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 3, 2017

Acknowledgements

[[acknowledgements]] Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

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 3, 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

[[abstract english]] This is the second paragraph. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Keywords **[[keywords en]]**

Abstrakt

[[abstract czech]] And after the second paragraph follows the third paragraph. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Klíčová slova [[keywords cz]]

Contents

Introduction **1**

1 About GNU R **3**

1.1 Language features of R 4
1.2 AST interpreter 11
1.3 BC interpreter and compiler 15
1.4 Why is R hard to optimize 19

2 About RIR **21**

2.1 Why is RIR slow 21

3 Improvements **23**

3.1 Instruction set extensions 23
3.2 Compiler modifications 23
3.3 Interpreter refactoring 23

4 Evaluation **25**

Conclusion **29**

Bibliography **31**

A Acronyms **33**

B Contents of the enclosed CD **35**

List of Figures

4.1 History of runnning times 26
4.2 Overview of the speedup vs. GNU R 27
4.3 History of average speedup vs. GNU R 28

List of Listings

1.1 Anonymous function 4
1.2 Arithmetic operators are function calls in R 5
1.3 Promise lazy evaluation 6
1.4 Superassignment 6
1.5 Coercion to the most flexible type 7
1.6 Recycling shorter vector 8
1.7 Object attributes 8
1.8 Argument handling 9
1.9 S3 object system 10
1.10 Basic subsetting / subassignment 11
1.11 Other subsetting operators 12
1.12 AST of a simple expression 14
1.13 Disassembling a BC object 16

3.1 Safe `break` 24
3.2 Context for `break` required 24
3.3 [[write listing caption]] microbenchmark (run with jit enable 2) . . 24

List of Tables

1.1 Some common types of internal R objects 13
1.2 Description of some GNU R bytecodes 17

Introduction

**[[write intro; cite:
https://www.tiobe.com/tiobe-index/
http://pypl.github.io/PYPL.html]]**

This thesis is structured in the following way:

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

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 done to RIR in this thesis.**[[?]]**

The chapter *Evaluation* discusses how the performance of RIR changed after the changes done in this thesis. It describes how the measurements were done and how the performance compares to GNU R.**[[?]]**

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

CHAPTER 1

About GNU R

GNU R[1] is a programming language used mainly for statistical computations. 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 recognised 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. **[[cite]]**

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 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[2] that provides, for example, syntax highlighting and quick access to documentation through a web-like browser. 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.

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

1.1 Language features of R

[[cite: http://r.cs.purdue.edu/pub/ecoop12.pdf]]

[[cite: http://adv-r.had.co.nz/]]

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 (more than one, in fact!) for object oriented programming, and is heavily influenced by functional programming languages, notably Scheme (a dialect of Lisp).

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 arithmetic operators being just syntactic sugar for function calls, as can be seen in listing 1.2[3]. 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.

```
> typeof(`+`)
[1] "builtin"
> `+`
function (e1, e2)  .Primitive("+")
> `+`(1, 2)
[1] 3
> 1 + 2
[1] 3
```

Listing 1.2: Arithmetic operators are function calls in R

All actual arguments to a function are lazy evaluated by default. When applying a closure, parameters are wrapped in promises. A promise is simply 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.

Delayed evaluation is demonstrated in listing 1.3, 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[4]. 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).

These features highlight the functional approach of R by minimizing side-effects. However, R supports assignment[5] which enables programmers to change a

[3]As a side note, backticks are used it 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.

[4]`cat` is short for *Concatenate and Print*

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

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

Listing 1.3: Promise lazy evaluation

function's local state by modifying its bindings and thus the imperative programming style. Also, the superassignment operator `` `<<-` `` makes it possible to change non-local bindings (as shown in listing 1.4) and thus brings the side-effects back into play.

```
> 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 1.4: Superassignment

The basic data type in R is an atomic vector. Vectors are collections of homogeneous values (i.e., 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 built from vectors and lists under the hood. Vectors can be created in R by calling the function `c` (the name stands for *combine*).

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 1.5, 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[6].

```
> 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 1.5: 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 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 1.6, where R even issues a warning about recycling.

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

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

```
> 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 1.6: Recycling shorter vector

extensions as they provide a way of encoding arbitrary additional metadata for objects. As an example, in listing 1.7 a vector of 4 elements is created, then changed into a 2 by 2 matrix[7] and finally changed back to vector (that also has its elements named).

```
> 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 1.7: Object attributes

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

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

or may not cause an error. Variable number of arguments is supported as well by using the ellipsis `` `...` ``. The ellipsis in an argument list matches any number of arguments, and later in the fucntion 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 1.8.

```
> 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 1.8: Argument handling

As was already mentioned, R has not one but three different object systems. The simplest is called *S3* and it uses a class[[verb]] 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, 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 1.9).

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

Listing 1.9: S3 object system

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 and their objects are modified in place (as opposed to the standard pass by value semantics).

An important and powerful feature of R is its subsetting and subassignment 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[8]. 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 1.10.

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 (i.e., the dollar version expects a name, and it need not be quoted). Also, the dollar operator does partial matching, similar to

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

```
> 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 1.10: Basic subsetting / subassignment

how function argument names are handled. These operators are demonstrated in listing 1.11.

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 (with some exceptions, such as assignment that sets its result to be invisible). Then the reader is again invoked and the process repeats.

```
> 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 1.11: Other subsetting operators

Every object in R is internally represented by a C structure called SEXPREC[9] (actually, R passes the objects around as pointers to this structure, which are called SEXP). This structure contains a header with metadata about the object, such as its type, reference counter or infromation 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.

[9]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.

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 the parsed expression. This tree is called the abstract syntax tree (AST) and its nodes are all SEXPs. An example AST is shown in the listing 1.12[10]. 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.

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.

[10] `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."

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

Listing 1.12: AST of a simple expression

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 perfomed 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).

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 funcions are the core language constructs, such as control flow (conditionals and loops). They take their arguments unevaluated in a list and evaluate them as needed while running. This is necessary for example for the if**[[verb]]** 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 integer 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 longjump 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 interpreter and compiler

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 **[[verb]]**compiler. 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 [2]. However, it was not used by default until version 3.4.0, released in late April 2017[11] [3].

[[cite compiler pdf]] The BC compiler itself is implemented in R, and walks the abstract syntax tree of an expression being compiled in a similar manner

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

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 instruction. 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 1.13. The object is printed as a 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 arguments to the instructions (the first element is an exception, as it encodes the version of the BC stored in the given object).

```
> 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 1.13: 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

Instruction	Description
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 infromation 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 funcion 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

[[cenv]] [[assumptions]] [[inlining during function call]]

1.4 Why is R hard to optimize

[[TODO]][1] [4] [5] **[[cite: https://cran.r-project.org/doc/manuals/R-lang.html section 6 and 2, hadley]]**

Firstly, R is not the fastest language out there. 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).

On top of that, R is a very dynamic language that gives a very high degree of freedom to a programmer. The design decisions behind R have an impact on performance. 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[[about ggplot2]]).

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

Also, R has its semantics defined by its one major implementation (although attempts have been made to formalize the language, e.g. [4]). This further complicates things as there is no formal description of the language.

CHAPTER 2

About RIR

[[Introduce RIR]] RIR[12] is an alternative compiler for the R language. 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.

[[history: research project, northeastern? first appearence?]]

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.

The architecture is very similar to GNU R. The compiler is

[[write about rir bytecode]]

[[how is rir bc different]]

[[optimizations, ai framework...]]

2.1 Why is RIR slow

[[TODO]]

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

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.

3.1 Instruction set extensions

GNU R bytecode compiler assumes certain invariants about the code when compiling, as was described in section 1.3.1. Of course, the instruction set of the default compiler reflects this. In fact, having specialized bytecode instructions for specific tasks is where the compiler gets most of its speedups. Specifically, not inlining the primitive R functions of type **[[verb]]**special causes the call mechanism to fall back to the same C routines that the AST interpreter uses, where the expression tree is examined and parts of it evaluated as needed.

3.2 Compiler modifications

3.3 Interpreter refactoring

As it turned out, a lot of speedup could be gained by changing the RIR bytecode interpreter.

[[to compiler, to ir, to interpreter, use code snippets, describe microbenchmarks, theory (threaded code...)]]

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

Listing 3.1: Safe `break`

```
function(n) {
    repeat {
        foo(if (n <= 0) break else 3)
        n <- n - 1
    }
}
```

Listing 3.2: Context for `break` required

[[everywhere: motivation - how it helped in microbenchmarks, then how in real]]

[[relational operators, fast paths for logical args, unary plus minus not, loop contexts, bc cleanup, colon, superassing, inlining of instructions in main loop, threaded code, inline stack funcs, loops refactor, disable guardfuns]]

```
f <- function() {
    i <- 10000000L
    while (i > 0) {
        i <- i - 1
    }
}
system.time(f())[[3]]  # jit everything
t <- c()
for (x in 1:15) t <- c(t, system.time(f())[[3]])
mean(t[5:15])  # only include measurments after warmup
```

Listing 3.3: **[[write listing caption]]** microbenchmark (run with jit enable 2)

CHAPTER 4

Evaluation

[[discussion of results, add figures]] [[discuss interesting point in measurements - naive nbody and threading etc.]]

[[plot times without warmup]]

[[plot how times change with successive invocations]]

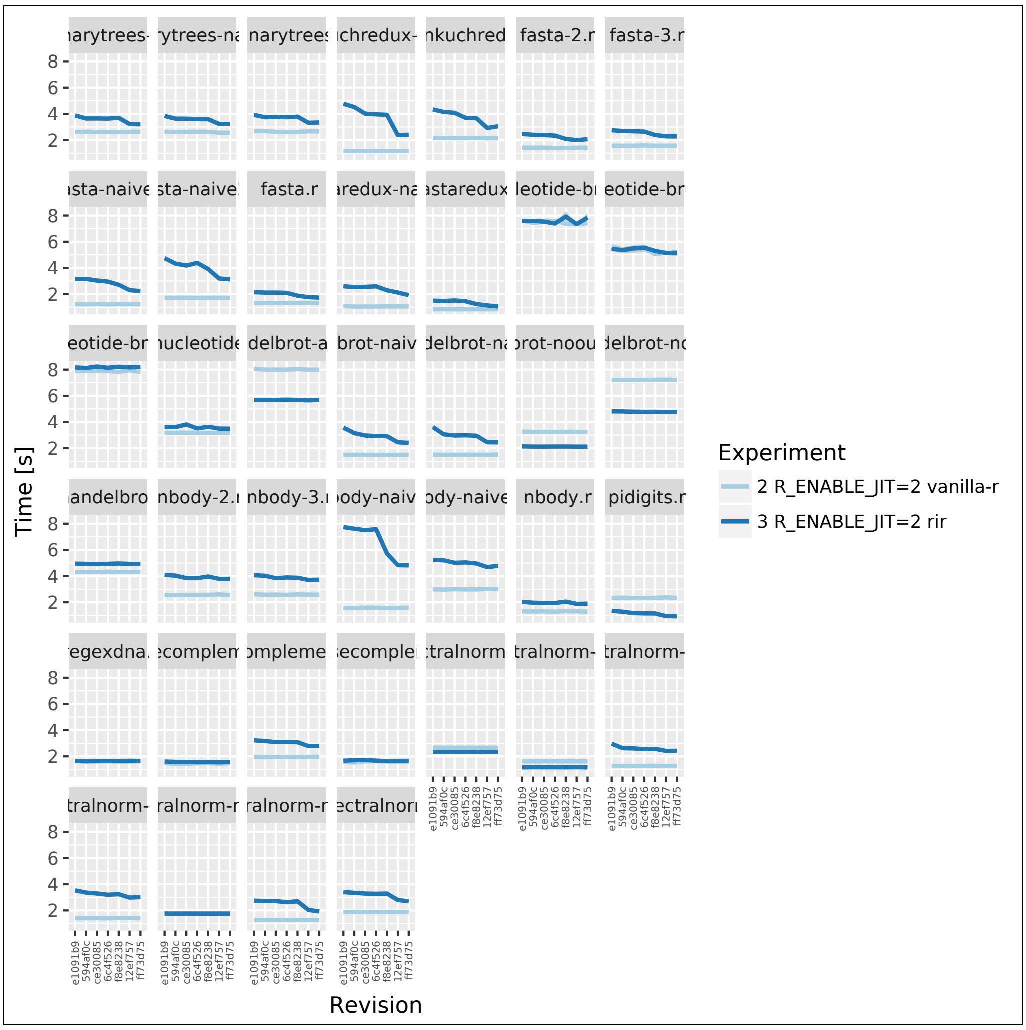

Figure 4.1: History of runnning times

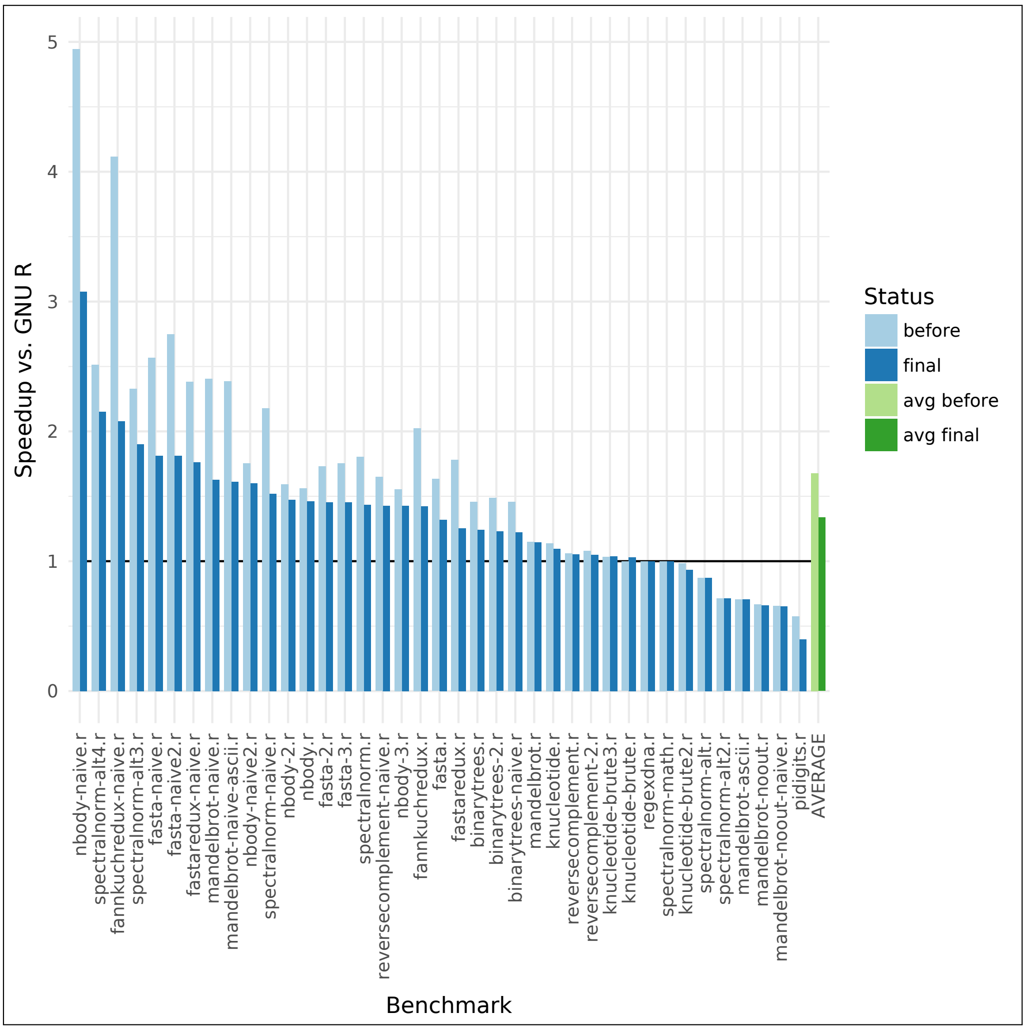

Figure 4.2: Overview of the speedup vs. GNU R

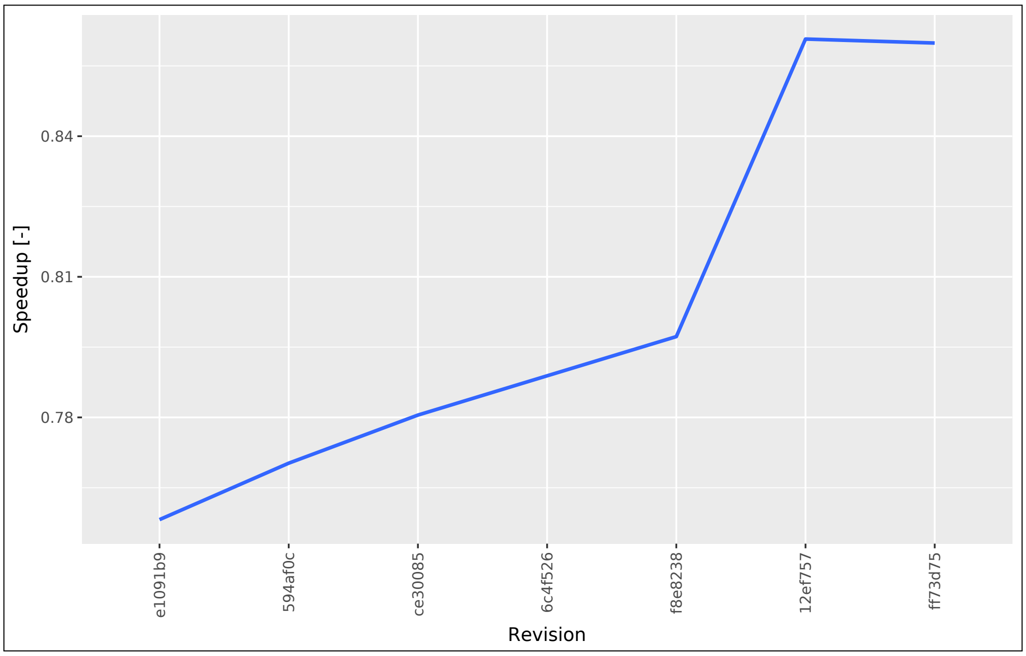

Figure 4.3: History of average speedup vs. GNU R

Conclusion

[[conclusion, future work, related work, fails - promises of consts, stoke etc.]]

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. 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`.
5. 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!
JIT	Just-in-time compilation
REPL	Read-eval-print loop
VM	Virtual machine

APPENDIX B

Contents of the enclosed CD

[[contents of cd]] After this fourth paragraph, we start a new paragraph sequence. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.