DSC 2001 Proceedings of the 2nd International Workshop on Distributed Statistical Computing
March 15-17, Vienna, Austria
http://www.ci.tuwien.ac.at/Conferences/DSC-2001
K. Hornik & F. Leisch (eds.) *ISSN 1609-395X*

Compiling R: A Preliminary Report

Luke Tierney*

Abstract

This paper outlines an initial implementation of a byte code compiler for R. The compilation process is illustrated on a simple example. Semantic issues raised by the compilation process are discussed and sketches of the current virtual machine implementation and compiler design are given.

1 Introduction

This paper outlines an initial implementation of a byte code compiler for R. This implementation is intended as a proof of concept and a starting point for working out issues rather than as a final design.

The primary motivation for compilation is to improve speed of computation. But there are additional benefits. The R language semantics are currently defined primarily by the implementation. Writing a compiler forces the careful examination of these semantics and can help identify unusual or perhaps unintended features. Features that are difficult to compile often lead to programs that are difficult for human readers to understand. A compiler can also help to identify possible programming errors. An additional potential benefit is that compilation may make it easier to support some of the features currently under consideration for R, such as an improved exception handling mechanism or some form of light-weight thread support.

*School of Statistics, University of Minnesota. Research supported in part by grant DMS-9971814 from the National Science Foundation.

2 A Simple Example

The compiler generates byte codes for a simple stack-based virtual machine. As an example, here is a simplified version of the normal density function dnorm:

```
f<-function(x, mu=0,sigma=1)
    (1/sqrt(2 * pi)) * exp(-0.5 * ((x - mu)/sigma)^2) / sigma
```

This function does not support the log argument of dnorm and also does not check that the arguments are real. This function can be compiled with

```
fc<-cmpfun(f)
```

An assembly version of the resulting byte code is shown in Table 1.

LDCONST 0.398942280401433	push constant $1/\sqrt{2\pi}$ on stack
LDCONST -0.5	push constant -0.5 on the stack
GETVAR x	get value of x and push on stack
GETVAR mu	get value of mu and push on stack
SUB	pop top two values, subtract, and push result
GETVAR sigma	get value of sigma and push on stack
DIV	pop top two values, divide, and push result
LDCONST 2	push constant 2 on stack
EXPT	pop top two values x, y, compute and push x^y
MUL	pop top two values, multiply, and push result
EXP	pop top value x, compute and push e^x
MUL	pop top two values, multiply, and push result
GETVAR sigma	get value of sigma and push on stack
DIV	pop top two values, divide, and push result
RETURN	pop top value and return as function result

Table 1: Byte code for simplified normal density function.

The compiler examines the code and recognizes that it uses one global variable, pi, the global functions sqrt and exp, along with the arithmetic operators -, *, /, and ^. Since the function f is defined at top level, the variable pi is assumed to refer to the global constant for π in the base package. This means that the expression for $1/\sqrt{2\pi}$ is a constant expression and can be computed at compile time. Another constant expression that is evaluated at compile time is -0.5, which R parses as the unary minus function applied to $+0.5$. Similarly, the global functions used by f are assumed to refer to the arithmetic functions defined in the base package. These have special opcodes in the byte code virtual machine that allow them to be called more efficiently than generic functions.

Some simple timings can be obtained using

```
system.time(for (i in 1:100000) fc(x))
```

with analogous expressions for f and dnorm. Results are given in Table 2. In this case most of the improvement of fc over f comes from constant folding.

Function	`x = 1`	`x = seq(0,3,len=5)`
`f`	7.74	8.86
`fc`	3.49	4.42
`dnorm`	2.35	3.75

Table 2: Timings in seconds for normal density computation.

3 The Virtual Machine

Using a virtual machine with a byte code machine language has a long history going back at least to the UCSD Pascal system. It forms the basis of the Python [20] and Perl [26] runtimes and is also used in many Scheme implementations. The Java [13] and .Net [15] platforms are also designed around such machines.

The virtual machine targeted by the R compiler is a stack-based machine with a simple set of instructions. The external representation of byte code objects produced by the compiler, the representation that is used when a byte code object is stored in a workspace, consists of a vector of integers representing the instructions and a vector of generic R objects representing the constant pool. The runtime system contains defined numerical values for different opcodes such as the ones shown in Table 1. With defines like

```
#define RETURN 0
#define LDCONST 9
#define GETVAR 13
#define ADD 37
```

a function body of the form `x+3` would have its code object constructed by a call of the form

```
mkCode(code = as.integer(c(13, 0,  # GETVAR x  (x is const[0])
                           9, 1,   # LDCONST 3 (3 is const[1])
                           37,     # ADD
                           0)),    # RETURN
       const = list(quote(x), 3))
```

These code objects can be executed by a byte code interpreter. A simplified version of such an interpreter is shown in Figure 1. This interpreter has a certain amount of overhead in its `switch` loop. One way to avoid this is to use threaded code, where the internal representation of the instructions contains addresses to jump to rather than integers to be used in a `switch`. Threaded code interpreters cannot be implemented in portable C, but the GNU C compiler has extensions designed for this purpose. In particular, it is possible to store the address of a label in a variable, and `goto` can be given a variable containing such an address as its jump target. With careful use of macros one can write a single interpreter that will use threaded code when compiled with GNU C and a portable `switch`-based implementation when compiled with other C compilers. The macros used are based on those described by Piumarta and Riccardi [19].

```
SEXP bceval(SEXP code, SEXP env)
{
    int *pc = instructions(code);
    SEXP *consts = constantPool(code);
  loop:
    switch(*pc++) {
    case RETURN: return(popStack());
    case LDCONST: pushStack(const[*pc++]); goto loop;
    case GETVAR: pushStack(findVar(const[*pc++], env)); goto loop;
    case ADD: { SEXP y = popStack(), x = popStack();
                pushStack(do_Add(x, y)); goto loop; }
    ...
    }
}
```

Figure 1: A simple byte code interpreter.

4 Compiler Operation

Currently the compiler is written almost entirely in R. Only the function for creating an internal code object is written in C. The compiler walks the source tree and recursively emits code into a code buffer which is then collected to form the final code object. The operation of the compiler is driven by a table keyed on the names of global functions. When compiling a global function call, this table is checked. If the table contains an entry for the function, then that entry is called to generate code for the call. Otherwise, code for a generic call is generated. This approach makes it simple to gradually add optimizations for special handling of different functions.

The code generators for a class of basic arithmetic functions attempt to constant fold their arguments. If all arguments are scalar constants, then the function is called by the compiler and the resulting value is entered in the constant pool.

Almost all internal functions of class `SPECIAL` and many of class `BUILTIN` have corresponding opcodes in the virtual machine. The code generators for these functions just emit the appropriate opcode sequences.

Many R functions have R definitions that are simple wrappers around `.Internal` calls. An example is `dnorm`, which is defined as

```
dnorm <- function (x, mean = 0, sd = 1, log = FALSE)
    .Internal(dnorm(x, mean, sd, log))
```

For most of these functions the overhead of a generic call to these closures can be avoided by inlining. Thus a call of the form

```
dnorm(y, 2, 3)
```

is replaced by the call

```
.Internal(y, mean = 2, sd = 3, log = FALSE)
```

and then compiled. Once again, special opcodes are available for many `.Internal` functions. Unfortunately the choice of which internal functions to inline cannot be automated since inlining changes the depth of the call stack and a few internal functions rely on knowing the structure of the call stack.

There are a few other minor optimizations in the compiler, but the single recursive pass used in compilation limits the amount of optimization that can be done. The reason for using a single pass is to allow the compiler to be quite fast. This would be essential if we were to replace the interpreter entirely; in this case `eval` would first compile its argument expression into a function of no arguments and then call the function. This approach is used by Python and many other systems with simple byte code compilers. It may be essential if we decide to implement light-weight threads as discussed in Section 8.

5 Compiler Warnings

Compilation provides an opportunity to examine code for possible errors. Given the way lazy evaluation is used in R, it is difficult to determine with certainty whether an expression is intended to be evaluated or not. In most cases it would therefore not be appropriate to signal errors, but warning messages may be quite useful. The warning messages currently produced are based in part on some discussion of common programming errors on the R mailing lists in Fall 2000.

As an illustration, here is an example of a function that may contain some errors:

```
g<-function(x, exp = TRUE) {
    if (exp)
        exp(x+3) + ext(z-3)
    else
        log(x, bace=2)
}
```

Compiling the function produces

```
> cmpfun(g)
Note: global variables used: z
Note: possible error in log(x, bace = 2):
      unused argument(s) (bace ...)
Note: local functions used: exp
Note: undefined functions used: ext
```

The use of global variables, in particular undefined ones, usually indicates a mis-spelled identifier, so this produces a warning.

In current R programming practice, local variables rarely hold functions, so again a warning is given; in this case the logical argument `exp` is probably unintentionally shadowing the base function by the same name. The compiled code will still execute correctly by selecting the first function value named exp in the environment chain, but it will be less efficient since it cannot determine at compile time that the function used will always be the exp function in the base package. This particular kind of warning should be more refined and distinguish between cases where there are explicit local function definitions and when arguments or non-function local variables are used as functions and their names shadow defined global functions.

Uses of global functions with known definitions are checked against their argument lists and inconsistencies result in a warning. In this case it would appear that the base argument to the log function has been mis-spelled.

A warning is also given if a global function that is called does not have a definition. This can lead to many spurious warnings if a package is compiled unless the package is loaded before compilation.

Compiler warnings can be very useful in debugging, but care is needed to achieve the right level of warnings for different situations. Some mechanism for customizing and tuning the warnings that are produced would clearly be useful.

6 Timings and Performance

Current performance of the compiler and runtime system represents a lower bound on what can be achieved. Nevertheless, in a few cases results are already quite promising. For example, for some of the examples in the bootstrap package compiling the functions in the base and bootstrap packages cuts execution time in half.

As another example, recently the interpreted function lapply was replaced by a partially internal C version to improve performance. Byte compiling the interpreted version is able to recover about half of the improvement gained by creating a C version. There is some hope that with further improvements in the compiler and runtime system the byte compiled version will be close enough in performance to the C version to make a C implementation in this and similar situations unnecessary.

There is, however, still considerable room for improvement. Some time ago Radford Neal posted several small benchmarks to one of the statistics usenet news groups. One of these is shown here:

```
test5 <- function (n) {
    k <- 0
    for (j in 1:n) {
        v <- c(1,2,j,j,3)
        if (v[2]==2) k <- k+1
    }
}
```

As with all benchmarks these need to be taken with a grain of salt; this particular one may be useful for assessing performance for certain applications where C or FORTRAN style looping is used heavily. Table 3 shows run times for these benchmarks when interpreted and compiled. Also shown are analogous results for XLISP-STAT [25] on the same machine. For both test3 and test5 compilation provides a substantial improvement, but in both cases the XLISP-STAT compiler

		R	R	XLISP-STAT	XLISP-STAT
	n	interp.	compiled	interp.	compiled
test2:	10000	4.54	4.50	4.73	4.56
test3:	10000	8.34	3.11	2.20	0.43
test4:	1000	2.07	1.60	2.07	1.92
test5:	100000	4.24	2.13	2.19	0.66

Table 3: Execution times in seconds for Radford Neal's benchmarks.

and runtime are able to do considerably better. Exploring the reasons for this gap should help in improving the R compiler and runtime performance.

One area where the current implementation is definitely lacking is in looking up variable values. Currently this involves a linear search through environments. With appropriate semantic conventions it should be possible to determine the positions of variables in the environment at compile time. This would allow variable values to be found with one or two array references.

Another area for possible improvement is function call conventions. Compiled code now uses the interpreter's calling mechanism, which involves a large amount of intermediate storage allocation. Improvements, in particular for simple calls with no named arguments, should be possible.

7 Semantic Issues

Up to now the discussion has glossed over an important point: There are semantic differences between compiled code and interpreted code. The main difference is due to the fact that new variable bindings can be created using `assign` and removed using `rm`, both in a function's own frame and in frames of its callers. As a result, it is impossible to determine with certainty whether a variable is global or local. For example, in the top level definition

```
f <- function(x, z) { g(); if (z) x + 1 else log(x) }
```

it is likely that the variable `z` in the body is local and that the variable `log` is global. However, the function `g` *could* reach into it's callers environment, add a local function called `log` and remove the local variable `z`. Then during the execution of the `if` expression `z` would be global and `log` would be local.

In order to reliably identify global functions the compiler needs to be able to determine which variables are local, or at least which ones might be local. Currently the compiler assumes that no new variables will be created in local environment frames by explicit calls to `assign`. To make this approach legitimate it might be useful to develop a concept of sealed environments that can be applied to both interpreted and compiled code. A sealed environment would only be allowed to contain bindings for an explicitly specified set of variables. It need not contain bindings for all of these variables—some can be marked as undefined. Sealed environments could use a vector representation that would allow fast access to their variable values.

Sealing the environment of a function would involve a code analysis that identifies as potential local variables all argument names and all names that appear on the left hand side of assignment expressions in the function body. An interpreted function could then in principle be marked as having a sealed environment, and the compilation process described here would then produce a compiled function with identical semantics to a sealed interpreted one.

Once we have identified a function variable `log`, say, as a global variable, we still cannot safely conclude that it refers to the `log` function in the base package. The base package is always at the end of the search list. It is in principle possible for a package `foo` loaded with the `library` command to define its own `log` function. Interpreted code that calls `log` will then call the function `log` in `foo` instead of the one in the base package. Occasionally this can be useful, but in most cases, especially if the code calling `log` is itself in a different package, this is not what is wanted. To avoid this form of name clash among global variables it would be useful to develop some form of name space management system for R. Most languages now include some form of name space mechanism. Java, Perl and Tcl [27] have fairly simple systems. The ML language family has a very rich mechanism [14]. The Scheme48 [21] and MzScheme [9, 10, 8] module systems (MzScheme calls them units) are somewhere in between. These should provide a basis for designing a name space mechanism for R.

Providing mechanisms for sealing environments and name space management will make it easier to compile R code without affecting the semantics. But it will also make it easier for human readers of R code to reason about what the code will do. If a function is defined within a particular name space and declared to have a sealed environment then the meanings of its variables are clearly determined. Without sealing and name spaces readers can have a pretty good idea of that the variables are intended to mean, but must always keep in the back of their minds that strangely behaving callees and unusual loaded packages could change things completely. This is a good example of a case where a feature that complicates compilation also complicates human readability of code.

A few other issues are related to non-local exits and their interaction with lazy evaluation. Currently compiled and interpreted code behave quite similarly in these settings. But developing a compiler brings these setting into clear relief and raises the question whether the current behavior is in fact what it should be. As one example, consider the expression

```
for (i in 1:n) f(break)
```

Currently, for both interpreted and compiled code, if `f` is a closure and `f` evaluates its argument, then the interpreter will signal an error (unless `f` is interpreted and the argument evaluation happens to occur inside a loop). The compiler issues a warning when it encounters this. Similar issues arise for explicit `return` calls that occur in function arguments and for `break` calls in functions defined within a loop.

This may seem like an esoteric issue, but it does potentially have some serious consequences. The `switch` function is implemented as an ordinary function. It might at times be useful to use `switch` in a loop and have some of the `switch` clauses call `break` to exit from the loop. Currently this will not work in interpreted or compiled R code. To address this there is a need to examine what the semantics of a `break` call should be in this context. Common Lisp [24] and Dylan [23] both have very elaborate non-local exit mechanisms that give meaningful semantics to exiting from a loop from a procedure called within the loop. These can serve as a model for R as well.

8 Future Directions

In principle the byte code produced by the compiler could be used as intermediate code for generating C code. Whether the resulting C code would be efficient enough to be worth the effort needs to be investigated. It would however be interesting to compare this approach to the approach taken by Calder to compile a limited subset of the S language to C [5].

Other approaches to machine code generation may be worth investigating. One interesting approach presented by Piumarta and Riccardi [19] is to create code for fused byte code instructions by copying bits of code from the byte code interpreter. This approach is particularly well suited for distributed computing where the byte code representation can be used as the portable, machine-independent representation used in code migration, and can then be transformed into more efficient, machine dependent code once it reaches its execution platform [3].

The basic design of the virtual machine is a stack architecture. This seems to make more sense for a single pass compiler for R than the register architecture used in the XLISP-STAT compiler. It should also simplify compiling to the Java or .NET virtual machines. On the other hand, the stack architecture does complicate generating C code and also may be less suited for certain optimizations.

Name space management is an important issue both for more reliable development of packages and for clarifying compilation semantics. Clarification of the issues in this area is a major near-term objective.

Adding a declaration mechanism of some form may allow the compiler to generate more efficient code. Specifying that certain variables are scalars may allow some expressions to be simplified. It may also help identifying certain looping idioms, such as

```
for (i in 1:n) ...
```

where generating the sequence `1:n` can be avoided. Being able to declare a function as strict, i.e. as forcing the evaluation of all its arguments, may allow a more efficient function calling mechanism to be used.

As the compiler is developed further some consideration needs to be given to the appropriate set of techniques to use. The compiler methodology used so far is quite simple and very much in the spirit of the simple Scheme compiler presented in Norvig [16, Chapter 23]. The more sophisticated XLISP-STAT compiler is based on techniques for compiling strict, mostly-functional languages as described for example by Appel [1]. It is not clear whether this is the right direction to go for R as well. Since R uses lazy evaluation for function arguments, methods for compiling non-strict languages like Haskell may be useful [2, 12, 18]. On the other hand, R is mostly strict since local assignments to variables and data structures and creation of data structures all force evaluation. R is also in many ways a very dynamic language, and it is likely to get a more object-oriented flavor in the near future, so techniques developed for Self and other object-oriented languages may be appropriate as well [11, 7]. One unusual feature about all statistical languages is that they deal heavily with vector data. Once again, specialized compilation techniques and perhaps language extensions from languages like NESL [4] may be useful.

An issue currently under discussion is whether support for some form of multi-threading can be added to R. One direction taken by many high level languages is so implement very light-weight user-level threads [17, 22]. This requires being able to save and restore the current computational state of a thread of execution. If all R code is compiled, then the state is captured by the current stack, which in R could easily be implemented as a linked list of frames, the currently executing code object, and the index of the current instruction. This makes capturing and switching the state of execution considerably easier than it is for an interpreter that recursively calls a C evaluation function.

This paper and this research project have concentrated on the R language. Some of the implementation issues are specific to R, but others carry over more generally to the S language [6] and might be used in developing a compiler for the S-plus implementation.

One longer term objective to explore is whether it is possible to design a runtime system that can execute both R and XLISP-STAT code, and whether it is possible to design a runtime system abstraction that can be implemented either in C or on the Java or .NET platforms.

9 Implementation and Availability

The changes in R source code needed to support the byte code compiler are fairly minimal (at least compared to the recent memory management changes) but since they are very likely to change substantially they have not yet been merged into the R CVS tree. Snapshots will be made available periodically until the design is stable enough to be added to the CVS tree.

References

[1] Andrew W. Appel. *Compiling With Continuations.* Cambridge University Press, 1992.

[2] Andrew W. Appel. *Modern Compiler Implementation in C.* Cambridge University Press, 1998.

[3] Carine Baillarguet and Ian Piumarta. An highly-configurable, modular system for mobility, interoperability, specialization, and reuse. http://www-sor.

inria.fr/publi/HCMSMISR_ecoop99.html, 1999. 2nd ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS'99).

[4] Guy E. Blelloch. NESL: A nested data-parallel language (3.1). Technical Report CMU-CS-95-170, Carnegie Mellon University, September 1995.

[5] Matt Calder. Scompile. http://people.ne.mediaone.net/mncalder/scompile/scompile.html.

[6] John M. Chambers. *Programming With Data: A Guide to the S Language*. Springer Verlag, 1998.

[7] Jeffrey Dean, Greg DeFouw David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 83–100, October 1996.

[8] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional programming*, pages 94–104, September 1998.

[9] M. Flatt and M. Felleisen. Cool modules for HOT languages. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 236–248, June 1998.

[10] Matthew Flatt. PLT MzScheme: Language manual. http://www.cs.rice.edu/CS/PLT/packages/doc/mzscheme/index.htm, August 2000.

[11] Urs Hölzle and David Ungar. A third-generation Self implementation: Reconciling responsiveness with performance. In *Proceedings of the ACM OOPSLA '94 Conference*, Portland, OR, October 1994. ACM.

[12] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[13] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000.

[14] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3), 2000.

[15] Microsoft. MSDN online .NET developer center. http://msdn.microsoft.com/net/.

[16] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, 1992.

[17] S. L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages*, pages 295–308, St Petersburg Beach, Florida, January 1996. ACM.

[18] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Phil Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Thechical Conference*, Keele, UK, 1993.

[19] Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 152–161, June 1998.

[20] The Python language. `www.python.org`.

[21] Jonathan Rees. Another module system for Scheme. `http://www.eecs.tulane.edu/www/Jennings/apla/module.ps`, 1994.

[22] John H. Reppy. *Concurrent Programming in ML*. Cambridge Univ Press, 1999.

[23] Andrew Shalit, David Moon, and Orca Starbuck. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, 1996.

[24] Guy L. Steele, Jr. *Common Lisp: The Language*. Digital Press, 2nd edition, 1990.

[25] Luke Tierney. *Lisp Stat: An Object Oriented Environment for Statistical Computing and Dynamic Graphics*. Wiley, 1990.

[26] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, 3rd edition, 2000.

[27] Brent B. Welch. *Practical Programming in Tcl and Tk*. Prentice Hall, 3rd edition, 2000.