RICE UNIVERSITY

RCC: A Compiler for the R Language for Statistical Computing

by

John Garvin

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE
Master of Science

APPROVED, THESIS COMMITTEE:

John Mellor-Crummey, Senior Faculty Fellow, Chair
Computer Science

Bradley Broom, Research Scientist
Computer Science

Keith Cooper, Professor
Computer Science

Ken Kennedy, Ann and John Doerr Professor in Computational Engineering
Computer Science

Walid Taha, Assistant Professor
Computer Science

Houston, Texas
May, 2004
© Copyright
John Garvin
2004
RCC: A Compiler for the R Language for Statistical Computing

John Garvin

Abstract

R is a programming language for statistics that enables users to express computation at a high level of abstraction. Until now, its only implementation has been the R interpreter. Though interpretation is convenient for interactive use, it hampers the performance of computation-intensive programs. This thesis describes the design and implementation of RCC, a compiler that translates R into C to improve performance and enable future optimization.

RCC uses runtime libraries of the open-source R interpreter, combining compiled and interpreted code to achieve a complete translation. Function definitions and control flow in R are translated directly into C, while operations such as dynamic function modification remain interpreted.

RCC-generated code in the current version achieves over a threefold speedup compared to the R interpreter. Hand-coded experiments suggest that optimizing the generated code using knowledge about the runtime libraries could improve performance by a factor of 100.
Acknowledgments

I am deeply grateful to my advisor John Mellor-Crummey for his guidance and inspiration. Much thanks also go to Ken Kennedy and Keith Cooper for their knowledge, time, and energy. In addition to much helpful advice, Walid Taha provided a valuable theoretical perspective, for which I am grateful.

Bradley Broom contributed an astonishing amount of his time and energy to provide ideas, implementation help, and valuable knowledge. Without him this thesis would not have been possible.

Gracious thanks also go to Cheryl McCosh for providing useful comments on early versions of this work.

Finally, I gratefully thank my family—steadfast as always—and my friends, whether across the hall or across the country, for their constant support and encouragement.
Contents

Abstract i
List of Tables v

1 Introduction 1
1.1 Thesis ........................................ 1
1.2 Motivation ...................................... 2

2 Background 5
2.1 The R Language .................................. 5
   2.1.1 Functions in R ............................. 6
2.2 The R Interpreter ............................... 8
2.3 Garbage Collection ............................. 11

3 Related Work 13
3.1 Compiling R-like Languages .................. 13
3.2 Other Related Work ............................ 14

4 Methods 16
4.1 Translating R into C ............................ 16
4.2 Implementation of RCC ....................... 18
   4.2.1 Function Definitions ..................... 18
   4.2.2 Protecting Objects From the Garbage Collector .......................... 19
   4.2.3 Avoiding Redundant Creation ............. 20
4.3 Optimizations .................................. 21
   4.3.1 Stack Allocation of Lists ............... 21
   4.3.2 Direct Function Calls ................... 23
5 Experiments and Results  26
  5.1 Understanding Performance ..................................  26
  5.2 Performance Results ..........................................  27
  5.3 Code Shape .....................................................  29

6 Future Work ......................................................  31
  6.1 Improving Performance .........................................  33

7 Conclusions ........................................................  35

A Definition of Translation ........................................  36
  A.1 Grammars for R and C .........................................  36
  A.2 Reconstruction .................................................  38
  A.3 Translation .....................................................  39

Bibliography ..........................................................  45
Tables

5.1 Performance of variants of RCC relative to that of the R interpreter. Values greater than 1 are speedups over the R interpreter; values less than 1 represent decreased performance relative to the interpreter. "Heap and "local" describe the allocation of R objects: "heap" denotes allocation of objects using the R interpreter; "local" means local allocation is used when possible. "Closure" and "direct" refer to function calling: "closure" means all calls to compiled functions are made using R's mechanism; "direct" means that applicable functions are called directly.
Chapter 1

Introduction

R is a programming language that enables users to express statistical calculations at a high level of abstraction. It is used to solve many kinds of statistical problems. Researchers at the M.D. Anderson Cancer center use R to analyze clinical trial data. Analysts in the U.S. Department of Defense use the S language, a predecessor of R, to analyze intelligence data. Online R resources such as the Comprehensive R Archive Network [9] and Bioconductor [6] contain hundreds of R packages contributed by volunteers around the world. The standard implementation of R, an open-source interpreter, provides rapid prototyping of statistical calculations. For computation-intensive programs, however, the performance penalty of interpretation is substantial. One typical R program from M.D. Anderson, when interpreted, is slower by a factor of about 100 than an equivalent program written in C [7]. An optimizing compiler for R would enable users to express calculations at a high level in R while achieving performance comparable to that of C. This thesis describes RCC, a compiler that translates R into C with the aim of improving performance and establishing a basis for future analysis and optimization.

1.1 Thesis

My thesis is that a compiler for R can improve performance over that of the R interpreter and serve as a basis for further optimization. To support this thesis, I have created RCC, a program that translates R into C to improve performance and enable future optimization.
1.2 Motivation

An example illustrates some inefficiencies in the interpreter's actions that can be improved with compilation. Consider the following piece of R code, which defines and calls a function:

```r
foo <- function(x) x + 2
a <- 1
foo(a)
```

This code defines a function named "foo" that returns the sum of its argument and 2; binds the variable "a" to 1; and outputs the application of foo to a. The implementation of the function application foo(a) exhibits some of the inefficiencies in the R interpreter. After the definition of foo and a, the interpreter evaluates foo(a) in the following way:

- R looks up the foo function in the current R environment. An environment is implemented in the R interpreter as a dictionary where variables can be looked up to find their values. In this case, lookup of foo returns a function definition.

- R constructs a promise—an R object representing delayed computation—to hold foo's argument, a, to be evaluated later. In R, a function is called before its arguments are evaluated; arguments are evaluated only if they are needed, at the time they are referenced. R's unusual function call semantics are discussed in Section 2.1.1.

- The environment of foo is extended to create a new environment in which the formal argument x is bound to the promise containing the argument a. The foo function body, x + 2, is evaluated in this new environment.

- The expression x + 2 is evaluated. During evaluation, the argument x is looked up in the new environment, causing the promise to be evaluated.

- The promise is evaluated. The variable a is looked up in the caller's environment, returning its value, 1.
- R's internal addition function adds the objects representing 1 and 2. The `foo` function returns the answer, 3.

There are many reasons this process is much less efficient than the equivalent program in C or Fortran would be:

- The function application and the function body are constructed as large data structures that must be interpreted, in contrast to machine code produced by a compiler.

- Constructing the promise to delay evaluation of the argument is unnecessary in this case; since the argument is referenced in the function body, and its evaluation is a simple variable lookup with no side effects, the output is the same whether the argument is evaluated before or after the function call.

- R's mathematical functions such as the addition function are designed to work on vectors and matrices as well as scalars. They also check for missing or erroneous values whether or not such a check is necessary. Checking for these possibilities takes extra time when the variables involved are known to be bound to legal values.

- Storing and looking up variables in explicit environments is time-consuming. In compiled code, it is desirable to store variables directly in registers and memory, where they can be accessed more quickly.

These inefficiencies in R evaluation may not all be eliminated by a compiler, but they point out the opportunities for a compiler to improve interpreted R code.

Presently, the most common way to improve performance of an R program is to rewrite and optimize it by hand in a lower-level language such as C or Fortran. This can result in a large performance improvement, but it requires a great deal of effort on the part of the programmer [18]. Hand optimization also makes code much harder to understand, reuse, and modify.
The R interpreter is open-source and available online [29]. This presents an opportunity to use parts of the R interpreter code to form RCC. RCC generates compiled code that operates using the interpreter’s runtime library functions, including object construction, mathematical operations, and variable lookups. Additionally, internal R functions are used to construct and evaluate R syntax trees to produce correct code when compilation is difficult or infeasible.

RCC will eventually be part of the Telescoping Languages project [18]. This project aims to improve both performance and compile time when compiling code that uses domain-specific libraries. The large base of user scripts and statistical libraries written in R makes an attractive target for compilation in a telescoping context.

Chapter 2 provides relevant background knowledge of R internals. Chapter 3 discusses other work related to RCC. Chapter 4 details the specific methods of the translation. Chapter 5 discusses experimentation and performance results. Chapter 6 discusses future directions for R compilation, and Chapter 7 concludes.

\[^{1}\text{R is released under the GNU General Public License.}\]
Chapter 2

Background

RCC uses parts of the R interpreter to compile the R language. Before describing RCC, then, it is useful to discuss the R language and the workings of the R interpreter, with particular emphasis on the knowledge required to understand the workings of RCC.

2.1 The R Language

R is a high-level programming language for statistical calculations designed to enable programming at a high level of abstraction [5]. R users do not have to handle many of the low-level details that users must cope with when programming in languages like C and Fortran. In R, vectors, matrices, and the operators that work with them exist as primitives. Matrix-matrix multiplication, for example, can be expressed with a single operator in R, whereas it would typically be written as a nested loop in C or Fortran. The R language and interpreted environment, like its predecessor, S, facilitates open-ended interactive use [4, 16]. R also includes a vast library of powerful functions for statistics, making R a popular environment for statistical programming.

R shares many similarities with a more well-known language, Matlab [14]. Like R, Matlab has built-in support for multidimensional array and matrix computations. It includes many special-purpose libraries that define domain-specific languages. Matlab is designed for general-purpose mathematics, while R is designed specifically for statistical calculations. Since S was invented in the 1970s, S and R have been more popular among statisticians.

R is a descendant of a language called S and its interpreted environment [4]. S is also part of a commercial interpreter and environment called S-PLUS, a product of Insightful Corporation [10]. The variant of S defined by the R interpreter includes
some changes and improvements to those defined by S-PLUS and the original S interpreter. Because of these differences, I use "R" to refer to the dialect of S defined by the R interpreter as well as the interpreter itself. Despite their differences, however, R and S-PLUS share enough similarity that most of the ideas in RCC could be applied to compilation of S-PLUS as well.

The most important difference is the treatment of scoping. S and S-PLUS implement simple two-level scoping semantics, in which every variable is either global or local to its closest enclosing function. This means that nested functions in S act as if they were defined at the top level. R implements true lexical scoping, in which the hierarchy of variable scopes matches the hierarchy of nested functions.

Though R implements lexical scoping by default, R's flexibility gives it ways to circumvent the default scoping rules. Given a function, it is possible for the programmer to access and modify any environment in the chain of lexical scopes enclosing that function's definition. R even allows access and modification of environments of functions in the call stack, actions normally allowed only with dynamic scoping. These capabilities complicate the task of compilation.

2.1.1 Functions in R

Functions in R have many interesting features. R supports named arguments, a feature found in Fortran 90 and other languages. Instead of supplying arguments in a single defined order, a call to a function can supply arguments in any order by specifying the formal arguments to which they are bound. Function arguments can also have default values supplied in the function definition: a function can be called without supplying the argument, and the default value will be used instead. The following example illustrates named arguments and default values:
foo <- function(a, b=0, c=1, d=FALSE) [definition]

foo(10,1,3,TRUE)
foo(c=1, d=TRUE, b=3, a=100)
foo(50)
foo(12, c=100)

All four function calls above are valid calls to foo.

A function definition can include the symbol "..." at the end of the list of formal arguments to denote a variable number of arguments, similar to C functions that use the varargs library. In addition to the explicit arguments, a function call can supply zero or more arguments in the position of the "..." object. Inside the function definition, the variable args is automatically defined as a list of the actual arguments that were mapped to the "..." object.

Unlike most languages, R uses normal order when evaluating function calls. In other words, actual arguments are evaluated lazily. Normal-order evaluation is used in pure functional languages such as Haskell [17]. Actual arguments that are passed to a function, instead of being evaluated before the function call, are only evaluated if and when they are used inside the function. If a formal argument is never used in the function body, evaluation never occurs. If the argument does appear in the body, the argument is evaluated in the caller’s environment at the time the argument is referenced. Evaluation of each argument happens at most once in a function call; once an argument is evaluated, the resulting value is used for all subsequent uses inside the function. This behavior is commonly called call-by-need semantics [1].

The current draft of the R language definition [30] erroneously states that R implements call-by-value semantics. In true call-by-value semantics, arguments would be evaluated before a function call. What the authors appear to mean is that R does not implement call-by-reference. In other words, modifying an argument in a function does not change the value of the corresponding variable in the caller.

Call-by-need semantics are normally discussed in the context of pure functional languages such as Haskell, which perform computation without mutation. In this
context, since all functions given the same arguments return the same result, call-by-need is simply an optimization of call-by-name, in which an argument is evaluated each time it is referenced. In R, in which mutation is a central feature, call-by-need results in different semantics than call-by-name.

R is unusual in that its functions are non-extensional. An extensional function can be differentiated from another function only by applying it to arguments. It is an opaque object whose inner structure is hidden. Functions in most languages are extensional. R functions, by contrast, are objects with separable, modifiable components. The list of function arguments, the associated environment, and the body of the function can be accessed and modified. In R parlance, treating code objects as data is called "computing on the language." This aspect of functions presents obvious difficulties in compilation, since the structure of the components of functions is outwardly visible in addition to the behavior. RCC's solution is described in Section 4.2.1.

R includes interfaces to outside code. The R function .External is an interface to external code written in C. The programmer supplies the name of the C function and the arguments to pass. The function dyn.load loads a dynamic library. RCC uses these features to create compiled code that can be executed from within R.

2.2 The R Interpreter

The R interpreter is composed of two principal parts: a parser and an evaluator. The parser analyzes an R program and produces a code object representing the syntax tree of the program; the evaluator then traverses the code object and performs the appropriate computation. Both RCC and the C programs produced by RCC have much in common with the R evaluator, especially in the internal functions and data types used. R uses a data type called an SEXP, a variation of S-expressions used in LISP [13], to represent both code and data. In fact, almost all objects in R are represented as SEXP. The output of the R parser is one SEXP representing the input program; RCC analyzes this SEXP to determine the C code to produce. The C code generated by RCC constructs SEXP and uses R's internal libraries to perform
computation on SEXP s. It is instructive, then, to examine the structure of SEXP s in order to understand the workings of RCC.

Internally, SEXP is a C data type defined as a pointer to a structure called SEXPREC. The first field of an SEXPREC structure is an integer tag that encodes the type of the R object. R types include scalar and vector data (integer, floating point, complex, and Boolean), lists, R code, environments, closures, and an assortment of special objects such as the “...” object. An SEXPREC structure also includes an ATTRIB field. This is used to attach arbitrary extra information to objects in the form of attributes. An ATTRIB field is implemented as a (possibly empty) list of symbolic attribute tags bound to values. Attributes are used to implement object-oriented programming, to give dimensions to matrices, and for other purposes.

Other components of an SEXPREC are used for various purposes according to the type of the object. These can contain data or other SEXP s (pointers to other SEXPREC structures). An SEXP is said to have type $T$ if it points to an SEXPREC structure whose type field is a code for $T$.

SEXP s that represent data vectors can be integer, real (floating-point), Boolean, or complex. The SEXPREC structure of an SEXP of vector type contains a pointer to the array. Scalar values are represented as arrays of length 1.

An SEXP of symbol type represents variables and other names. A symbol’s SEXPREC structure contains two SEXP s: a pointer to a string representation of the symbol’s name and a pointer to the value to which the variable is bound, if the symbol is a variable bound to a value.

Another type of SEXP is an environment, which enables lexical scoping. Each environment contains a frame, a dictionary mapping a set of variables to their values. Variables in the R runtime library are defined in one global environment. Every other environment includes a pointer to a parent environment. The environments of functions defined at the top level point to the global environment; nested functions’ environments point to the environments of their enclosing functions. When a variable is looked up in an environment, R first tries to find the variable’s binding in the current
frame. If the variable is not found in the frame, R traverses the chain of environment pointers until either the variable’s value is found or the chain ends, making the variable unbound.

One very common SEXP is a cons cell. Just as in LISP [13], a linear chain of cons cells form a list. Each cons cell includes a CAR and a CDR, the CAR pointing to an element of the list and the CDR pointing either to the next cons cell in the list or to a nil value signifying the end of the list. (The nil value in R is a separate SEXP type with a single element named R_NilValue.) In R, unlike LISP, each cell also contains a TAG field used for names. The TAG field gives a name to the CAR of the cell. TAG fields are used, for example, when passing named arguments to a function.

Code objects are SEXP’s that represent pieces of R code. Atomic code objects can be variables (represented as symbols) or constants (represented as strings, real values, or boolean values). Larger code objects, representing function calls and syntactic structures, are recursive. They are encoded as lists using special “language” cons or lcons cells; these are like ordinary cons cells but have a different code in the type field. A list with an lcons as the first cell is a language list; it represents a function application or syntactic construct. The CAR field of the lcons points to the function being called, in the form of a closure or a symbol naming the function. The CDR field is a list containing the arguments of the function. For example, the representation of the function application foo(a, 12) in R might be constructed by a process similar to the following pseudocode:

```r
lcons('foo, cons('a, cons(12, ())))
```

where 'foo and 'a are symbols, 12 is a real constant, () is the nil value, and the lcons and cons constructors create an lcons cell and a cons cell, respectively. We can use comma-separated elements in brackets to denote a list whose first cons cell is an lcons. In this notation, the same object can be represented like this:

```
['foo, 'a, 12]
```

Syntactic constructs, such as if statements, sequences of expressions, and subscript expressions, are considered functions in R; expressions using these constructs
are represented internally as language lists whose arguments are code objects. To illustrate a larger example, consider the following for-loop in R:

```r
for(i in 1:20) { a <- 12 ; foo(a)}
```

This expression is parsed to produce the following code object in the form of nested language lists (in bracket notation):

```
["for", "i", [":", 1, 20], ["{", ["<-", "a", 12], ["foo", "a"]]]
```

Internally, for is an SEXP of symbol type; it is a variable bound to a function that takes three arguments: the loop variable, the range of values the loop variable takes, and the code object representing the interior of the loop. Functions such as foo and syntactic constructs such as for both appear internally as symbols bound to functions at the head of a language list containing their arguments.

The output of the R parser is a single SEXP that represents the entire program input. The most general evaluation function is named `eval`; it looks at the type of an SEXP to determine the appropriate action, usually calling another function designed to handle a specific type of SEXP. When an SEXP contains one or more code objects, as do language lists, `eval` is called recursively.

### 2.3 Garbage Collection

R uses a generational mark-sweep garbage collector [20, 2]. Storage for R objects is divided into a series of spaces called *generations* according to their age. Each object is tagged with a *generation number* that indicates its generation. When objects are allocated, they are placed in the youngest generation, which is collected the most often; after an object lives through some number of collections it becomes part of the next older generation. Each generation is implemented as a circular doubly linked list. Each R object contains two pointers used to link to the next and previous objects in a generation. Objects are added to and removed from generations by changing these
pointers; in this way, collection can proceed without moving data\(^1\).

The collector starts at the set of roots—the objects through which pointers can be followed to reach all known R objects. These include global expressions, the current expression, objects on the R function call stack, etc. The collector traverses compound data structures, setting a mark bit on each object found. All unreachable objects are garbage; their storage can be reclaimed.

Outside code that does not use the R allocation mechanism, such as RCC output, would ordinarily be unreachable from the roots and thus collected as garbage. To avoid memory corruption, the code must explicitly inform the garbage collector that data is to be preserved. A stack of explicitly protected objects is included in the roots. A macro called `PROTECT` pushes its argument onto the protection stack. `UNPROTECT_PTR` removes the given object from the stack, releasing it from protection when it is no longer being used. Another macro, `UNPROTECT`, is more commonly used in small C functions written by users to interact with R. It pops a given number of objects from the top of the protection stack. `UNPROTECT_PTR` is used in code, including parser functions and RCC, that constructs recursive objects. In such code, an object is often built from sub-objects. While the larger object is protected at the top of the stack, the sub-objects underneath must be unprotected. `UNPROTECT_PTR` must be used instead of `UNPROTECT`, which would remove the larger object as well.

\(^1\)R objects are also divided into classes by type and size, in addition to the division into generations by age; each class has a separate hierarchy of generations implemented as circular linked lists. The class distinction is not important for this discussion.
Chapter 3

Related Work

Many other projects have compiled languages similar to R. This chapter discusses these predecessors and other work related to RCC’s task.

3.1 Compiling R-like Languages

Bartlett implemented a compiler for Scheme that uses C as its target language [3]. Bartlett’s compiler can translate nearly all of Scheme’s features, including function calls with optional arguments, initialization of variables in the runtime environment, nested function environments, and continuations. Bartlett mentions one source of incompleteness in the Scheme-to-C translation: Scheme requires optimization of tail calls to avoid stack overflow, but the C functions produced by the compiler implement tail calls using the stack. Bartlett does not mention any special handling of Scheme’s intentionality (reflection or coding on the language), whereas RCC’s translation specifically handles R’s reflection capabilities. Other Scheme compilers and optimizers use ideas applicable to optimizations done in RCC [19, 8].

Matlab compilation is a rich subject area. The mcc compiler, distributed with the Matlab interpreter, translates Matlab scripts into C code that makes calls to internal Matlab functions [21]. It is not intended mainly as an optimization tool; rather, it is intended as a way to create stand-alone applications from Matlab scripts. Beginning with version 6.5, Matlab has included a bytecode compiler that optimizes control flow, memory allocation for arrays, and calling of built-in functions [14]. This improves performance of loops involving scalar computation, small arrays, and built-in function calls. De Rose and Padua [11, 24] developed the FALCON compiler. It translates Matlab to Fortran 90 using static and dynamic type inference to specialize array computations for performance. McCosh [22] translated Matlab linear algebra
codes to FORTRAN or C. Type inference was used to produce specialized versions of user code and libraries rather than performing total inlining. One of the goals of RCC is to support similar type inference in R. Plans for future work are discussed in Chapter 6.

Tierney [31] implemented a compiler of a subset of R into a stack-based bytecode. This compiler performs some optimizations, such as inlining for R functions that are wrappers to C function calls. Tierney describes the possibility of variable lookup without explicit environments and the difficulty of such optimizations with R's semantics. Currently, there is no reliable way to guarantee that a variable reference will refer to a specific definition. R's reflection mechanism allows a user to change the environment of a closure after it is defined. Furthermore, the package system in R works in such a way that even references to variables defined in the global environment cannot be assumed to remain the same. Tierney [31] proposes ideas to change the R language to allow better compilation: scaled environments, which can be altered only from the function associated with it in the closure; and a reworking of the R package system.

3.2 Other Related Work

Stichnoth and Gross [27] used code composition as a way for compilers to generate code for high-level operations such as parallel array operations. Their system uses code templates, which produce code variants based on characteristics of input code. For example, one code template generates a nest of for loops given the number of levels, the loop bounds, and the inner loop code. The set of output functions in RCC that consume an SEXP and output a block of C code are analogous to code templates.

LISP-like lists are the predominant data structure in R and RCC; they are used for both data and code. A number of ways to improve list allocation have been proposed. Shao et al. [26] proposed list unrolling to reduce the number of pointer dereferences in list accesses. McDermott et al. [23] proposed a variety of ways to improve list allocation in LISP, including allocating on the stack when possible instead of the heap. This is part of the "local allocation" optimization performed in RCC.
Multi-stage languages [28] are designed to support writing *meta-programs*: programs that manipulate other programs. Multi-stage languages are designed to be type-safe themselves as well as guaranteeing syntactic soundness and type safety in all object programs generated. Multi-stage programming is made possible by type-safe meta-languages such as MetaML and MetaOCaml and type-safe object languages such as ML and OCaml. Applying multi-stage language theory directly to RCC is problematic because both the C representations of R code generated by the parser and the object language, C, are type-unsafe. A hypothetical language Meta-C that could generate C code that is guaranteed to be correct would be a good candidate for an RCC implementation language, where “correct” C code is defined as both syntactically correct and subject to some reasonable set of weaker typing requirements. Instead, RCC uses encapsulation and careful design of output functions to achieve type-safety and correctness as much as possible in the output code, though correctness is not formalized as in MetaML or MetaOCaml.

RCC is designed for use with a compilation strategy called Telescoping Languages, proposed by Kennedy et al. [18]. Library-based languages, such as Matlab and R, require interprocedural analysis for good performance, but recompiling large libraries along with user scripts greatly increases compilation time even for small user programs. In a telescoping system, library functions are compiled into multiple variants corresponding to specific contexts in which they are called. When a user script is compiled, the compiler optimizes with respect to the optimized library. This idea can be iterated with multiple libraries built on one another; this is the source of the term “telescoping.” R’s heavy use of libraries makes it a good candidate for optimization in a telescoping context.
Chapter 4

Methods

This chapter covers the definition of the translation of R into C and the design and implementation of RCC. A formalized translation appears in Appendix A.

4.1 Translating R into C

Translation is the process by which RCC translates a code object into C code. Part of the process is the related process of reconstruction. Given an R object, reconstruction generates C code that, when compiled and executed, will construct an identical R object. Reconstruction is useful for code objects for which compilation has not yet been implemented or is infeasible. While compilation is often complicated to implement, reconstruction can be done relatively easily on any object. RCC performs reconstruction by generating calls to the constructor functions used internally by the R parser. Reconstructed code objects are typically passed to library functions that call eval, R’s evaluation function, to evaluate the object.

For reconstruction of atomic objects, a call to a constructor is generated. Calls to the constructor functions ScalarReal, ScalarLogical, mkString, and install are generated to reconstruct numerical constants, logical (Boolean) constants, strings, and symbols respectively. The nil value is represented as the constant R_NilValue.

To reconstruct a recursive object, RCC performs a postorder traversal of the object. Code is generated to reconstruct each sub-object and assign it to a temporary variable. The original object is constructed by generating a call to the appropriate constructor, using the temporary variables representing its sub-objects. This separation, with each object assigned to a temporary variable, is necessary to allow each object to be protected from the garbage collector.

Now we can define the main translation function. A code object must be translated
into C code that implements the computation the object represents. We keep a handle to the environment in the generated program. This is a C variable; at the top level it is defined as `R_GlobalEnv`, but it may also be a variable referring to a new environment. Symbols, which represent variable references, are translated into calls to the function `findVar` to look up the variable in the current environment. Recursive code objects are translated recursively into C code that performs the code's intended action. Code for the larger object is combined with the translation of the sub-objects in different ways, depending on the object.

The definition in Appendix A is simplified for clarity. Certain aspects of the translation are not present in the definition:

- Variable declarations are assumed. The output includes a variable declaration `SEXP v;` at the top of the bracketed expression for each variable `v` used.

- A `scope` in C is delimited by a set of curly braces (`{""}`). C programs with large numbers of variables in a scope can cause unreasonably long compilation times. In some cases, they can cause compilers to fail due to memory overflow. To prevent this problem, RCC defines many objects in separate scopes. Certain expressions such as lists are put inside a smaller scope within a larger one, separated with curly braces. A new temporary variable is created in the outer scope. RCC generates an assignment at the end of the scope assigning the final value to the variable; this carries the final inner value outside the scope.

- Many unnecessary `PROTECT` and `UNPROTECT_PTR` statements are eliminated in the generation of lists. In the formal algorithm, every cons cell except the outermost is created, protected, and immediately used as an argument to the next `CONS` constructor. The `cons` function in R protects its arguments before allocating space. Thus, a cons cell that is immediately used as an argument to another `cons` will not be treated as garbage in the interim, and thus protecting these intermediate cons cells is unnecessary. When RCC generates lists, `PROTECT` is called only on the final, outermost `cons` cell.
• In function definitions, objects that are constant, i.e., independent of function arguments, are defined outside the function body, in an initializer that is run when the code is loaded. This avoids having the same object created multiple times in different function calls. This transformation is described in Section 4.2.3.

• Many objects are allocated using local stack space instead of using R’s allocation mechanism. Large spaces for lists are allocated once per function. Multiple objects that are guaranteed not to be live at the same time can be allocated in the same space, avoiding allocation and garbage collection overhead. This technique is explained greater detail in Section 4.3.1.

4.2 Implementation of RCC

RCC uses version 1.6.2 of R with some modifications to interact with compiled code. The modifications are invisible when running interpreted code. R was configured as a shared library (using the --enable-shlib option to R’s configure script) as well as a standalone program, to enable executable code generated by RCC to be linked with the R shared library. RCC output is compiled as a dynamic library that can be called through R using the dyn.load function. R has a special mechanism to allow code to be executed upon loading a dynamic library. If a dynamic library includes a function with a name of “R.init.” concatenated with the filename, R automatically executes the function when dyn.load is called. RCC generates such an initialization function that executes the compiled R program. After an R script is compiled, a user can use dyn.load from within R to call the compiled script.

4.2.1 Function Definitions

When the R interpreter encounters a function definition, it calls a function constructor. This constructor builds a closure containing a pointer to the function body, a list of the formal arguments, and a pointer to the current environment. If RCC compiled a function definition naively, it would simply reconstruct the function body and use it
as an argument to the function constructor (along with the function arguments and the current environment). This process would generate correct code, but it would prevent compilation of all code inside functions. Instead, RCC compiles R functions into C functions that can be called using R's built-in C interface. RCC produces a code object representing a call to the compiled C function using R's .External interface. Since R functions are non-extensional (their internals can be inspected and modified at runtime), the original interpreted code must be preserved to allow reflection. The compiled code is added as an ATTRIB attribute to the closure representation. I modified the code for applications of closures in the interpreter to check for the existence of the compiled-code attribute and execute it if it exists. This approach combines interpreted and compiled code, allowing each to be used for the appropriate purpose.

4.2.2 Protecting Objects From the Garbage Collector

Because RCC constructs objects that the R interpreter cannot reach through ordinary means, RCC must use the PROTECT macro immediately after an object is constructed to protect the object from the garbage collector. After use, the object should be unprotected to allow it to be garbage collected once it is dead. In RCC, an object is most commonly unprotected because it no longer needs protection from the garbage collector when it becomes a child of another protected object. When RCC builds an expression with subexpressions, it protects the object and its sub-objects in the following way:

1. RCC uses PROTECT to protect each subexpression immediately after it is defined.

2. RCC uses the appropriate constructor to create the larger expression out of the subexpressions and immediately protects the new object.

3. The subexpressions, as part of the protected larger expression, no longer need to be protected. RCC calls UNPROTECT_PTR on each subexpression to release the space in the protection stack.

Some information must be passed from subexpressions up to their parent expressions. For example, an expression can be lifted to the constant pool if all its
subexpressions are constant. Therefore, during compilation of the larger expression, RCC must know whether the compiled subexpressions are constant. RCC’s recursive code generation functions return this information to their callers using a structure called an Expression. It includes the following:

- a string: the variable containing the expression
- a Boolean: whether the expression is dependent on function arguments (If it is part of a function but independent of function arguments, it can be lifted to a constant pool.)
- a Boolean: whether the expression is visible (should be printed when a top-level R expression)
- a string: the C code that should be output to release protection, etc., when the expression goes out of scope. This is most often a call to UNPROTECT_PTR that releases the expression from the protection stack.

4.2.3 Avoiding Redundant Creation

In the R interpreter, the code object representing the entire program is constructed exactly once by the parser. In RCC, however, there is a danger of creating many redundant copies of the same code object when translating function definitions. This can significantly degrade performance of compiled code. Suppose a piece of R code exists as a code object inside an R function definition. In the interpreter, the code object is created once and used multiple times. If RCC naïvely translated the function into C, however, the C function would contain the code to create the code object, and the same object would then be created anew on every call of the function. To avoid this redundancy, RCC generates a pool of constants outside function definitions. RCC must determine whether objects created inside function definitions can be lifted to the constant pool. If an object contains some argument to the function, or any other variable that must be looked up in the function's environment, the object must be created inside the function. But if all parts of an object are independent of function arguments, then object instantiation can be lifted.
4.3 Optimizations

What is described so far is a relatively simple compiler that produces code similar to internal interpreter code. As the produced code largely performs the same actions as the interpreter, it achieves roughly the same performance. The next step is to find opportunities for optimization. RCC optimizes code by allocating certain objects locally and improving function calls.

4.3.1 Stack Allocation of Lists

In R, all objects are allocated on the garbage-collected heap. Many of these objects are short-lived lists that are dead immediately after use. For example, when an internal library function is called, the SEXPs representing the arguments are packaged inside a list. The CONS cells that make up this list are guaranteed to be dead as soon as the function returns. We can bypass the R allocator in these cases to use our own space, decreasing allocation and garbage collection overhead.

RCC reuses local storage to improve memory locality and to decrease initialization and garbage collection overhead. Most parts of list structures are the same from list to list: CDRs are pointers to the next cell; the TYPEOF fields are usually LISTSXP, representing a list of data, or more rarely LANGSXP, representing a code object; the TAG and ATTRIB fields are usually empty. Rather than allocating a new list for each temporary, we can allocate space for one general list at the beginning of a function and reuse it for many temporary lists that are allocated during execution of the function. The general list is filled in once with default values at the top of the function. When each temporary list is created, RCC fills in the fields that differ from the default and restores them to the default values once the list is no longer in use. In the future, some redundancy could be eliminated when multiple temporary lists use the same non-default value consecutively in one general list.

Within each scope there must be at least as many general lists allocated and initialized as the maximum number of temporary lists live at any point in time. Each general list must be large enough to fit the largest temporary list stored there. RCC keeps track of the lists currently in use and their lengths. Each temporary list is
matched to the first unoccupied general list. At the top of the function, each general list is allocated and initialized with enough space to store as many cons cells as the maximum length of a temporary list stored there.

General lists are implemented as arrays of cons cells. The cells appear in reverse order; the first cell’s cdr is \texttt{RNilValue}, and every other cdr points to the previous cell. Thus, if a memory location \texttt{mem0} stores a general list, a temporary list of length 3 will occupy \texttt{mem0}, \texttt{mem0+1}, and \texttt{mem0+2}; a pointer to \texttt{mem0+2} is returned as the value of the list. Non-cdr values are initialized to default values at the beginning. When a general list is filled in with a specific list, the appropriate fields are filled in. In this way, RCC avoids many unnecessary allocations.

To ensure maximum overlap of memory locations would require \textit{escape analysis} [25], which is beyond the scope of this thesis. RCC implements a conservative approximation by locally allocating only lists created as argument lists to function calls, which are guaranteed not to escape the scope of the function being called.

A potential issue can arise when locally allocated objects are reached by the garbage collector. When the garbage collector traverses the data structures, it must be prevented from trying to add the locally allocated objects to the allocation pools. If the garbage collector tried to free memory that was not allocated in R, memory corruption would result.

Specifically, corruption may occur if the garbage collector tries to \textit{move} locally allocated objects from locally managed storage to a generation in R’s garbage-collected space. The non-moving R garbage collector does not move the actual data; it “moves” an object by changing the pointers that link it to the other objects of its generation in a circular linked list. The garbage collector moves data in two cases: when an object lives long enough to be moved to an older generation and when an old object that points to a new object is collected.

To prevent age-related moving of local objects, new objects are set to have generation numbers older than all R generations. Since they appear to the garbage collector to be older than all R-allocated objects, the garbage collector will never move them to an older generation.
When the garbage collector collects a new generation, in addition to the ordinary roots, it needs to consider references from old objects to new objects. (Such references are rare; subexpressions are generally older than the larger expressions built out of them [32].) When the collector encounters an object in an old generation that contains a reference to an object in a newer generation, it moves the old object to a special memory area reserved for old-to-new pointers.

RCC prevents this kind of move of locally allocated objects by adding a special case to the detection of old-to-new pointers in the garbage collection code. Each locally allocated object has NULL in place of pointers to the next and previous objects in the circular doubly linked list. I added a check for NULL to the R runtime code that checks for old-to-new pointers. If an object has a NULL pointer as its NEXT field, it is never moved as the old object in an old-to-new situation. This change to R code does not change normal operation of the interpreter.

4.3.2 Direct Function Calls

In R, evaluation of function applications involves a great deal of overhead. The procedure for evaluating a closure applied to arguments is as follows:

1. R matches the formal arguments with the actual arguments, taking into account default arguments, named arguments, positions in the argument list, and the "..." object.

2. R creates promises out of the actual arguments using the current environment.

3. The formal and actual arguments form a set of new variable bindings. R creates a new environment by adding a frame containing these new bindings to the closure's environment.

4. R defines a context—a target for control flow to jump from return statements inside the function.

5. R evaluates the function body using the new environment.
Even for function applications as simple as the `.External` call used to call compiled functions, evaluation takes a significant amount of time. A great deal of time can be saved by bypassing the closure system when possible. For functions with certain characteristics, RCC can generate a direct function call to the C function. The function name is not looked up in an environment at runtime; instead, the first function definition in the global environment with the given name is used. Argument matching and promise creation are not done; the list of arguments is simply evaluated and passed to the function. RCC takes a list of applicable functions as command line arguments. These R functions are translated into two different C functions: one following the closure interface and one to be called directly. This enables function calls visible to RCC to call the function directly, while enabling calls buried in R code objects to call the function as well.

In order to create direct function calls, the user must ensure that it is legal to bypass the closure mechanism. The following conditions must hold in order for a direct function call to be legal:

- The function must have only one definition.
- At all call sites, arguments occur in the order specified in the function definition. This means direct function calls cannot take advantage of named arguments.
- The function must be defined in the global environment, not nested inside another function definition.
- The function must cause the same behavior whether call-by-need or call-by-value semantics are used. This requirement is fulfilled if the function is `strict` in all its formal arguments (it always evaluates all its arguments) or if all actual arguments terminate without causing side effects, as do constants and bound variables.

A survey of public R code suggests that the many, if not most, user-written R functions and function calls fulfill these requirements. In future versions, there is ample opportunity for RCC to improve direct function calling. First, it may be
possible to relax the requirements for direct function calls. Argument matching, for example, might be done at compile time, enabling named arguments. Also, careful analysis, including strictness analysis [15], could automatically detect when direct function calls are legal, eliminating the need for the user to specify the functions. Chapter 6 describes these opportunities more fully.
Chapter 5

Experiments and Results

5.1 Understanding Performance

To understand the performance of RCC-generated code, standard performance monitoring tools are not enough. R code uses many library functions that make heavy use of low-level utility functions. Analysis that reveals the use of library functions is more useful than details about low-level functions. Knowing that a lot of time is spent in `strcmp`, for example, reveals little about the usage pattern of libraries that use `strcmp`. It is more useful to profile the entire function call stack.

The R interpreter has a profiling mechanism that can be used to discover where time is being taken during interpretation. When profiling is enabled, R periodically stops interpretation (by raising the signal SIGPROF) and prints the contents of the R function call stack to a file. This is useful for determining which R functions consume the most time when running a program with the interpreter. For compiled code, however, since this mechanism profiles the stack of R functions, not the stack of underlying C functions used to implement R, this shows nothing but the call to the dynamic loader that runs the code. A more involved technique is needed to discover the call stack profile of RCC-generated code.

I used the GNU debugger, GDB, to generate C stack traces of the R interpreter. Under GDB, it is possible to execute commands when signals are caught. I used GDB to print a C stack trace every time SIGPROF was thrown by the R interpreter. This produced a call stack profile of the interpreter implementation.

I used a similar method to generate a call stack profile of code generated by RCC. As expected, RCC translates R code that uses the profiling mechanism into C code that uses the same profiling mechanism. I ran compiled code through GDB, catching SIGPROF as before, to generate a call stack profile for compiled programs.
I analyzed the GDB call stack profiles to understand the effect of compilation and the optimizations done in RCC. The results are analyzed in the next section.

5.2 Performance Results

I compared the performance of RCC-compiled code to that of the R interpreter. I measured the effect of the two optimizations performed in RCC: the effect of local allocation for temporary lists versus R’s garbage-collected allocation mechanism, and the effect of calling compiled functions directly versus using R closures.

The test codes used were the following:

- **for** is a simple arithmetic function call inside a long-running for loop.

- **fib** is an implementation of the naïve recursive algorithm to find the \( n^{th} \) Fibonacci number.

- **array** is a program with a multidimensional array with a large amount of data, modified with an index array.

- **mixture** is a program from the M.D. Anderson Cancer Center that performs Bayesian mixture analysis. Its purpose is to analyze a mixture to determine the proportions of the original components.

- **trials** is a program from the M.D. Anderson Cancer Center that performs calculations relating to clinical trial design.

For, fib, and array are simple benchmarks that are less than 20 lines of R code each. The mixture and trials codes are about 500 lines and 50 lines long, respectively; they are representative of the computations performed by statisticians at M.D. Anderson.

Tests were run on a 2.4-GHz Pentium 4 running Linux. The R process running the tests did not exceed the available memory. The results of these tests appear in Table 5.1.

Both the R interpreter and RCC-generated codes take a noticeable amount of time to initialize variables, define library functions, etc., before evaluation of the
Table 5.1: Performance of variants of RCC relative to that of the R interpreter. Values greater than 1 are speedups over the R interpreter; values less than 1 represent decreased performance relative to the interpreter. “Heap and “local” describe the allocation of R objects: “heap” denotes allocation of objects using the R interpreter; “local” means local allocation is used when possible. “Closure” and “direct” refer to function calling: “closure” means all calls to compiled functions are made using R’s mechanism; “direct” means that applicable functions are called directly.

<table>
<thead>
<tr>
<th></th>
<th>for</th>
<th>fib</th>
<th>array</th>
<th>mixture</th>
<th>trials</th>
</tr>
</thead>
<tbody>
<tr>
<td>R interpreter</td>
<td>33.98s</td>
<td>21.80s</td>
<td>18.36s</td>
<td>11.12s</td>
<td>52.79s</td>
</tr>
<tr>
<td>heap/closure</td>
<td>0.7848</td>
<td>1.027</td>
<td>1.001</td>
<td>1.080</td>
<td>1.316</td>
</tr>
<tr>
<td>heap/direct</td>
<td>2.005</td>
<td>2.926</td>
<td>0.9797</td>
<td>1.082</td>
<td>1.322</td>
</tr>
<tr>
<td>local/closure</td>
<td>0.8762</td>
<td>1.021</td>
<td>0.9582</td>
<td>1.081</td>
<td>1.431</td>
</tr>
<tr>
<td>local/direct</td>
<td>2.062</td>
<td>3.375</td>
<td>0.9597</td>
<td>1.083</td>
<td>1.438</td>
</tr>
</tbody>
</table>

script occurs. Startup time, measured by measuring the running time of an empty R file, took the same amount of time in the interpreter and with RCC, approximately 0.33 seconds. Since we are intending to improve the performance of long-running R programs, startup time is subtracted from all time measurements.

Using local memory allocation often improves performance by allowing reuse of memory locations. It was most effective for programs like fib and trials that make many consecutive library calls within the same scope. Local allocation did not always improve performance over R allocation. This occurs for several reasons. First of all, RCC misses some opportunities to merge storage of multiple lists. Discovering whether lists can share memory locations is a hard problem. The current version of RCC merges lists when possible in individual scopes, but it does not share memory across scopes. Second, not all lists can be locally allocated. Lists passed as arguments to the NewEnvironment function, for example, must be R-allocated because they are destructively modified in such a way that local memory would become corrupted.

Direct function calling significantly reduces function call overhead. As expected, it improves the performance of programs such as for and fib with many function calls.
Improvement was minimal for programs that spend large amounts of time within math libraries, as does array. There were no R function calls to call directly, and local allocation did no good because there were few opportunities for reuse. Optimization of such programs requires a different approach. Section 6 describes a way of replacing library calls with more specific versions based on the context in which they are called.

An interesting result was that fib, mixture, and trials experienced improved performance simply through plain compilation without either optimization. The trials example experienced an impressive 31% improvement. This appears to be due to eliminating calls to the eval function in the interpreter, since trials is a scalar program that spends much of its time traversing nested code structures.

Plain compilation can also degrade performance, as in the for benchmark. Profiling data suggests that the slowdown is due to overhead associated with protecting and unprotecting objects from the garbage collector.

The optimizations currently performed in RCC improve performance on some codes, but more analysis and transformation is required to achieve performance close to the hand-translated code while preserving R’s semantics.

5.3 Code Shape

In addition to improving efficiency, RCC aims to improve the ability of later passes to analyze its output. RCC especially targets the Telescoping Languages project, which intends to improve code that makes heavy use of libraries. In the R interpreter, calls to library functions are hidden in the implementation of evaluation functions on various code objects. In RCC’s output, calls to library functions appear explicitly in the code, facilitating analysis.

RCC generates output that is relatively easy to analyze. Control flow is expressed in C, not in R syntax trees. With few exceptions, each subexpression is stored in a separate temporary variable in a scheme similar to A-normal form [12]. When lists are R-allocated, each list is allocated in a separate scope separated using C’s curly braces { }. This keeps temporary cons calls private; only the temporary representing the list itself is used outside the scope.
PROTECT and UNPROTECT_PTR calls are used for most subexpressions. Each protected subexpression is protected immediately after it is defined, and it is released with UNPROTECT_PTR once it is no longer in use. The only exception is inside for loops; a separate protection mechanism is used for the induction variable, matching the interpreter's evaluation of for loops. This aspect of for loops will be simplified in later versions of RCC.
Chapter 6

Future Work

Many opportunities exist to make RCC generate faster code and reduce usei work. The performance of RCC-generated code shows that the optimizations done in RCC were often significant but not enough to achieve performance close to that of a hand-coded translation. Local list allocation involved relatively small-scale changes to list representations of R data; more large-scale improvements, such as passing arguments as they are instead of in a list, could achieve a greater improvement. Direct function calling greatly improves code that uses many function calls. Most user code, however, uses few function calls, and so reducing function call overhead did not achieve a noticeable improvement on these programs. Type inference could improve much more common vector programs.

One example from the mixture code illustrates the inefficiencies in interpreted R that can be eliminated by analysis and transformation within RCC. A loop in mixture contains several sum reductions of the following form:

\[ n \leftarrow \text{sum}(r == 0) \]

The variable \( r \) is a vector 800 elements long containing 0 and 1 values. The purpose of this line is to count the number of 0s in \( r \) for use in later, more involved computations. Implemented in C using a vector of integers, all such sum reductions in the program together would take less than 0.1 seconds. In R, however, profiling data shows that sum reductions of this form take a total of about 3 seconds, or about 30% of the time to interpret the program. The interpreter performs the reduction in the following way:

- When the vector \( r \) is created, it is stored as a vector of floating-point values.
• R performs the vector equality test, represented by the expression `r==0`. R checks whether each array element is equal to 0 using the floating-point equality operator.

• During the equality test, R checks each element of the vector for special values N/A and NaN. These values represent the result of unusual situations that arise during computation. The object N/A represents a nonexistent value, which results from out-of-bounds array accesses. NaN, short for “not a number,” is the result of arithmetic computations that produce errors, such as division by zero. R treats these unusual values specially; thus, arithmetic operations must explicitly check for such values.

• The output of the equality test is an indicator vector of TRUE or FALSE values. For every element of the `r` vector, R sets the corresponding element of the indicator vector to TRUE if the element is equal to 0 and FALSE otherwise.

• The `sum` function counts the TRUE values in the indicator vector by summing the values, which are 1 for TRUE and 0 for FALSE.

• As the `sum` function processes the indicator vector, it once again checks each element of the vector for N/A.

Detailed analysis, including type and shape inference similar to that described in McCosh's work [22], could determine that many of the R interpreter's actions can be optimized away. It could discover that the only objects stored in `r` are integers—in fact, that they take only values of 0 or 1—allowing `r` to be stored as a vector of integers instead of floating-point numbers. The equality test could then use fast integer equality instead of the more time-intensive floating-point equality. Shape analysis could determine that all elements of the vector exist and that all accesses to `r` are in bounds, eliminating the need to check explicitly for N/A. All expressions stored to the vector are constants, never the results of computation that might give erroneous values, so the check for NaN is also unnecessary. Even if N/A and NaN checks were required, analysis could determine that only one check is required; the
indicator vector produced by the equality test will be the same length as \( r \) and contain only TRUE and FALSE values, so a second check is unnecessary. Most significantly, analysis could completely eliminate the need for an indicator vector. Instead of creating an indicator vector to record which elements are equal or unequal to 0 and then totaling the TRUE values, a simpler algorithm could count the 0 values in one pass, performing the equivalent computation in less time and space.

This example points out opportunities to use RCC to further improve performance.

### 6.1 Improving Performance

Direct function calling can be improved. The current version of RCC requires the user to specify the functions to be called directly, and it places restrictions on the characteristics of those functions. RCC can improve effectiveness and usability by automatically detecting and implementing direct calls and by expanding the set of functions that can be called directly. Interprocedural analysis, including *strictness analysis* [15], can enable us to determine when a function can be called directly.

The step of matching actual arguments to formal arguments is a time-consuming process that seems possible to do interprocedurally at compile time. If so, it would eliminate the need for in-order arguments as well as improving performance. For all call sites, even if the values are not known at compile time, the order of the arguments, which arguments are named, how many arguments match the "..." object, etc., are generally known. It should be possible to put the arguments in order, resolve default arguments, etc., at compile time so that when the call is executed, the function will receive the arguments in the correct order. More investigation is needed to determine the feasibility of this idea.

A great deal of time in RCC-generated code is spent in storing and looking up variables. In the *mixture* program with the R interpreter, lookup and definition of variables consumes about 12% of the time. RCC eliminates many lookups of function names with direct function calling, but variable definition and lookup times are still time-consuming: variable operations account for about 8% of the time taken by *mixture* compiled by RCC. RCC generates definitions and lookups that work the
same way R does, by accessing explicit environments via the functions `defineVar` and `findVar`. RCC should perform analysis to determine when C variables can be used instead. In many cases, `defineVar` can be replaced with a simple C assignment, and `findVar` can be replaced with a reference to the defined variable. This transformation would improve code shape in addition to improving performance.

Unfortunately, due to R’s flexible reflection capabilities and package handling, it is impossible, in general, to determine when a function can implement environments implicitly in C variables rather than using explicit R environments. With sealed environments and an improved package management system, ideas proposed by Tierney [31], variable lookup and definition could be optimized while preserving correct R semantics. A feature similar to Java’s `final` would enable a programmer to prohibit modification of certain functions. This could greatly improve performance by allowing the compiler to make more assumptions about the code.

Another future objective is to compile more code objects into C. As one important example, the current version of RCC cannot translate an assignment whose left-hand side is an array subscript, a field of a structure, or other expression more complex than a variable. Assignments of this type, especially assignments to array subscripts, are particularly common in mgg00 and other test cases. The interpreter performs a complex transformation on the assignment code object in such cases. Unraveling its actions to generate the right C code, while difficult, is important in order to produce fast code.

Eventually, RCC will be part of the Telescoping Languages project. It can fit in the language-building compiler section, translating libraries that are written in R into C, and also as the script translator, translating user code from R into C code to be compiled using knowledge gained from library compilation. Since most of the R domain-specific libraries are themselves written in R, RCC could be involved in intensive compilation of libraries to produce efficient variants based on the contexts in which they are expected to be used. User scripts in R must also be translated into a common language such as C as a target for the enhanced language compiler; RCC will be useful in that phase as well.
Chapter 7

Conclusions

RCC is, to my knowledge, the first compiler built for R. RCC translates R into C, improves performance, and provides a basis for further optimization.

For a language with as many features as R, building a complete compiler from scratch would be an enormous undertaking. Because the interpreter source code is available, however, RCC can use the R library for parsing, construction, and interpretation. Using the R library, RCC can generate compiled code as well as R code objects that can be interpreted. RCC compiles function definitions in a way that preserves R semantics, even with inspection and modification of function internals.

The code generated by RCC is designed to have straightforward, analyzable code shape. Control flow and function definitions are translated into their C counterparts, allowing library calls to appear in the C code not embedded in R object constructors. This facilitates later stages of analysis that can transform library calls to more specific variants. Further optimization will require more detailed analysis and transformation.

RCC generates code that achieves performance equal to or better than that of the interpreter. The two optimizations done by RCC, local list allocation and direct function calling, improve performance substantially for many programs. With further analysis, existing optimizations can be improved and new optimizations added. More work needs to be done before RCC can generate code with performance comparable to that of a hand-coded translation. With RCC, however, the potential exists to achieve powerful analysis and optimization of the R language.
Appendix A

Definition of Translation

A.1 Grammars for R and C

Translation of R into C is defined in terms of a function mapping code objects to C code. The set of code objects is the subset of SEXP's representing R code; it can be defined as follows:

\[
\begin{align*}
\text{constant} & \quad k \in \text{Reals} \cup \text{Strings} \\
\text{symbol} & \quad s \in \text{Symbols} \\
\text{tag} & \quad t ::= s | \text{nil} \\
\text{constructor label} & \quad k ::= \text{CONS} | \text{LCONS} \\
\text{code} & \quad c ::= \text{nil} | k | s | k c t c
\end{align*}
\]

Recall that CONS and LCONS are separate constructors used for data and language objects, respectively. In addition to a CAR and CDR, they take a tag \( t \), which, if it is non-nil, is a symbol used to name the first argument. All compound code objects are implemented in terms of lists with LCONS at the head; for example, a sequence of 10 R commands is implemented as an LCONS with a CAR pointing to the symbol “{” (left brace) and a CDR pointing to the list of the ten commands.

Much of the output C code is based on code from R interpreter functions; the output uses primitives and functions from the R infrastructure. The subset of C used as the output of RCC is defined as follows:
type ::= SEXP | int | PROTECT_INDEX | RCONTXT

xp^2 ::= + | - | * .

xp^1 ::= &

string ::= '" Cstring "'

declaration ::= type symbol ;

declarations ::= e | declaration declarations

choice ::= case e : statement | default : statement

choices ::= choice | choice choices

fundef ::= type x(arglist) b

fundefs ::= e | fundef fundefs

arglist ::= e | NEarglist

NEarglist ::= e | e,NEarglist

symbol ∈ {R.NilValue, R.GlobalEnv, context

CTXT_RETURN, CTXT_LOOP, R.ReturnedValue} U Symbols

f^1 ∈ {PROTECT | UNPROTECT_PTR

| CAR | CDR | CADR | CDDR | TAG | TYPEOF

| ScalarReal | ScalarInt | mkString | install | endcontext

| isList | isNull | length | LENGTH | allocVector

| LOGICAL | INTEGER | REAL | SETJMP}

f^2 ∈ {cons | lcons | mkPRIMSPXP

| allocVector | REPROTECT | PrintValueRe...

| setVar | findVar | findFun

| PROTECT_WITH_INDEX}

f^3 ∈ {tagged_cons | tagged_lcons | mkCLOSPXP

| defineVar | NewEnvironment}

f^6 ::= begincontext
\[\text{app} \quad ::= \quad f^1(e) \mid f^2(e, e) \mid f^3(e, e, e) \mid f^6(e, e, e, e, e)\]
\[\text{e} \quad ::= \quad c \mid \text{symbol} \mid \text{string} \mid e=e \mid e[e] \mid e++ \]
\[\quad \mid e \text{op}^2 e \mid e \text{op}^1 e \mid \text{app} \mid \text{sizeof(SEXPREC)}\]
\[\text{statement} \quad ::= \quad e; \mid b \mid \text{for}(e; e) \mid \text{if}(e) \mid \text{return}(e); \mid \text{break;} \]
\[\quad \mid \text{switch}(e) \{\text{choices}\} \mid x; \mid \text{goto symbol}\]
\[\text{statements} \quad ::= \quad e \mid \text{statement statements}\]
\[\text{block} \quad ::= \quad \{\text{declarations statements}\}\]

Translation is shown in the form of rule instances common in programming languages literature. A horizontal bar represents a logical inference; given the conditions above the bar (i.e. the results of translating sub-expressions), the result for a given expression is given below the bar.

### A.2 Reconstruction

Reconstruction, the \(R[]\) function, is defined as a function from a code object to a tuple of three objects \((c, v, p)\):

1. \(c\): a block of C code that constructs the object and assigns it to a temporary variable “handle”

2. \(v\): the variable handle representing the object

3. \(p\): a Boolean indicating whether the object is being protected from the garbage collector

The \(P[]\) and \(U[]\) functions protect and unprotect their arguments from the garbage collector.
\[ P : e \to \text{statement} \]
\[ P[e] = \text{PROTECT}(e) \]
\[ U : (e, \text{Boolean}) \to \text{statement} \]
\[ U[e, \text{TRUE}] = ([\text{UNPROTECT_PTR}(e)]U[e, \text{FALSE}] = []) \]
\[ C : \{\text{integer} \cup \text{float} \cup \text{symbol}\} \to e \]

The C function represents unary constructors:

\[ C[i] = \text{ScalarInteger}(i) \quad C[f] = \text{ScalarReal}(f) \quad C[s] = \text{install}(s) \]

\[ K^2 : k \to f^2 \]

\[ K^2[\text{CONS}] = \text{cons} \quad K^2[\text{LCONS}] = \text{lcons} \]

\[ K^3 : k \to f^3 \]

\[ K^3[\text{CONS}] = \text{tagged_cons} \quad K^3[\text{LCONS}] = \text{tagged_lcons} \]

\[ R : \text{code} \to (\text{statements}, \text{symbol}, \text{Boolean}) \]

\[ \text{Nil} \]

\[ R[\text{Nil}] = ([], \text{R_NilValue}, \text{FALSE}) \]

\[ \text{Constants/Symbols} \]

\[ R[c] = ([P[v = C[c]];], v, \text{FALSE}) \]

\[ R[e_1] = (c_1, v_1, p_1) \quad R[e_2] = (c_2, v_2, p_2) \]

\[ R[k\ e_1\ \text{R_NilValue}\ e_2] = (\begin{bmatrix} c_1; c_2; \\ P[v=k(v_1, v_2)]; \\ U[v_1, p_1]; U[v_2, p_2] \end{bmatrix}, v, \text{TRUE}) \]

\[ \text{Cons/lcons} \]

\[ R[e_1] = (c_1, v_1, p_1) \quad R[e_2] = (c_2, v_2, p_2) \quad R[t] = (c_t, t, \text{FALSE}) \]

\[ R[k\ e_1\ t\ e_2] = (\begin{bmatrix} c_1; c_t; c_2 \\ P[v=k(v_1, t, v_2)]; \\ U[v_1, p_1]; U[v_2, p_2] \end{bmatrix}, v, \text{TRUE}) \]

\[ \text{Tagged cons/lcons} \]

\[ \text{A.3 Translation} \]

Translation, the \( T[] \) function, is defined as a function mapping a code object and a reference to an environment to a tuple that contains four elements. The first element is a (possibly empty) list of function definitions. The tuple also contains the objects produced by \( R[] \): the C code generated, a variable handle, and a boolean value that
indicates whether the object is protected.

Many code constructs that are translated are language objects: lists composed of one icons cell and one or more cons cells. For clarity’s sake, I will use “LANG” with a variable number of arguments to indicate a list where the first cell is icons and the other cells cons, and in which all tags are nil. For example, LANG if e1 e2 is equivalent to LCONS if nil (CONS e1 nil (CONS e2 nil nil)).

\[ T : code, environment \to (fundefs, statements, symbol, Boolean) \]

\[ T[\text{nil}]_\rho = ([\text{;}], [], \text{R.NilValue, FALSE}) \]

\[ T[e_1] = (L_1, C_1, v_1, p_1) \quad T[e_2] = (L_2, C_2, v_2, p_2) \]

\[ T[\text{CONS } e_1 e_2] = \left( \begin{array}{c} L_1 \\ L_2 \end{array} \right), \quad \left[ \begin{array}{c} C_1 \\ C_2 \\ P[\text{out=cons}(v_1, v_2)] \\ U[v_1, p_1] \\ U[v_2, p_2] \end{array} \right], out, TRUE \]

\[ T[\text{constants}]_\rho = ([\text{;}], [P[v = C[e];]], v, FALSE) \]

\[ T[\text{symbols}]_\rho = ([\text{;}], [P[v=\text{findVar}(s, \rho);]], v, FALSE) \]

Two different If rules represent if-then and if-then-else statements.
If \( T[e_1]_\rho = (D_1, C_c, v_c, p_c) \) \( T[e_2]_\rho = (D_2, C_T, v_T, p_T) \)

\[
T[\text{LANG if } e_1 e_2]_\rho = \left( \begin{array}{c} D_1 \\ D_2 \\ C_c \\ P[v=\text{asLogicalNoNA}(v_c)] \\ U[v_c, p_c] \\ \text{if } (v) \{ \\ \quad U[v, \text{TRUE}] \\ \quad C_T \\ \quad P[v'=v_T] \\ \quad U[v_T, p_T] \\ \} \text{ else } \{ \\ \quad U[v, \text{TRUE}] \\ \quad P[v'=\text{R.NilValue}] \\ \quad U[v_T, p_T] \\ \}\end{array} \right), (v', \text{TRUE})
\]

If \( T[e_1]_\rho = (D_1, C_c, v_c, p_c) \) \( T[e_2]_\rho = (D_2, C_T, v_T, p_T) \) \( T[e_3]_\rho = (D_3, C_F, v_F, p_F) \)

\[
T[\text{LANG if } e_1 e_2 e_3]_\rho = \left( \begin{array}{c} D_1 \\ D_2 \\ D_3 \\ C_c \\ P[v=\text{asLogicalNoNA}(v_c)] \\ U[v_c, p_c] \\ \text{if } (v) \{ \\ \quad U[v, \text{TRUE}] \\ \quad C_T \\ \quad P[v'=v_T] \\ \quad U[v_T, p_T] \\ \} \text{ else } \{ \\ \quad U[v, \text{TRUE}] \\ \quad C_F \\ \quad P[v'=v_F] \\ \quad U[v_F, p_F] \\ \}\end{array} \right), (v', \text{TRUE})
\]

Macro for switch choices in the translation of for:

\[
\text{Choice}[L_1, L_2] = \left[ \begin{array}{c} \text{case } L_1: \\ \text{REPROTECT}(v=\text{allocVector}(\text{TYPEOF}(v_T), 1), vpi); \\ L_2(v)[0]=L_2(v_T)[i]; \\ \text{setVar}(v_{var}, v, \rho); \\ \text{break}; \end{array} \right]
\]
\[ T[e_1]_\rho = (D_1, C_r, v_r, p_r) \quad T[e_2]_\rho = (D_2, C_b, v_b, p_b) \]

\[ C[\text{var}] = v_{\text{var}} \quad L = \text{unique label} \]

For

\[ T[\text{LANG for var } e_1 \ e_2] = \]

\[
\begin{bmatrix}
D_1 \\
D_2
\end{bmatrix},
\quad \left[ v, v', \text{TRUE} \right]
\]

\[
\begin{align*}
defineVar(v_{\text{var}}, \text{R.NilValue}, \rho); \\
\text{if (isList}(v_r) \| \text{isNull}(v_r)) \{ \\
\quad \text{n}=\text{length}(v_r); \\
\quad \text{PROTECT.WITH_INDEX}(v=\text{R.NilValue}, &v_{\text{pi}}); \\
\} \text{ else } \{ \\
\quad \text{n}=\text{LENGTH}(v_r); \\
\quad \text{PROTECT.WITH_INDEX}(v= \\
\quad \quad \text{allocVector(TYPEOF}(v_r), 1), &v_{\text{pi}}); \\
\} \\
\quad v'=\text{R.NilValue}; \\
\quad \text{PROTECT.WITH_INDEX}(v', &v_{\text{api}}); \\
\begin{align*}
\text{begincontext}(&\text{cntxt}, \text{CTX_LOOP,}
\quad \text{R.NilValue}, \rho, \text{R.NilValue}, \text{R.NilValue}); \\
\text{switch} (\text{SETJMP}(\text{cntxt}.\text{cjmpbuf})) \{ \\
\quad \text{case} \text{CTX_BREAK: goto for_break_L; } \\
\quad \text{case} \text{CTX_NEXT: goto for_next_L; } \\
\} \\
\text{for (i=0; i<n; i++) } \{ \\
\quad \text{switch(TYPEOF}(v_r)) \{ \\
\quad \quad \text{Choice[LGLSXP, LOGICAL]} \\
\quad \quad \text{Choice[INTSXP, INTEGER]} \\
\quad \quad \text{Choice[REALSXP, REAL]} \\
\quad \quad \text{Choice[CPLXSXP, COMPLEX]} \\
\quad \text{case STRSXP:} \\
\quad \quad \text{REPROTECT}(v=\text{allocVector}(\text{TYPEOF}(v_r), \\
\quad \quad \quad \quad 1), v_{\text{pi}}); \\
\quad \quad \text{SET.STRING.ELT}(v, 0, \text{STRING.ELT}(v_r), i); \\
\quad \quad \text{setVar}(v_{\text{var}}, v_r, \rho); \\
\quad \text{case EXPRSXP: case VECSXP:} \\
\quad \quad \text{setVar}(v_{\text{var}}, \text{VECTOR.ELT}(v_r, i), \rho); \\
\quad \quad v_r=\text{CDR}(v_r); \\
\quad \text{case LISTSXP:} \\
\quad \quad \text{setVar}(v_{\text{var}}, \text{CAR}(v_r), \rho); \\
\quad \quad v_r=\text{CDR}(v_r); \} \\
\quad \quad \text{C}_b \\
\quad \quad \text{REPROTECT}(v'=v_b, \text{api}); \\
\quad \text{U}(v'); \\
\quad \text{for_next.L:} \\
\quad \} \} \\
\text{for_break.L:} \\
\text{endcontext}(&\text{cntxt}); \\
\text{U}(v_r); \text{U}(v); 
\end{align*}
\]
\[ R[e_1] = (C_{\text{args}}, v_{\text{args}}, p_{\text{args}}) \quad T[e_2]_{\rho} = (D, C_{\text{comp}}, v_{\text{comp}}, p_{\text{comp}}) \]

\[ R[e_2] = (C_{\text{interp}}, v_{\text{interp}}, p_{\text{interp}}) \]

\[ T[\text{LANG function } e_1 e_2]_{\rho} = \]

\[ D \]

SEXP \( f \) (SEXP full_args) {
    \( P[\text{env}=\text{CADR}(\text{full_args})] \);
    \( P[\text{args}=\text{CDER}(\text{full_args})] \);
    \( P[\text{newenv}=\text{NewEnvironment}(v_1, \text{args}, \text{env})] \);
    if (SETJMP(context.cjmpbuf)) {
        \( P[\text{out}=R._{\text{ReturnedValue}}] \);
    } else {
        begincontext(&context, CTXT\_RETURN, R._{\text{NilValue}},
            newenv, env, R._{\text{NilValue}});
        \( C_{\text{comp}} \)
        \( P[\text{out}=v_{\text{comp}}] \);
        \( U[v_{\text{comp}}, p_{\text{comp}}] \);
        endcontext(&context);
    }
    \( U[\text{env}, \text{TRUE}] \); \( U[\text{args}, \text{TRUE}] \);
    \( U[\text{newenv}, \text{TRUE}] \); \( U[\text{out}, \text{TRUE}] \);
    return out;
}

\[ C_{\text{args}} \]

\[ C_{\text{interp}} \]

\[ P[v = \text{mkCLOSXP}(v_{\text{args}}, v_{\text{interp}}, \rho)] \];
\[ P[v' = \text{cons}(\rho, v_{\text{args}})] \];
\[ P[v'' = \text{cons}(f, v')] \];
\[ U[v', \text{TRUE}] \]
\[ P[v'' = 1\text{cons}(\text{.External}, v'')] \];
\[ U[v'', \text{TRUE}] \]
\[ P[v_{\text{call}} = \text{mkCLOSXP}(v_{\text{args}}, v'', \rho)] \];
\[ U[v'', \text{TRUE}] \]
\[ U[v_{\text{args}}, p_{\text{args}}] \]

setAttrib(v, RCC\_CompiledSymbol, v_{\text{call}});
\[ U[v_{\text{call}}, \text{TRUE}] \]
Assignment

\[ T[e_1] = (D, C, v, p) \]

\[ T[\text{LANG} \gets s e_1] = (D, \begin{bmatrix} C \\ \text{defineVar}(C[s], v, \rho) \\ U[v] \end{bmatrix}, S[s], \text{TRUE}) \]

Sequence

\[ T[e_1] = (L_1, C_1, v_1, p_1) \ldots T[e_n] = (L_n, C_n, v_n, p_n) \]

\[ T[\text{LANG} \{ e_1 \ldots e_n \} = ( \begin{bmatrix} L_1 \\ \vdots \\ L_n \end{bmatrix}, \begin{bmatrix} C_1 \\ U[v_1, p_1] \\ \vdots \\ C_{n-1} \\ U[v_{n-1}, p_{n-1}] \\ C_n \end{bmatrix}, v_n, \text{TRUE}) \]

The \( F \) function maps the name of a function/operator in R to the internal library function that implements it.

\[ F[\gets] = \text{do.set} \quad F[\text{if}] = \text{do.if} \quad F[+] = \text{do.arith} \ldots \]

App

\[ s \in \text{builtin} \quad T[e] = (L, C, v, p) \]

\[ T[\text{LANG} f e] = (L, \begin{bmatrix} C \\ P[[\text{out}=(F[f])(\text{R.NilValue}, s, v, \rho)]]; \\ U[v, p] \end{bmatrix}, \text{out}, \text{TRUE}) \]

App

\[ R[e] = (L, C, v, p) \]

\[ T[\text{LANG} f e] = (L, \begin{bmatrix} P[f=\text{findFun}(s, \rho)]; \\ C \\ P[\text{args}=v]; \\ P[\text{call}=\text{lcons}(f, \text{args})]; \\ P[\text{arglist}=\text{promiseArgs}(\text{args}, \rho)]; \\ U[\text{args}]; \\ P[[\text{out}=\text{applyClosure}(\text{call}, f, \text{arglist}, \rho, \text{R.NilValue})]]; \\ U[\text{call}, \text{TRUE}] \\ U[f, \text{TRUE}] \\ U[\text{arglist}, \text{TRUE}] \end{bmatrix}, \text{out}, \text{TRUE}) \]
Bibliography


