CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

# ASSIGNMENT OF MASTER'S THESIS

| | |
|---|---|
| **Title:** | Improvements of the RIR bytecode toolchain |
| **Student:** | Bc. Jan Ječmen |
| **Supervisor:** | Ing. Petr Máj |
| **Study Programme:** | Informatics |
| **Study Branch:** | System Programming |
| **Department:** | Department of Theoretical Computer Science |
| **Validity:** | Until the end of winter semester 2018/19 |

## Instructions

Familiarize yourself with the R language, its bytecode compiler, and interpreter architecture. Familiarize yourself with RIR, an alternative bytecode format, compiler, and interpreter for the language. The R bytecode compiler assumes certain invariants (such as built-in meaning of control flow statements and certain operators) about the code to make the compiled code faster. Analyze similar assumptions that are used by RIR and extend RIR to use assumptions made by GNU-R as well. Identify and implement improvements to the RIR (compiler, bytecode format, and interpreter). Discuss your results.

## References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague March 1, 2017

Czech Technical University in Prague

Faculty of Information Technology

Department of Theoretical Computer Science

Master's thesis

# Improvements of the RIR bytecode toolchain

***Bc. Jan Ječmen***

Supervisor: Ing. Petr Máj

May 9, 2017

# Acknowledgements

I would like to express my gratitude to Petr Máj for supervising this thesis, for always finding time to help and for his input and advice.

Besides my advisor, I thank Olivier Flückiger and Jan Vitek for their insights.

Lastly, I would like to thank my family for their unending patience and support.

# Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60(1) of the Act.

In Prague on May 9, 2017 …………………

Czech Technical University in Prague

Faculty of Information Technology

## Citation of this thesis

JEČMEN, Jan. *Improvements of the RIR bytecode toolchain.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017. Available also from WWW: ⟨`https://github.com/JanJecmen/dip`⟩.

# Abstract

R is a dynamic programming language that, despite being over 20 years old, is still widely used. RIR is an alternative to its bytecode compiler and interpreter that aims to facilitate adding static analyses and optimization passes easily. RIR is under development and does not currently match the performance of standard R. This thesis attempts to amend the situation. It extends the RIR internal representation, adds new features to its compiler and refactors its interpreter. The average slowdown versus standard R is brought down by about one half in the Shootout benchmarks.

**Keywords** R language, RIR, bytecode compiler & interpreter, optimizations

# Abstrakt

R je dynamicý programovací jazyk, navzdory svému stáří dnes stále oblíbený. RIR je alternativní implementace kompilátoru a interpretu R bajtkódu, která umožňuje snadno provádět statickou analýzu a přidávat optimalizace. RIR je ve vývoji a zatím nedosahuje výkonu standardního R. Tato diplomová práce se pokouší přiblížit výkon RIR k výkonu standardního R. V jejím rámci byly přidány nové instrukce do RIR bajtkódu a nová funkcionalita do jeho kompilátoru a došlo k přepracování jeho interpretu. Průměrné zpomalení na sadě benchmarků Shootout proti standardnímu R bylo sníženo o polovinu.

**Klíčová slova** jazyk R, RIR, kompilátor & interpret bajtkódu, optimalizace

# Contents

# List of Figures

# List of Listings

# List of Tables

# Introduction

R is a programming language that, despite being very old, seems to be steadily gaining popularity in the last years. In 2012, it was estimated [9], that there were about 2000 package developers maintaining over 4000 packages, and over 2 million end users.

Since then, various programming language popularity rating sites report (even though they ought to be taken with a grain of salt) that R only rises.[1]

R is a very dynamic language that is easy to pick up quickly. Unfortunately, it can also be orders of magnitude slower than optimized C code and is notoriously memory hungry.

RIR is a research project at Northeastern University supervised by prof. Jan Vitek. Its long term goal is to provide a fast implementation of R through the means of its own bytecode representation, compiler and interpreter. It is designed in a way that allows for implementing analysis and optimization passes over the RIR bytecode easily.

However, at present, it is lacking the performance of the official GNU R bytecode virtual machine. To make it a viable alternative, improvements in this direction are needed.

This thesis explores where the speed difference comes from and proposes changes to be made to lower it. These changes are implemented and evaluated.

The chapter *About GNU R* gives an introduction to the R language and discusses its features. It also goes under the hood and describes the inner workings of its interpreter and bytecode compiler – the GNU R.

[1] See, e.g., `https://www.tiobe.com/tiobe-index/` and `http://pypl.github.io/PYPL.html`

The chapter *About RIR* introduces an alternative bytecode compiler for the R language, talks about the motivation behind it, its architecture and design choices, the differences to the original and its shortcomings.

The chapter *Improvements* describes in depth the changes that were made to RIR.

The chapter *Evaluation* discusses how the performance of RIR changed after the improvements were made. It describes how the measurements were done and how the performance compares to GNU R.

The results are discussed in *Conclusion*, as well as the direction of future efforts regarding RIR.

# CHAPTER 1

# About GNU R

GNU R [14] is a programming language used mainly for statistical computations.[2] It is an open-source dialect of S, an older statistical language created in 1976 by John Chambers at Bell Laboratories. R has been around from 1993 and was designed by Ross Ihaka and Robert Gentleman, both recognized statisticians. It is a part of the GNU software family and is still actively developed by the R Core Team today. It is a popular alternative to the other major implementation of the S language, S-PLUS, which is a commercial version shipped by TIBCO Software Inc.

R comes with a software environment built around it, which allows for easily manipulating data, carrying out computations and producing quality graphical outputs such as plots and figures. Although at its heart R is used via a command line interface, there are also more user-friendly graphical IDEs available. One of the most widely used is RStudio that provides, for example, syntax highlighting and quick access to documentation through a web-like browser.[3] This, together with R's readable syntax and a vast collection of extension packages available through The Comprehensive R Archive Network (CRAN) makes it possible for new users to step in and start working quickly.

[2] Homepage: `https://www.r-project.org/`

[3] Homepage: `https://www.rstudio.com/`

## 1.1 Language features of R

This section was written while consulting [16, 9, 1, 11, 12].

R is, as far as programming languages go, very interesting and has some quite unusual semantic features. It is an interpreted language, and is dynamically typed and garbage collected. It supports multiple programming paradigms: users can use procedural imperative style, but at the same time R provides an object system for object oriented programming, and is heavily influenced by functional programming languages, notably Scheme (a dialect of Lisp).[4]

Functions are, in accordance with functional languages, first-class values, so they can be passed around as call arguments, returned as results of function calls and created dynamically at runtime. R uses lexical scoping (which it adopted from Scheme) and R functions are closures that capture their enclosing environment at creation time. Arguments are passed by value (although a variant of reference counting is implemented, so that deep copies are only created as needed, e.g., when an object is modified).

Functions are by design anonymous, i.e., they are not named when created. This is unlike many languages (e.g., C or Python) and follows the approach of lambda calculus. Instead of creating named functions, R programmers create anonymous ones and, if they choose so, then use regular assignment to bind them to names. Example can be seen in listing 1.1.

```
> (function(x) x + 1)(2)
[1] 3
> f <- function(x) x + 1
> f(2)
[1] 3
```

**Listing 1.1:** Anonymous function

Interestingly, everything that happens in R is in fact a function call. This goes as far as all control flow statements and arithmetic operators being just

[4] In fact, R has multiple OO systems: 3 builtin and others available as packages, such as R.oo (https://cran.r-project.org/web/packages/R.oo/) or R6 (https://cran.r-project.org/web/packages/R6/).

syntactic sugar for function calls, as can be seen in listing 1.2.[5] In this spirit, even assigning into a variable, evaluating a block of code inside curly brackets or grouping expressions with parentheses translate to calling the respective functions. As such, everything can be redefined, as in listing 1.3.

```
> is.function(`while`)
[1] TRUE
> is.function(`break`)
[1] TRUE
> is.function(`if`)
[1] TRUE
> `if`(TRUE, 1, 0)
[1] 1
> `if`(FALSE, 1, 0)
[1] 0
> `+`
function (e1, e2)  .Primitive("+")
> `+`(1, 2)
[1] 3
```

**Listing 1.2:** All that happens in R is function call

```
> ((0))
[1] 0
> `(` <- function(x) x + 1
> ((0))
[1] 2
```

**Listing 1.3:** Redefinition of parentheses grouping

All actual arguments to a function are lazy evaluated by default. When applying a closure, parameters are wrapped in promises. A promise is an object that contains the unevaluated expression and an environment in which the expression should be evaluated. Promises are only evaluated when the value is actually needed (which is called forcing the promise). Also, once the promise is forced, it remembers the result, so that subsequent uses of the value do not evaluate the expression again.

[5] As a side note, backticks are used in R to denote symbols (i.e. names). Because some symbols have special syntactic meaning (like the `+` being an infix binary operator), they can cause a syntax error if they appear unquoted in a place where the parser does not expect them.

Delayed evaluation is demonstrated in listing 1.4, where the function in question only evaluates its first argument and does not touch the second. For the first call, the block of code in the curly brackets is never executed and so the side-effect of printing a greeting does not happen.[6] In the second call, the arguments are swapped, and the side-effect can be observed in the output. It is possible to pass a block of code as a function argument, since the `` `{` `` is bound to a function that sequentially evaluates all its arguments and returns as its result the value of the last expression (or `NULL` if no expressions are passed in).

```
> f <- function(a, b) a
> f(1, {cat("Hello\n"); 2})
[1] 1
> f({cat("Hello\n"); 2}, 1)
Hello
[1] 2
```

**Listing 1.4:** Promise lazy evaluation

R supports assignment which enables programmers to change a function's local state by modifying its bindings and thus the imperative programming style.[7] Also, the superassignment operator `` `<<-` `` brings into play non-local side-effects (as shown in listing 1.5).

```
> x <- 1
> f <- function() {
+     x <- 2  # local
+     x <<- 3  # lookup in the enclosing environment
+     x  # local
+ }
> x
[1] 1
> f()
[1] 2
> x
[1] 3
```

**Listing 1.5:** Superassignment

[6] `cat` is short for *concatenate and print*.

[7] An interesting quirk is R's left to right assignment: the code `1 -> x` assigns to `x` and is equivalent to `x <- 1`.

The basic data type in R is an atomic vector. Vectors are collections of homogeneous values (a given vector can only hold objects of one particular type), that preserve the order of their elements. R also provides a list type which is heterogeneous. Higher-dimensional types such as matrices and data frames, as well as objects, are internally built on vectors and lists. Vectors can be created in R by calling the function `c`.[8]

Atomic vectors can have one of these six types: logical, integer, double, character, complex and raw. Since R targets data analysis, a special "not available" value `NA` is provided for these. Because all values in a vector must be of the same type, R performs coercion when an attempt is made to combine vectors of different types. In listing 1.6, combining a numeric vector with a character vector results in a character vector, and summing a logical vector coerces to integers (`TRUE` becoming `1` and `FALSE` becoming `0`).

In R there are no scalar types, as scalar values, such as individual numbers and strings, are considered to be vectors of length one. This holds for character vectors, too, which can cause confusion if one expects C-like behavior of strings and tries subscripting, because in R the subscript belongs to the vector that holds the string and not the string itself.[9]

```
> x <- c(1, 2, 3)
> y <- c("a", "b", "c")
> typeof(x)
[1] "double"
> typeof(y)
[1] "character"
> c(x, y)
[1] "1" "2" "3" "a" "b" "c"
> typeof(c(x, y))
[1] "character"
>
> sum(c(TRUE, TRUE, FALSE, TRUE))
[1] 3
```

**Listing 1.6:** Coercion to the most flexible type

Despite its inspiration in functional world, R does not optimize tail recursion, which is the standard approach in functional languages since they typically

[8] The name stands for *combine*.

[9] In such a case, one needs to use the `substr` function.

use recursion in the place of iterative loops. Instead, R encourages vectorized operations. Hence, most of R builtin functionality works element-wise with vectors, while recycling the elements as needed (e.g., when adding vectors of different lengths). This is demonstrated in listing 1.7, where R even issues a warning about recycling.

```
> numeric(10)
 [1] 0 0 0 0 0 0 0 0 0 0
> 1:3
[1] 1 2 3
> numeric(10) + 1:3
 [1] 1 2 3 1 2 3 1 2 3 1
Warning message:
In numeric(10) + 1:3 :
  longer object length is not a multiple of shorter object length
```

**Listing 1.7:** Recycling shorter vector

In R, every object can have arbitrary attributes associated with its data. Attributes are basically a hidden map that assigns names to values. Some of the most important attributes are names (a character vector that assigns names to elements), dimensions (a vector specifying the dimensions and thus effectively distinguishing vectors from matrices and arrays) and class (for implementing one of R's object systems). Attributes are used a lot in R for many purposes and extensions as they provide a way of encoding arbitrary additional metadata for objects. As an example, in listing 1.8 a vector of 4 elements is created, then changed into a two by two matrix and finally changed back to vector (that also has its elements named).[10]

True to its dynamic nature, R is very liberal in handling arguments in function calls. The language supports both positional and named matching of arguments, as well as default argument values. R even understands when the argument names are abbreviated, as long as the arguments can still be uniquely matched.

Moreover, R lets users call functions and not provide the specified arguments. It is only while executing the function body that the missing arguments may (or may not) cause an error. Variable number of arguments is supported as

[10] Here it can be seen here that R uses column-major order for storing the elements.

```
> x <- 1:4
> x
[1] 1 2 3 4
> attr(x, "dim") <- c(2, 2)
> x
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> attr(x, "dim") <- c(4)
> attr(x, "names") <- c("a", "b", "c", "d")
> x
a b c d
1 2 3 4
```

**Listing 1.8:** Object attributes

well by using the ellipsis `` `...` ``. The ellipsis in an argument list matches any number of arguments, and later in the function body refers to the list of matched arguments. Special symbols can be used to access the ellipsis arguments, such as `` `..1` ``, `` `..2` ``, etc. Argument handling is demonstrated in listing 1.9.

As was already mentioned, R has multiple object systems. The simplest is called S3 and it uses a class attribute to implement ad hoc polymorphism (also known as function or operator overloading). It does not have formal classes, but instead uses a special function called a generic function that decides what to call based on the value of the class attribute. A typical example is printing, where the `print` function is generic and its body consists of a call to a dispatcher `UseMethod("print")`. Specialized versions for different types can be defined, such as `print.data.frame` for data frames, or, given an object with class set to `"foo"`, `print.foo` (as is shown in listing 1.10).

S4 is a more formal system than S3, and it allows for true class definitions, describing class representation and inheritance. It has multiple dispatch, which means that dispatchers can pick which method to call based on multiple arguments. R also has reference classes that implement message passing style. Their objects are modified in place (as opposed to the standard pass by value semantics).

```
> f <- function(a, b, a.very.long.argument.name)
+     a.very.long.argument.name
> f(1, 2, 3)
[1] 3
> f(a = 3)
Error in f(a = 3) :
  argument "a.very.long.argument.name" is missing, with no default
> f(a. = 3)
[1] 3
>
> f <- function(a, b) a
> # b is not required
> f(1)
[1] 1
> # a is required
> f()
Error in f() : argument "a" is missing, with no default
> f(b = 2)
Error in f(b = 2) : argument "a" is missing, with no default
>
> f <- function(...) ..2
> f()
Error in f() : the ... list does not contain 2 elements
> f(1, 2, 3, 4, 5)
[1] 2
```

**Listing 1.9:** Argument handling

```
> x <- list()
> class(x) <- "foo"
> print(x)
list()
attr(,"class")
[1] "foo"
> print.foo <- function(...) cat("Printing foo!\n")
> print(x)
Printing foo!
```

**Listing 1.10:** S3 object system

An important and powerful feature of R is its subsetting and subset assignment mechanics. Parts of objects can be retrieved and even changed by the operators `` `[` ``, `` `[[` `` and `` `$` ``. These behave differently for different types of objects.

The first version is a general subset function that supports all kinds of indexing.[11] For example, integer vector specifies which elements to get and in what order, even allowing duplication. If negative indices are used, these elements are omitted from the result. Logical vectors can be used to select only elements at positions where `TRUE` occurs. If the object is named, character vectors can be used to select by name. In combination with assignment (and superassignment), objects can be modified. Subsetting works also for higher-dimensional structures by simply providing indices for each dimension, separated by commas. Some examples are in listing 1.11.

```
> x <- 101:110
> x
 [1] 101 102 103 104 105 106 107 108 109 110
> x[c(3, 5, 1, 1)]
[1] 103 105 101 101
> x[-c(2, 3, 8)]
[1] 101 104 105 106 107 109 110
> x > 105
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
> x[x > 105]
[1] 106 107 108 109 110
> x[x %% 2 == 0] <- NA
> x
 [1] 101  NA 103  NA 105  NA 107  NA 109  NA
> m <- matrix(1:9, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> m[-2, 2:3]  # omit row 2 and get columns 2 to 3
     [,1] [,2]
[1,]    4    7
[2,]    6    9
```

**Listing 1.11:** Basic subsetting / subassignment

[11] As opposed to many languages (e.g., C or Python), R starts indexing elements from 1 instead of 0.

The second version, `` `[[` ``, returns only a single element of an object, and is used to get elements out of a list. The `` `$` `` is then just a shorthand for `` `[[` `` when used with character subsetting (that is, the dollar version expects a name, and it need not be quoted). Also, the dollar operator does partial matching, similar to how function argument names are handled. These operators are demonstrated in listing 1.12.

```
> l <- list(sq = 1:3, str = "a", bool = FALSE, na = NA)
> l
$sq
[1] 1 2 3

$str
[1] "a"

$bool
[1] FALSE

$na
[1] NA

> l[1]
$sq
[1] 1 2 3

> typeof(l[1])
[1] "list"
> l[[1]]
[1] 1 2 3
> typeof(l[[1]])
[1] "integer"
> l[["bool"]]
[1] FALSE
> l$bool
[1] FALSE
```

**Listing 1.12:** Other subsetting operators

## 1.2 AST interpreter

In its core R uses a classic architecture for an interpreted language. After initialization, the user enters R's read-eval-print loop (REPL), that lets them type in expressions and have R evaluate them. First, a reader, or parser, waits for the user input and reads it line by line. If, at the end of line, it has read a syntactically complete expression, it passes it to an evaluator (otherwise it waits for more input). After the evaluator returns the evaluated expression, a printer is invoked that displays the result.[12] Then the reader is invoked again and the process repeats.

Every object in R is internally represented by a C structure called `SEXPREC` (actually, R passes the objects around as pointers to this structure, which are called `SEXP`).[13] This structure contains a header with metadata about the object, such as its type, reference counter or information for garbage collector, and then a union of other structures that represent different types of R internal objects. Some of these types are listed it table 1.1.

**Table 1.1:** Some common types of internal R objects

| Type | Usage |
|---|---|
| `NILSXP` | the singleton **`NULL`** object |
| `SYMSXP` | symbols (or names) |
| `LISTSXP` | lists of dotted pairs |
| `CLOSXP` | closures |
| `ENVSXP` | environments |
| `PROMSXP` | promises |
| `LANGSXP` | language constructs (typically closure application) |
| `SPECIALSXP` | special forms (typically control flow) |
| `BUILTINSXP` | builtin non-special forms (e.g., arithmetic operators) |
| `INTSXP` | integer vectors |
| `REALSXP` | real vectors |
| `STRSXP` | string vectors |
| `BCODESXP` | object compiled to bytecode |

The parser, when it scans the stream of input characters, checks that it is syntactically correct and at the same time builds a tree structure that represents

[12] However, results can also be hidden, e.g., by the **`invisible`** function.

[13] The name refers to S-expressions, or symbolic expressions, as known from Lisp, although the classical linked lists built from dotted pairs are mostly used internally, and vectors are implemented as C arrays for efficiency reasons.

the parsed expression. This tree is called the abstract syntax tree (AST) and its nodes are all SEXPs. An example AST is shown in listing 1.13.[14] In the listing, parentheses denote function calls (i.e. LANGSXP node), the first child being the callee (typically a SYMSXP, i.e., a name that is bound to a function) and the rest its arguments.

```
> pryr::ast(x <- (y + 3) * f(z))
\- ()
  \- `<-
  \- `x
  \- ()
    \- `*
    \- ()
      \- `(
      \- ()
        \- `+
        \- `y
        \-  3
    \- ()
      \- `f
      \- `z
```

**Listing 1.13:** AST of a simple expression

The evaluator is a recursive function that gets as its input two SEXP objects, one representing the AST of the expression that is to be evaluated, and the other the environment in which to evaluate the expression. The evaluator walks the given AST and based on the type of nodes it encounters, performs some action. The result is then returned to be processed by the caller of eval.

Some nodes are self-evaluating, meaning that no action needs to be performed and the node itself is the result. These are for example the **NULL** object, atomic vectors or environments.

If the eval function sees a symbol node, a lookup for its binding is performed in the provided environment. If it is not found there, because of lexical scoping, the parent environment is searched, and so on, until either the binding is found or an empty environment is reached (the empty environment serves as a sentinel parent of all environment chains and does not have a parent itself).

[14] pryr (homepage: https://github.com/hadley/pryr) is a package created by Hadley Wickham that allows to "pry back the surface of R and dig into the details."

One other prominent type of nodes is LANGSXP. R has internally three types of functions, called *special*, *builtin* and *closures* (or user-defined functions). These have different behavior when they are applied, and the eval function handles that.

Special functions are the core language constructs, such as control flow (conditionals and loops). They take their arguments as ASTs in a list and evaluate them as needed while running. This is necessary for example for the **if** statement, because, since R has side-effects, only one of the conditional branches must be evaluated to preserve the correct semantics.

Builtins, on the other hand, are known to evaluate all their arguments, so it is not necessary to create promises from their arguments. Instead, a list of evaluated arguments is created and passed to the builtin function. Examples of builtin functions are arithmetic operators or the colon operator for generating sequences.

The last group are closures. Closures are user-defined functions written in R, and they adhere to the lazy evaluation semantics. All arguments to a closure are therefore allocated as promises: the expressions are bundled together with the enclosing environment.

As opposed to specials and builtins, which, being C routines, are called directly after preparing their arguments, closures need the interpreter to do some additional work. First, the actual unevaluated arguments have to be matched to the formal arguments of the closure. Then, a new environment has to be created and filled with the matched pairs of arguments. Only after this can the body of the closure be evaluated in the new environment. Also, a long jump target is set here to catch any explicit return calls from within the body.

Finally, the dispatch to the bytecode interpreter for objects compiled to bytecode is also found in the eval function.

## 1.3 BC compiler and interpreter

In an attempt to make R faster, a special internal representation for R code was developed, and a compiler from R to this bytecode was added in a package called compiler [15]. This also required some minor changes to the original

AST interpreter, namely adding a new `SEXP` type for the compiled objects, called `BCODESXP`, and handling of bytecode objects in the evaluator. A new evaluator was also added for interpreting the bytecode which is invoked by the AST version when it needs to evaluate a compiled object.

The compiler was written by Luke Tierney, and added as a standard package to R in 2011 in version 2.13.0. However, it was not used by default until version 3.4.0, released in late April 2017 ([2] and [3]).[15]

The BC compiler itself is implemented in R, and walks the abstract syntax tree of an expression being compiled in a similar manner that the AST interpreter does (but, of course, it does so at the R level by using introspection). However, instead of evaluating the code as it traverses the tree, it produces a code object. The code is then later executed by the BC interpreter, a separate virtual machine runtime system from the AST version. The compiler uses just a single pass, meaning that it only looks at the compiled expression once, and while doing so, produces a stream of instructions. A multi-pass version that would add optimization passes for the internal representation is planned to be explored in the future.

The bytecode objects produced by the compiler consist of two components. The first is an integer vector that encodes the code itself in the form of instruction opcodes interleaved with the instruction operands. The second is a general list that represents a constant pool. In the constant pool, important objects are stored, such as the source for the compiled expression, small constant objects, or promises. The compiler is designed such that each bytecode object has its own constant pool.

The compiler can be used explicitly to compile an expression or a closure. However, a more convenient way is to enable just-in-time compilation (JIT). Doing so causes the AST interpreter to invoke the compiler automatically when calling a closure that is not yet compiled.

The compiler comes with a disassembler that makes it possible to inspect the bytecode of an object, as is shown in listing 1.14.[16] The object is printed as a

[15] The default packages were compiled, but for additional packages and user code JIT was disabled and had to be explicitly enabled.

[16] The output was reformatted for the sake of readability.

list that starts with the `.Code` symbol, then follows the code vector (with the opcodes decoded), and last comes the constant pool.

The integers in the code that are not instructions represent immediate arguments to the instructions (the first element being an exception, as it encodes the version of the BC stored in the given object). In this particular instance, all immediates represent constant pool indices.

```
> f <- compiler::cmpfun(function(n) n + 1)
> f
function(n) n + 1
<bytecode: 0x367ee40>
> compiler::disassemble(f)
list(.Code,
     list(8L,
          GETVAR.OP, 1L,
          LDCONST.OP, 2L,
          ADD.OP, 0L,
          RETURN.OP),
     list(n + 1,
          n,
          1)
)
```

**Listing 1.14:** Disassembling a BC object

The virtual machine that executes R bytecode uses a stack oriented architecture. This means that a stack is used by the instructions at runtime to get their arguments and store their results. For example, the instruction that performs addition expects its two operands at the top of the stack. When it is executed it removes these two objects from the stack, adds them together and puts the resulting object back on the top of the stack.

The VM is implemented as a C routine that gets a bytecode object and an environment as arguments (similar to the AST interpreter). It verifies the BC version and then enters a loop that looks at the instruction stream in the BC object and dispatches to code that implements the given instruction. The loop is very carefully optimized by various techniques, such as using C preprocessor macros and threaded code. This will be discussed later in chapter 3.

The instruction set of the internal representation is designed to allow big parts of the AST interpreter internals to be reused. There are currently 123 instructions, some of which are described in table 1.2.

**Table 1.2:** Description of some GNU R bytecodes

| Instruction | Description |
|---|---|
| `RETURN.OP` | Take the top of stack and return it as a result |
| `GOTO.OP` | Unconditionally jump to a label |
| `BRIFNOT.OP` | Conditionally jump to a label |
| `POP.OP` | Remove the top of stack value |
| `LDCONST.OP` | Push a constant from the constant pool |
| `GETVAR.OP` | Look up the symbol binding and push it |
| `SETVAR.OP` | Update the symbol binding |
| `MAKEPROM.OP` | Create promise from a call argument |
| `CALL.OP` | Do function call |
| `CALLBUILTIN.OP` | Call builtin function |
| `CALLSPECIAL.OP` | Call special function |
| `MAKECLOSURE.OP` | Create closure (with environment) |
| `ADD.OP` | Arithmetic binary plus |
| `LT.OP` | Relational less than |
| `STARTASSIGN.OP` | Prepare for subassignment |
| `ENDASSIGN.OP` | Clean up after subassignment |
| `ISNULL.OP` | Test if top of stack is **`NULL`** |
| `COLON.OP` | Create integer sequence |

The compiler itself has in its heart the recursive function `cmp` that visits the AST of a given expression. It passes along a code buffer object (that contains the instruction stream and the constant pool) and a context object. When generating code, it writes into the code buffer, and uses the context to guide the compilation (it carries along information such as whether the expression should be followed by a return, if the result is ignored or not or if the expression is in a loop).

The expressions that are not self-evaluating are function calls, variable references, bytecode objects and promises. The rest is treated as being a constant. Bytecode objects and promises should not appear as literals in code, so they cause a compilation error if encountered.

Constant expressions are compiled by inserting the object into constant pool and generating a load instruction such as `LDCONST.OP` (that takes as an argument the index into the constant pool of the object).

For variable references, the symbol is inserted into the constant pool and then a `GETVAR.OP` instruction is emitted (although there is a special instruction for the "dot-dot-names" such as **`..1`**).

Everything else is a function call. When compiling a function call, multiple steps are required. First, the function to call has to be compiled. Usually this involves emitting an instruction that looks up the function by its name, but sometimes also compiling an expression that evaluates to a function. Then the arguments are compiled and code that prepares them on the stack is emitted. Finally, the call instruction is generated.

Since the compiler uses only a single pass, it has limited options of optimizing the generated code. The only optimization it performs, apart from those described in the next section, is constant folding. This is a very useful transformation that attempts to replace subtrees of an expression's AST that are constant (not only self-evaluating, but rather always evaluating to the same result) with the constant result.

Currently, constant folding is performed (depending on compiler options) on "small" expressions that consist of self-evaluating expressions, select base variables (like **`pi`**) and calls to some select base math functions (like **`sqrt`** or **`sin`**). In the current version, no deoptimization is possible.

### 1.3.1 BC compiler assumptions

When dealing with a language as dynamic as R, one needs to be very careful to distinguish compile time from execution time, and in particular to realize that between the two there is a window during which the state of the program can change.

It was explained earlier (see listing 1.2) that everything that happens in R is a function call, even constructs that have a special syntactic meaning, like **`if`**, **`while`**, **`break`** etc.

Furthermore, R uses lexical scoping. When a user starts an interactive session, they work in the global environment, which is the first environment in the hierarchy of environments known as the search path. The base environment (where all the default R bindings are defined) as well as any loaded packages are its ancestors in the search path. It is therefore possible to shadow the default bindings by creating new bindings in the global environment.[17]

Taking all this into consideration, to be conservative the compiler would have to compile everything as a function call, and then, at runtime, look up the current value of the callee binding and perform the call. Unfortunately, this almost completely negates the benefits of having a compiler at all, because the call is exactly what the AST interpreter does.

For this reason, the compiler assumes about certain functions that their meaning does not change in the meantime between compilation and execution. This is quite a reasonable assumption, since bindings for these special and builtin functions are rarely redefined, and it allows them to be properly compiled.

As a downside, the compiler produces code that is incorrect if the assumptions do not hold. Moreover, the bytecode interpreter has no way of telling that the code became incorrect, so it just executes it without any warning or error. To demonstrate, consider the code in listing 1.15.[18]

The reason for this behavior can be seen in listing 1.16 which shows the relevant part of the disassembled `fc` function. The `` `+` `` was inlined at compile time (because at the time, it really was an addition). When executing the function for the second time, however, the meaning of `` `+` `` was actually subtraction, and this was not reflected in the bytecode.

[17] The base is in fact the last environment in the search path. Its parent environment is a special empty environment.

[18] JIT needs to be turned off for this.

```
> f <- function(n) n + 1
> fc <- compiler::cmpfun(f)
> c(f(1), fc(1))
[1] 2 2
> `+` <- `-`
> c(f(1), fc(1))
[1] 0 2
```

**Listing 1.15:** Breaking the compiler

```
list(8L,
     GETVAR.OP, 1L,
     LDCONST.OP, 2L,
     ADD.OP, 0L,
     RETURN.OP)
```

**Listing 1.16:** Reason for the erroneous result

To reduce the chance of such errors, the compiler keeps track of local and global bindings visible at compile time. If a shadowing is detected the function in question is not inlined.

## 1.4 Why is R hard to optimize

This section is based on [9, 16].

Firstly, R is hard to optimize and that makes it slow. Of course, being an intepreted language, one cannot expect the performance of lagnuages like C that are compiled to native machine code. This is because during runtime, there is the inherent overhead of managing the virtual machine that executes a given program. For the AST interpreter, it entails walking the tree again with every evaluation of an expression. For the BC compiler and interpreter, there is first the compilation itself (which only happens once), but then dispatching of the instructions needs to be done in the interpreter loop.

Another matter is that R is inherently single threaded, it is very memory hungry (as it needs a lot of metadata about its internal structures) and all memory is allocated on the heap and managed and garbage collected by the runtime system of R.

Just to give an example, incrementing a variable in C is done in one machine instruction (and possibly one memory store if the variable is not allocated in a register). An AST interpreter has to navigate a tree data structure in memory which, for this example, would probably consist of at least one assign node, two variable lookup nodes, one constant node and one arithmetic operation node. For the bytecode interpreter, the amount of work is considerably reduced, but still there is the large gap between machine instructions and the same bytecode instructions executed by a virtual machine (that must, for example, perform all type checking and possibly type promotions dynamically at runtime).

Furthermore, all values are boxed in R (often more than once), so the interpreter has to work its way through multiple levels of indirection. For instance, a value can be boxed in a promise `SEXP`, which when forced produces a vector `SEXP`, which in turn contains the actual data.

That said, what makes R hard to optimize is that it is a very dynamic language that gives a very high degree of freedom to a programmer. At runtime, users have full access to all of the program data and representation. This means, for example, that not only the values of a function's arguments can be accessed, but also the code that is used to compute them.

Even though R did not go as far as Lisp which makes no distinctions between programs and data, it provides ways to transform code into text and the other way round (namely, the functions **`parse`** and **`deparse`**).

Non-standard evaluation and metaprogramming are possible by leveraging delayed evaluation and using funcions like **`substitute`** (that returns the parse tree for an unevaluated expression and at the same time possibly modifies it), **`quote`** (that simply returns its argument unevaluated) and **`eval`** (that evaluates an expression in a specified environment). R also provides means for creating embedded domain specific languages (e.g., the formula specification or the "grammar of graphics" of ggplot2).[19]

To conlude, R being the mixture of different paradigms that it is makes it quite difficult to reason about the code and optimize it. Adding together functional style, object systems, laziness, introspection, dynamic evaluation, computation

[19] `ggplot2` (homepage: `http://ggplot2.org/`) is another package by Hadley Wickham. It is widely used for data visualization.

on the language itself, explicit environment manipulation and more creates a very complex result.

Additionally, R has its semantics defined by its one major implementation (although attempts have been made to formalize the language, e.g., [9]). This further complicates things as there is no formal description of the language.

# CHAPTER 2

# About RIR

RIR is an alternative compiler for the R language.[20] It comes with its own internal representation, an interpreter for its bytecode and an abstract interpretation framework which provides a way to easily implement static analyses on top of the RIR bytecode.

RIR acts as a drop-in replacement for the GNU R bytecode compiler. It requires a patched version of GNU R that makes some slight adjustments that allow the standard GNU R expression evaluator function to interface with the RIR bytecode compiler and interpreter. RIR is written in C and C++ and is compiled as a shared library that can be dynamically loaded by R.[21]

Listing 2.1 shows how to manually load RIR (although a script *tools/R* is provided that does this automatically and turns on JIT, too).

The architecture is very similar to GNU R. The compiler generates bytecode for a stack oriented virtual machine, which is later executed by a bytecode interpreter. However, RIR is designed to be multi pass, and provides a framework for adding new analyses, transformations and optimizations.

The RIR bytecode was designed with analyses in mind. This lead to some design decisions that are quite different from GNU R.

[20] Homepage: `https://github.com/reactorlabs/rir`

[21] The interpreter uses C, the reason being that it could eventually become a part of GNU R. The compiler and analysis framework are intended as packages and are written in C++.

```
> dyn.load("~/rir/build/librir.so")  # path to the shared object
> source("~/rir/rir/R/rir.R")  # load the API for RIR compiler
> # RIR is now ready:
> f <- rir.compile(function() {})
> f
function() {}
<bytecode: 0x34b4510>
> rir.disassemble(f)
0x2f80538
   guard_fun_  { == 0x2077cd8
   push_  23 # NULL
   ret_
```

**Listing 2.1:** Loading RIR at runtime

In general, where GNU R uses complex instructions that have to cover a lot of work, RIR aims to achieve the same results by using lighter and more specialized instructions (reminiscent of the reduced instruction set ideals).

Firstly, the bytecodes have statically defined how they behave, i.e. if they have observable side-effects and how many elements they pop off and how many they push on the stack. This information allows the analyses to better reason about the code.

Secondly, RIR instructions aim to be predictable and self-contained.

For instance, when compiling a call, GNU R emits `MAKEPROM.OP` instructions for non-constants. Later at runtime, when a list of arguments is built on the stack, their behavior depends on the type of the callee. Each decides at runtime whether it will allocate a promise (for closures), evaluate the code right away (for builtins) or do nothing at all (for specials).

RIR behaves much more regularly in such a case. It prepares the arguments at runtime and either evaluates them all eagerly or allocates them all as promises.

Another example is a **`for`** loop. GNU R uses three instructions that handle everything: one for initializing the loop variable, one for advancing it along the loop sequence, and one to clean up.

RIR on the other hand emits instructions for every operation, such as incrementing the index variable, checking its bounds, jumping out of the loop conditionally, extracting the appropriate element of the loop sequence, storing it in the loop variable etc.

Furthermore, as opposed to the assumptions used by the GNU R compiler (see 1.3.1), RIR uses guard instructions that check at runtime whether the assumption holds or not (i.e. whether the inlined function is still the same as it was at compile time).

RIR is at present slower than GNU R. This is caused by several factors. First, RIR falls back to the AST interpreter more often than GNU R. Second, RIR allocates more memory than GNU R. For example, GNU R does not create promises out of constant arguments passed to a closure. RIR, on the other hand, creates promises from each argument. Moreover, the GNU R bytecode interpreter loop is very optimized and efficient, as opposed to RIR's.

Some examples of RIR instructions are listed in table 2.1.

**Table 2.1:** Description of some RIR bytecodes

| Instruction | Description |
|---|---|
| `push_` | Push on top of stack |
| `pop_` | Pop off the stack |
| `ldvar_` | Look up binding |
| `stvar_` | Update binding |
| `close_` | Create closure |
| `ret_` | Standard return |
| `return_` | Non-local return |
| `asbool_` | Convert top of stack to **`TRUE`** or **`FALSE`** |
| `brtrue_` | Jump based on top of stack |
| `brobj_` | Jump if top of stack is object |
| `br_` | Jump unconditionally |
| `dup_` | Duplicate top of stack |
| `dup2_` | Duplicate two top elements |
| `add_` | Arithmetic addition |
| `guard_fun_` | Check function is same as at compile time |

| Instruction | Description |
|---|---|
| `extract1_` | Get an element of vector |
| `inc_` | Increment integer on top of stack |

# CHAPTER 3

# Improvements

In this chapter I will discuss in detail the changes made to RIR in an attempt to bring it up to speed with GNU R byte-compiled code. The improvements can be divided into several categories:

The first group consists of the extensions of the bytecode instruction set itself. Here, new instructions are added for some functions that GNU R inlines but which were missing in the RIR (and thus were compiled as standard calls).

The next section talks about changes made to the RIR compiler. These consist mainly of loop context handling.

Finally, the RIR interpreter loop itself was refactored to run more efficiently, and this is described in the last section of this chapter.

Throughout, for ensuring that the improvements had positive effects, a kind of microbenchmarks, such as the one in listing 3.1, were checked by hand in fresh sessions of GNU R (with JIT disabled and with JIT set to 2) and RIR (with JIT enabled). In the code, a function is defined and measured repeatedly. The final reported time is computed as arithmetic mean of only a tailing part of the runs. This is to ensure a proper warmup (i.e. everything is byte-compiled by the JIT and possibly the processor branch predictors warm up).

```
f <- function() {
    i <- 10000000L
    while (i > 0) i <- i - 1
}
t <- c()
for (x in 1:10) t <- c(t, system.time(f())[[3]])
mean(t[5:10])
```

**Listing 3.1:** Microbenchmark code

## 3.1 Instruction set extensions

The GNU R bytecode compiler assumes certain invariants about the code at compile time, as was described in section 1.3.1. Of course, the instruction set of the default compiler reflects this. Having specialized bytecode instructions for specific tasks is where virtual machines generally get a lot of speedup [8].

The first step was to work through the documentation of the GNU R bytecode compiler [15]. Here, all the inlining done by default by the compiler was determined and experiments were carried out to compare their list to RIR, usually by using the disassemblers of both GNU R and RIR, examining the results of compilation of different calls and studying the C source codes.

The GNU R compiler has four different levels of the *optimize* option (from 0 to 3) and the amount of inlining performed is directly influenced by this setting. The default value is 2, at which the compiler inlines functions in the base packages (including those that are syntactically special or considered core language functions) that are not shadowed at compile time (by function arguments and local bindings). Some of these are listed in table 3.1.

**Table 3.1:** Examples of inlined functions

| Function | Optimize level | Note |
|---|---|---|
| \`+\`, \`-\`, \`*\`, \`<\`, \`==\` etc. | 2 | Operators |
| \`(\` | 2 | Expression grouping |
| \`{\` | 2 | Block |
| \`if\` | 2 | Control flow |
| \`repeat\`, \`while\`, \`for\` | 2 | Control flow |

| Function | Optimize level | Note |
|---|---|---|
| `` `break` ``, `` `next` `` | 2 | Control flow |
| `return` | 2 | Control flow |
| `function` | 2 | Closure constructor |
| `sin`, `cos`, `floor`, `sign` etc. | 3 | 1 arg. math |
| `is.null`, `is.atomic` etc. | 3 | Predicates |

The inlining happens in both GNU R and RIR when the compiler sees a function call. Both first try to inline calls to special and builtin functions. If the inlining fails, the compilers fall back to the standard call mechanism, and that means calling the same C routines that the AST interpreter uses, effectively running interpreted code.

Thus the way to speed things up is to add more special cases that handle code that originally fell back to the AST interpreter, the trade-off being that the generated code is not always safe (see listing 1.15).

When adding a new bytecode instruction, several steps have to be performed. First, the instruction has to be added to the instruction list in the *insns.h* header file. The new instruction needs to have a name and also have some properties specified.

These are *imm* (the number of immediate arguments that the instruction expects – immediates are inserted directly into the code stream), *pop* and *push* (the number of elements that the instruction removes from and adds to the stack, respectively) and *pure* (a flag that says if the instruction has side-effects, e.g., it can force a promise – purity is good to know for optimization purposes).

The format is displayed in listing 3.2. `DEF_INSTR` is a C preprocessor macro that needs to be defined each time this header file is included, and can be used to get information about the instructions. For instance, a method that returns the number of immediates is shown in listing 3.3.

Second, the instruction has to be manually added at some places in the class that implements the bytecode instruction type and is used throughout the compiler and the analysis framework. This step is mostly mechanical.

```
DEF_INSTR(gt_, 0, 2, 1, 0)
```

**Listing 3.2:** Adding new opcode to RIR bytecode

```
static unsigned immCount(Opcode bc) {
    switch (bc) {
#define DEF_INSTR(name, imm, opop, opush, pure)                              \
    case Opcode::name:                                                       \
        return imm;
#include "insns.h"
    default:
        assert(false);
        return 0;
    }
}
```

**Listing 3.3:** Getting number of immediates for instructions

After that, the compiler must be taught to use the instruction. This is done during the inlining step. A special case for the function call that the instruction implements has to be added. In this special case, the instruction opcode, together with any other instructions needed (such as the guard instructions explained in chapter 2) are inserted into a code stream. See listing 3.4 for a code snippet that demonstrates this.

It is also here while the instructions are being added into the stream that constant pool is filled (since the indices of constant pool objects are needed as immediates).

```
bool compileSpecialCall(Context& ctx, SEXP ast, SEXP fun, SEXP args_) {
    RList args(args_);
    CodeStream& cs = ctx.cs();
    // ...
    if (fun == symbol::Add && args.length() == 1) {
        // emit instructions...
        return true;
    }
    // ...
    return false;
}
```

**Listing 3.4:** RIR compiler inlining

Finally, the instruction itself has to be implemented and added to the dispatching mechanism in the interpreter. This is where the code that executes the instruction is located. The skeleton of the RIR evaluator can be seen in listing 3.5. BEGIN_MACHINE, INSTRUCTION and LASTOP are all C macros that implement dispatching.

```
SEXP evalRirCode(Code* c, Context* ctx, SEXP env, unsigned numArgs) {
    /* ... */
    BEGIN_MACHINE {
        /* ... */
        INSTRUCTION(eq_) { /* body */ }
        /* ... */
        LASTOP;
    }
eval_done:
    return ostack_pop(ctx);
}
```

**Listing 3.5:** RIR evaluator

Summary of the various newly added instructions is in table 3.2. One that does not fit anywhere is nop_. As one would expect, it does not do anything, takes no immediates or stack arguments and is, by definition, pure. It will be useful later when eliminating loop contexts. The rest are described in the following sections.

**Table 3.2:** Newly added BC instructions

| Name | *imm* | *pop* | *push* | *pure* |
|---|---|---|---|---|
| nop_ | 0 | 0 | 0 | yes |
| gt_ | 0 | 2 | 1 | no |
| le_ | 0 | 2 | 1 | no |
| ge_ | 0 | 2 | 1 | no |
| eq_ | 0 | 2 | 1 | no |
| ne_ | 0 | 2 | 1 | no |
| uplus_ | 0 | 1 | 1 | no |
| uminus_ | 0 | 1 | 1 | no |
| not_ | 0 | 1 | 1 | no |
| colon_ | 0 | 2 | 1 | no |

| Name | *imm* | *pop* | *push* | *pure* |
|---|---|---|---|---|
| `ldvar2_` | 1 | 0 | 1 | no |
| `stvar2_` | 1 | 1 | 0 | no |

### 3.1.1 Relational operators

Originally, RIR only had `` `<` `` out of the six usual relational operators. The rest, `` `>` ``, `` `<=` ``, `` `>=` ``, `` `==` `` and `` `!=` `` were being compiled as calls to the builtin C routines, but were newly added.

All these instructions behave in the same way as arithmetic binary operators do. They take no immediate arguments, and instead expect their operands to be left for them at the top of the stack. They pop two values off the stack, compute the respective operation, and push one result back. They are impure, since their arguments may be promises in which case they would have to force them.[22]

The speedup of adding these operations as standalone instructions lies in the fact that they are quite often called with "scalar" arguments, and as was mentioned before, R does not have any scalar values, since they are simply boxed in vectors of length one. If the operands are of a suitable type, and both have a single element, then a fast path can be taken and a call to the standard C builtin avoided.

Listing 3.6 shows the body of the `eq_` instruction. `INSTRUCTION`, `DO_RELOP` and `NEXT` are C preprocessor macros. The code first gets the two operands from the stack, then it performs the operation, puts the result back on the stack, and after that moves the control to the next instruction.

In listing 3.7 the fast paths are shown. After checking the types and lengths of the two operands, either a **`NA`** value is assigned to the result, or the particular operation is performed. This is possible because internally R uses the C types **`int`** and **`double`**, so the actual comparison can be done in plain C.

[22] Because R has **`eval`**, any arbitrary code may be run when forcing a promise.

```
INSTRUCTION(eq_) {
    SEXP lhs = ostack_at(ctx, 1);
    SEXP rhs = ostack_at(ctx, 0);
    DO_RELOP(==);
    ostack_popn(ctx, 2);
    ostack_push(ctx, res);
    NEXT();
}
```

**Listing 3.6:** The eq_ instruction

For the `<` operator, only combinations of integer and double scalars had fast paths implemented. This was amended and for all relational operators there is now also a fast path for comparing two logical values. Furthermore, the relational operators do not need to allocate a new vector for their result, since the R logical objects are singletons and they are simply assigned to the result.

```
#define DO_RELOP(op) do {                                              \
    if (IS_SIMPLE_SCALAR(lhs, LGLSXP) &&                               \
        IS_SIMPLE_SCALAR(rhs, LGLSXP)) {                               \
        /* handle NA */                                                \
        res = *LOGICAL(lhs) op *LOGICAL(rhs) ? R_TrueValue             \
                                             : R_FalseValue;           \
        break;                                                         \
    } else if (/* ... */) {                                            \
        /* handle real + real, real + int, int + real, int + int */    \
    }                                                                  \
    BINOP_FALLBACK(#op);                                               \
} while (false)
```

**Listing 3.7:** The DO_RELOP macro

If the fast paths fail, the instruction falls back to the standard routine that handles vectorized operations, vector recycling, type promotion and also any possible problems (e.g., concerning incompatible operand types). This is shown in listing 3.8. For every instruction, a static variable is used in this case to cache the builtin function (or rather a pointer to it), so that the lookup (which is an expensive operation) is only performed once. Here, the operands are added to a linked list (that the builtin C routine expects), and the builtin is called.

```
#define BINOP_FALLBACK(op) do {                                       \
    static SEXP prim = NULL;                                          \
    static CCODE blt;                                                 \
    if (!prim) {                                                      \
        /* look up builtin */                                         \
    }                                                                 \
    SEXP call = getSrcForCall(c, pc - 1, ctx);                        \
    SEXP argslist = CONS_NR(lhs, CONS_NR(rhs, R_NilValue));           \
    ostack_push(ctx, argslist);                                       \
    res = blt(call, prim, argslist, env);  /* call builtin */         \
    ostack_pop(ctx);                                                  \
} while (false)
```

**Listing 3.8:** The BINOP_FALLBACK macro

### 3.1.2 Unary operators

Unary operators are in principle the same as binary. If their operand has one element and appropriate type, a fast path can be taken. The logical negation has a fast path for logical scalars in addition to numeric.

R has two arithmetic unary operators, `+` and `-`, and logical negation `!`. These are all rather straightforward, they do not have immediates, pop one argument and push one result back.

Since the instruction bodies correspond closely to those of binary operators (except they only have a single operand), a snippet of a compiler code that emits these is shown instead (listing 3.9). It is taken out of a function that performs the inlining. It returns true if successful, and false otherwise, in which case the compilation proceeds to the standard call mechanism (i.e. emitting a ldfun_ instruction, compiling the call arguments as promises and finally emitting a call_).

In the compiler, inserting into the code stream cs is done using the overloaded **operator** <<. Factory methods are used to create the bytecode instruction objects (from their arguments, the immediates for the instructions are generated). A reference to the original AST of the compiled call is saved into the code stream, too.

The bytecode runtime system uses a stack architecture, therefore using recursion ties in elegantly. First, a guard instruction is inserted (see chapter 2 for explanation). Then the bytecode that computes the value of the operand is emitted recursively. Lastly, the operator instruction is added that processes the result left for it on the stack.

```
if (args.length() == 1 &&
    (fun == symbol::Add || fun == symbol::Sub ||
     fun == symbol::Not)) {
    cs << BC::guardNamePrimitive(fun);
    compileExpr(ctx, args[0]);
    if (fun == symbol::Add)
        cs << BC::uplus();
    else if (fun == symbol::Sub)
        cs << BC::uminus();
    else if (fun == symbol::Not)
        cs << BC::Not();
    cs.addSrc(ast);
    return true;
}
```

**Listing 3.9:** The piece of code emitting unary operators

### 3.1.3 The colon operator

The colon operator `` `:` `` in R provides a convenient way to generate sequences. It is used very often, notably in **`for`** loops as an integer control sequence for the loop variable. The values it generates can be both increasing and decreasing, and they differ by 1. If the starting value is an integer, then the vector is also integer.

A `colon_` instruction was added that, similarly to arithmetic operators, takes no immediates, expects two operands on the top of the stack and pushes back the resulting sequence object. It is impure, because, same as the operators, its operands could be unevaluated promises.

Adding it to the compiler was actually the same as adding an arithmetic binary operator. The instruction itself adds a fast path for combinations of integer and double operands (the doubles apply only if they represent an integer up to a rounding error). The fast path allocates an integer vector of appropriate length and fills it with the sequence values.

```
INSTRUCTION(colon_) {
    /* ... */
    if (IS_SIMPLE_SCALAR(lhs, INTSXP)) {
        int from = *INTEGER(lhs);
        if (IS_SIMPLE_SCALAR(rhs, INTSXP)) {
            /* ... */
        } else if (IS_SIMPLE_SCALAR(rhs, REALSXP)) {
            double to = *REAL(rhs);
            if (from != NA_INTEGER && to != NA_REAL &&
                    R_FINITE(to) && INT_MIN <= to &&
                    INT_MAX >= to && to == (int)to) {
                res = seq_int(from, (int)to);
            }
        }
    } else if (IS_SIMPLE_SCALAR(lhs, REALSXP)) {
        /* real + int, real + real */
    }
    if (res == NULL) {
        BINOP_FALLBACK(":");
    }
    /* ... */
}
```

**Listing 3.10:** The `colon_` instruction

### 3.1.4 Superassignment

Superassignment operator ``` `<<-` ``` differs from normal assignment in that it works in an enclosing environment. Thus, the local environment where superassignment occurs is skipped during the binding lookup. For simple assignment of the form `x <<- y`, that is all there really is.

The semantics of a complex subset assignment are a bit more complicated, as is shown in listing 3.11 (taken from [13]), because it is not a matter of simply creating or changing a binding, but a part of a binding's object has to be extracted and modified first. This model applies recursively for still more complex assignments.

The target object is looked up and stored into a variable ``` `*tmp*` ``` (it is not recommended to use this name for anything in user code, since, as a side-effect, this will overwrite and then delete it). Then, a function name is constructed from the left-hand side call expression by appending an assignment arrow. The resulting name must refer to a function that takes the same arguments as the

```r
# The result of this command...
x[3:5] <- 13:15
# ... is as if the following had been executed
`*tmp*` <- x
x <- "[<-"(`*tmp*`, 3:5, value=13:15)
rm(`*tmp*`)
```

**Listing 3.11:** Complex subset assignment

original one as well as an additional value argument. This function is called, the right-hand side expression is passed as the value, and its result is stored to the target. Then the temporary binding is removed.

For superassignment the same principles apply, however, the target binding (and only the target binding) is looked up in an enclosing environment of the expression.

Two new bytecode instructions were added for handling the superassignment semantics of looking up bindings. The first is for loading a symbol. It takes one immediate argument, an index into the constant pool where it finds the symbol to look up. It does not pop anything from the stack but pushes one object, the value of the binding. The second is symmetrical to the first, it takes an immediate constant pool index, pops one object, does not push anything. Both are impure: the load may evaluate a promise, and the store modifies non-local bindings (unlike normal `stvar_` that writes only into the local environment and hence is pure).

These new bytecodes were added to the compiler at a point where it inlines normal assignment and subset assignment.

In the instructions themselves, the environment for looking up bindings gets replaced with its enclosing environment. This is shown in listing 3.12, where the call to R internal `setVar` function takes as the last argument the enclosing environment of the current one. The symbol is read from the constant pool at the index determined by the immediate. The value to store is taken from the stack.

One additional detail comes up: how the program counter (PC) is manipulated. With every instruction, the dispatcher reads its opcode from the current position of the PC and moves the PC to the next position (opcodes take up a single byte). If the instruction has any immediates, it is up to the instruction code to manipulate the PC accordingly. This is done by the advanceImmediate macro.

```
INSTRUCTION(stvar2_) {
    SEXP sym = readConst(ctx, readImmediate());
    advanceImmediate();
    SEXP val = ostack_pop(ctx);
    INCREMENT_NAMED(val);
    setVar(sym, val, ENCLOS(env));
    NEXT();
}
```

**Listing 3.12:** The stvar2_ instruction

## 3.2 Compiler modifications

In this section, some changes that were made directly to the compiler are discussed. Included at the end is a section about refactoring loops compilation in a way that saves some jumps. Ultimately, this was not used because it turned out that it had a negative impact on performance.

### 3.2.1 Loop context removal

One of the consequences of lazy evaluation in R is that the control flow statements **break** and **next** can in fact be non-local jumps (i.e. across multiple stack frames).[23] Listing 3.13 presents a possible way of producing non-local **break**. The arguments to the foo function are wrapped in promises, and may or may not be evaluated. However, if the second argument ever gets evaluated, it has to break out of the **repeat** loop that called foo.

To implement this behavior, C long jumps have to be used. Thus, both GNU R and RIR have a bytecode instruction that sets up the long jump, and another that cleans up after the loop finishes.

[23] This is true for **return** as well, however non-local returns are not handled by loop contexts.

```
function(n) {
    repeat {
        foo(n, break)
        n <- n - 1
    }
}
```

**Listing 3.13:** Context for `break` required

Quite often the runtime loop context is not needed, as is shown in listing 3.14. In such cases only local jump suffices because the `break` can only ever be evaluated in the same stack frame.

```
function(n) {
    repeat {
        if (n <= 0) break
        n <- n - 1
    }
}
```

**Listing 3.14:** Safe `break`

The rule for when the runtime context is needed and when not is as follows. If a loop contains any `break` or `next` statements that are wrapped in a promise, the loop context is needed. However, if all such statements are wrapped inside their promises in another loop (or the closure constructor `function`), the context can still be left out.

RIR used to create the runtime loop contexts for all loops, so a mechanism was added to enable skipping them in the safe cases. The compiler uses a `Context` object that holds a stack of `CodeContext` objects. Each code context in turn contains a code stream for the generated code and its own stack of `LoopContext` objects. The original structure can be seen in listing 3.15.

A new `CodeContext` was pushed when the compiler entry point was invoked, and then each time a promise was compiled. The system of mutually recursive compiler functions then worked in the code context that was on the top of the stack. When a loop was encountered, a new `LoopContext` was pushed to the stack of the top code context.

```
class Context {
  public:
    class LoopContext { /* ... */ };
    class CodeContext {
      public:
        CodeStream cs;
        std::stack<LoopContext> loops;
        // ...
    };
    std::stack<CodeContext> code;
    // ...
};
```

**Listing 3.15:** RIR compiler context

The loop context objects contain the target labels for the local control flow. A new flag was added that signals whether a runtime loop context is needed for the given loop. The class is shown in listing 3.16.

```
class LoopContext {
  public:
    LabelT next_;
    LabelT break_;
    bool context_needed_ = false;
    LoopContext(LabelT next_, LabelT break_)
        : next_(next_), break_(break_) {}
};
```

**Listing 3.16:** Loop context class

To implement the rules for skipping runtime loop contexts, a `PromiseContext` class was added that is used when the compiler compiles a promise. This is shown in listing 3.17.

```
FunIdxT compilePromise(Context& ctx, SEXP exp) {
    ctx.pushPromiseContext(exp);  // instead of CodeContext
    compileExpr(ctx, exp);
    ctx.cs() << BC::ret();
    return ctx.pop();
}
```

**Listing 3.17:** Compiling a promise

The `CodeContext` was changed to store a pointer to its parent (a context below it on the stack). This change allows for finding out transitively if the compiler is in any loop, not just one from the current context, and also to set the `LoopContext` flag in the first enclosing loop. The new functionality is presented in listing 3.18

```
class CodeContext {
  public:
    // ...
    CodeContext* parent;
    bool inLoop() {
        return !loops.empty() || (parent && parent->inLoop());
    }
    void setContextNeeded() {
        if (loops.empty() && parent)
            parent->setContextNeeded();
        else
            loops.top().context_needed_ = true;
    }
};
```

**Listing 3.18:** Hierarchy of `CodeContext` objects

The `PromiseContext` class inherits from `CodeContext`. When compiling **`break`** or **`next`** in a promise, the parent context (i.e. where the promise was created) is notified to set the loop context flag. This is shown in listing 3.19.

```
bool loopIsLocal() override {
    if (loops.empty()) {
        parent->setContextNeeded();  // set flag in parent
        return false;  // and do not inline
    }
    return true;
}
```

**Listing 3.19:** Promise context class

One last missing piece is being able to remove an instruction from a code stream. The stream originally did not have this functionality, but it was added to allow removing unnecessary loop context instructions. The trick used is to replace the instruction opcode together with its immediates by `nop_` instructions (which get removed later during BC cleanup).

Finally, when compiling the loops, the instruction `beginloop_` is emitted just as before. However, after the loop is compiled, it is removed from the stream if the context is not needed. The code for the **repeat** loop is shown in listing 3.20.

```
if (fun == symbol::Repeat) {
    // ...
    ctx.pushLoop(loopBranch, nextBranch);
    unsigned beginLoopPos = cs.currentPos();
    cs << BC::beginloop(nextBranch)
       << loopBranch;
    compileExpr(ctx, body);
    cs << BC::pop() << BC::br(loopBranch)
       << nextBranch;
    if (ctx.loopNeedsContext())
        cs << BC::endcontext();
    else
        cs.remove(beginLoopPos);
    cs << BC::push(R_NilValue) << BC::invisible();
    ctx.popLoop();
    return true;
}
```

**Listing 3.20:** **repeat** loop inlining

Compiling **next** and **break** then involves checking with the context if inlining is ok (i.e. the statement is not in a promise or it is in a loop in a promise). The code for inlining **break** is in listing 3.21.

```
if (fun == symbol::Break) {
    if (!ctx.inLoop()) return false;
    if (ctx.loopIsLocal()) {
        cs << BC::guardNamePrimitive(fun)
           << BC::br(ctx.loopBreak())
           << BC::push(R_NilValue);
        return true;
    }
}
```

**Listing 3.21:** **break** inlining

### 3.2.2 BC cleanup

A few small improvements were made to the optimization pass that RIR runs on the bytecode after it is compiled. The recursive compiler and possibly other transformations applied to the bytecode sometimes produce sequences of BC instructions where some are redundant or can be replaced. Examples of this include loading a variable twice or pushing to the stack and immediately popping.

The modified compilation of loops described previously produces for one some nop_ instructions. Moreover, the pair of instructions pick_ 1; pick_ 1 occurred quite often. The pick_ instruction removes the stack element at the index given by its immediate and puts it to the top of stack. If the code picks the element just under the top of stack twice in a row, the effect is as if nothing happened. Therefore, the code in listing 3.22 was added to the BC cleanup pass to fix these.

The two methods are located in an analysis class that implements the visitor pattern over instructions. Thus adding new cleanup code means simply overriding the particular bytecode instruction.

```
void nop_(CodeEditor::Iterator ins) override {
    CodeEditor::Cursor cur = ins.asCursor(code_);
    cur.remove();
    return;
}
void pick_(CodeEditor::Iterator ins) override {
    if (ins != code_.begin() &&
        ins.asCursor(code_).bc().immediate.i == 1) {
        auto prev = ins - 1;
        if ((*prev).is(Opcode::pick_) && *ins == *prev) {
            CodeEditor::Cursor cur = prev.asCursor(code_);
            cur.remove();
            cur.remove();
            return;
        }
    }
}
```

**Listing 3.22:** nop_ and double pick_ 1 elimination

Finally, the cleanup analysis is run iteratively until a constant number of runs happens or the analysis reaches a fixpoint (i.e. it does not change the code anymore), whichever comes first. The constant limiting the number of runs was previously set to 2, which often left a lot of easy cleanup unfinished. This limit was therefore increased to 10.

### 3.2.3 Loop refactoring

Theoretically, it is possible to optimize the code generated for **while** loops by swapping the loop's body and its condition. The intuitive way is to compile the condition first, then insert a conditional jump to the end of the body, then compile the body and jump unconditionally to the top again. In pseudocode this is shown in listing 3.23.

```
COND:
    // code for the condition
    brfalse_ END
BODY:
    // code for the body
    br_ COND
END:
```

**Listing 3.23:** **while** loop bytecode

However, if the condition and the body are swapped, half of the jumps can be saved. The body is compiled first, then follows the condition and a conditional jump back to the body. Additionally, a single unconditional jump is inserted before the body. Listing 3.24 demonstrates this.

```
    br_ COND
BODY:
    // code for the body
COND:
    // code for the condition
    brtrue_ BODY
END:
```

**Listing 3.24:** Refactored **while** loop bytecode

Similar change can be done to **for** loops. The **repeat** loops are infinite by design and do not have a condition, so nothing can be done there.

Unfortunately, this change proved to be a slowdown rather than a speedup in anything more complicated than a loop with an empty body, and even there the gains were quite small.

This is probably caused by changing the backward jump from unconditional to conditional. Because RIR profiles the backward jumps (they signal the presence of a loop), it seems that one forward conditional jump combined with one backward unconditional are cheaper than a single conditional backward jump.

## 3.3 Interpreter refactoring

As it turned out, the biggest speedup was gained by refactoring the RIR bytecode interpreter. Originally, the interpreter was quite straightforward. The main evaluator function `evalRirCode` contained in its core an infinite loop, and in its body there was a large switch statement with one case for each bytecode instruction.

In the switch cases there was a call to a function that implemented the instruction. These functions were defined with the C **`inline`** and **`static`** modifiers, and to make sure the compiler really inlined them, the GCC specific attribute `__attribute__((always_inline))` was also used.

However, the compiler clearly had trouble with optimizing this version, possibly due to alias analysis being degraded (e.g., the interpreter passed the PC around as a pointer to a pointer, which means double indirection).[24]

In the interpreter refactoring, I have built on the version where instruction functions were replaced by macros and directly used in the interpreter loop. Further, the infinite loop was replaced by a label and a **`goto`**. The dispatching was left to the **`switch`** statement as before. One level of indirection was removed from the PC handling and the bytecode access functions that use the PC were also turned into macros.[25]

[24] The inlining itself was actually happening, because the compiler would otherwise complain [5].

[25] Done by Oli Flückiger.

To demonstrate the difference, a microbenchmark and its results are presented in listings 3.25 and 3.26.

```
> f <- function() {
+     i <- 10000000L
+     while (i > 0) i <- i - 1
+ }
> t <- c()
> for (x in 1:10) t <- c(t, system.time(f())[[3]])
> mean(t[5:10])
[1] 1.141833
```

**Listing 3.25:** Effects of inlining instructions by hand – before

```
> # same code as before...
> mean(t[5:10])
[1] 0.6458333
```

**Listing 3.26:** Effects of inlining instructions by hand – after

Following the example of GNU R, where the interpreter almost never calls any functions and instead inlines the code via C preprocessor macros, some functions that manipulated the bytecode stack or retrieved the constant and source pool objects were converted into macros.

To further improve the interpreter, a switch based dispatch technique was replaced with threaded code dispatching. The technique is popular in languages such as Forth, and is described in, e.g., [4].

Dispatching is the mechanism that transfers control in a virtual machine interpreter between individual bytecode instructions. When a `switch` statement is used, the dispatching takes place at one location (the switch header), where the address to jump to for a given instruction is determined (by looking into a jump table). After the instruction finishes, another jump is needed to transfer the control back to the switch.

This can be made more efficient by eliminating this last jump. Since the code that does the dispatching is typically very short, it can be replicated at the end of every instruction. If a jump table can be created with the addresses of all instructions, then the `switch` can be removed, and completely replaced by looking into the jump table (which is indexed by the instruction opcodes) and jumping to the next instruction. This is called indirect threading because of the need to look for the addresses in the jump table.

There is no way to implement threading in standard C. Fortunately, GCC supports an extension called computed goto [6], that allows for taking the address of a label. Using this, it is easy to create the jump table. This is shown in listing 3.27. The `INSTRUCTION` macro becomes simply a label, the `NEXT` macro uses the jump table to get the address of the next instruction and computed goto to jump there. The jump table is constructed statically using the list of all instructions.

GNU R uses a similar technique, called direct threading. The difference is in removing even the lookup of a label address in the jump table, and instead encoding it directly into the opcodes of instructions. When finalizing the bytecode object, GNU R substitutes all the opcodes by the addresses of their code. Thus, the address to continue computation can be obtained by simply dereferencing the PC.

This was not implemented in RIR, mainly for the reason that doing so causes the opcodes to become longer, changing from a single byte to the size of a pointer on a given machine (i.e. to 8 bytes on a 64-bit architecture), and the benefits of changing switch dispatching to threading seemed quite small.

```
#define INSTRUCTION(name) op_##name:
#define NEXT() (__extension__ ({goto *opAddr[advanceOpcode()];}))
static void* opAddr[numInsns_] = {
#    define DEF_INSTR(name, ...) (__extension__ && op_##name),
#    include "ir/insns.h"
#    undef DEF_INSTR
};
```

**Listing 3.27:** Threaded dispatching

Lastly, some experimenting was done with forcing the C compiler to allocate certain variables in registers. Most of the local variables defined in the instructions were factored out to the beginning of the evaluator function. Then the form `register OpcodeT* pc asm ("r12");` was used to keep some of the often used variables in specific registers.

However, as described in [7], the compiler takes this only as a hint. No consistent improvement was achieved and, on the contrary, specifying two or more variables to be in registers seemed to degrade the performance.

It is probably best to allow the register allocator do its job without any influencing, as is recommended in the documentation.

# CHAPTER 4

# Evaluation

This chapter discusses how the changes implemented in chapter 3 impact the performance of RIR.

For evaluation, the Shootout benchmarks were used (see [8, 10]). These are small programs that focus on different parts of R, such as recursive calls, loops, vector arithmetic, string manipulation etc. The details are in table 4.1, taken from [8].

**Table 4.1:** The Shootout benchmark suite

| Benchmark | Description |
|---|---|
| binarytrees | Allocates and traverses binary trees<br>*GC benchmark, recursive calls, recursive lists* |
| fannkuchred | Solves a combinatorial problem<br>*Loops, indexing short vectors* |
| fasta | Generates DNA sequence by copying, rand. selection<br>*String operations, scalar arithmetic* |
| fastaredux | Solves same problem as fasta<br>*Adds more loops, vector indexing and arithmetic* |
| knucleotide | Finding patterns in gene sequences<br>*Uses environment as a hashmap, string operations* |
| mandelbrot | Calculates a Mandelbrot set (fractal image)<br>*Vector arithmetic on complex numbers* |

| Benchmark | Description |
|---|---|
| nbody | Solves the N-body problem (simulation)<br>*Arithmetic, Math with short vectors* |
| pidigits | Calculate digits of pi using spigot algorithm<br>*Arbitrary precision arithmetic in R (diverse code)* |
| regexdna | Matching, replacing regex-specified gene sequences<br>*Regular expressions (falls back to regex library)* |
| reversecompl | Computing reverse-complements for gene sequence<br>*String vector indexing using string names* |
| spectralnorm | Computing eigenvalue using power method<br>*Loops, function calls, scalar arithmetic* |

The benchmark problems come from The Computer Language Benchmarks Game.[26] The R language mutation was added by Leo Osvald [10]. There are several versions of most problems, the naive ones being more or less a literal translation from C or Java to R. The alternative implementations solve the same problems, however, they use different styles of programming that leverage various features available in R, as experienced R programmers would.

The benchmarks were run on a machine with *Intel(R) Core(TM) i5-6500* (4 cores) and 8 GB of memory. For every benchmark, three experiments were measured: GNU R interpreted code (JIT set to 0), GNU R byte-compiled code (JIT set to 2 and optimize to 2) and RIR compiled code (JIT set to 2).

Each experiment took place in a fresh R session. The benchmark code was sourced, then it was executed 12 times, and the last 8 measured times were logged. This ensured that the machine was warmed up properly. Any compile time overheads were disregarded. The same measurements were performed for every set of added features.

Figure 4.1 shows the summary of how much the difference between RIR and GNU R was lowered for each benchmark, as well as an overall average (smaller values are better).

[26] See `http://benchmarksgame.alioth.debian.org/`.

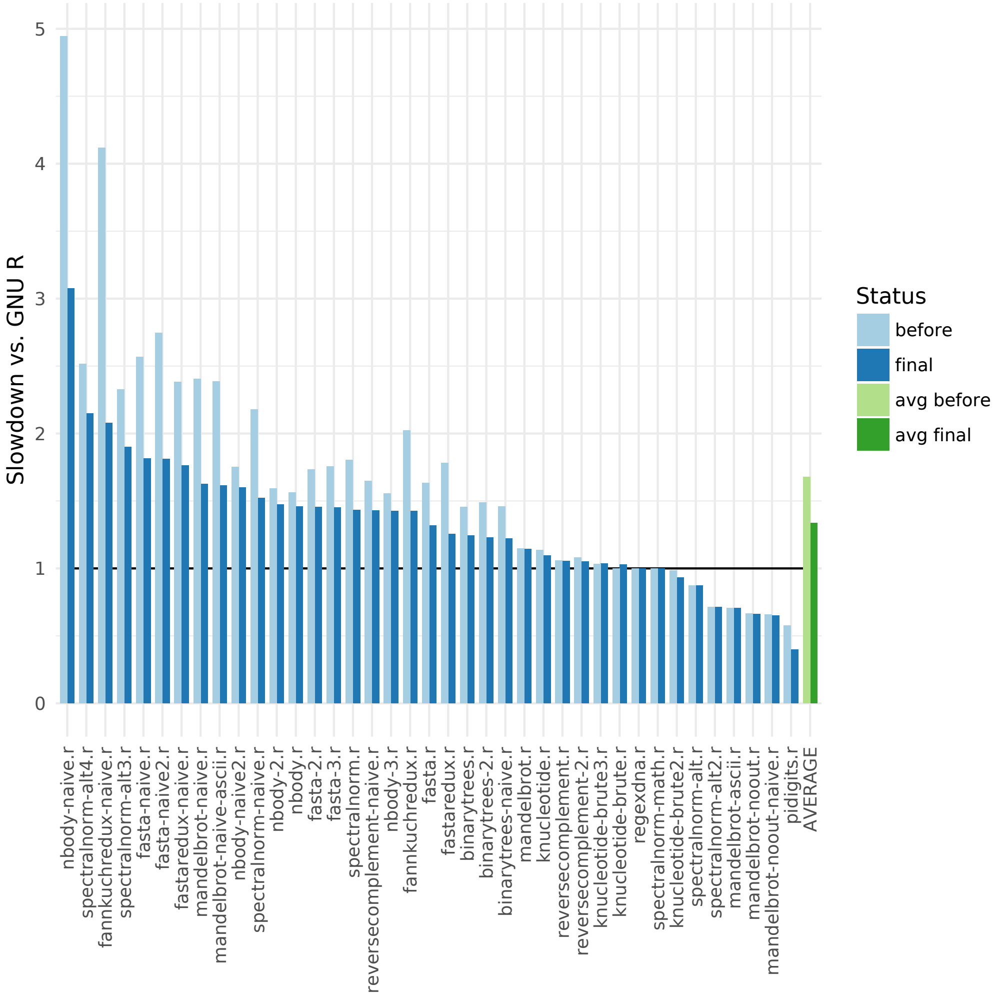

**Figure 4.1:** Overview of the slowdowns vs. GNU R

The lighter color refers to the state before any feature was added, the darker after all the changes described in chapter 3.

The slowdowns are computed relative to the running time of GNU R byte-compiled code, which was normalized to 1 (and is indicated by the solid black line).

Overall, an average slowdown against GNU R was lowered by about 50 % (from a factor of 1.678 to 1.336).

It can be clearly seen that the most speedup was gained for the naive versions of benchmarks. These are the programs that use a lot of nested loops, lot of arithmetic operations and have relatively long functions, which is exactly the area where the efforts of chapter 3 were focused.

The revisions that were used to monitor the progress, together with their features, are listed in table 4.2. The table also shows the average speedup over all benchmarks of each feature set, relative to the first revision.

**Table 4.2:** Git revisions used in the benchmarks

| Hash | Features | Avg. speedup |
|---|---|---|
| e1091b9 | State before the first changes | 1.0 |
| 594af0c | Added relational and unary operators | 1.029 |
| ce30085 | Loop contexts removal | 1.043 |
| 6c4f526 | BC cleanup and colon operator | 1.054 |
| f8e8238 | Superassignment operator | 1.081 |
| 12ef757 | Interpreter loop refactoring | 1.199 |
| ff73d75 | Use indirect threading | 1.206 |

Figure 4.2 captures the running times of each benchmark over the course of the work described in chapter 3.

In the figure, the light blue color represents performance of the GNU R bytecode. As all measurements were carried out on the same version of GNU R (namely 3.3.2) the times are identical.

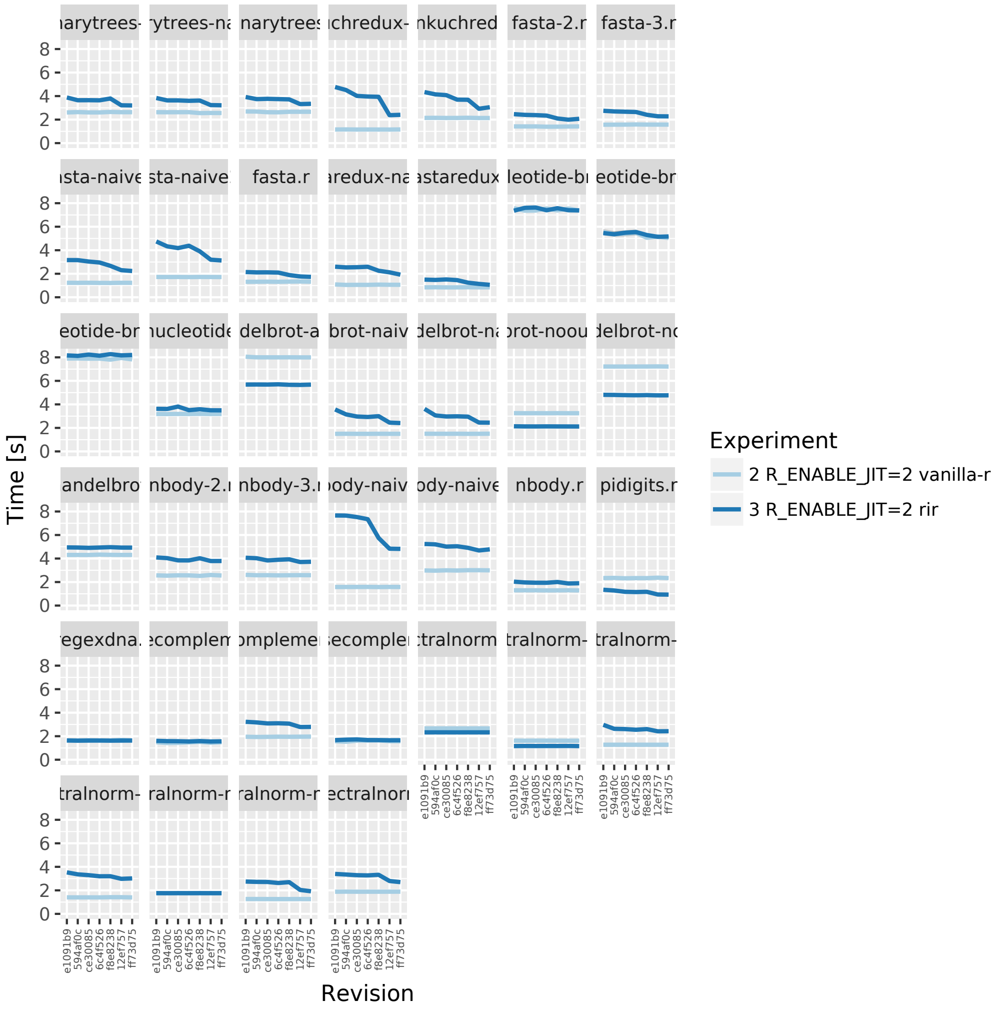

**Figure 4.2:** History of running times

RIR performance for some benchmarks was comparable to GNU R even before the first modifications, with some (notably *pidigits*) outperforming GNU R. In particular, the *pidigits* code is the longest of the benchmarks (in terms of lines of code), has a lot of user-defined functions and seems to spend most of the time executing the bytecode. Together with the *mandelbrot* versions where RIR is faster that do mostly just arithmetic in loops, this suggests that when there are few calls into plain R and all work is done in the bytecode interpreters, RIR is competitive.

Other benchmarks have remained more or less constant throughout. This is because they do not use that much bytecode instructions and instead spend the majority of time inside builtin C code. Therefore changes to the bytecode have almost no effect on them.

Generally, the biggest improvements were brought by inlining the instructions into the interpreter loop. With the exception of some benchmarks not affected by any changes, all others gained a boost. The nature of this change is such that it speeds up every bytecode instruction a tiny bit. Thus all benchmarks that spend some considerable time in the bytecode interpreter see an improvement.

For some benchmarks, a particular revision brings minor slowdowns. The reasons are not entirely clear and require some more investigation. For instance, an operator that checks a fast path and then falls back to the builtin call anyway may be slower than calling the builtin directly. On the other hand, taking into consideration branch prediction in the CPU and the fact that the call instruction has some overhead too (e.g., for preparing the argument list), this seems unlikely.

Two of the benchmarks were rerun with 20 iterations and their plots then zoomed to present a more detailed view. These were *nbody naive*, to show the steep bump, and *knucleotide brute*, because it originally had very wide confidence intervals. This suggested that perhaps the CPU was loaded by some other work during the original measurements.[27]

[27] Unfortunately, rerunning all of the benchmarks was not possible due to time constraints on the test server.

The detailed view of the *nbody naive* benchmark is in figure 4.3. The benchmark gained the most with inlining superassignment, as it uses it heavily in nested loops. Another large speedup was due to the interpreter refactoring and is so distinctive because the produced bytecode for the main hotspot (the function `advance`) is very arithmetic heavy with a lot of loops and subsetting and few calls. This means that long sequences of instructions get executed at once, thus leveraging the improved interpreter loop.

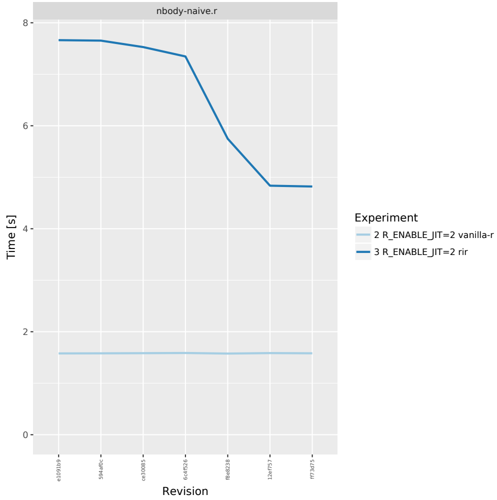

**Figure 4.3:** Detail of the *nbody naive* benchmark

As for the *knucleotide brute* benchmark in figure 4.4, the confidence intervals were indeed a byproduct of the CPU load and disappeared after the rerun. However, there are still some inexplicable fluctuations for the GNU R version, which will require future investigation.

The average speedup versus GNU R over the measured revisions is captured in figure 4.5. The result was obtained by averaging the measurements for each benchmark and then taking the average of those values. Of the changes, the most debatable is threading. It seems that this feature only starts to pay off for larger and longer programs.

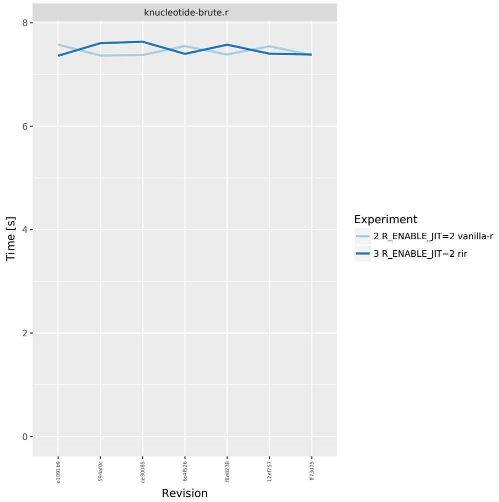

**Figure 4.4:** Detail of the *knucleotide brute* benchmark

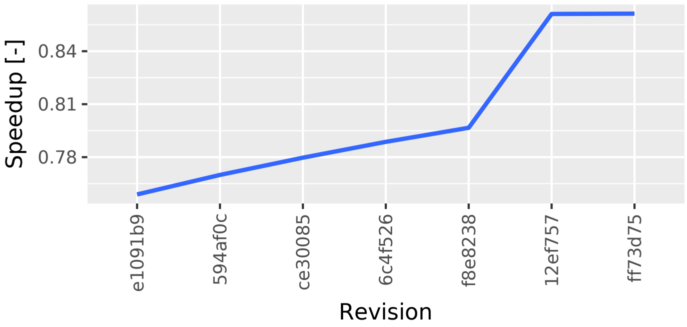

**Figure 4.5:** History of average speedup vs. GNU R

# Conclusion

The goal of this thesis was to bring RIR, an alternative bytecode compiler and interpreter for the R language, closer to R's reference impelemtation in terms of performance.

First, the R language and RIR were explored and their internal workings examined and compared. Experiments were carried out to discover their similarities and differences. Improvements to RIR infrastructure were implemented in three areas:

New bytecode instructions were added to its instruction set and employed in its compiler.

The compiler was extended to be more aware of its context and this was used to eliminate generating loop contexts in the common case when all loop control is local.

Finally, the interpreter loop was refactored and its dispatching mechanism changed from switch-based to threaded code.

All the implemented changes were evaluated using the Shootout benchmarks. Overall, the performance deficiency of RIR relative to GNU R was decreased by about 50 %.

RIR is still under active development and further work is needed.

The benchmarks where RIR lags behind would deserve a still more thorough investigation to uncover the remaining bottlenecks.

The call mechanism of RIR is generally slower than GNU R and ways to improve it should be explored. A way to avoid the unnecessary promise allocations for constant arguments while preserving the ideas of RIR should be found.

Still more extensions to the instruction set are possible. It could be worth looking into superinstructions (i.e. identifying what common pairs, maybe even triples of instructions occur often and combining them).

The abstract interpretation framework of RIR is another direction for future efforts. It can be improved and extended on one hand, and on the other be used to implement new analysis and optimization passes for RIR bytecode. For instance, promises could be avoided if an interprocedural analysis showed that the callee for sure evaluates its arguments.

In the early stages of this thesis, unsuccessful experiments were carried out with a tool called STOKE.[28]

STOKE is a stochastic superoptimizer for the x86-64 architecture developed at Stanford University. It tries to optimize code sequences by randomly searching the space of possible program transformations. Repeatedly applying minor changes can surprisingly produce code that runs faster than the original. New and non-obvious ways to compute equivalent results can sometimes be discovered.

STOKE is able to optimize code for performance or size, syntesize implementations from scratch and verify equivalence between different code sequences for all possible inputs.

Unfortunately, it appears that it has trouble with running on dynamically linked code, and thus could not be used directly. New ways of applying it to RIR could be explored.

[28] See `https://github.com/StanfordPL/stoke`

# Bibliography

1. BURNS, Patrick. *The R Inferno* [online]. 2011 [visited on Apr. 30, 2017]. Available from: `http://www.burns-stat.com/documents/books/the-r-inferno/`.
2. DALGAARD, Peter. *The R-announce Archives: R 2.13.0 is released* [online]. 2011 [visited on Apr. 30, 2017]. Available from: `https://stat.ethz.ch/pipermail/r-announce/2011/000538.html`.
3. DALGAARD, Peter. *The R-announce Archives: R 3.4.0 is released* [online]. 2017 [visited on Apr. 30, 2017]. Available from: `https://stat.ethz.ch/pipermail/r-announce/2017/000612.html`.
4. ERTL, Anton. *Threaded Code* [online] [visited on Apr. 30, 2017]. Available from: `https://www.complang.tuwien.ac.at/forth/threaded-code.html`.
5. FREE SOFTWARE FOUNDATION, INC. *Using the GNU Compiler Collection (GCC): An Inline Function is As Fast As a Macro* [online]. 2017 [visited on Apr. 30, 2017]. Available from: `https://gcc.gnu.org/onlinedocs/gcc/Inline.html`.
6. FREE SOFTWARE FOUNDATION, INC. *Using the GNU Compiler Collection (GCC): Labels as Values* [online]. 2017 [visited on Apr. 30, 2017]. Available from: `https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html`.
7. FREE SOFTWARE FOUNDATION, INC. *Using the GNU Compiler Collection (GCC): Specifying Registers for Local Variables* [online]. 2017 [visited on Apr. 30, 2017]. Available from: `https://gcc.gnu.org/onlinedocs/gcc/Local-Register-Variables.html`.
8. KALIBERA, Tomáš; MÁJ, Petr; MORANDAT, Floréal; VITEK, Jan. *A Fast Abstract Syntax Tree Interpreter for R* [online]. 2014 [visited on Apr. 30, 2017]. Available from: `http://janvitek.org/pubs/vee14.pdf`.

9. MORANDAT, Floréal; HILL, Brandon; OSVALD, Leo; VITEK, Jan. *Evaluating the Design of the R Language: Objects and Functions For Data Analysis* [online]. 2012 [visited on Apr. 30, 2017]. Available from: `http://r.cs.purdue.edu/pub/ecoop12.pdf`.

10. OSVALD, Leo. *r-shootout: R version of language shootout game* [online]. 2014 [visited on Apr. 30, 2017]. Available from: `http://r.cs.purdue.edu/hg/r-shootout`.

11. R CORE TEAM. *R Internals* [online]. 2017 [visited on Apr. 30, 2017]. Available from: `https://cran.r-project.org/doc/manuals/r-release/R-ints.html`.

12. R CORE TEAM. *R Language Definition* [online]. 2017 [visited on Apr. 30, 2017]. Available from: `https://cran.r-project.org/doc/manuals/r-release/R-lang.html`.

13. R CORE TEAM. *R Language Definition: Subset assignment* [online]. 2017 [visited on Apr. 30, 2017]. Available from: `https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Subset-assignment`.

14. THE R FOUNDATION. *What is R?* [online]. 2017 [visited on Apr. 30, 2017]. Available from: `https://www.r-project.org/about.html`.

15. TIERNEY, Luke. *A Byte Code Compiler for R* [online]. 2016 [visited on Apr. 30, 2017]. Available from: `http://homepage.divms.uiowa.edu/~luke/R/compiler/compiler.pdf`.

16. WICKHAM, Hadley. *Advanced R* [online]. 2014 [visited on Apr. 30, 2017]. Available from: `http://adv-r.had.co.nz/`.

# APPENDIX A

# Acronyms

| | |
|---|---|
| **API** | Application programming interface |
| **AST** | Abstract syntax tree |
| **BC** | Bytecode |
| **CLI** | Command line interface |
| **CRAN** | The Comprehensive R Archive Network |
| **GNU** | GNU's Not Unix! |
| **GCC** | GNU Compiler Collection |
| **IDE** | Integrated development environment |
| **JIT** | Just-in-time compilation |
| **PC** | Program counter |
| **REPL** | Read-eval-print loop |
| **VM** | Virtual machine |

# APPENDIX B

# Contents of the enclosed CD

- `thesis` ........ sources for the thesis in XeLaTeX
  - `benchmarks` ........ scripts for benchmarks and plots
  - `data` ........ measured data
  - `images` ........ plots used in the thesis
- `MT_Jecmen_Jan_2017.pdf` ........ PDF of the thesis
- `revisions.txt` ........ list of revisions that added new features

**Directory structure B.1:** Contents of the enclosed CD