA, May 1984. Corresponds to LeLisp version 15.

[6] Richard P. Gabriel. *Performance and Evaluation of LISP Systems.* Research Reports and Notes, Computer Systems Series. MIT Press, Cambridge, MA, 1985.

[7] Martin L. Griss and Anthony C. Hearn. A portable LISP compiler. *Software Practice and Experience*, 11:541–605, 1981.

[8] R. Halstead. Multilisp: A language for concurrent symbolic computation. In *ACM Trans. on Prog. Languages and Systems*, pages 501–538, October 1985.

[9] Robert H. Halstead, David A. Kranz, and Eric Mohr. Mul-T: A high-performance parallel Lisp. In *SIGPLAN 89 Symposium on Porgramming Language Design and Implementation*, June 1989.

[10] Jack Holloway, Guy Lewis Steele Jr., Gerald Jay Sussman, and Alan Bell. The Scheme–79 chip. Technical Report AI Memo 559, Mass. Inst. of Technology, Artificial Intelligence Laboratory, 1980.

[11] D. A. Kranz et al. Orbit: An optimizing compiler for Scheme. In *Symposium on Compiler Construction*, pages 219–233. ACM SIGPLAN, June 1986.

[12] Mass. Inst. of Technology, Cambridge, MA. *MIT Scheme Reference, Scheme Release 7*, 1988.

[13] James Miller. *MultiScheme: A Parallel Processing System Based on MIT Scheme.* PhD thesis, Mass. Inst. of Technology, August 1987. Available as MIT LCS/TR/402.

[14] James Miller. Implementing a Scheme-based parallel processing system. *International Journal of Parallel Processing*, 17(5), October 1988.

[15] James Miller and Christopher Hanson. *IEEE Draft Standard for the Programming Language Scheme.* IEEE. forthcoming.

[16] James Miller and Guillermo Rozas. Free variables and first-class environments. *Journal of Lisp and Symbolic Computation*, to appear.

[17] Guillermo Rozas. Liar, an Algol-like compiler for Scheme. Bachelor's thesis, Mass. Inst. of Technology, 1984.

[18] Guy Lewis Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, Mass. Inst. of Technology, 1978.

A Performance Measurements and Code Comparison

A detailed analysis and explanation of Gambit's performance is beyond the scope of this paper. This appendix provides a brief sketch of our performance results and compares the code generated by Gambit to that of MIT's Liar compiler and T's Orbit compiler. Figure 5 shows the results of running Clinger's version of the Gabriel benchmark programs[6]. For the most part the figures need no explanation. The following points may help the reader to better interpret them:

Motorola MC68000

	Gambit	Orbit	Liar	cc
boyer	4.74	×2.09	×1.96	
browse	1.04	×1.65	×6.12	
cpstak	.56	×1.93	×1.57	
dderiv	2.29	×2.04	×1.99	
deriv	2.10	×1.76	×1.82	
destruct	.65	×1.49	×3.71	
div-iter	.41	×2.29	×1.39	
div-rec	.58	×2.14	×1.57	
puzzle	3.51	×1.39	×3.59	× .61
tak	.14	×1.30	×3.15	×1.71
takl	1.18	×1.53	×1.86	×1.80
traverse	32.01	× .80	×1.68	
triangle	67.83	× .89	×1.96	× .83

MIPS R2000

	Gambit	Orbit	Liar	cc	scc
tak	.06	×1.17	×2.83	×1.17	×2.00
takl	.28	×1.14	×2.75	×1.79	×2.36
triangle	15.41	×1.02	×2.60	× .85	×1.77

Note: Timings for Gambit are absolute, in seconds. All others times are relative to Gambit.

Figure 5: Performance Comparison

1. The measurements for the MC68000 family were taken on a Hewlett-Packard 9000/340 system with a 16Mhz MC68020 CPU, 16 megabytes of memory, and a local disk. The measurements are based on the HP/UX time functions which deliver an estimate of user CPU time in units of 20 milliseconds. System time and time for garbage collection (if any) are *not* included in these numbers. All measurements were taken in full (multi-user) operating mode, but with only a single user logged in.

2. The measurements for the MIPS R2000 CPU were taken on a Digital Equipment Corporation DECstation 3100 with 16 megabytes of memory and medium-speed local disks, running under a preliminary release of the CMU Mach 2.1 operating system. Again, measurements are based on the Mach timing functions, omit system and garbage collection time, and were taken under multi-user conditions.

3. The column labeled "scc" contains timings from Joel Bartlett's Scheme to C compiler [2] of August 25, 1989. The column labeled "cc" contains measurements for some of the benchmarks that were hand coded in C and compiled (with the –O switch for optimization) using the vendor-supplied C compiler, "cc."

4. All of the benchmarks are executed five times and the mean is reported. In our experience, the measured times are repeatable to within a few percent.

5. All benchmarks were run as supplied by Clinger but with two differences. On all systems, a (compiler-dependent) declaration was supplied that caused arithmetic to be performed in fixnum mode (exact integers with no overflow detection) only. In addition, each benchmark was compiled both as written and enclosed in a `let` expression to allow each compiler to take advantage of any flow analysis it performs. The *best* timings for a given compiler are recorded here. We are unable to find a consistent pattern to explain which form of the program will perform better for a given compiler.

6. A number of procedures are used routinely by the benchmarks and their performance can dominate the performance of the entire benchmark. This is particularly noticeable in the case of the `get` and `put` operations in the Boyer benchmark. In order to compensate for this we wrote specialized version of the procedures `symbol->string`, `gensym`, `get` and `put`. While the details of the code are system dependent (since they require non-standard procedures) the algorithms used are the same on all systems.

We now turn to a more detailed look at the actual code produced by the three compilers. Figure 6 shows the results of compiling (for the MC68000) the following Scheme program by Gambit (version 1.3), Orbit (version 3.1), and Liar (Beta release of 7.0):

```
(define (reverse-map f l)
  (define (loop l x)
    (if (pair? l)
        (loop (cdr l) (cons (f (car l)) x))
        '()))
  (loop l '()))
```

The code in Figure 6 has been slightly modified for presentation purposes. We have converted the instruction sequences from each system's private assembler notation into "standard" Motorola syntax. In addition, the code from all three compilers actually includes interspersed words used by the garbage collector and interrupt handlers. These have been removed to make the code easier to read. They do not affect performance, since they are not executed in the usual case.

We do not pretend to have undertaken a detailed study of the code from these three compilers. However, from examination of a number of programs and after discussions with several of the implementors (David Kranz for Orbit, Chris Hanson and Guillermo Rozas for Liar) we can supply the following observations that account for a large part of the differences in output code. These comments apply to Gambit used with the MC68000 and MIPS back ends:

Object Representation

Gambit use the three low bits of a data item for the type tag, with 0 representing fixnums and other type

Gambit

```
; L1 is entry point
; on entry:
;  d1=F, d2=L, a0=CONT (return adr)

L1:
  beq    L8         ; arg count = 2 ?
  jsr    20(a6)     ; arity error
L8:
  move.l d6,d3      ; X <-- ()
  btst   d2,d7      ; (pair? L)
  bne    L6
  bra    L5         ; jump to LOOP
L7:
  move.l d1,-(a3)   ; ARG3 <--
  move.l a3,d1      ;  (cons RESULT X)
  move.l (sp)+,-(a3)
  move.l d1,d3
  move.l (sp)+,a1   ; ARG2 <-- (cdr L)
  move.l -(a1),d1
  move.l d1,d2
  move.l (sp)+,d1   ; ARG1 <-- F
  move.l (sp)+,a0   ; restore CONT
  cmp.l  64(a5),a3  ; heap overflow?
  bcc    L9
  jsr    32(a6)     ; heap overflow
L9:
  btst   d2,d7      ; (pair? L)
  bne    L6
L5:
  move.l a0,-(sp)   ; save CONT
  move.l d1,-(sp)   ; save F
  move.l d2,-(sp)   ; save L
  move.l d3,-(sp)   ; save X
  move.l d2,a1      ; ARG1 <-- (car L)
  move.l (a1),d1
  lea    L7,a0      ; setup CONT
  move.l 8(sp),a1   ; jump to F
  moveq  #-1,d0     ;  with 1 arg
  jmp    (a1)
L6:
  move.l d3,d1      ; RESULT <-- X
  jmp    (a0)       ; return
```

Orbit

```
; D15 is entry point
; on entry:
;  a1=F, a2=L, (sp)=CONT (return adr)

D15:
  move.l a1,-(sp)   ; save F
  move.l #65649,-(sp)
  move.l a2,a1      ; ARG1 <-- L
  move.l d7,a2      ; ARG2 <-- ()
  bra    LOOP_12    ; jump to LOOP
C_27:
  move.l a2,a1      ; RESULT <-- X
  lea    8(sp),sp   ; deallocate frame
  moveq  #-2,d5     ; return with 1 arg
  move.l (sp),a5
  jmp    (a5)
C_38:
  cmp.l  d7,a1      ; (null? L)
  bne    C_15
  bra    C_27
C_15:
  move.l a2,-(sp)   ; save X
  move.l a1,-(sp)   ; save L
  pea    D20        ; setup CONT
  move.l 1(a1),a1   ; ARG1 <-- (car L)
  move.l 16(sp),a0  ; jump to F
  moveq  #2,d5      ;  with 1 arg
  jmp    *21(d7)
D20:
  jsr    *9(d7)     ; TEMP <--
  move.l a1,1(a4)   ;  (cons RESULT X)
  move.l 8(sp),-3(a4)
  move.l 4(sp),a0   ; ARG1 <-- (cdr L)
  move.l -3(a0),a1
  move.l a4,a2      ; ARG2 <-- TEMP
  lea    12(sp),sp  ; deallocate frame
LOOP_12:
  move.l a1,d6      ; (list? L)
  and.b  #3,d6
  cmp.b  #3,d6
  bne    C_27
  bra    C_38
```

Liar

```
; reverse_map is entry point
; on entry:
;  4(sp)=F, (sp)=L, 8(sp)=CONT (return adr)

reverse_map:
  cmp.l  (a6),a5        ; interrupt check
  bge    interrupt
  clr.l  -(a7)          ; ARG2 <-- ()
  move.l 8(a7),-(a7)    ; ARG1 <-- L
  bra    loop_5
loop_5:
  cmp.l  (a6),a5        ; interrupt check
  bge    interrupt
  move.l (a7),d0        ; (pair? L)
  bfextu d0{0:6},d1
  cmp.b  #1,d1
  beq    label_14
  move.l 4(a7),8(a6)    ; RESULT <-- X
  lea    16(a7),a7      ; deallocate frame
  and.l  d7,(a7)        ; return
  rts
label_14:
  pea    continuation_2 ; setup CONT
  or.b   #-0x60,(a7)
  and.l  d7,d0          ; ARG1 <-- (car L)
  move.l d0,a0
  move.l (a0),-(a7)
  move.l 16(a7),-(a7)   ; jump to F
  jmp    0x68(a6)       ;  with 1 arg
continuation_2:
  cmp.l  (a6),a5        ; interrupt check
  bge    interrupt
  move.l a5,d0          ; ARG2 <--
  or.l   #0x4000000,d0  ;  (cons RESULT X)
  move.l 8(a6),(a5)+
  move.l 4(a7),(a5)+
  move.l d0,4(a7)
  move.l (a7),d0        ; ARG1 <-- (cdr L)
  and.l  d7,d0
  move.l d0,a0
  move.l 4(a0),(a7)
  bra    loop_5         ; jump to LOOP
```

Figure 6: Code Comparison

tags chosen to optimize references to the `car` and `cdr` of pairs and direct jumps to procedures. Orbit uses the two low bits for the type tag, and also chooses 0 for fixnums. Liar uses the top six bits for a type tag, with 0 representing `#F`. Orbit and Liar use a single object to represent the empty list and `#F`. Gambit distinguishes between these two objects and only `#F` counts as false.

Free Pointer Alignment
Gambit keeps the free pointer octa-byte aligned at all times, potentially wasting space when large numbers of small objects are created. Orbit and Liar maintain only quad-byte alignment.

Consing
Gambit performs consing by in-line code expansion, as does Liar. Orbit performs this with a call to an external procedure in order to allow GC checking to be done when the allocation occurs.

GC Detection
Gambit detects the need for garbage collection by performing a test at the end of any basic block in which allocation occurs. Orbit places this test in the code that performs the allocation itself, while Liar tests at both the entry to a procedure and the entry to every continuation point. Gambit's garbage collector is not yet fully functional on all of the back ends. The code and measurements reflect the full cost of detecting the need for GC, but none of the benchmarks actually invoked the garbage collector.

Interrupt Testing
Gambit does not test for interrupts, since it assumes a stand-alone environment rather than a program development environment. This will be changed for the parallel implementations in order to allow timer interrupts to produce a fair scheduler. Liar combines the garbage collection and interrupt check into a single short code sequence executed at the start of every procedure and continuation. We do not know how this is handled in Orbit.

"Unknown" Procedure Call
When calling a procedure that can't be identified at compile time, Gambit loads 3 arguments, the continuation and the argument count into registers (the stack is used to hold the other arguments if there are any). Any procedure that may be called in this manner will begin with a procedure label and the code will compare the number of arguments passed with the number of parameters expected and will move the arguments or trap as appropriate. Liar passes the arguments and the continuation on the stack. It uses an elaborate mechanism that distinguishes calls to procedures named by variables at the top level of a compilation unit from other procedure calls, and the interpreter supplies a number of trampolines that are used to combine a link-time arity test with runtime argument motion. An explanation of one part of this mechanism can be found in [16]. Orbit passes the arguments in registers and the continuation on the stack. A mechanism similar to, but somewhat simpler than, that of Liar is used for arity checking.

T Compatibility
Orbit is actually the compiler for a distinct language, T, that is closely related to Scheme. All of the benchmarks were run in Scheme compatibility mode, whose performance cost is not clearly understood. We rewrote `tak` and `takl` in T and compared the actual code and found no differences between the native T version and the Scheme compatibility mode version.

There was one very noticeable cost in the T implementation that is not shared by Gambit or Liar. This is the coding of the primitive procedure `pair?`. Orbit's two bit type tags do not distinguish pairs from the empty list (so as to optimize T's `list?` operation). Thus `pair?` is expensive with Orbit when compared to the other compilers.

B PVM Usage Statistics

The MC68000 back end allows programs to be compiled in a way that gathers measurements of dynamic usage of each of the PVM instructions and the types of operands used. This information can be used for performance analysis, and has been used to allow us to choose what parts of the actual implementation of PVM deserve careful optimization. The mechanism is both simple and very efficient: as each basic block is constructed, the front end counts the number of each kind of PVM instruction and operand class used in the basic block. The back end creates a counter for each basic block and generates code to increment that counter when the block is entered at runtime. At the end of a run, these counters are used to recreate the statistics.

The resulting code runs 30 to 40% slower than unmeasured code, allowing sizable programs to be measured and analyzed. See Figure 7 for a synopsis of the dynamic measurements taken from running Gabriel's[6] version of the Boyer-Moore theorem prover benchmark.

Operand Usage Frequency

reg	stk	obj	lbl	glob	mem
66.4%	24.4%	4.5%	4.0%	0.4%	0.0%

Instruction Usage Frequency

COPY	LABEL	APPLY	COND	JUMP	MAKE_CLOSURES
42.9%	20.4%	16.1%	11.9%	8.5%	0.0%

Detailed Instruction Breakdown

LABEL		APPLY		COND	
simple:	61.2%	car:	45.0%	pair?:	38.8%
continuation:	28.1%	cdr:	25.4%	null?:	28.9%
procedure:	10.5%	cons:	11.1%	eq?:	13.0%

Figure 7: Dynamic Measurements for Boyer Benchmark

In gathering these measurements, we used a version of Boyer that is enclosed in a `let` expression. This accounts for the lack of global variable references. The procedures `put` and `get` (which dominate the performance of the benchmark) were implemented using `assq`.

There are only a few comments to be made on these results. First, label operands can appear either directly in a `JUMP` instruction or as a source operand to another instruction (for example, it may be stored into a local variable for later use). In this benchmark, labels appeared almost exclusively (99.9% of the time) in this latter context. The primary use of direct jumps is to branch around other arms of a conditional when the conditional isn't in the tail position of an expression. Most conditionals in Boyer appear in tail position; we don't know how common this is in general Scheme code.

The breakdown of the label code is interesting, since it shows the dynamic execution frequency of the various types of label. Recall that simple labels and continuations actually generate no code, so there is no runtime associated with their use. A procedure label, however, requires an arity check and may require moving values from argument locations to parameter locations.

The `APPLY` instruction is used by the front end to request open coding of a primitive that the back end supports. Figure 7 shows the breakdown by primitive procedure of these operations that occur when Boyer runs. The table shows only those open coded primitives that account for more than 10% of the run time, although the actual statistics contain numbers for all primitives. In fact, a good deal of detail has been omitted from all of these tables to make the presentation more tractable.

`COND` is used for all conditionals. In the case of Boyer, 80.7% of the predicates encountered were open coded versions of `pair?`, `null?`, or `eq?`. Of the remaining 19.3% of the predicates, 18.8% are not open coded and the remaining open coded predicates occur under 0.1% of the time.