# An Introduction to the Rush Language

Adam Sah[*], Jon Blow, and Brian Dennis

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

May 29, 1994

## Abstract

Rush is a new language that looks and feels much like Tcl [**?**]; we offer a compiler that executes scripts a hundred times faster than Tcl 7.x, allowing programs to run at speeds close to their C-language counterparts. Rush incorporates many features from Tcl and contains new features. From Tcl, Rush acquires its syntax, "everything is a string" model, set of core commands and datatypes, scoping rules, C callout facility, and support for popular libraries, including Tk and Tcl-DP. New features in Rush include pass-by-reference, first class closures and production rules. A generalization of operator syntax allows users to code in either command-style or operator-style syntax; a converter program provides a translation between the two forms. We introduce the language as a set of changes to Tcl, with a focus on performance issues and discussion of new features.

## 1 Introduction

### 1.1 Design Rationale

Rush is a new language whose syntax, semantics and runtime libraries mirror those of Tcl. Our goal is to maximize the performance and utility of our language without abandoning the features that make Tcl popular. To that end, Rush largely preserves Tcl compatibility while increasing performance two orders of magnitude and adding several new language features.

Rush arose from a desire for a faster Tcl that would run applications at speeds near those achieved by their C equivalents. Realistically, this requires compilation of time-critical pieces of scripts into efficient machine code, a process not practical for Tcl given its semantics.

Rather than adopt Tcl syntax verbatim, we decided to make modifications to it, in response to frequently-voiced complaints that Tcl syntax is sometimes too verbose for expert users. Rush provides terse syntax for mathematical operations, variable assignments and other common forms. Rush still supports Tcl's familiar command syntax, though, and the availability of familiar Tcl core commands helps ease the transition to Rush for Tcl programmers.

Rush supports the features that have made Tcl popular. Specifically, these include Tcl's "everything is a string" type model, a scoping system that allows variables to be used without predeclaration, the C callout mechanism, and integrated support for user libraries, notably Tk (the user-interface library) and Tcl-DP (the distributed programming library).

Rush also adds new language features, the most important of which are *production rules*. Though production rules are unconventional in general-purpose languages, we have already found several interesting uses for them. We hope that our implementation will make them inexpensive enough and easy enough to use as a general programming tool.

In short, Rush provides much compatibility with Tcl, adds new features, and runs substantially faster. The remainder of this paper will discuss our motivation for wanting a high performance scripting language, give an overview of Rush, describe our current implementation, present performance results, and discuss future work.

*This work was supported in part by NSF grant #IRI9107455. This work originally appeared as part of the Tcl'93 Workshop.

## 1.2 Motivation for High Performance

The desire for improved performance in the execution of Tcl scripts is often voiced on the Tcl newsgroup, `comp.lang.tcl`. Tcl offers some of the poorest performance of interpreted scripting languages, such as Perl[**?**], Python[**?**], and xlisp. The argument is that Tcl performance doesn't matter. This is true for Tcl procedures that are tied to I/O operations such as network reads and user-interface events. The time required to perform these operations is large compared to the overhead of executing a simple Tcl script, thus effectively hiding Tcl's sluggish performance.

In practice, there are a few cases where this "speed doesn't matter" philosophy breaks down. One case is that when an application starts up, it typically must initialize large tables of information. Since this involves little I/O and the user must wait until its completion, startup times for Tcl applications can be long and frustrating. Also, slowness in a scripting language lessens its utility in coding even moderately expensive algorithms; in Tcl, the programmer is forced to write such functions in C, negating some of the benefit of using Tcl in the first place. For example, the *Perspecta Presents!* slide-making program consists of ~29,000 lines of C code and only ~13,000 lines of Tcl code. The tendency to write large sections of code in C for performance reasons was verified in the results of a `comp.lang.tcl` survey [**?**]

Rewriting Tcl code in C can be a heavy time sink, and once is done, the incremental development offered by high-level script interpretation is lost forever. Some procedures are messy to rewrite in C; this is especially true for procedures that interact with the interpreter state (mutating Tcl variables) or perform the string operations that the Tcl language makes so easy. In these cases, it is clear that Tcl is the preferred language for the job, yet the programmer is forced to code in C. In the most painful case an entire application must be rewritten, as with the Caste object-oriented library [**?**].

Finally, many programmers would like to write entire applications directly in Tcl. For example, TkMan, the Unix manual page browser, is written entirely in Tcl. In such cases, the choice to rewrite code in C complicates the installation process and raises concerns of portability.

## 1.3 Previous Work: Optimizing Type Conversions

One attractive feature of Tcl is that all values can be treated as strings; Tcl commands parse their arguments from strings into the types they want and then operate on the parsed values. Internally, the original implementation of Tcl (hereafter called *Tcl7.x*, to allay confusion; *Tcl* refers to the language definition and not a specific implementation) stores all values as strings and passes these strings as arguments, which produces this behavior naturally. However, conversion between strings and other value types is slow. The following Tcl code is a good example:

```
for {set x 0}  {$x<1000}  {incr x} {
   ... loop body here ...
}
```

All of the arguments to `for` are strings. At every iteration of the loop, `$x<1000` must be tested to see if the looping should continue, and the `incr x` must be run as a command. In each iteration, Tcl7.x parses `$x<1000` into a math expression before testing for loop continuation. Since the value stored in `x` will be a string, it must convert the value to an integer (and do the same with `1000`) and compare them. Every time Tcl7.x reaches the end of the loop, it must likewise parse `incr x` as a command and call `incr`. `incr` will convert the string value of `x` to an integer, add 1, convert that value back to a string, and store the result in `x`. In addition, the body of the `for` loop is a string and must be parsed as a sequence of commands at every iteration of the loop.

This adds up to a lot of unnecessary conversion, the expense of which often dwarfs the amount of useful computation being performed. This is why Tcl7.x executes loops so slowly. In the Tcl compiler project, TC [**?**] [**?**], we remedied this problem by caching the results of type conversions with variables. Running under TC, the example above would parse `x` as an integer once and operate only on integers thereafter. In the end, TC offered a 5x-10x speed improvement at the expense of major changes to Tcl's internals.

## 1.4 General Compiler Optimization

A big component of optimization is code transformation. Optimizing compilers often change the code the programmer has written, moving expressions and eliminating redundant calculations. The better the optimization, the less the final product resembles the original program.

Some of Tcl's features assume things about the structure of running code, severely limiting the optimizations a compiler can perform. We have done our best to replace such features with less-devastating forms of the same expressive power. A full description of possible compiler optimizations and their interaction with language features is well beyond the scope of this paper; at appropriate times, however, we present short summaries of the ways in which features interact with optimization. Those who remain unconvinced should see [?] and [?] for rigorous explanations.

## 2 Language Overview

### 2.1 Basic Types in Rush

The Rush type system behaves like Tcl's, complete with string representations for all datatypes and automatic type conversion as needed by primitives. Its implementation echoes that of the Tcl compiler, with native types stored whenever possible, and dynamic type conversion used to provide strings and / or other types when primitives demand them. Like TC, Rush uses one conversion procedure per type. Rush even offers a nearly identical set of base types as Tcl and TC:

#### 2.1.1 Strings

Strings in Rush are delimited by double quotes. Like Tcl, a special set of operators is available for string substitution within double-quoted strings. This lends nearly identical behavior to Tcl for such values.

```
set a 5; puts stdout "a is $a"
```

#### 2.1.2 Numbers

Numbers in Rush are expressed in the usual way. Functions are provided to operate on numbers, such as add and mult. For each such command, we provide a syntactic operator borrowed from C (see the section on “Generalizing Operator Syntax”). Thus,

```
a = 4; a++; b = $a*5
```

is identical to:

```
set a 4; incr a; set b [mult $a 5]
```

Thus, the expr command in Tcl serves little purpose in Rush, although we provide one for backward compatibility.

#### 2.1.3 Procedures

Rush offers procedures as first-class data types and reserves curly braces for their syntactic representation. Although braces in Tcl are used to enclose literal strings, these literal strings are often Tcl scripts passed as arguments to commands; this guides our choice of syntax. Here is an example:

```
set a 5
set x {incr a; puts stdout "a is $a."}
puts stdout "About to run 'x':"

# note that no "eval" is needed here,
#   since Rush already knows about $x
#   as a script: it's a proc taking no
#   arguments.
$x
```

This will print:

```
About to run 'x':
a is 6.
```

The first line sets the value of x to be a script, and the last line runs x as a command. Procedures in Rush are just variables that refer to scripts. They can be created with names or without. The ability to define procedures without names is often convenient when using a procedure as a return value or the value of a variable. Here is an example of the latter:

```
# Standard Tcl-like use of "proc"
proc add1 (x, y) { return $x+$y }
set add2 [proc (x, y) { return $x+$y }]

add1 4 5
add2 4 5
# The answer to both is 9.
```

Braces without the proc keyword are a shorthand for creating procedures with no arguments; these are often used by commands like for :

```
for {set i 0} {$i<1000} {incr i} {...}
```

#### 2.1.4 Lists

In order to distinguish lists from procedures, Rush lists are delimited by parentheses:

```
set a 6
lindex (5, $a, "hello") 2
```

yields the answer 6. List items may be of any type, although string constants require double quotes. Likewise, in the example above, the first element is the integer 5, not the string “5”. This is important for performance, because a procedure that assigns the same constant list to a local variable should not be forced into assigning strings, which later require parsing before use.

When converting from strings to lists, Rush uses a similar system to Tcl, only commas delimit items, and items have leading and trailing whitespace removed.

```
[list "a, (b, c), d, e"] ==
      ("a", ("b", "c"), "d", "e")
```

### 2.1.5 Associative Arrays

Associative arrays are collections of associated keys and values. We provide a syntax for array literals, as well as one for indexing arrays. The example below maps strings to values, but our current implementation supports both string and integer keys, and values can be of any data type. The ability to use integer keys is provided for high performance in special cases. Since all Rush values have string representations, any type can be effectively used as an array key, though it will be converted to string form.

```
set a ( "cliff steele" :: 16384,
        "crazy jane"   :: 69105,
        "mr nobody"    :: 90028 )

set b a:("crazy jane")
# The result is 69105
```

## 2.2 Using the Type System

The Rush type system allows explicit checking and assertion of the type of a given value. For each type that the system understands, an *is-a function* exists whose name is of the form `typename?`. This function accepts one argument and returns a boolean describing whether the value passed is of the desired type. For example, if we were to construct a type called `number`, the is-a function might be defined as follows:

```
proc number? (x) {
   return [int? x] || [float? x]
}
```

Each type also has an assertion function whose name is that of the type itself. The function accepts either one or two arguments, with the first being the value to coerce and the optional third argument being a procedure to call if the coercion fails. If no such function is provided, a system default is used, issuing messages similar to Tcl's: "expected $typename but got $value".

Rush coercion functions can be implemented as user-level procedures. Here is the coercion function for the `number` type above:

```
proc number (x, errfun = def_errfun) {
     if {number? x} {return x}

     # if it's not already a number, go to
     # string form, then convert back.
     # note: since all values have string
     #  representations, no error function
     #  is needed when cvt'ing to string.
     s = [string x]

     # pass the error function onto the
     #   floating point converter.
     return [float x errfun]
  }
```

Type assertions are therefore a handy tool for localizing errors in the under-constrained type system provided by languages like Tcl and Scheme. Without them, malformed data can pass through large sections of code unchecked, erroring only when passed to a core command. Instead, the lightweight syntax of assertions makes it easier to capture such errors. An example of

```
# typed declaration of new variables
proc foo {
   # create a new variable 'a', assign
   #  it the value 7 and coerce it to
   #  an integer.
   int a = 7

   # find or create the global variable
   #  'b'.  If it exists, coerce it to
   #  an integer.
   global int b

   ...
}
```

Likewise, we added type assertions into our procedure declaration syntax as optional argument modifiers:

```
proc foo (int x, y) {...}
```

is equivalent to

```
proc foo (x, y) {int x; ...}
```

While this is a minor change, as a matter of practicality such lightweight syntax is very important because it lowers the barriers to adding type information to code. The hypothesis is that in a dynamically typed system, it is by definition impossible to force programmers to supply type information everywhere, but it is highly preferable to do so in select cases, from both optimization and software engineering standpoints.

The existence of typechecks which do no useful work in correct code may spark concern about performance. However, we provide a compiler flag which ignores debugging typechecks; production code will not be slowed by such debugging aids.

## 2.3 Scoping in Tcl and in Rush

Tcl offers three kinds of scope transitions for accessing variables: global, uplevel and local. Globally scoped items are those defined in the toplevel interpreter script, and must be explicitly imported into a procedure's local variable list. Uplevel variables are those in a procedure's caller's scope. Since all procedures are ultimately called from the global scope, global variables may be thought of as a special case of uplevel variables. Local variables in Tcl are all other variables that are defined using the "set" command. Unlike C, C++, and Pascal, Tcl requires no declaration of variables prior to their use.

Rush offers nearly identical facilities to Tcl for global and local variables, including the ability to create new global variables from within procedures. However, Rush removes the ability to use uplevel variables, and thus removes `uplevel` and `upvar` from its lexicon. In the `upvar` and `uplevel` commands, Tcl offers a feature that the programming language community calls *dynamic scoping*. This variable scoping technique allows procedures to locate variables based on the runtime calling sequence: procedures can peer into your caller's scope. Since calling sequences can only be known at runtime, dynamic scoping requires runtime support, dramatically increasing the overhead of local variables. By comparison, under lexical scoping (where you can only peer into textually surrounding blocks), the positions of visible variables can be fully determined at compile time. C employs lexical scoping; Lisp and Tcl employ dynamic scoping.

It is difficult to implement dynamic scoping efficiently. The ability to perform operations like `uplevel` and `upvar` requires that variables in higher calling scopes be indexable at runtime; the overhead of maintaining data structures to make this possible dramatically slows the runtime system because all functions must create and maintain this table, which in Tcl is implemented as a hash table of variable names and values. Also, if procedures are allowed to modify their callers' scopes unpredictably, the values of variables must be stored in predictable and modifiable locations, which means that they cannot usually be held in registers; nor can variables be eliminated from a procedure body. For a complete discussion of the troubles of dynamic scoping, please see [**?**].

The commands `uplevel` and `upvar` are used fairly often in Tcl, so if we wish to remove them, we must provide a mechanism that meets the same needs. A survey of comp.lang.tcl readers showed that `upvar` and `uplevel` are used mainly for two things: modifying the values of parameters to a procedure and executing blocks of code passed to the callee, where the evaluation must occur *in the caller's scope* to be correct. This latter usage is typical of user level control constructs, commands that control flow over a body of code, with the usual example being the `for` command.

To enable procedures to change the values of their arguments on the caller's side, we use the idea of *reference parameters*. In the tradition of languages like Pascal, reference parameters are denoted by the keyword `var` in an argument list. When a reference parameter is changed, the value of the argument on the caller's side changes as well. Here is a sample implementation of the command `incr` in its two-argument form, where both the variable and the value to add are given as arguments.

```
# Tcl implementation of "incr"
proc incr {x i} {
    upvar x; set x [expr $x+$i]
}

# Rush implementation of "incr"
#  the "var" indicates that x is
#  a reference parameter.
proc incr (var x, i) { set x ($x+$i) }
```

As we have seen, curly braces enclose lists of commands to be treated like procedures. The variables mentioned inside such a script refer to values in the procedure that where the script is created, not the one where it is called. Here is an example:

```
proc run_as_command <x> {
   set a "*in local scope*"
   puts stdout "Running command:"
   x
}

set a "*in global scope*"
run_as_command {
   puts stdout "a is $a."
}
```

In this example, we have a global variable `a` with the value "*in global scope", and another variable `a` local to `run_as_command` that holds the value "*in local scope*". Running this example will result in the following output:

```
Running command:
a is *in global scope*.
```

because the anonymous function that is passed to `run_as_command` was formed in the global scope, the value of `a` inside it refers to the global variable `a` , which holds 5.

By composing this mechanism with the Rush exception handling mechanism, it becomes possible to efficiently implement user level control commands like `for` . In fact, the core commands `for` and `while` are implemented as compiled Rush code.

## 2.4 Generalizing Command Syntax

Rush procedure calls are written in Tcl's cmd-arg-arg-arg style ("command syntax"). However, we felt the need to respond to complaints about the verbosity of this format. For example, where `x++` suffices in C to increment an integer variable, Tcl users must specify `incr x` . When indexing arrays, Tcl requires one to write `[lindex $list $index]` where the C equivalent is `list[index]`.

We decided that special syntax to express common language idioms would improve readability. This is not a radical change; Tcl itself already supports a few syntactic operators. For example, Tcl array variables are essentially hash tables keyed on strings; therefore, we can expand the operator form of `$array(index)` into `[hashtab_find $array index]`. Likewise, dollar-sign substitution within quoted strings is shorthand for a call to `concat`: `"hi $a"` expands to `[concat "hi" $a]`. With this in mind, it seems reasonable that syntax for commonly-used commands like `set` and `expr` might be effective in reducing verbosity. The dividing line of too much operator syntax, beyond which code becomes unreadable, is a fairly arbitrary separation; we are careful to draw this line conservatively and not to overstep.

However, command syntax offers users new to the language or feature set the ability to easily identify the function being invoked for a given statement. It is then possible to find this command in a manual, for example. By comparison, `a--`$\rangle$`b` is legal in C and yet indescipherable to non-experts.

A solution to this dilemma is to assign all operators to unique commands. Then it becomes possible to convert from command form to operator form on an operator-by-operator basis. If the user specifies which operators she likes, all other operators can be expanded into their command forms. The inverse transformation can also be applied. If this transform program is invoked before each call to the user's editor, then code may be seamlessly transferred from one user to another, even if the two users have chosen different sets of operators to display.

In most cases, Rush core commands and syntax mirror Tcl's, so code is generally compatible. For example, though Rush variables need not be delimited by dollar signs, we offer dollar-sign notation for backward compatibility and because it is necessary when substituting items into double-quoted strings.

## 2.5 Rules in Rush

### 2.5.1 Introduction to Rules

A rule is a predicate attached to a list of commands. [**?**] Any time during the course of program execution that the predicate becomes true, the list of commands is run. This is unlike an 'if' command, which only tests a predicate at one point in the program. A rule is like an 'if' command that is always watching.

Here is an example of a rule declaration in rush:

```
on { $x < 10 } do { puts stdout $x }
```

This means that whenever `x` acquires a value below 10, the value will be printed.

Rules in Rush are implemented with a mechanism akin to Tcl's write traces, though traces in Rush are much faster. One could attempt to implement the `on` command in Tcl using write traces, but it would be difficult and very slow because Tcl's dynamic scoping and unavailability of the "old" value when capturing write traces.

We will now discuss some interesting uses of rules. We will speak of rules "firing"; when a rule fires, its predicate is tested, and if it is found to be true, the action will be run. The rule above will fire whenever the value of `x` changes, though the action will only be run when `x` is less than 10.

### 2.5.2 Expressing Invariants Using Rules

An invariant is a condition that must always be true if a program is correctly running. If an invariant ceases to be true, either the program is faulty or the algorithm it implements is ill-conceived. Expressing invariants in a program can be a great aid to debugging, especially when implementing large systems.

Some languages provide mechanisms for checking invariants at specific points of a program; for example, C defines the macro `assert` in its standard library. These traditionally work as `if` statements; but the 'on' command in Rush is a natural way to express *continuous* invariants, that is, invariants that should always be true for a given set of variables, at any point in the program where those variables exist.

Since rules are placed on variables, reference parameters interact with rules the way one might expect: that is, a rule placed on a reference parameter is also placed on the variable on the caller's side. This allows us to create procedures that place rules on variables. Here is an example of a procedure that attaches an upper-limit to a variable, so that a warning message is printed if the limit is ever exceeded:

```
proc upper_limit (var x, int lim) {
   on { $x > $lim } do {
      puts stdout "value over limit."
   }
}
```

Rush provides a way to talk about the "old" value of a variable (the value as it was before it was modified by the change that fired the current rule). The unary operator "∼" before a variable name indicates that it refers to an old value.

Here is a rule declaring that the variable `x` should never decrease for any reason:

```
on { $x < $~x } do { warn_about $x }
```

The predicate `$x < $∼x` means "is the new x less than the old x?".

At the end of this paper we delve a bit deeper into the use of rules for expressing invariants.

### 2.5.3 Using Rules in General Purpose Code

Predicates can be arbitrarily complex. We now show some uses of production rules as general-purpose tools that create less bulky code. Our first example is a self-refilling list; suppose we wish to extract items from a list and have the list automatically refill itself when it becomes empty:

```
on {[llength $l] == 0} do {refill_list l} }
```

In a sense we have created a new data type with this 'on' statement, turning the list 'l' into a new type of "self-refilling list". This list can be accessed and modified using the commands and operators supported by Rush, and it will perform as desired; no new accessor functions are required. Unlike standard encapsulation techniques, ours is reversible: such rules can be easily removed (although we haven't shown the syntax for doing so), yielding the original behavior.

Our second example is a *ring buffer*: suppose we wish to treat a list as a ring structure: that is, we want to move around an index to that list, and if the index goes off one end we want it to wrap to the other. Ring buffers is often useful when implementing features such as a shell's history list, or remembering scrollback for a text window.

If we change the value of the list index often in the code, it becomes ugly to explicitly check to see if the value should wrap after each of these changes. Instead, we could use 'on':

```
int i
list l

# If we've scrolled off the right,
#  wrap to the left.
on {i >= [llength l]} do {
    i -= [llength $l]
}

# If we've scrolled off the left,
#  wrap to the right.
on {$i < 0} do {
    i += [llength $l]
}
```

And due to the magic of reference variables, we can encapsulate these calls in a procedure, so that we could use them all over the place. In this modified version we use some of Rush's extended operators for brevity.

```
proc wrapping_index <var int i,
                     var list l> {

   on {i >= [llength l]} do {
      i -= [llength l]
   }

   on {i < 0} do {
      i += [llength l]
   }
```

Because the `i` and `l` in `wrapping_index` are reference parameters, the two rules created by each call to `wrapping_index` are attached to the variables on the caller's side.

We can use `wrapping_index` whenever we have a list and an index for it:

```
int i
```

```
  list l

  wrapping_index $l $i
```

In essence, wrapping_index binds a list and an integer together into an active data structure, on in which l and i automatically respond to each other's values.

One limitation of rules in Rush is that they assume that procedure calls depend only on the values of their parameters. A procedure whose return value depends on other factors (such as the values of global state variables) may cause a rule's predicate to become true, yet the rule will not be fired. We say that such a procedure does not preserve *referential integrity*. For an easy illustration you might imagine the following rule:

```
# "random" is a procedure that returns
  #     either true or false, at random.
  on {random 10 == 5} do {
     puts stdout "Hai-Keeba!"
  }
```

This rule is nonsensical because there is no way to know when it should be fired. There are rules which do not preserve referential integrity yet make more intuitive sense. We do not attempt to support these cases because we consider them relatively unimportant in terms of usefulness, and because under our implementation scheme deciding when to fire such rules is computationally intractable.

A thorough treatment of rule semantics, including a description of rule composition, will be given in a forthcoming paper.

## 3 System Architecture

Rush consists of a parser that emits Scheme code, a set of runtime libraries written in Scheme, and a small amount of C code to interface to the operating system. A Tcl interpreter is compiled into these libraries, which provides the system with the functionality of Tk and Tcl-DP. The parser reads a statement of Rush code at a time and emits sets of Scheme code, usually one or two Scheme statements per Rush statement. We have used this interface both to write the Rush$\rightarrow$Scheme compiler, as well as for the interpreter. Multiple-statement optimizations are expected to be performed by the Scheme or C compilers.

### 3.1 Development Environment

The build process of a Rush-based application is begun by writing code in Rush, Scheme, and/or C, and providing a subset of this code to be compiled into an interpreter. Our parser translates Rush code into Scheme, which is compiled together with the runtime libraries and any Scheme code provided by the user. The Scheme compiler chosen was Bartlett's Scheme$\rightarrow$C system [**?**] and so the output of this compilation is a stream of C code. This C code is then compiled into either an interpreter or standalone program. This process is nearly identical to Tcl, in that C-based additions are presented to a compiler for inclusion into a new version of the interpreter. Like Tcl, if no code is to be compiled, then a precompiled interpreter may be used (ie. the Rush equivalent of "wish"). The advantage of the Rush build process over Tcl's is only that ours is designed to be portable and seemless.

In the future, it may be possible to adopt a compilation model where the user works directly from a command line and each statement is compiled before execution. Then, only the few C procedures needed by the system would be compiled into the "compiling interpreter"- the remaining code could be either interpreted or compiled on the fly. It would also be possible to place traps on procedures and compile them if they have been executed many times or if they contain loops or other expensive operations.

The first version of Rush was implemented between January and May of 1994, a surprisingly short period of time for a full-fledged language. Although much of the design work was done in the year preceding this, this period also included a flurry of design decisions as well. In the end, Rush consists of about 12,000 lines of code in three major modules. The parser is 4,000 lines of C code and yacc script; the runtime library of core commands, types, type converters, and rule code is another 4,000 lines of compiled Scheme; the glue code that includes the main loop, the Tk and DP bindings, and so on is another 3,500 lines of compiled Scheme and 500 of C. The runtime shared libraries compile to 2.5MB, libtcl and libtk (build fm the standard distributions) compile to another 1.2MB. The non-sharable executable for the Rush interpreter compiles to 24K. The interpreter uses 1.4MB of RAM on startup; Tcl requires 350K by comparison.

### 3.2 External Callouts

We recognize the need to interoperate with other languages. Interfacing to the operating system and to extremely efficient code are just two examples. As well, reusing previously existing libraries of code requires this capability. To support such needs, the Rush runtime has been engineered to support calls to foreign functions otherwise known as *callouts*.

At the current time, we provide these callout functions with little support for examining the state of the Rush runtime. While we believe that callout functions should be low level primitives and not often need such capabilities, we will be exploring adding such *callin* features to the Rush runtime. Our hope is to add callins without compromising performance and possible compiler optimizations of Rush code.

Callout support is based on the capability of Scheme$\rightarrow$C to externally call predeclared C functions. With predeclared signatures, Scheme$\rightarrow$C can make most callouts directly to the C function without having to go through an argc/argv style interface. Scheme$\rightarrow$C has the capability to export some of its datatypes, numbers, strings, and opaque pointers, to C directly. Opaque pointers are essentially handles to machine pointers. These opaque pointers can be used to represent such things as C function pointers. While they can not be mutated from the Rush runtime, they can be stored in Rush variables, lists, associative arrays, etc. and later passed to other callouts. The export capability saves string conversions when calling out to external functions. Similarly, certain foreign return values are automatically converted into Scheme datatypes. Integers, floats, strings and C arrays are such datatypes. While it is possible to retarget our system to work with other Scheme systems, our "native" C callout mechanism has not been made to be portable, owing to the lack of standardization among Scheme implementations of the C-based interface for examining Scheme objects. We also offer an argc/argv interface like Tcl, and porting this interface to other Scheme implementations should not be difficult.

External code can call into the Rush runtime through one function which receives a string form and evaluates the form in the toplevel runtime state. The result is returned as a string. An appropriate analogy is using only the `send` command to execute Tcl code, so as not to interact with compiled procedures. In the future, we plan to support interpretation from within a given procedure frame; unlike Tcl, our implementation would not penalize the performance of any other procedure except the one(s) where interpretation is possible. While we have a design for this mechanism, it has not been implemented, so we defer its discussion for a future paper.

Using these callout and callin capabilities we have constructed support for interacting with Tk and Tcl-DP. In our current version of distributed Rush, we embed a Tcl-DP and Tk capable interpreter into the Rush runtime. Using our callout capabilities we export as Rush primitives, commands to manipulate the interpreter. This has allowed us to prototype and demonstrate distributed *Rush* programming. Using Tcl-DP as a low level transport mechanism, we can send and receive Rush programs across a network of hosts. Similarly, callin and callout let us perform graphical user interface manipulations with Tk.

## 4 Performance

Here we validate our performance claims. In the benchmarks below, there are three Rush numbers for each case. "Interpreted Rush" is code that was entered and run from a command-line interpreter. "Compiled Rush" is code that was compiled into native machine code, linked with the command-line interpreter, and run from there. "Optimized Rush?" is a purely speculative number measured by running compiled code that was hand-optimized in Scheme. We include it as a sort of theoretical limit of the speeds we could hope to achieve given a smart Rush-to-Scheme compiler (with the same Scheme back-end).

There are five interesting cases we chose to study. The first two are examples where traditional scripting languages perform poorly, relative to C. The latter three are microbenchmarks of the new features we implemented.

### 4.1 Fibonacci Test

The first test is a recursive implementation of a Fibonacci number function, a common performance metric among the denizens of `comp.lang.tcl`. In theory, this should stress the function call mechanism of the language. In practice, scripting languages which implement all operations as function calls, as Tcl does, will also expend a large portion of effort calling primitive operations. Compiled Rush and C programs avoid this expense by minimizing function call overhead and by inlining small, frequently-called procedures. This test demonstrates that we can compile mathematics and control flow operations into efficient machine code, and that once we do, the cost of Rush procedure calls is not exorbitant.

**Performance: recursive fib(20)**

| | | |
|---|---|---|
| Tcl | 7,300 msec | |
| perl4 | 5,800 msec | |
| TC | 1,400 msec | 5x Tcl |
| interpreted Rush | 800 msec | 9x Tcl |
| compiled Rush | 26 msec | 280x Tcl |
| optimized Rush? | 5 msec | 1500x Tcl |
| optimized C | 3.4 msec | 2100x Tcl |

Here are the Tcl and Rush versions of `fib` used in the above tests:

```
# Tcl version
proc fib {n} {
    if {$n<2} then {
        return $n
    } else {
        return [expr
          [fib [expr $n-1]]+
          [fib [expr $n-2]]]
    }
}

# Rush version
proc fib (int n) {
    if {n < 2} then {return n}
    return [fib n-1]+[fib n-2]
}
```

## 4.2 Integer Summation Test

Our second test is the summation of the first thousand integers "the hard way", using a `for` loop to increment a summation variable. This is even worse than `fib` for scripting languages that use function calls to execute primitives, because compilers for more efficient languages like C will convert the entire operation into a tight inner loop of a few machine language instructions. For Rush, this shows that we can reasonably compile such tight inner loops. While these numbers are not spectacular, we consider them adequate for the time being; they can be improved arbitrarily by better Scheme compilers.

**Performance: integer summation**

| | | |
|---|---|---|
| Tcl | 240,000 $\mu$sec | |
| interpreted Rush | 25,000 $\mu$sec | 10x Tcl |
| TC | 24,000 $\mu$sec | 10x Tcl |
| compiled Rush | 1,100 $\mu$sec | 220x Tcl |
| optimized Rush? | 480 $\mu$sec | 500x Tcl |
| optimized C | 19 $\mu$sec | 12600x Tcl |

## 4.3 upvar vs. pass-by-reference

To help validate our claims about pass-by-reference, we must show that using real pass-by-reference is no slower than if we used Tcl's `upvar` command to reference a variable by name from a called procedure. Two numbers are shown: the first is the steady-state time measurement of one procedure passing a reference to another. For Tcl, this would necessitate calling `upvar` in each procedure, although it would be possible to optimize away this call for all procedures but those that actually need to use the reference, in which case the former number drops to about 15 $\mu$sec to account for the overhead of passing one additional parameter to the called procedure; this extra parameter would be the level at which to find the original variable.

For Rush, the overhead is in creating the reference, which is actually implemented as two closures, a "getter" and a "setter", which also eases rule implementation. Once created, these getters and setters can be passed as normal parameters with no additional overhead. The runtime libraries check a tag bit to see if objects are "boxed" in this way.

When accessing reference variables, Tcl has to traverse a simple pointer; the remaining overhead is the cost of string substitution in Tcl. For Rush, this cost becomes the cost of a function call through a closure (the getter in these tests).

**do upvar/make reference**

| | | |
|---|---|---|
| Tcl | 43, 43 $\mu$sec | |
| interpreted Rush | 0, 21 $\mu$sec | 2.0x Tcl |
| compiled Rush | 0, 12 $\mu$sec | 3.6x Tcl |

**use upvar/reference variable**

| | | |
|---|---|---|
| Tcl | 9.0 $\mu$sec | |
| interpreted Rush | 7.9 $\mu$sec | 1.1x Tcl |
| compiled Rush | .8 $\mu$sec | 11.3x Tcl |

## 4.4 uplevel vs. closures

The uplevel test mirrors the upvar test, only in this case, we need to pass a set of code to a procedure and then invoke this code. In Rush, we pass closures instead of source code.

The results are fairly straightforward, except for the first case where we pass code in Tcl. In this case, the cost is roughly linear in the size of the source text being passed, because Tcl has to pass code as strings. The high overhead of creating closures in interpreted Rush is an artifact of Scheme$\rightarrow$C . As an experiment, we are porting the runtime library to SCM, a small, popular, and portable Scheme interpreter. Under SCM, closure creation time dropped to around 3$\mu$sec, although other numbers suffered and hence we did not include SCM in our test results. We believe that the individual weaknesses of these two Scheme interpreters is a result of the portability-for-performance tradeoff that each made in using native C facilities for their implementations. In less portable systems like Screme [?], we would expect better performance.

| pass code/create closure | | |
|---|---|---|
| Tcl | 45 + .29$\mu$sec / char | |
| interpreted Rush | 84.0 $\mu$sec | |
| compiled Rush | 3.1 $\mu$sec | 15+ x Tcl |

| uplevel call/execute closure | | |
|---|---|---|
| Tcl | 74.0 $\mu$sec | |
| interpreted Rush | 5.0 $\mu$sec | 13x Tcl |
| compiled Rush | .35 $\mu$sec | 211x Tcl |

## 4.5 Rules

Our test for rules shows a marked improvement of Rush over Tcl, where Tcl rules are hand-written using write traces. Our Tcl implementation of rules for this test is severely limited; it only applies to simple predicates of one variable, and that variable must be global. Also, Tcl provides no way for write traces to see the "old value" of a variable, so our cases did not include that sort of expression. Furthermore, since Rush executes the predicates and actions much more quickly than Tcl, it would not be fair to include anything more than one simple predicate and one simple action.

This is the best we could do without producing ridiculous test code. Though the features of these two implementations vary widely, we think there is some small basis for comparison.

| simple rule: action executes | | |
|---|---|---|
| Tcl | 322.0 $\mu$sec | |
| interpreted Rush | 101.0 $\mu$sec | 3.2x Tcl |
| compiled Rush | 8.6 $\mu$sec | 37.4x Tcl |

| simple rule: action doesn't execute | | |
|---|---|---|
| Tcl | 271.0 $\mu$sec | |
| interpreted Rush | 69.0 $\mu$sec | 3.9x Tcl |
| compiled Rush | 5.1 $\mu$sec | 53.1x Tcl |

The following is the source code to the rule test:

```
# Tcl version
trace variable a w n
proc n {a b c} {
  upvar #0 $a d
  if {$d<7} {global e; set e 1}
}
# action executes.
time {set a 3} ...

# action does not execute.
time {set a 9} ...

#------------------------------
# Rush version
on {a<7} do {b=1}

# action executes.
time {a=3} ...

# action does not execute.
time {a=9} ...
```

## 5 Future Work

In the introduction, we said that we have been careful to make Rush optimizable. That is so, but our current compiler does not take advantage of most of these opportunities! A more heroic Scheme optimizer, perhaps one that massages code in continuation passing style [?] [?], might bring us close to our theoretical limit.

A sophisticated debugging environment would be nice for scripting languages such as Rush and Tcl, which tend to catch fewer errors at compile-time than more rigid languages like Pascal and C. We are at work on an interactive system that makes helpful comments about your rush code as you type it. Also, we are developing methods for viewing and debugging rules, since rules have the potential to interact with each other in mysterious ways.

One long-term goal for Rush is to provide a runtime compilation system that generates very efficient code [?]; such a system would perform many of the operations of traditional optimizing compilers, but for statements entered on the command line. This would allow on-the-fly composition of code from unpredictable sources (for example, code represented as text delivered by network messages) into a representation that runs almost as quickly as statically-compiled code, and which runs much more quickly than the interpreted fare offered by traditional "eval" features. Such a runtime compiler would also speed up the rule system.

A short-term goal is to retrofit TC (the Tcl compiler) so that it can act as a Tcl-to-Rush translator. This translator would not be a universal program that works on all Tcl code; the Tcl idioms which Rush has abandoned for performance reasons would not be supported. Our hope is that migration to Rush will be easy.

All in all, Rush seems to provide many new opportunities for interesting research, and we likewise hope that it will be useful in developing real applications. Since Rush was developed for the Mariposa distributed database project we will soon have the experience to guide future design decisions.

We would like to thank Michael Stonebraker, Raph Levien, John Ousterhout, and the Mariposa research group for their insightful comments and feedback.

## References

[ASU86] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. “Compilers: Principles, Techniques and Tools.” pp.422-423 introduce the issues in implementing dynamic scope, 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.

[ARZ] Fran Allen, Barry Rosen, and Kenneth Zadeck. *Optimization in Compilers*. This is a highly-detailed point-by-point description of modern optimization techniques. ACM press, forthcoming.

[WF90] Jennifer Widom and Sheldon Finkelstein. Set-Oriented Production Rules in Relational Database Systems. Proc. ACM SIGMOD. Atlantic City, NJ, May 1990.

[Oust94] John Ousterhout. *An Introdution to Tcl and Tk*. Addison-Wesley, New York, NY, 1994.

[Bart89] Joel Bartlett. “Scheme$\rightarrow$C : a portable Scheme-to-C compiler”, DEC WRL Technical Report #89/1, Jan, 1989.

[Brav93] Michael Braverman. “Caste: a class system for Tcl”, Proc. Tcl’93 Workshop. June, 1993.

[SB93] Adam Sah and Jon Blow. “TC: A Compiler for the Tcl Language”, Proc. Tcl’93 Workshop. June, 1993.

[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.

[Surv94] Adam Sah, ed. *USENET survey of Tcl usage on comp.lang.tcl.* Unpublished. April, 1994.

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

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

[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.

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

[Sto93] Michael Stonebraker. The Integration of Rule Systems and Databases. UC Berkeley tech report #UCB/ERL M93/25. 1993.

[Ros93] Guido van Rossum. "An Introduction to Python for UNIX/C Programmers". Proceedings of the NLUUG najaarsconferentie 1993.

[WS91] Larry Wall and Randal L. Schwartz. *Programming Perl*. O’Reilly and Associates, 1991.

[YS93] Curtis Yarvin and Adam Sah. "Binary Optimization for Portable Runtime Code Generation in C." UC Berkeley Tech Report #UCB-CSD-93/792. December 1993.