A New Architecture for the Implementation of Scripting Languages

Adam Sah and Jon Blow
Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720
{asah, blojo}@cs.Berkeley.EDU

September 12, 1994

Abstract

Nearly all scripting languages today are implemented as interpreters written in C. We propose an alternate architecture where the language is translated into the dynamic language Scheme [R4RS]. The plethora of high quality, public domain Scheme implementations give the developer a wide selection of interpreters, byte compilers, and machine code compilers to use as targets for her VHLL.

Our VHLL, Rush, provides high-level features such as automatic type conversion and production rules [SHH86][Ston93]. Performance benchmarks show that our system runs with acceptable speed; in fact, Rush programs run much more quickly than their equivalents in languages such as Tcl and Perl4. Whereas those languages are coded in C, Rush takes advantage of Scheme's existing high-level features, saving development time. Since the features provided by Scheme are among those most VHLLs share, we expect this approach to be widely applicable.

1 Introduction

We propose that VHLLs be implemented as source-to-source translators emitting a high-level language such as Scheme [R4RS] and which use publicly available implementations of the target language to execute the program. In other words, we advocate implementing VHLLs that use Scheme as their assembly language. Though others have indeed suggested that Scheme has properties useful as a compiler representation language, such observations and proposals have rarely gone beyond speculation. We advocate the approach because we have tried it and it works.

We think of VHLLs as supporting some of these features:

- Automatic memory management.
- Comfortable syntax when interpreted from a command line.
- Primitive support for complex data structures.
- Easy translation of data to string form and back.

*supported in part by a grant from the National Science Foundation #NSF-IRI9107455

- A large and useful standard procedure library.
- Ability to define procedures at runtime.
- ...other "high level" features like variable traces.

Implementing any single high level feature is not difficult, but putting them together can be challenging. Most VHLLs in use today, such as Tcl and Perl, incur large performance penalties to provide their features. Scheme, on the other hand, provides many of these features with implementations that are fairly efficient and interact cleanly; by using Scheme as a back-end the VHLL designer can adopt desired features with minimal effort. This allows him to concentrate development time on the features that matter most, the ones specific to the VHLL being implemented, rather than reinventing old wheels.

We designed a VHLL called Rush and implemented it as a Scheme-generating compiler. This paper explains how we arrived at the decision to use Scheme as an intermediate form (section 2), lists some of the problems that we encountered using Scheme (section 3), and shows how some high level Rush features map elegantly into Scheme, where such a mapping would be far more difficult into other languages (section 4). In section 5 we present a small performance analysis of Rush, showing that it executes code far more quickly than similar languages such as Tcl and Perl4, even though our system wasn't hand-tuned. In section 6 we examine the source trees for Perl4, Tcl7.3, Rush, and SCM4d3 (a public domain Scheme interpreter) in an attempt to pinpoint the work we saved by using Scheme as our runtime environment.

2 Scheme as an Intermediate Representation

2.1 What is an IR?

An Intermediate Representation (IR) is a language that lies between the parsing module of a compiler and the target-code generator. Source code in the high level language is compiled into the IR, and then the IR code is either compiled into machine code or interpreted on the fly. The benefits of IRs are numerous. First, an IR makes the source language more portable, since the IR→machine code back-end can be retargeted without changing the front end.

As an example, the GNU C Compiler (gcc) converts C source code into an intermediate representation called RTL ("register transfer language"). RTL is a very low level representation, almost a generic assembly language. The compiler analyzes and optimizes the RTL code, then uses one of many back-end programs to convert RTL into machine-specific assembly language. To retarget gcc, one need only write a new RTL translator; to adapt gcc to compile a new language, one need only write a translator from that language to RTL.

A careful selection of the IR can make analysis and optimization easier; this is widely believed to be the case with three-address codes like Single Static Assignment [RZ88] [CFRWZ91] (for Pascal-like languages) and Continuation-Passing Style [Shiv88] [App92] for Lisp-like languages. As we discuss in sections 3 and 4, analysis can be very important when designing and implementing VHLLs; it is a fallacy to assume that perforance doesn't matter. The choice of IR directs how analysis will be performed, and to what degree existing implementations of the IR already perform the desired analyses.

Problems can arise from a bad choice of IR. A poor matchup between the source language and the IR will require the language implementor to expend a lot of effort bridging semantic gaps. For example, translating Scheme into C is frustrating because of the need to support call-with-current-continuation (call/cc) and tail-recursion elimination, both of which have unpleasant interactions with memory allocation and C's stack-based procedure frame model, as implemented in virtually every C compiler today[Bart89].

If the IR you select has already been implemented, you save the implementation time for the back end. As an example of this, it has become common practice to write compilers that emit C code which is then compiled into an executable, rather than emitting machine code directly. (The INTERCAL-C Compiler [WL73] is such a program.) At the very least this approach saves the language implementor from having to write register allocation algorithms, some of the most complex code in modern optimizing compilers.

2.2 An Introduction to Scheme

Throughout this paper we will discuss the Scheme output code of our compiler system. We cannot provide here a full description of Scheme; for that we refer the reader to [SICP] and [SF89]. However, to give a taste of the way Scheme works, we present a short description of the behavior of Scheme's procedures in comparison to C's. Those familiar with Scheme may skip this section with impunity.

Procedure calls in Scheme are written as nested lists; each list is a series of expressions, the first denoting the procedure to call and the rest listing the arguments. For example:

```
(foo 4 (bar 3))
```

calls the procedure `foo` with two arguments: `4` and the result of calling `bar` with `3`. This is equivalent to the C expression:

```
foo(4, bar(3))
```

Variables in Scheme can be defined with the keyword `define`. Procedures can be created with the constructor `lambda`, which takes as parameters a list of arguments and an expression to act as the procedure body. Here is the definition of a "square" function:

```
(define square (lambda (x) (* x x)))
```

This is equivalent to the C function:

```
int square(int x) { return x*x; }
```

Procedures in Scheme can be nested. Scheme is lexically scoped. By combining lexical scoping and use of `lambda` we can create objects with state in a simple way, like so:

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

This means that `make-adder` is a function of one argument `x` that, when called, returns another procedure (using "`lambda`") that takes one argument `y` and adds it to `x`. Here is an example of its use:

```
(define adder1 (make-adder 6))          ; adder1 is now a function
(adder1 5)                              ; returns 11

(define adder2 (make-adder 10))
(adder2 6)                              ; returns 16
(adder1 (adder2 10))                    ; returns 26
```

Each procedure has its own value of `x` which it remembers for as long as it exists. Such state values could even be modified. Using "`lambda`" this way is perfectly natural in Scheme.

2.3 Why Scheme?

Before espousing the benefits of Scheme, we should make it clear that Scheme is not a panacea; neither is it the only logical choice as a code generation language for VHLLs. Some discussion of other possible languages is presented at the end of this paper.

Scheme's structure is simple, consisting of command/argument lists enclosed in parentheses. This syntax has led many critics to complain that the language is unreadable; however, for machine-generated code readability is not so important, and syntactic simplicity is a boon since it means that the code-generation routines of the VHLL compiler can also be simple.

In the segments below we discuss our reasons for using Scheme; first we talk about the availability and versatility of Scheme implementations, then we discuss at length the semantic advantages that Scheme provides.

2.4 Current Scheme Implementations

Many Scheme implementations are available, both commercially (as Chez Scheme) and as freeware on the Internet (eg. SCM, Scheme→C, Bigloo, MIT-Scheme, schemelib, etc.). More importantly, these are not weekend hacks or student projects; they are well-documented, portable systems. This provides a wide selection of targets to choose from.

2.4.1 Scheme Speed

Scheme can be implemented efficiently [VP89] and this is proven in practice by the above systems. By comparison, using other high-level languages (such as, say, Tcl) might be a bad idea since such languages' semantics guarantee that they will be untowardly difficult to optimize.

Deciding which languages would make suitable back-ends requires one to set a goal about how fast the VHLL should be. In terms of Rush, we measured speed relative to C, deciding that Rush programs should run no more slowly than 10x-50x their C counterparts. In contrast, Tcl is a factor of 1000x-10000x away from C, and pathologic cases (such as array references) even have different computational complexities [Sah94]. In setting a speed goal, we accept that the VHLL we're designing could not compete with C or C++ for speed, but we also insist that it mustn't turn a DEC Alpha into an Atari 800.

2.4.2 Scheme Architecture

Basically all implementations of Scheme are centered around a read-eval-print loop: a routine that reads a string of text, evaluates it as Scheme, and translates the result into a string. The VHLL implementor can use this mechanism directly, parsing the code from her language into Scheme and feeding that Scheme to the read-eval-print routine.

The demo version of the Rush interpreter was a parser connected to a Scheme interpreter via a Unix pipe. This was conceptually dirty but it meant that we could concentrate on writing a correct yacc script to parse Rush and a code generation module to emit valid Scheme. This mechanism took a few hours to write, debug, and test; it's a negligible amount of code. It has since been discarded; now we link the Scheme interpreter as a shared library and call the read-eval-print loop as a procedure from C.

2.5 Advantageous Scheme Semantics

2.5.1 Scoping

Scheme is lexically scoped, which means that a variable is defined and accessible in a program only at the level of nesting where its definition is written, and in routines contained within that definition.

Lexical scoping is generally fast (compared to dynamic scoping, which is used in languages like Tcl and some older versions of Lisp) [ASU86].

2.5.2 First-class Procedures

We described first-class procedures in a previous section introducing Scheme. In Rush first class functions are used to provide many important features. More details about those features can be found in section 4.

2.5.3 Continuations

A continuation is a procedure that represents the of "the rest of the program"— what computations would follow from the current state of program execution. In Scheme, continuations are captured by using the procedure `call-with-current-continuation` (abbreviated `call/cc`). *call/cc* takes a procedure as an argument; it invokes that procedure, passing the current continuation as an argument. Implementationwise, continuations contain the place in the program that execution has reached as well as the state of all active environments (in other words, the values of just about every live variable, except perhaps for globals).

In practice, continuations can be used to implement coroutines, exception handling, and asynchronous event handling, as well as simple control flow (like "goto"). Cooperative multithreading becomes trivial with use of continuations; a context-switch is just a `call/cc` to the next thread on the run queue.

2.5.4 Powerful Macros

One commonly available Scheme macro system is Expansion Passing Style (XPS) [DFH86]. XPS is Turing-complete; it is more expressively powerful than the C preprocessor or even the m4 macro expander. This power gives the user an increased choice of where to compile a language construct. For example, one could write one's entire VHLL as a parser that desugars the VHLL code into parenthesized forms (a pidgin Scheme code), and then use the macro expansion facility to convert this into real Scheme code, with the macros performing all transformations and analyses. From experience, XPS is not hard to use in this way, though we did not use it so heavily for the Rush parser. Instead we emit terse code that is easy to read, using macros to expand forms that would otherwise obfuscate the code.

2.5.5 List-Handling Primitives and Varargs

Linked lists are a fundamental data structure in Scheme. Most common list-manipulation operations are provided as primitives. Some more powerful Scheme procedures take advantage of first-class functions; one example is `map`, which applies a function to every member of a list and returns the resulting list.

Scheme's varargs facility is more versatile than C's; when a varargs procedure is called, Scheme creates a list that contains the values of all the arguments passed, like so:

```
; varargs in scheme:
; myfun requires two arguments, but can take any number beyond that.

(define myfun (lambda (arg1 arg2 . varargs) (map square varargs))

(myfun 1 2 3 4 5)            ; returns (9 16 25)
```

Compare this ease-of-use to C where one must use special accessor macros to get arguments one at a time, knowing the types of each beforehand. An object-level facility has been written for C to make varargs easier; it's called AVCALL, but it isn't universally available [BT93]. Also, it is nearly impossible to use C's native varargs facility to provide varargs as a feature for a VHLL, so the programmer coding the interpreter in C must come up with her own facility.

2.5.6 Garbage Collection

Every Scheme implementation contains a garbage collector. This can be a great benefit. Almost every VHLL provides some form of automatic memory management so that the user does not have to concern himself with explicitly allocating and freeing memory. The authors have yet to hear the benefits of automatic memory management questioned by programmers who have used it in production settings. The single valid complaint is that auto memory management can be slow, but the 5-20% of runtime that it typically consumes for "fast" languages drops well into the noise when the slowness of current VHLLs is considered.

Automatic memory management can be effected in one of two ways. The first (and simplest) way is that the VHLL can be designed with invariants on its data structures such that memory can be reclaimed trivially. The second is that can allocate memory from a garbage collected heap. Actually, current research suggests a third alternative where analysis is used to eliminate the need for garbage collection, but this analyzer will be even more complicated to write than a collector, and is still unproven in practice [TT94].

The first (and worst) approach is used by most VHLLs being written in C; these VHLLs are generally seeking *some* form of memory management, but do not try to implement completely automatic management, since that is more difficult. Tcl, for example, limts its available data structures to those representable by strings; by not offering the notion of a data reference in the language, the Tcl interpreter can easily tell when a value will no longer be used, freeing its memory. The trouble with this approach is twofold. First, it limits the sorts of data structures the language can have and how they can be implemented. In the case of Tcl, this necessitates some copying of data as part of its pass-by-value calling convention. The second problem is that this approach requires the C programmer to maintain strange invariants during callouts from the VHLL. The same problem arises when using a garbage collector, although the selection of a C-compatible garbage collector such as Boehm-Weiser [BW88] might eliminate this need.

2.5.7 Smarter Arithmetic

Scheme makes a tradeoff of speed-for-features in its arithmetic library, but the features provided are very useful.

First, Scheme provides integers of unbounded size ("bignums"), so arithmetic overflow simply does not happen. Most Scheme implementations will cause integers to flow smoothly between several representations depending on their size, all invisibly to the user; an integer less than 2^{31} might be stored in the same form as a C integer, but a larger number would be stored in bignum format.

Scheme also provides precise notions of real and rational numbers, rather than C's floating-point approximations. Again, this has the effect of shielding the casual user from bizarre errors arising from simple math over- and under- flow.

The strength of Scheme's arithmetic handling can be useful in maintaining consistent language semantics. Here we have an example from Tcl 7.3, where evaluating the command:

```
[expr 1234567890*1234567890]
```

produces different results on different systems: on an Alpha running OSF/1 v2.1, the answer is 1524157875019052100, whereas on an Intel 486 running Linux, the answer is 304084036. This is because registers are 64 bits wide on an Alpha but only 32 bits wide on a 486. Perl4 seems to implement integer arithmetic using double-precision floating point, so "print 1524157875019052000-10" results in "1524157875019052000" (which, by the way, is Perl4's answer for the above multiplication.) Since VHLLs generally attempt to abstract the machine from the user, such behavior is in conflict with the spirit of the languages.

Detecting math overflow in C is nontrivial, as is writing the infinite precision math library. To give credit, the Free Software Foundation's "gmp" package is starting to fill this gap for floating point, and Perl4 has already implemented infinite precision integer arithmetic.

2.5.8 SLIB

Aubrey Jaffer's SLIB is a portable, R4RS compliant Scheme library that implements nearly every library feature that's present in VHLLs like Perl4 and Tcl. Without producing a white sheet for it, we will summarize a few of its features:

- String handling and formatting library.
- Data structures: collections, hash tables, queues and priority queues, trees, and records.
- Bitwise manipulation, list routines, random number generation, and sorting.

A third party pattern matching library is also available from `titan.cs.rice.edu:/public/wright/match.tar.Z`. As we will explain later, a C callout mechanism is assumed for our VHLL, so additional C libraries should be callable from Scheme. For example, making a Scheme binding for the Tk graphical user interface is reasonably straightforward (and in fact has already been done).

3 Some Downsides to Scheme

Scheme has several flaws from a VHLL standpoint that make it an imperfect code generation language. We believe that these do not outweigh the benefits provided. In this section, we detail the major flaws we encountered during the implementation of Rush.

3.1 C Callouts Made Hard

Providing a callout mechanism to C is important to many VHLL designs; it can be invaluable because of the enormous body of code written in C and the comparative difficulty of getting at this code through other means.

Unfortunately, Scheme has no standard facility for calling foreign functions. Sadly, neither does any other dynamic language we looked at that runs at reasonable speed, so this is a generic problem. Part of our current work is to specify and build this mechanism. It is not difficult to add this to any individual Scheme implementation; our goal would be to target this for a number of Schemes. The major problem is that C programs need to have standard ways to access and build Scheme data structures, and they must do it in such a way that the garbage collector is able to keep track.

The other reasonable option is to provide a string-based callout mechanism, as Tcl does. In that case the information you send to C is converted into a string and passed to C routines which translate the strings into native C structures. This incurs a tremendous performance penalty. (For Tcl 7.3 this is not a problem since all Tcl data is in string form; Tcl has already accepted the performance hit.) This string-passing method only works well for simple data structures; for example, creating a string representation of a closure is very hard [SG90]. In the current version of Rush, we have implemented a string-based callout mechanism, but it does not support closures.

3.2 'goto' Considered Useful

Whereas 'goto' has often been considered harmful, programmers typically demand it in limited forms; the most popular are constructs like "break", "return", "continue", and exception handlers. In Scheme the natural way to alter control flow like this is to create a continuation at the *destination* of the jump and then call that continuation at the source. (This works much like the C procedures `setjmp` and `longjmp`.) However, setting up a continuation can be very slow under some Scheme implementations. On an Alpha using Scheme→C, for example, we were measuring values of 80+ μsec (8000+ instructions) in typical cases. This is because Scheme→C stores call frames on the stack, so `call/cc` must copy the stack into the heap to save a continuation. On systems where `call/cc` is not so slow, call frames are allocated on the heap, causing more frequent garbage collections, slowing down the system in general.

Since we refused to use continuations to implement loop control in Rush, we devised a code manipulation that converts nonlocal gotos into return values from nested scopes. This necessitates a few tricks: First, we require that no code may follow a nonlocal goto statement in any single basic block. Since such code would be unreachable, it deserves a compiler warning anyway.

The second and more challenging trick is to properly capture the results of each code block that *could* result in a nonlocal goto. For example, here is some Rush code:

```
proc foo () {
   for {i=0} {i<100} {i++} {
      if {i>90} {
         break;
      }
      puts "hi"
   }
}
```

In this example, the "break" performs a "goto" to the end of the `for` loop body; in order to implement it, the compiler generates code of this form:

```
...
; then => 'break      else => 'blah
(let ((result (if (> i 90) 'break 'blah)))
  (case result
      ((break) 'break)
      ((else <puts "hi", increment i, recursively call loop>))
)
```

This results in straight-line code becoming nested when multiple statements in a block can perform nonlocal gotos. For example,

```
...
if {...} { break }
if {...} { break }
...
```

will emit two case statements, one nested within the other. The code generated for these cases quickly grows disgusting. The continuation solution, though slower, is simpler.

3.3 Balancing Parentheses

We stated earlier that Scheme's proliferation of parentheses and otherwise general lack of readability was not a big problem with machine-generated code: who wants to look at it anyway? Well, the truth is that it is an annoyance when you are trying to debug your VHLL-to-Scheme code generator. When working on Rush, we'd see the following Scheme emitted:

```
...))))))))))))
```

When these parens didn't balance with previous open parens, the Scheme interpreter would issue an error like "invalid number of arguments to *foo*", "unbalanced parentheses", "unexpected end of input" or "argument #x is not a *type*". Trying to track this from the above output can be difficult; it requires more sophisticated debugging and output. The solution is to make sure that the translator emits properly indented and commented Scheme code so that you can read it manually when debugging.

3.4 Garbage Collection and Callouts

High-quality garbage collectors interact with low-level data structures in the target language. For example, collectors generally need to know the internal data structure of each value in memory, in order to follow any pointers contained inside the structure. Providing a description for each structure is "a small matter of code", again detracting from time spent designing the VHLL. By comparison, any data structures built on top of Scheme's base structures are automatically collected properly and already have proper type tags.

Perhaps even better is the Boehm-Weiser collector, which can garbage collect arbitrary C programs with reasonable accuracy against memory leaks, and near-complete safety against corruption. The problem is that the Boehm-Weiser collector needs to scan through the entire heap to collect, because it attempts to treat *every* word in memory as a pointer to be followed, comparing that pointer against an internal table of valid data pointers. Leakage occurs when an non-pointer word accidentally corresponds to a valid heap object address, or when pointers are hidden (eg. the classic trick of XORing the "next" and "previous" pointers in a doubly-linked list to save one word per link). A worse problem than leakage is the BW collector's performance. It must run atomically; this can cause "hiccups" when used on a large heap. Boehm and Weiser suggest some ways around this, but the common solution of using an incremental collector is difficult to implement in this environment, since it requires significant cooperation from user code [BW88]. One can use virtual memory support instead, but this tends to be slow and less portable.

3.5 Lack of Sophisticated Optimizers

Machine-generated code that is not optimized typically contains many redundancies, common subexpressions, dead code, unnecessary typechecks, and so on. Ideally, to save effort, one would like to emit dumb Scheme code and have the Scheme backend perform most of the omptimizations. To do otherwise can be a hassle since complex optimizations are notoriously difficult to implement and test.

While the Scheme literature discusses many algorithms for optimizing Scheme code, and proofs have shown ways to apply many standard optimization algorithms to Scheme code [Shiv88], such facilities are far from common in Scheme implementations. Indeed, many Scheme implementations perform no optimizations at all, implying that the Scheme code presented to the compiler is literally what is output from that compiler, translated into machine code. The one exception is tail recursion elimination, mandated by R4RS, but this can be minor compared to other optimizations.

Using Scheme→C as an example target, we describe the problem faced by Rush in the above break-return-continue problem. In this case, the only way to avoid wrapping *every* statement with the trap for nonlocal gotos is to perform static analysis over the code to see what subcomputations can effect such a goto. While not difficult for the seasoned compiler optimization expert, this is rather painful in light of what should be a far easier task. In the above case, the flow equations are:

```
BREAKS(stmt)- does stmt break?

BREAKS(stmt) = case stmtType of
 IF:                        BREAKS(true-branch) v BREAKS(false-branch)
 FOR, WHILE, FOREACH:       FALSE
 BREAK:                     TRUE
 FUNCALL:                   BREAKS(arg1) v ... v BREAKS(argn)
 CONTINUE, RETURN, ... :    FALSE
```

with similar equations for each of "continue" and "return". Then, we only emit the trap if a statement could break, return, or continue. In the case of the v.1.0 of the Rush→Scheme compiler, we chose to emit a single style of case trap, so we combined the equations to check for `NONLOCAL_GOTO(stmt)`. The tradeoff is that for some cases, we produce nonoptimal case statements, as in the above example, where `'return` and `'continue` are not possible, and so can be removed from the possible goto results:

```
; a more efficient implementation of the previous example.
(case (if (> i 90) 'break)
 ((break) 'break)
 ((else (<puts "hi", increment i, recursively call loop>)))
)
```

We wish we were able to count on the Scheme compiler to detect dead code in a case statement (code bound to values that can never be produced by the predicate). Even better would be a more heroic optimizer that would reduce the above code into a single `if` statement:

```
(if (> i 90)  'break  (<puts "hi", incr i, recursive call>))
```

which may sound incredible but is well within the capabilities of better C compilers; both gcc v.2.5.8 and DEC native cc under OSF/1 v.2.1 were able to emit this reduced version for an equivalent piece of C code.

In general, the better a job the target language optimizer does, the less work you have to put into your translator to get good performance. This is as important a consideration as the final performance of your VHLL; if performance doesn't matter, neither does optimization. In the case of Rush, we claimed that performance counts inasmuch as scalability matters, but not so much that we want to try to match C. It's fine to not lose sleep about the efficiency of these case statements, but it's a bad idea to wrap every statement with such checks.

3.6 The Effectiveness of Optimization in Scheme

It is generally futile to rely on the C compiler to optimize the output of a Scheme→C translation. This is because unoptimized translator output will necessarily be tainted with false references to environments. In the loop example, our Scheme code might look like:

```
(define (foo)
 (let ((i 0))
   (letrec ((forloop (lambda ()
     (case (if (> i 90) 'break)
       ((break) 'break)
       (else (display "hi") (set! i (+ i 1)) (forloop))))))
   (forloop)))
)
```

Without analysis to determine that "i" doesn't escape the current context and isn't used later in its defining context, the Scheme implementation cannot convert this into the more efficient form:

```
(define (foo)
 (letrec ((forloop (lambda (i)
    (case (if (> i 90) 'break)
      ((break) 'break)
      (else (display "hi") (forloop (+ i 1)))))))
   (forloop 0))
)
```

In this version, "i" need not be stored in memory; it can reside in a register, leading to a performance improvement.

To test for this behavior, we passed the previous scheme code through Scheme→C and discovered that the output was more pathological than we'd thought:

```
TSCP t_foo()
{
  TSCP X2, X1;
  X1 = _TSCP(0);
  X1 = CONS(X1, EMPTYLIST);
L2040:
  if (LTE (_S2CINT (PAIR_CAR (X1)), _S2CINT (_TSCP (360))))  goto L2044;
  X2 = x2013;
  goto L2045;
L2044:
  X2 = FALSEVALUE;
L2045:
  if (EQ (_S2CINT(X2), _S2CINT(c2013))) goto L2042;
  scrt6_display(c2015, EMPTYLIST);
  X2 = _TSCP (PLUS ( _S2CINT (PAIR_CAR (X1)), _S2CINT ( _TSCP (4))));
  SETGEN (PAIR_CAR (X1), X2);
  GOBACK (L2040);
L2042:
  return (c2013);
}
```

In this code, X1 ("i" in the source) is represented by a pair, where a direct reference would do; the C compiler is unable to detect and eliminate this unnecessary level of indirection.

How does this affect the VHLL author? Consider that the above loop, when we exclude the display call, essentially counts from 0 to 90. The "fast" version took 85μsec on a 486dx2/50 running Linux; the slower version took 145μsec. On the Alpha, the times were 22μsec and 41μsec. Considering the number of such "minor" optimizations that apply to any given situation, the VHLL implementor cannot ignore them if he cares about the speed of his language. It is not sufficient to rely on the machine code translation process to decrease runtimes.

4 Rush as an Example Language

The Mariposa database project needed an extension language with the feel of Tcl but which also ran quickly (within a factor of 100 of the speed of C). Faced with a time table allowing us five months to go from design to demo, we were left with two options: either finish the existing Tcl Compiler

(TC) and add the features we needed, or build a new language from scratch. TC was too slow for our needs; trying to use it would have been far worse than embarking on a new project.

So far, the Rush project has been very successful. Not only did we produce a working interpreter and optimizing compiler, but at our release demo we operated “hands free”, letting users play with the system directly. What makes this exciting to us is that at no time have we had to concern ourselves with the way procedures are declared, invoked, or stored. We’ve never had to worry about the semantics of our memory manager or debugging its implementation; Rush inherited a generational copying collector from our chosen Scheme implementation. We have never seen the assembly output from the Rush compiler; we never needed to.

Here we will describe some of what we did with Rush. This section is divided into two parts. The first part talks about Rush rules (the most Very High Level feature of Rush) and “boxing”, the technique we used for implementing them. The second part discusses optimizing away the need for these boxes to increase language performance.

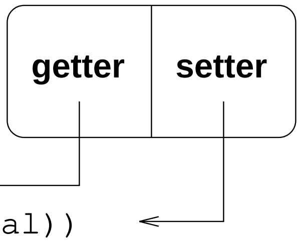
4.1 Boxing

To provide a more in-depth case study of what we found useful in Scheme, we present the design and implementation of “boxing” and how it works in Rush. From the beginning Rush was required to support commands of the form “on <event> do <action>”, where <event> is an arbitrary predicate and <action> is an arbitrary block of statements. These “production rules” are sort of like “if” statements that are always watching. Here is a sample rule:

```
on {x>3 && y<10} do {notify x y z}
```

The action `[notify x y z]` will be performed whenever the predicate `[x>3 && y<10]` becomes true *any time over the course of program execution.* For a more in-depth description of how these rules work and why they’re useful, see [SBD94]. The most intuitive way to implement rules is to monitor the changes in relevant variables. For this we chose a *boxing* mechanism.

Rush “boxes” are pairs of functions, a “getter” and a “setter”, which are called to retrieve values from and store values to a memory location. Rush semantics require that every variable behave like a box. For normal variables, the getter and setter access the location directly, like one would expect. Figure 1 shows a diagram of a getter/setter pair for a boxed variable “foo” that is doing nothing tricky.




```
(let ((val *initial-value*))
   (cons
      (lambda () val)
      (lambda (newval) (set! val newval))
))
```

Figure 1: A Rush variable (the box is a Scheme “pair” of pointers.)

If we wish to create a read-only variable, we just modify the setter of its box so that instead of changing the value, the setter generates an error. This configuration is shown in figure 2.
Likewise we can easily imagine a “read trap” as a modified getter function.

These boxes are easy to create and manipulate because of Scheme’s first-class functions and automatic memory management.

```
(let ((val *readonly-value*))
   (cons (lambda () val)
         (lambda (newval) (throw-error))
   )
```

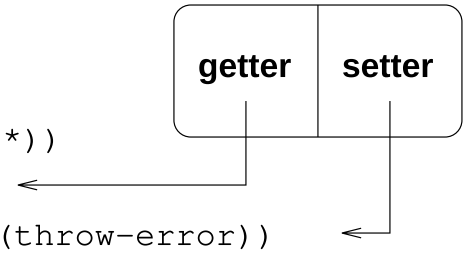


Figure 2: A read-only variable in Rush. The overall function still returns a box. When its getter is called, it errors; when its setter is called, the box is rewritten to be a normal variable.

We also use boxes to implement rules. A rule is represented by a set of functions which are added to the setters of relevant boxes. Since the rules are hidden inside boxes, they are never seen unless the boxes they effect are modified. Therefore the implementation scales well; that is, programs do not slow down as rules are added.

One beautiful feature of rules that proceeded naturally from their implementation in Scheme is that they are scoped; a rule has the same scope as the box it is attached to. Here is some sample Rush code that uses a local rule:

```
proc foo (list m) {
    on {[llength m] == 0} do {error "list has run out of elements!"}

    i = 0
    while {i < 100} {
        if {member? i m} {remove i m}
        i++
    }
}
```

If the list `m` ever becomes empty while evaluating `foo`, an error will be thrown; but if `foo` exits without throwing an error, the rule will be forgotten because it is out of scope.

4.2 Optimizing Away Boxes

If every variable in Rush were actually boxed, at least one Scheme procedure call would be necessary to get the read a variable's value; this is an unacceptable performance penalty. Therefore we adopted a convention that variables may be either boxed or unboxed. We created a macro called `get-value` which checks to see whether a variable is boxed; if so, the getter is called, otherwise the value itself is used. Then we modified the code generator to make sure that only values that needed boxes (that is, values whose getters or setters did something unusual) were ever boxed. Since `get-value` is a simple `if` statement, it compiles to fast code in the general case (the case where a value does not need to be boxed).

5 VHLL Comparisons

First we present a breakdown of the effort and line-count of the Rush source code by topic of implementation, comparing it to the source trees for a few other public domain VHLLs. Then we show a few performance figures for Rush in an attempt to demonstrate that our use of Scheme as an IR actually helped us make a *faster* language than coding directly in C. We've discussed many of Scheme's virtues and shortcomings; now we present empirical results.

5.1 Code Comparison Between Three VHLLs

The following is a line count comparison between the source trees of a set of VHLL implementations. In some sense, this comparison is inaccurate since in almost every case we're comparing apples to oranges: the features provided by these languages are fairly different from each other. Both Rush and Tcl conform rigorously to the Sprite coding conventions[Ous89], which include a fair volume of comments. Of the other two columns, Perl v4 is a byte-compiling interpreter by Larry Wall and SCM is a public domain Scheme interpreter that we've ported Rush to. Please note that the measurements for Perl4 are especially inaccurate; the source code has little documentation so we have based our figures on educated guesses. We have inflated the line counts for Perl to compensate for the lack of comments.

You may notice that the Rush parser is large. This reflects our efforts to unify the command syntax of Tcl with the operator syntax of C, so as to minimize the verbosity of the language. Details will be provided in a forthcoming technical report. Again, the point is that our use of Scheme freed us to explore this aspect of the language that we would otherwise not have had time for.

VHLL implementation sizes (in lines of code)

n/a = feature not provided by the VHLL.

description	perl4	Tcl7.3	Rush	SCM4d3
read-eval-print loop	4,700	2,300	100	350
main and runtime lib	2,500	3,600	1,500	3,200
parser	5,000	1,300	5,600	200
memory management	600+	600	n/a	1,900
regexp library	3,300	1,900	Tcl	250
arithmetic library	300	2,300	300	1,900
basic data structs, types	4,600	1,800	2,200	4,800
compiler/codegen	3,000	n/a	2,500	n/a
strings and conversion	3,700	2,600	450	n/a
var traces/rules	n/a	1,700	1,500	n/a
exception handling	300+	300+	660	300
total of all rows	28,000	18,400	14,800	12,900
total for language	40,700	26,700	17,600	20,000

Some notes:

- Rush entries labeled "Tcl" are stolen from an embedded Tcl interpreter, which is linked to Rush as a shared library and called through Tcl's C interface.
- The "memory management" row only accounts for the central controller of memory. Perl and Tcl don't seem to offer fully-automatic memory management, so some memory management code exists outside the core. The large size of Scheme's memory management facility is due to the sophisticated garbage collection algorithm.
- "Basic data stucts and types" include hash tables, lists, booleans, and common operations on them. This mostly demonstrates that while Scheme can save some of the labor, a typical VHLL still needs to tailor primitive types to its purposes. In the case of Rush, boxing required us to duplicate much of this sort of code.

- In most systems, error handling code is distributed across many places in many functions; these estimates are surely undercounts of this difficult-to-measure item. Again, the point is that a new VHLL will likely need its own exception and error handling mechanisms.

- Again, these numbers are *very* rough estimates, and shouldn't be used except to say that a feature took "a lot" or "a little" code to implement in the given language. Assume that they may be off by 25-50%. Each system has 6,000-10,000 lines missing; they are typically from "portability and OS interface", and reflect more how widely ported the system is (or how portable the code is) and what OS interface features are implemented (ie. networking).

5.2 An Example: Conway's Life

We implemented Conway's game "Life" using the naive $\theta(n^2)$ algorithm commonly used by novice programmers and by experts prototyping such an program. Some notes:

- "tcl" refers to the Tcl7.3 implementation. "rush-interp" refers to a nearly-identical version for Rush (syntactic changes only). "rush-compiled" refers to the same code compiled into Scheme, then into C by Scheme→C.

- "rush-compiled" is still a Rush interpreter, but with the time-consuming set of Life primitives compiled into the executable. This is directly analogous to Tcl, which lets users bind Tcl commands to C functions; in Rush, we also let users compile Rush functions into the interpreter. Like Tcl, they can be redefined at runtime.

- We consider these numbers very poor for Rush, owing to the high number of assignments done in the program code. Such assignments (eg. to update the next generation's board) incur high overhead because they interact unpleasantly with Scheme→C's garbage collector.

These comparisons are highly Tcl-centric because we developed Rush as a descendant of Tcl. Additional benchmarks can be found in [SBD94]. While we didn't measure other systems such as Perl4 or Python, various accounts from the comp.lang.tcl USENET newsgroup suggest that their speeds are similar to Tcl's.

10x10 board size			20x20 board size		
tcl	1560μsec	1.0x Tcl	tcl	8400μsec	1.0x Tcl
rush-interp	477μsec	3.3x Tcl	rush-interp	1880μsec	4.4x Tcl
rush-compiled	62.2μsec	25.1x Tcl	rush-compiled	264μsec	31.0x Tcl

40x40 board size		
tcl	65200μsec	1.0x Tcl
rush-interp	7600μsec	8.5x Tcl
rush	1060μsec	61.0x Tcl

5.3 Other Languages Besides Scheme?

As time goes on we see that other languages might make IRs as effective as Scheme. Probably the most promising of these is ML.

We had several reasons for originally choosing Scheme over ML. Scheme has a simpler syntax which, as we pointed out, makes code generation easier; Scheme's stated claim of orthogonal functionality seemed like a pretty good idea for a high-level IR.

ML compiles to very efficient code, usually more efficient than Scheme. However, much of this efficiency is due to ML's system of strong typing— that is, that the type of an identifier in ML must be known at compile-time and must remain constant. For Rush, we wanted to offer dynamic typing: we wanted identifiers that could always hold values of any type, and that could change type on the fly. This means that in the ML output code of our compiler each identifier's type would have to be the union of all types. This would keep the ML type inference system from being able to infer anything useful, thus discarding much of the point of using ML.

There were also far more Scheme implementations to choose from than ML. When we started this project, the only full-scale optimizing ML compiler we knew of was Standard ML of New Jersey (SML/NJ). Whereas SML/NJ is a high quality implementation, it is also quite big. At the same time there seemed to be many viable Scheme implementations, most of them fairly small. Since then, two smaller ML implementations have been announced for a variety of platforms, CAML and Moscow ML.

As time goes on, we expect ML to be a much more viable alternative to Scheme; we also expect other languages to become good possibilities.

6 Conclusions

Scheme offers the VHLL designer many advantages which together can drastically reduce the effort needed to design new languages. The same arguments can be used to claim that Scheme provides similar benefits to the modifier of a VHLL, since more of the underlying engine is "standard" (non-proprietary).

There are disadvantages to using Scheme, but the benefits seem to outweigh them. With Rush we found the savings dramatic, compressing a multi-year development cycle into a relaxed half-year; this included a parser, an interpreter, a compiler, a binding to the Tk graphical user interface library and a binding to the Tcl-DP distributed computing library, and ports to several platforms and two Scheme back-ends.

Much of this paper focused on performance issues. Our stance concerning language performance is fairly simple: if you're going to make changes to the design or implementation that cause the resulting system to be slower, it should be for a good reason. Adding "high level" features, making the system easier to use, etc. are all reasonable excuses to slow the system down. In the implementation of VHLLs, performance has often been traded away for greater ease of design and speedier implementation of the VHLL. We suggest that Scheme provides the VHLL designer with a feature-rich, fast environment, obviating the need for this tradeoff.

6.1 Future Work

In the future we're looking to expand and complete the Rush environment. Part of this will necessitate building a debugger, another painfully laborious project that we'd like to wish into existence (as opposed to "debug into existence"). We have no idea how we're going to leverage off of existing debuggers, since they tend to be language specific, but that's never stopped us before, and the deadline is even worse this time.

Students and researchers interested in the system should send email to the authors. We do not expect to publicly release the system until we're satisfied that its syntax and semantics will remain fairly constant.

7 Biography

Adam Sah is a PhD student at UC Berkeley, working on the Mariposa massively distributed database. Rush will serve as the "glue" language and also as the expert system shell in which we will encode distributed storage management policies. His other interests include poetry, raving, and lazy afternoons with his cats.

Jon Blow is an about-to-be-paroled undergrad at UC Berkeley who's worked on an update to INTERCAL, the FMPL language, and most recently Rush. He's never commented code with `/* more deep magic */` or `/* you are not expected to understand this */`

8 Acknowledgements

The authors graciously acknowledge the work of Brian Dennis, who helped on the Rush prototype, and to Raph Levien, who suggested the use of the getter-setter mechanism. Thanks also to the program committee, whose feedback was invaluable, to John Ousterhout, Mike Stonebraker, Sue Graham and Paul Hilfinger, who each guided our understanding of the issues involved in the Rush language design.

References

[App92] Andrew Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[ASU86] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. "Compilers: Principles, Techniques and Tools." pp.422-423 introduce the issues in implementing dynamic scoping, although their text predates the acceptance of RISC architectures and the new importance on storing local variables in registers, as opposed to their maintenance in memory locations.

[Bart89] Joel Bartlett. "Scheme→C: a portable Scheme-to-C compiler". DEC WRL Technical Report #89/1, Jan, 1989.

[BT93] Bill Triggs. AVCALL 0.1 Documentation. ftp.robots.ox.ac.uk: /pub/gnu/avcall0.1.tar.gz

[BW88] Hans-Juergen Boehm and Mark Weiser. "Garbage Collection in an Uncooperative Environment." Software- Practice and Experience, Vol 18(9), Sept. 1988.

[CFRWZ91] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and Kenneth Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph." ACM Trans. on Prog. Lang. and Systems, 13:4, October, 1991.

[DFH86] R. Kent Dybvig, Daniel P. Friedman and Christopher T. Haynes. "Expansion-Passing style: Beyond Conventional Macros". ACM Conf. on Lisp and Functional Prog. 1986.

[Ous89] John Ousterhout. "The Sprite Engineering Manual." UC Berkeley Technical Report #89/512, 1989.

[R4RS] William Clinger and Jonathan Rees, ed. "The Revised$\hat{4}$ Report on the Algorithmic Language Scheme." Lisp Pointers IV (July-Sept 1991).

[RZ88] Barry Rosen and Kenneth Zadeck. "Global Value Numbers and Redundant Computations." ACM Princ. of Prog. Languages, January, 1988.

[Sah94] Adam Sah. "An Efficient Implementation of the Tcl Language". Master's Thesis, Univ. of Cal. at Berkeley tech report #UCB-CSD-94-812. May, 1994.

[SBD94] Adam Sah, Jon Blow and Brian Dennis. "An Introduction to the Rush Language." Proc. Tcl'94. June, 1994.

[SF89] George Springer and Daniel P. Friedman. *Scheme and the Art of Computer Programming* MIT Press, Cambridge, MA 1989.

[SG90] James W. Stamos and David K. Gifford. “Remote Evaluation.” ACM Trans. on Prog. Lang. and Systems, Vol 12, No. 4, October, 1990.

[SHH86] Michael Stonebraker, Eric Hanson, and Chin-Heng Hong. "The Design of the Postgres Rules System." UC Berkeley tech report #UCB/ERL M86/80.

[Shiv88] Olin Shivers. “Control Flow Analysis in Scheme”. ACM Prg Lang Des. and Impl. June, 1988.

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

[Ston93] Michael Stonebraker. “The Integration of Rule Systems and Databases.” UC Berkeley tech report #UCB/ERL M93/25. 1993.

[TT94] Mads Tofte and Jean-Pierre Talpin. “Implementation of the Typed Call-by-Value Lambda Calculus using a Stack of Regions.” Conf. Proc. of ACM Princ. of Prog. Lang., Portland OR, 1994.

[VP89] Steven Vegdahl, Uwe Pleban. “The Runtime Environment for Screme, a Scheme Implementation on the 88000”. 3rd Int’l Conf. ASPLOS. SIGPLAN Notices 24, April 1989.

[Wil92] Paul R. Wilson. “Uniprocessor Garbage Collection Techniques.” Intl. Workshop on Memory Management. St. Milo, FR. Sept, 1992.

[WL73] Donald Woods and James Lyon. “The INTERCAL Programming Language Reference Manual.” Technical report. Stanford University, 1973.