UNDERSTANDING FEEDBACK POLLUTION IN THE R PROGRAMMING LANGUAGE

Bc. Filip Říha

Master's thesis
Faculty of Information Technology
Czech Technical University in Prague
Department of Theoretical Computer Science
Study program: Informatics
Specialisation: System Programming
Supervisor: doc. Ing. Filip Křikava Ph.D.
May 9, 2025

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Understanding Feedback Pollution in the R Programming Language
Student:	Bc. Filip Říha
Supervisor:	doc. Ing. Filip Křikava, Ph.D.
Study program:	Informatics
Branch / specialization:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2025/2026

Instructions

Understanding Feedback Pollution in the R Programming Language

The heart of just-in-time compilation is the ability to specialize functions based on past behavior.
By recording information about types, callees, or control flow, a JIT compiler can generate efficient native code even for highly dynamic programming languages. However, the recorded feedback tends to become less precise over time, impacting the code quality and, in turn, performance. This thesis aims to study this phenomenon known as feedback pollution in the scope of the R programming language and explore ways to reduce it.

Tasks:
- Get familiar with Rsh, the JIT compiler developed at the PRL laboratory at FIT.
- Implement a tool for gathering data about feedback recorded by the VM.
- Explore how feedback behaves, how pollution occurs, and how to reduce it.
- Look how feedback is implemented in some other VM.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 22 January 2025 in Prague.

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Říha Filip. *Understanding Feedback Pollution in the R Programming Language*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2025.

I want to thank my supervisor Filip Křikava for guiding me through the creation of this thesis, Sebastián Krynski for being a great colleague and for his help with all of the many corner cases of R and Ř, and also Jan Vitek for his mentoring.
This work was funded by the Czech Science Foundation grant 23-07580X.

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. I further declare that I have created the thesis or part of it in the mode of employee work pursuant to Section 58 of the Copyright Act as an employee of the Czech Technical University in Prague. This fact does not affect the provisions of Section 47b of Act No. 111/1998 Coll., the Higher Education Act, as amended.

I declare that I have used AI tools during the preparation and writing of my thesis. I have verified the generated content. I confirm that I am aware that I am fully responsible for the content of the thesis.

In Prague on May 9, 2025

Abstract

Modern dynamic languages often leverage Just-in-Time (JIT) compilers to improve performance of frequently executed code. These compilers use the feedback collected at runtime to make speculations about future execution, making further optimization possible. However, over time, the accuracy of this feedback can degrade, leading to less precise speculations and performance degradation. This phenomenon is known as feedback pollution.

The R programming language is highly specialized for statistical computing and data visualization, and thanks to its features such as reflection and lazy evaluation, it can be difficult to optimize. The Ř JIT compiler tries to improve the performance of R by using speculations on the observed feedback in order to increase the performance of the compiled code.

The primary objective of this thesis is to develop a tool that allows for detailed observation and analysis of the behavior of the Ř JIT compiler, as otherwise the inner workings of the compiler are a black box. By having the facilities to examine the JIT internals, we open up possibilities for further research and analysis, including the understanding of feedback pollution and feedback usage explored in this thesis.

Keywords R, GNU R, JIT, compilation, feedback vector, type feedback

Abstrakt

Moderní dynamické jazyky často využívají JIT (Just-in-Time) překladače ke zrychlení opakovaně spouštěného kódu. Tyto překladače sbírají za běhu informace o chování programu a na jejich základě spekulují o budoucích bězích, což umožňuje lepší optimalizace. Postupem času však může přesnost těchto informací klesat, což vede ke zhoršení výkonu – tomuto jevu se říká znečištění zpětné vazby.

Programovací jazyk R je úzce zaměřený na statistické výpočty a vizualizaci dat. Kvůli vlastnostem jako reflexe a odložené vyhodnocování je však jeho optimalizace náročná. Ř je JIT kompilátor navržený pro R, který se snaží zvýšit výkon pomocí spekulací založených na pozorované zpětné vazbě.

Hlavním cílem této práce je vytvořit nástroj pro detailní pozorování a analýzu chování kompilátoru Ř, jehož vnitřní fungování je jinak černou skříňkou. Tím, že máme k dispozici prostředky pro zkoumání vnitřních částí kompilátoru, otevíráme možnosti pro další výzkum a analýzu, včetně pochopení znečištění a využití zpětné vazby, které jsou zkoumány v této práci.

Klíčová slova R, GNU R, JIT, kompilace, vektor zpětné vazby, typová zpětná vazba

Contents

Introduction 1

1 Background 3
- 1.1 The R Language . . . 3
- 1.2 GNU-R . . . 8
- 1.3 The Ř Compiler . . . 12
- 1.4 Related Work . . . 17
- 1.5 Corpus . . . 18

2 Recording Tool 19
- 2.1 Motivation . . . 19
- 2.2 Design . . . 20
- 2.3 Implementation . . . 22
  - 2.3.1 Hooks . . . 22
  - 2.3.2 Recorder . . . 22
  - 2.3.3 Events . . . 24
  - 2.3.4 Serialization . . . 24
  - 2.3.5 Interface . . . 27
- 2.4 Assessment . . . 28

3 Feedback Pollution 29
- 3.1 Motivation . . . 29
- 3.2 Methodology . . . 31
- 3.3 Analysis . . . 32
- 3.4 Pollution Prevention . . . 34

4 Feedback Usage 36
- 4.1 Definitions . . . 36
- 4.2 Methodology . . . 39
- 4.3 Observations . . . 40
- 4.4 Limitations . . . 43

5 Conclusion 45
- 5.1 Future Work . . . 46

A Bytecode Examples 48

Bibliography 51

Contents of the Attachment 54

List of Figures

1.1 Plot of listing 1.1 . . . 3
1.2 Structure of the GNU-R SEXP . . . 8
1.3 Overview of Ř architecture[10] . . . 12
1.4 Structure of the Ř runtime objects . . . 13
1.5 Composition of Ř runtime objects . . . 15

2.1 Idealized event log, each point represents a function invocation 19

3.1 Event log of listing 3.1 without contextual dispatch, each point represents a function invocation . . . 30
3.2 Event log of the listing 3.1 with contextual dispatch, each point represents a function invocation . . . 31
3.3 Function pollution in Kaggle script, each point represents a compilation[21] . . . 32
3.4 Pollution of functions in benchmarks[21] . . . 33

4.1 Illustration of the relationship between static, feedback, expected and assumed types . . . 37
4.2 Usage of slots across closure compilations . . . 40
4.3 Categorization of unused slots across closure compilations . . . 41
4.4 Categorization of used slots . . . 42

List of Tables

1.1 The different SEXP types[5, 1.1.1 SEXPTYPEs] . . . 9

3.1 Summary of the feedback pollution in the corpus[21] . . . 34

4.1 Overview of analyzed programs . . . 39

List of Code listings

1.1 R example[2] . . . 3
1.2 Demonstration of R special calls . . . 5
1.3 Example of R laziness . . . 6
1.4 Immutability example . . . 7
1.5 Example of environment mutability . . . 7
1.6 Example of malicious reflection[4] . . . 7
1.7 A truncated example of generated GNU-R and RIR bytecodes, full code in appendix A . . . 11
1.8 Example of a PIR instruction . . . 16

2.1 Simplified code of compilation logic in interpreter/interp.cpp . 21
2.2 Example of recording state management in a hook caller in the class `ObservedTest` in file runtime/TypeFeedback.h . . . 22
2.3 Simplified definition of `Record` and `FunRecording` classes . . . 23
2.4 Definition of the `Serializer` struct and its usage for type `bool` 25
2.5 Function definition of field serialization functions . . . 26
2.6 Example of using the fields serialization functions defined in 2.5 26

3.1 Motivating example for feedback pollution . . . 30

4.1 PIR code structure of type assumption . . . 37

A.1 GNU-R bytecode for listing 1.7.1, generated using A.3 . . . 48
A.2 RIR bytecode for listing 1.7.1 . . . 49
A.3 Code used for formating disassembled GNU-R bytecode . . . 50

List of abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
CRAN	The Comprehensive R Archive Network
GC	Garbage Collector
IR	Intermediate Representation
JIT	Just-in-time
NAN	Not a Number
OSR	On-Stack Replacement
SSA	Static Single-Assignment
SEXP	Symbolic Expression
VM	Virtual Machine

Introduction

Many modern dynamic languages are executed on a virtual machine (VM). In order to speed up the execution of programs, VMs traditionally include Just-in-Time (JIT) compilers, allowing them to improve the performance of frequently executed pieces of programs by compiling them to native code. To further enhance performance, most modern VMs record information about the runtime (called feedback), allowing them to predict future behavior. If the feedback information is useful, the compiler can speculate on it, leading to a more optimized code if the assumption holds. This is based on the thesis that what is past is prologue. However, over time, the recorded information tends to become less precise, resulting in more general speculations and slower code down the line. This trend is known as feedback pollution.

R is a high-level programming language specialized for statistical computing and data visualization. It is dynamically typed with function and object-oriented patterns, and features reflection and lazy evaluation. GNU-R, the reference implementation of R, contains both an abstract syntax tree (AST) interpreter and a bytecode JIT compiler and interpreter. Ř is an alternative JIT compiler for GNU-R. It uses feedback information from an interpreter to speculatively optimize the native compilation. However, the inner workings of Ř are sort of a black box. There is no simple way to track why and how are functions invoked and compiled, or why and where are deoptimizations triggered.

The main goal of this thesis is to implement a tool that would allow us to observe and analyze the behavior of the Ř compiler, allowing us to better understand the Ř internals and opening up possibilities for further research and analysis. Concretely, in this thesis, we are interested in understanding the feedback pollution as it happens in R.

Chapter 1 introduces the necessary background information, including the R language, the GNU-R virtual machine implementation, and the Ř compiler structure. Chapter 2 delves into the design and implementation of the recording tool, as well as its impact on the Ř codebase. Chapter 3 outlines the research done on feedback pollution, which was possible thanks to the record-

ing tool. Chapter 4 then further analyzes how the feedback information is used (or not used) during compilation, observing feedback patterns in the compiler, and connecting them to the feedback pollution.

Chapter 1

Background

In this chapter, we introduce the R language and its implementation. We also outline the implementation of Ř, an extension to R that is composed of a bytecode interpreter and a speculative JIT compiler. We compare how Ř collects and uses feedback information with the state of the art JIT compilers, namely the V8 for JavaScript and HotSpot JVM for Java. Finally, we describe the R programs used for experiments and analysis throughout the thesis.

1.1 The R Language

The *R*[1] programming language has been primarily designed for statistical computation, data analysis, and graphical visualization. It was developed by Ross Ihaka and Robert Gentleman at the University of Auckland as an open source alternative to the S language. It is part of the GNU Project, licensed as a free software under GNU GPL.

```
ggplot(
  data = gapminder,
  aes(
    x = gdpPercap, y = lifeExp,
    color = continent
  )
) +
  geom_point() +
  scale_x_log10()
```

Code listing 1.1 R example[2]

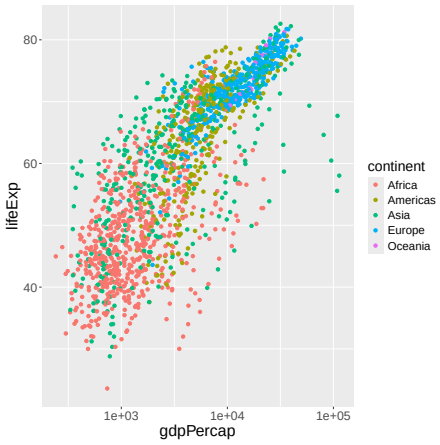

Figure 1.1 Plot of listing 1.1

The popularity of R stems from its domain-specific focus. Unlike general-purpose languages, R offers high-level abstractions for statistical operations, making complex analyses more accessible. As a consequence, many R users are statisticians rather than traditional programmers. An example of the R expressivity is in the listing 1.1, where with just a few lines of code, we can plot the relationship between GDP per capita and life expectancy from the `gapminder` data set[3].

The language is high-level, dynamic, object-oriented, functional, interpreted, and lazy, with automatic memory management.

The R syntax is very C-like, with `if` statements, `for` and `while` loops, array indexing, and function calls. For assignment, the `<-` operator is used. Statements are separated by a new line, and optionally by a semicolon if they are on the same line.

Types

The basic types of R are the *numeric types* (*integer*, *double* or also *real* and *complex*), *character type* (strings) and *logical type* (the boolean values `TRUE` and `FALSE`). R also has `NULL`, and a constant `NA` (*not available*) representing a missing value related to missing statistical observation.

R has no concept of scalars. Instead, all basic types are represented as *vectors*. To create a vector with multiple elements, the function `c` (standing for combine) can be used. All traditional mathematical operators are also vectorized.

For storing elements of different types, a *list* is used. Other commonly used types built on top of vectors and lists are *matrix* (two-dimensional vector), *array* (multidimensional vector), and *data frame* (matrix-like structure whose columns can have different types).

Every value can have associated *attributes*, a collection of name-value pairs. These are accessed through the functions `attr` or `attributes`.

R also has multiple *object models*. The most common ones are *S3*, which is controlled by setting a `class` attribute on any type, and *S4*, which defines more formal classes that must be created and instantiated and are internally represented by a distinct type.

Environments

Different scopes in R are separated using objects named *environments*. An environment consists of a *frame*, which is the collection of variables, and a pointer to an *enclosing environment* (also called a *parent*). The topmost environment has a pointer to a special *empty environment*, which has no other parent.

When a variable or a function is accessed, it is first looked for in the current environment. If not found, it is searched for recursively in each enclosing environment. Only then if it is not found, it is an error.

```
a <- if (TRUE) {
  1
} else
  2 * (3 + 4)
```

```
# This is equivalent to
# the previous statement
`<-`(a, `if`(
  TRUE,
  `{`(1),
  `*`(
    2,
    `(`(`+`(3, 4))
  )
))
```

Code listing 1.2 Demonstration of R special calls

The lookup for functions is separate from non-function lookup. An assignment of a variable in some environment does not shadow a function in its enclosing environment. So the code `c <- 42; c(c, c)` results in one vector with two numbers 42, because the definition of the variable `c` does not shadow the function of the same name from the base environment.

These environments are also first-class values. We can create new environments, access the value of the current enclosing environment, or even access and modify environments on the call stack.

Functions

A *function*, or also a *closure*, is composed of three parts—*formals*, which is the list of formal arguments and their optional default values, *body*, which is the code of the function, and *environment* defining the lexical scope of the function body.

New functions can be created with the `function` keyword, which is actually compiled to a function call. Most operations that happen in R are, in fact, function calls. This also includes control flow statements (like *if* or *while*), binary operations, assignment, and even surrounding an expression in parentheses or multiple expression with braces. This is demonstrated in listing 1.2. Note that builtin functions have to be surrounded with backticks in order to refer to them as identifiers.

Laziness

R uses a variation of call-by-need semantics, delaying the evaluation of function arguments until a value is needed. This means that when a function is called, the passed arguments are not instantly evaluated, but instead wrapped in a *promise*, a tuple of *expression*, *value*, and *environment*. The value is first set to the unbound value. When the promise is first accessed, the expression of the promise is evaluated within its environment (also called a *force*). The result

is cached in the value field, and every other access to the argument results in the cached value.

This can be seen in example 1.3. The call on line 12 first prints "Hello from g", then the parameter `x` is accessed, the promise is forced, and the text "Hello from f" is printed. The second call to the function `h` demonstrates a second behavior—when a parameter is never accessed, the promise is not evaluated, and nothing is printed.

```
1  f <- function(x) {
2      print("Hello from f")
3      x
4  }
5
6  g <- function(x) {
7      print("Hello from g")
8      x
9  }
10
11 # Prints "Hello from g", then "Hello from f"
12 g(f(42))
13
14 h <- function(x) 0
15
16 # Does not print anything
17 h(f(42))
```

Code listing 1.3 Example of R laziness

Being lazy in the arguments allows R to have very expressive domain-specific languages for ease of manipulating the data, as well as reducing memory overhead of the programs, as seen in the motivation example in the listing 1.1.

Promises can also be created manually by calling the `delayedAssign` function or by using the C API.

Immutability

All R values are *semantically immutable*, with the exception of environments. This means that when a variable holds a value, this value never changes unless the variable is reassigned. We say that non-environment values have a *copy-on-write behavior*—when a value is modified, a new copy is created with the modified parts.

Thanks to the syntax of R, it is possible to write code that looks like it is using mutability but keeps the copy-on-write behavior. In example 1.4, the statement on line 3 looks like it is mutating the vector in place, but instead

```r
x <- c(1, 2, 3)
y <- x
y[1] <- 42
# x is still vector (1, 2, 3)
```

Code listing 1.4 Immutability example

```r
e1 <- new.env()
e2 <- e1
e2$x <- 42
e1$x == 42
# TRUE
```

Code listing 1.5 Example of environment mutability

```r
bad <- function() {
  rm("x", envir=sys.frame(-1)) # Remove the variable x from the caller
  2
}

good <- function(y) {
  x <- 1
  z <- y # Here the promise is evalated
  x + z
}

good(bad())
# Error in good(bad()) : object 'x' not found
```

Code listing 1.6 Example of malicious reflection[4]

the function `[<-` is called, which creates a copy of the vector, modifies it and reassigns the variable x.

The only truly mutable values are environments, and they conform to a *reference behavior*. Listing 1.5 shows this—the variable `x` is added to both `e1` and `e1`, even though we only defined it on `e2`.

Reflection

As previously mentioned, environments are a first-class citizen. This allows any function to access or even modify any environment where it was created or called. Moreover, when combined with lazy arguments, a function can reflect on the environment in which it is being forced and modify it, sometimes in unpredictable ways.

This is demonstrated in listing 1.6, taken from [4]. We can see that evaluating the promise (the call to `bad`) removes a variable binding, resulting in an error.

R also has primitives for creating, manipulating, and evaluating expressions from the language itself. This allows for, among other things, evaluating a promise in a different environment, programmatically creating arbitrary pieces

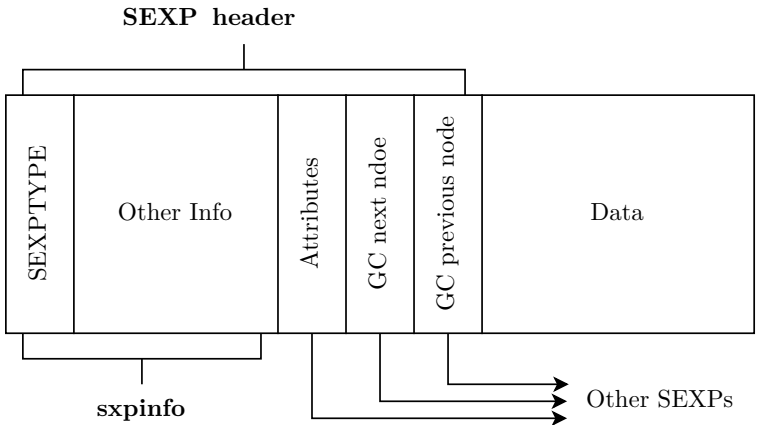

Figure 1.2 Structure of the GNU-R SEXP

of code, or instrumenting functions with additional calls.

Combining reflection, laziness, and side effects makes R a very complex language to optimize.

1.2 GNU-R

The R language is not formally specified, it only has a reference implementation that we will refer to as *GNU-R*. It is written in the C programming language, spans more than 250,000 lines of code and is currently maintained by the R Core Team.

Representation

The GNU-R represents all code and values as *symbolic expressions* (also *S-expressions* or *SEXPs*), a format for nested lists popularized by the Lisp languages. The type `SEXP` is a pointer to a `SEXPREC` or `VECTOR_SEXPREC` structure. Both of these structures contain a header of type `sxpinfo_struct`, a pointer to the attributes list, and previous and next nodes in the garbage collector. The `sxpinfo_struct` then contains the type of SEXP (`SEXPTYPE`, values are defined as in 1.1), as well as additional information about the type, garbage collection and debugging. The rest of the structure contains the actual data. The structure of a `SEXP` value is represented in figure 1.2.

As a note, through the thesis, when referring to a SEXP value, we mean the value of `SEXPREC` or `VECTOR_SEXPREC` and not the pointer to it.

no	SEXPTYPE	Description
0	NILSXP	NULL
1	SYMSXP	symbols
2	LISTSXP	pairlists
3	CLOSXP	closures
4	ENVSXP	environments
5	PROMSXP	promises
6	LANGSXP	language objects
7	SPECIALSXP	special functions
8	BUILTINSXP	builtin functions
9	CHARSXP	internal character strings
10	LGLSXP	logical vectors
13	INTSXP	integer vectors
14	REALSXP	numeric vectors
15	CPLXSXP	complex vectors
16	STRSXP	character vectors
17	DOTSXP	dot-dot-dot object
18	ANYSXP	make “any” args work
19	VECSXP	list (generic vector)
20	EXPRSXP	expression vector
21	BCODESXP	byte code
22	EXTPTRSXP	external pointer
23	WEAKREFSXP	weak reference
24	RAWSXP	raw vector
25	OBJSXP	objects not of simple type

Table 1.1 The different SEXP types[5, 1.1.1 SEXPTYPEs]

Interpreter

The GNU-R parses the code into an *abstract syntax tree* (*AST*), represented by SEXPs of the `LANGSXP` type, which is then interpreted.

Included with the GNU-R distribution is a *bytecode compiler*, which can be invoked either explicitly, when a package is installed, or as a Just-in-Time (JIT) compiler. The compiler is written mostly in R with few supporting functions written in C, while the bytecode interpreter is written in C. The bytecode is stack-based with fat instructions, such as specialized instructions for type checking, specialized loads for common constants like `NULL`, `TRUE` or `FALSE`, instructions for executing `for` loops or various subsetting operators. Branching is done via jumps to arbitrary code index. An example of the bytecode can be seen in listing 1.7.2.

The bytecode employs a series of optimizations, most notably constant folding and inlining of base and builtin functions. Inlining can be set to various levels, with higher levels being more optimized while assuming that the

functions in the base package and core language functions (like `if` or `` `{` ``) are not shadowed. This allows the compiler to translate control flow from function calls to bytecode jumps, as well as to use bytecode instructions to perform basic arithmetic and logical operations.

When a function is compiled, its `SEXP` is modified in-place, replacing the AST body with the bytecode. Every other call to this function is then interpreted by the bytecode interpreter.

Garbage Collector

R does not have primitives for managing memory, instead an automatic memory management provided by the runtime is expected. GNU-R uses a generational non-moving stop-the-world garbage collector with three generations[6].

Next to the GC, GNU-R also has a reference counter for each object, included in the object header. This is used for optimistic mutations—when an object is to be copied and mutated, but there is only one reference to it, it is instead mutated in place, avoiding unnecessary copies. This correctly preserves the copy-on-write behavior while improving performance.

Packages

A big advantage of using R is the vast library of packages, libraries, and data sets. These are hosted at *The Comprehensive R Archive Network* (*CRAN*)[7] package repository, which is curated and tested by the R Core Team. They can be installed very simply by calling `install.packages` and loaded with the `library` function.

The GNU-R base installation is also distributed with several packages, including, but not limited to, `base` containing the basic functions for using R, `stats` implementing statistical functions, `graphics` with base functions for manipulating graphical output, `compiler` implementing the mentioned bytecode compiler, or `Matrix` with definitions of dense and sparse matrix classes.

C Interface

In order to speed up certain packages, GNU-R allows parts of the code to be written in more low-level languages via a C interface. This includes the definitions for the SEXP types, a big set of functions and macros to create and manipulate values, and several evaluation functions. The SEXP structure is only exported as an opaque pointer, with the intention that individual fields are to be accessed and manipulated via the exported functions.

The interface is more of an afterthought rather than a deliberate choice, as indicated, for example, by the the main file with type definitions being called `Rinternals`. The functions expose a large portion of the internal structure of the interpreter, and this is subsequently used by package authors. Even parts of the code that are not directly exposed are commonly accessed by packages.

```
f <- function(x) {
  for (i in 1:10) {
    if (i + 2 > 1) {
      g(x)
    }
  }
  g(x)
}
```

(1.7.1) R code

```
Code:
  1 LDCONST 1
  3 STARTFOR 4 3 30
  7 GETVAR 3
  9 LDCONST 5
 11 ADD 6
 13 LDCONST 7
 15 GT 8
 17 BRIFNOT 9 28
 20 GETFUN 10
 22 MAKEPROM 12
 24 CALL 11
 26 GOTO 29
 28 LDNULL
 29 POP
 30 STEPFOR 7
 32 ENDFOR
 33 POP
 34 GETFUN 10
 36 MAKEPROM 12
 38 CALL 11
 40 RETURN

Constant pool:
...
12:
 Promise 0:
  Code:
    1 GETVAR 0
    3 RETURN
  Constant pool:
  0:
   symbol x
  1:
   language g(x)
  2:
   'expressionsIndex' ...
...
```

(1.7.2) GNU-R code

```
0:
      0   push_  1
      5   visible_
      6   force_
      7   push_  10
     12   visible_
     13   force_
     14   ; :(1, 10)
          colon_input_effects_
     15   pop_
     16   swap_
     17   colon_cast_lhs_
     18   [ <?> ] Type#0
     23   ensure_named_
     24   swap_
     25   colon_cast_rhs_
     26   ensure_named_
     27   [ <?> ] Type#1
     32   dup2_
     33   ; NULL
          le_
     34   [ _ ] Test#0
     39   brfalse_  1
     44   push_  1L
     49   br_  2

1:
     54   push_  -1L
          ...
7:
    287   popn_  3
    292   ldfun_  g
    297   [ 0, <0>, valid  ] Call#2
    302   mk_promise_  2
    307   ; g(x)
          call_  1
    324   [ <?> ] Type#11
    329   ret_

[Prom (index 0)]
0:
      0   ldvar_  x
      5   [ <?> ] Type#5
     10   ret_

[Prom (index 1)]
0:
      0   ldvar_  x
      5   [ <?> ] Type#9
     10   ret_
          ...
```

(1.7.3) RIR code

Code listing 1.7 A truncated example of generated GNU-R and RIR bytecodes, full code in appendix A

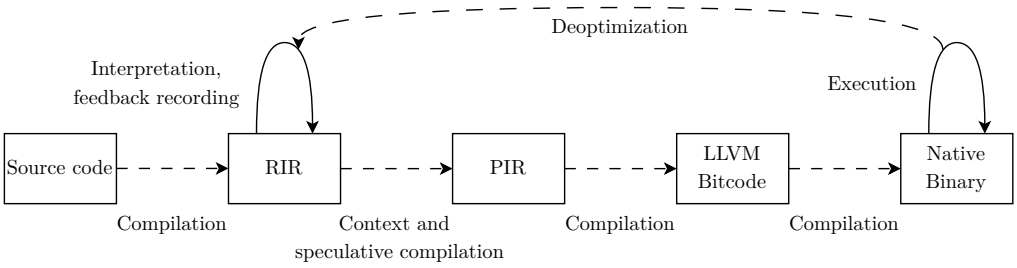

Figure 1.3 Overview of Ř architecture[10]

As an example, some packages rely on the hidden reference counter in order to optimize updates by modifying structures in-place.

This makes evolution of R very hard without breaking packages. It also complicates alternative implementations of R as they need to explicitly export the same functions as GNU-R if they want to support all available packages.

1.3 The Ř Compiler

Ř (also stylized as *Rsh*) is a JIT compiler for the R language, developed at Programming Languages Laboratory at Czech Technical University in Prague[1] and Programming Research Laboratory at Notheastern University in Boston[2]. The project is freely available and hosted on GitHub[8].

It is built as an extension to GNU-R, although it uses a slightly modified version of the codebase. It bypasses the GNU-R bytecode compiler and interpreter, instead using a custom one, while reusing the `SEXP` representation, AST interpreter, and garbage collector. For compilation into native code, the *LLVM Project*[9] is used. The compilation pipeline is outlined in figure 1.3.

Runtime Objects

Ř reuses the SEXP representation and memory management from GNU-R. All runtime objects are embedded into SEXP objects. This is one of the modifications that need to be applied to GNU-R, a new `SEXPTYPE` is added (`EXTERNALSXP`), along with the necessary changes to how it is garbage collected and how it references other objects.

The structure of Ř runtime object can be seen in figure 1.4. It contains the SEXP header and then the embedded Ř object, which always inherits from a `RirRuntimeObject` class. The Ř object starts with two `uint32_t` numbers, the first dictates where is the begining of area referencing other SEXPs, and

[1] `https://prl-prg.github.io/`
[2] `https://prl.khoury.northeastern.edu/`

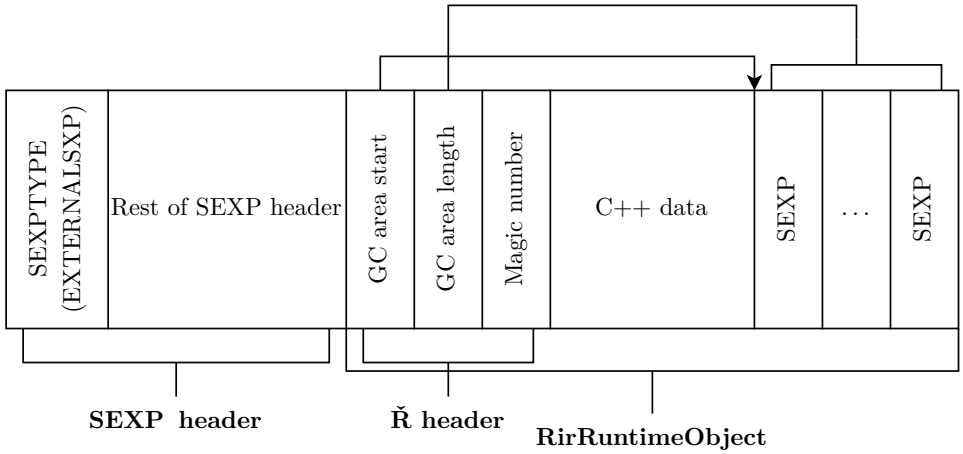

Figure 1.4 Structure of the Ř runtime objects

the second indicates how many pointers are there. The magic number dictates which Ř object it is.

There are helper macros to access the headers, as well as the pointers to other SEXPs. On the Ř side, there are functions to convert between a C++ pointer and a SEXP.

Similarly to the GNU-R compiler, when an R function is to be interpreted, it is compiled to a bytecode, and the body field of the closure is replaced by a Ř structure. The composition of Ř objects can be seen in figure 1.5.

The body of Ř compiled function contains a *dispatch table*. A dispatch table contains multiple entries of the compiled code, where the first version (also called a *baseline*) is always the bytecode representation, whereas the other versions are compiled into native code. The multiple compiled entries are used by the contextual dispatch, as explained in chapter 1.3.

Every dispatch table entry is represented by a `Function` structure. This holds the signature of the function, the call context, statistic about the execution like how many times it has been invoked or deopted, the body of the function and the feedback vector.

The body of a function is stored in a structure called `Code`. This is either the bytecode body or a pointer to the native function, and the constants pool.

The *feedback vector* collected from a bytecode interpretation is represented by the `TypeFeedback` structure[3]. It consists of *slots*, where one slot corresponds to information about one bytecode instruction. There are three different types of slots:

- *observed calls* records the destination of calling a function,
- *observed types* record the types of values that are loaded from environment,

[3] `TypeFeedback` is a missleading name as the interpreter also collect non-type information

forced, or are results of a function call,

- and *observed tests* has one of four values recording how a branch was taken (`None`, `OnlyFalse`, `OnlyTrue` and `Both`).

Since every function has different number of slots, the `TypeFeedback` object has a different size for each function and uses a *flexible array member*[11] to store observations.

RIR Bytecode Interpreter

The bytecode used by Ř is called *RIR*. It is a stack-based bytecode, interpreted by a Ř interpreter. Similarly to GNU-R, RIR assume the base functions are not shadowed, allowing it to have instructions for arithmetic operations and control flow instead of resolving them as function calls. Unlike GNU-R, the RIR instructions are much more granular, there are fewer of them, they are not as specialized, and a single instruction represents a much smaller piece of C code. An example of RIR compiled code can be seen in listing 1.7.3. Note that the example code is not complete and is only used to give impression between the size differences of the bytecodes and the full code listing is in appendix A.

For this thesis, the important bytecode instructions are `record_call_`, `record_type_`, and `record_test_`. These do the recording of feedback information about calls, types, and tests, respectively, by observing the top value on the execution stack and recording the information to the `TypeFeedback` structure. The instructions are printed in the listings as the current feedback slot value in square brackets followed by the label `Call#N`, `Type#N` or `Test#N`, where `N` is the slot index.

Apart from the feedback information, the RIR interpreter also records other information about the running program, most notably the number of times a function has been invoked, and the number of times a loop has been executed. These are used to determine which parts of the program are executed frequently, and thus are good candidates for compiling to native code.

PIR Compiler

When a function or a loop meets the compilation heuristics, it is compiled from RIR to *PIR*, an intermediate representation used for optimizations. PIR is composed of instructions in a *static single-assignment form* (SSA), organized in basic blocks, with each block being terminated with a (conditional) jump to another basic block or a speculation checkpoint.

Every PIR instruction has a *type* (also called a *PIR type*). These consist of one or several *R types* (like integer or logical) and type flags. For this thesis, the important type flags are

- *is scalar*, meaning the value is always a vector of size 1,

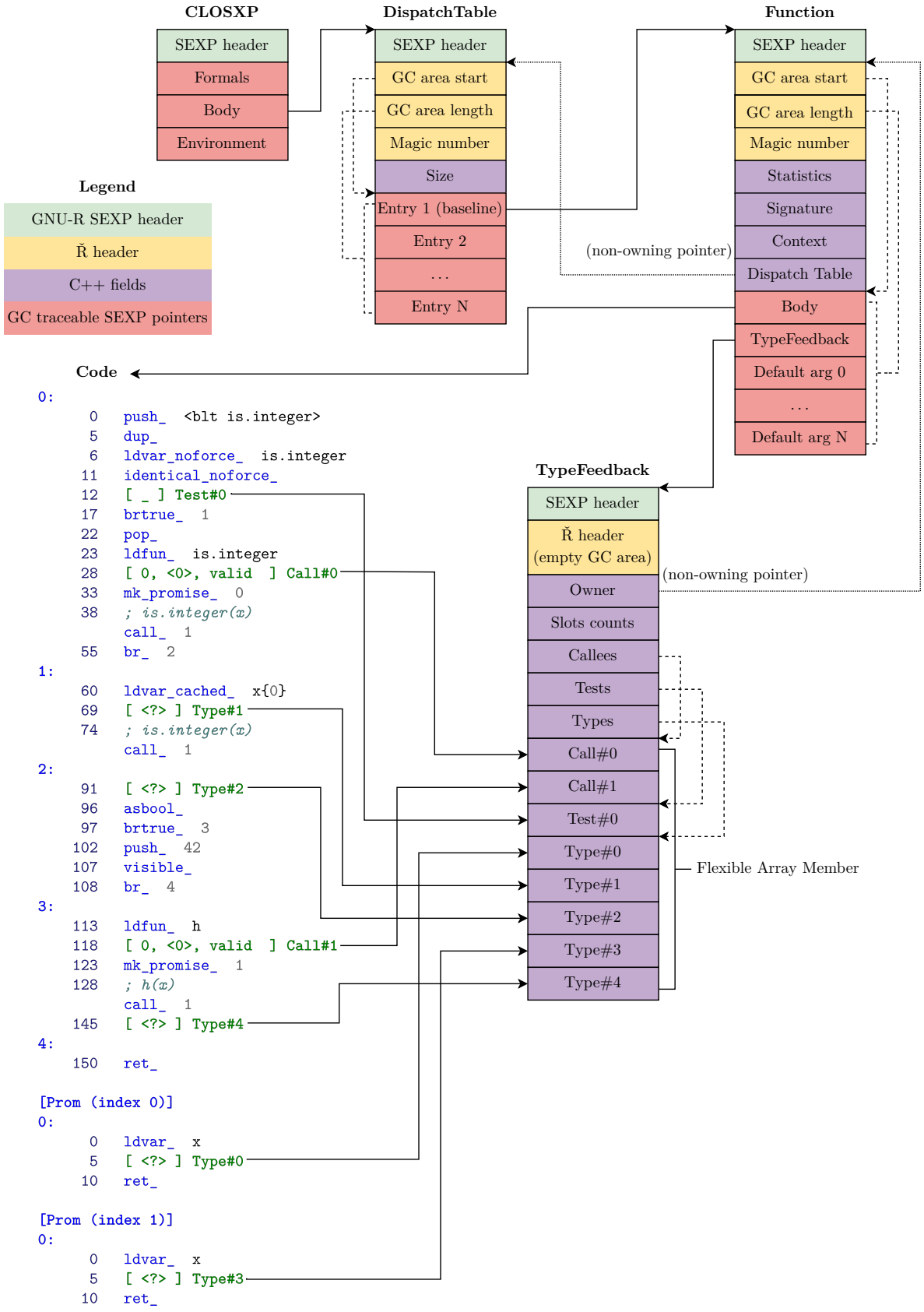

Figure 1.5 Composition of Ř runtime objects

- *is not object*, which means that the value is neither of the R object models,
- *has no attributes*, where the object does not have the associated R attributes (this implies it is not an object),
- *can be missing*, where the value can be an *unbound value*, raising an error if evaluated,
- *can be NA or NaN*, meaning the value can be `NA` or a not-a-number (NaN),
- and *maybe lazy*, where the value needs to be first evaluated before used.

All of these types form a complete lattice. There are also types that model the types internal to only the PIR code, used for the speculative optimizations.

All instructions also have *effect flags*, denoting what kind of observable side effects an instruction can emit. These, for example, include *deopt* (may trigger deoptimization), *warn* (might print a warning message), *error* (may produce an error), or *reflection* (might invoke reflection). By explicitly tracking these effects, the compiler is able to perform more optimizations while still preserving the correct R semantics.

In the listing 1.8, we can see how the PIR instructions are represented in text. It starts with the type (`real$-`), continues with the register by which the instruction is referred to (`%4.2`), then the name of the instruction (`Add`), followed by the effect flags (`d`), arguments (`%2.0, 2, elided`) and finally the feedback slot connected with this instruction (`<val?_>`). When the type of instruction is void, the register is omitted. Similarly, when the instruction does not have a feedback associated with it, it is not shown.

```
real$-    %4.2 = Add     d     2.0, 2, elided     <val?_>
```

Code listing 1.8 Example of a PIR instruction

Based on the observed feedback, the compiler performs *speculative optimizations*. When the feedback slot on a PIR instruction hold interesting information (e.g. the callee is only one target, branch was always taken or the observed type was a only integer), Ř *speculates* on this observation. It emits a *guard* which will assert at runtime that the speculation still holds. After compilation, if the guard fails, a *deoptimization* is triggered. This updates the feedback slot with the newly observed fact, marks the native code version with a flag, and continues execution in the bytecode interpreted version. A *frame state* is used to reconstruct the environment and execution stack to a corresponding state for the interpreter to correctly continue execution.

After all optimizations are completed, the PIR code is transformed into *LLVM bitcode*, the intermediate representation of LLVM. This is then passed to the *ORC JIT* compiler, which is part of the LLVM project, compiling the bitcode to the native code.

Contextual Dispatch

In addition to speculative optimizations, Ř employs another technique to optimize on dynamic types called *contextual dispatch*[12]. The idea is based on observing arguments on runtime and based on their types, creating a disjunct native version with the ability to specialize uniquely to the type.

For every function call, we create a *context*. It captures information about the first six arguments (if they are eager, nonreflective, not an object, or a simple scalar integer or double), the number of missing arguments, as well as if the arguments are correctly ordered, or if there are not too many arguments passed. These contexts form a partial ordering.

When a function compilation to native is triggered, it is compiled for a specific context, and the compiler is able to optimize on the information in the context. The resulting code with context (together called a *version*) is installed in the dispatch table.

When a function is invoked, its arguments are observed, and a call context is created. Based on this, the function is either dispatched to the version connected with the same context as the call context, or a less precise context which is ordered lower than the call context. If no such version is found, the call is dispatched to the baseline interpreted version.

1.4 Related Work

V8

V8[13] is the JavaScript and WebAssembly engine developed by Google and used in the Chrome web browser. V8 is composed of a bytecode compiler, a non-optimizing JIT compiler, and two optimizing JIT compilers, gradually compiling functions with more optimizations as they get more used.

V8 collects information about the shapes of dynamic JavaScript objects called *maps*. These describe the layout of objects like on which offset is a field stored. This information is recored in an *inline cache* (*IC*) for each load and store. Apart from being used for speculative optimizations, inline caches also speed up the interpreted instructions. All inline caches are stored per function in a *feedback vector*.

An initialized inline cache can be in one of three states—*monomorphic* meaning only one object shape was observed, *polymorphic* if multiple object shapes were observed, and *megamorphic* if too many object shapes were observed. When optimizing, V8 speculates on an inline cache unless it is megamorphic.[14]

Contrary to Ř, V8 does not speculate eagerly (that is, whenever feedback information allows it), but only when the information is used. So, for example,

while compiling the JavaScript function

```
function f(a, b) { return a + b }
```

it only assumes the types of arguments `a` and `b` when compiling the plus operator, but not while emiting load of the arguments. This ensures that every speculation is used.

In JavaScript, the functions are often polymorphic at the start of the program, eventually stabilizing[15]. In order to prevent that, V8 only starts recording the type feedback after the function is invoked multiple times. It also holds the observations in a weak pointer, discarding dead types and callees.

HotSpot

HotSpot[16] is a Java Virtual Machine (JVM) currently developed by Oracle. It uses two JIT compilers, C1 (split into multiple levels) and C2.

Although Java is a statically typed language, Java programs often use a large class hierarchies. The main speculative optimization is performed on method calls, where virtual table lookups are substituted by direct calls, and in the case of the C2 compiler to inlining.

The observations in HotSpot are either *monomorphic* (one observation), *bimorphic* (two observations), or *megamorphic*. When compiling a method call, the monomorphic and bimorphic observations are translated to a direct call. But HotSpot speculates even on megamorphic call sites, optimizing for the most common tagets.[17]

1.5 Corpus

For experiments and analysis, we use two codebases.

The first is a collection of Ř benchmarks[4], consisting of four different suites:

- *Are We Fast Yet*, a collection of both micro and macro benchmarks based on the cross-language compiler benchmark suite[18],
- *Real Thing*, a collection of real-world programs,
- *Shootout* benchmarks from a popular cross-language benchmarks game[19],
- and *Simple*, which are custom written short scripts used for microbenchmarking individual R features.

The second codebase is a script from a Kaggle competition about machine learning on the Titanic dataset[20]. This script was chosen as a representation of a more real-world program. It contains 108 lines of code extracted from an Rmarkdown notebook and uses some of the most popular R libraries like `ggplot2` or `dplyr`.

[4] `https://github.com/reactorlabs/RBenchmarking`

Chapter 2
Recording Tool

In order to observe the behavior of Ř, we had to develop a tool for capturing and recording various event happening in runtime. In this chapter, we introduce the main design goals of the tool, as well as the details of the implementation, along with the ways to interface with it. Finally, we assess the performance impact of running Ř with the recording tool.

2.1 Motivation

The problem we observed was that the behavior of Ř is a black box. There are logging utilities for the JIT compilation, but these only print the RIR, PIR, and LLVM code at the different stages of compilation. This is useful for manual observation of the produced code, or for tracking down bugs in the compilation, but it does not reflect the rest of Ř.

What we want is an event log that would reflect the behavior of the compiler, as in figure 2.1. We want information on what functions were invoked and which version was the call dispatched. When a function is compiled, we do not know what was the reason for the compilation, as there are multiple heuristics that can be met. When a deoptimization happens, we can log the final PIR instruction that triggers the deoptimization, but not the runtime value that causes it, nor the impact it has on a feedback vector.

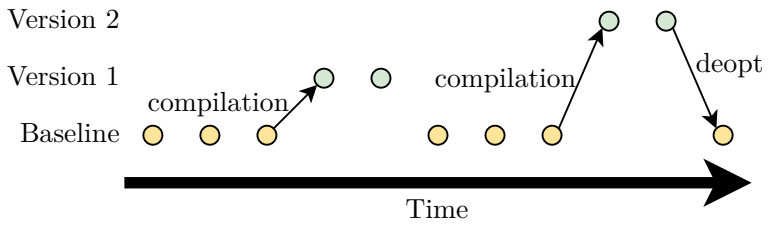

Figure 2.1 Idealized event log, each point represents a function invocation

In reality, the behavior is more complex than the motivating figure. When a compilation is triggered (we call that a start of *compilation session*), multiple closures can be lowered to their native version. At the time of compilation, the feedback vector of each function is static, but we have no idea of how we got to that state, thus we also have to observe all changes to of the feedback during the execution. We want to capture all of these events to properly analyze the behavior of Ř.

2.2 Design

One of the most important things about designing the recording tool was to minimize intrusiveness as much as possible, as observing the events is a matter of debugging, not a release build. Thus, we do not modify any function signatures, nor structure fields, and the recording tool has to be explicitly turned on when compiling Ř. When not enabled, it has basically zero impact on the compiler.

Another design goal was not to clutter the compiler code. Observing events is a cross-cutting concern impacting the whole codebase, similarly to logging. We do not want to have to keep around a state connected to the recording in the compiler because this would increase the code complexity.

What we ended up with is a series of *hooks*, very simple functions that are used by the rest of the codebase. These hooks are called at various points in the compiler, marking that an event has occurred. Internally, invoking a hook collects the information in a global state, but this is hidden from the caller. Since both the R and Ř code is sequential, we do not need to deal with any concurrency issues, but we still have to be careful with managing the global state. One feature that we leverage is that the C++ code is compiled without any exceptions, thus we can reliably use cleanup calls with confidence about the control flow of the program.

As an example, in the listing 2.1 in the function `doCall` we can see the logic around passing a function to the compiler when certain heuristics and conditions are met. `RecompileHeuristic` and `RecompileCondition` are pure functions, each having multiple heuristics or conditions, respectively, but the information about which one was triggered is not propagated anywhere. Therefore, when a condition or heuristic is met, we call to a hook that captures the information in the global state (also illustrated in the listing 2.1, in the function `RecompileCondition`). If a compilation occurs, we leverage this information from the global state with another hook. If a compilation does not occur, we reset the global state with the `recording::recordReasonsClear` hook.

This architecture of global state allowed us to make minimal changes to the codebase while still collecting important information during various stages of the compiler.

```
SEXP doCall(/* args */) {
  // ...

  if (!isDeoptimizing() && RecompileHeuristic(/* args */)) {
      if (RecompileCondition(/* args */)) {
          if (/* OSR condition */) {
              REC_HOOK(recording::recordOsrTriggerCallerCallee());
              call.triggerOsr = true;
          }
          DoRecompile(/* args */);
      }
  }
  REC_HOOK(recording::recordReasonsClear());

  // ...
}

bool RecompileCondition(/* args */) {
    if (fun->flags.contains(Function::MarkOpt)) {
        REC_HOOK(recording::recordMarkOptReasonCondition());
        return true;
    }

    if (!fun->isOptimized()) {
        REC_HOOK(recording::recordNotOptimizedReason());
        return true;
    }

    // ... Other conditions

    return false;
}
```

Code listing 2.1 Simplified code of compilation logic in interpreter/interp.cpp

```
void record(const SEXP e) {
    REC_HOOK(uint32_t old = seen);
    // Logic for modifying the `seen` member
    REC_HOOK(recording::recordSCChanged(old != seen));
}
```

Code listing 2.2 Example of recording state management in a hook caller in the class `ObservedTest` in file runtime/TypeFeedback.h

2.3 Implementation

The code is merged into the main branch of Ř and it is available on GitHub[8]. The whole implementation is in the folder `rir/src` in files `recording.{h, cpp}`, `recording_hooks.{h, cpp}` and `recording_serialization.h`, under the namespace `rir::recording`.

2.3.1 Hooks

All hook functions are defined in the file `recording_hooks.h`. This is the only file intended to be included by other parts of the compiler. Calling a hook either emits an event or adds information to the global state to be used by other hooks.

All calls to the hooks are surrounded by the `REC_HOOK` macro, which controls the conditional compilation of these hooks. If the recording is not enabled, the macro does not generate the call to the hook.

The only instances where a recording state is managed by the hook caller is when we need to capture information before calling the hook. These could be replaced by additional hook calls, but in this case, capturing the state is easier. For example in the listing 2.2, we check if the recording to the test feedback slots has changed the actual value. All other instances of state are managed by the hooks.

2.3.2 Recorder

The main orchestration is performed in the class `Record`, as defined in the listing 2.3. This class contains the observed events and closures they belong to.

We say that each event is connected to some closure, either a *Ř dispatch table*, *Ř function without a dispatch table* (when it is a top-level compilation), a *GNU-R compiled code* (represented by some SEXP), or a *primitive function*. When we first observe a closure, we create an associated `FunRecording` in the field `functions`. We try to find a name of the closure and the name of the environment in which it is defined, and serialize the closure (serialization has to be

```
class Record {
    std::unordered_map<const DispatchTable*, size_t> dt_to_recording_index_;
    std::unordered_map<int, size_t> primitive_to_body_index;
    std::unordered_map<SEXP, size_t> bcode_to_body_index;
    std::unordered_map<Function*, size_t> expr_to_body_index;

  public:
    std::vector<FunRecording> functions;
    std::vector<std::unique_ptr<Event>> log;

    template <typename E, typename... Args>
    void record(SEXP cls, Args&&... args);

    template <typename E, typename... Args>
    void record(const DispatchTable* dt, Args&&... args);

    template <typename E, typename... Args>
    void record(Function* fun, Args&&... args);
};

struct FunRecording {
    // Index into the array of primitive functions, or -1
    ssize_t primIdx = -1;
    // Possibly empty name of the closure
    std::string name;
    // Possibly empty name of the environment
    // in which the name was bound to the closure
    std::string env;
    // The serialized closure
    SEXP closure = R_NilValue;
    // The address of the closure
    uintptr_t address = 0;
};
```

Code listing 2.3 Simplified definition of `Record` and `FunRecording` classes

enabled). Every other time we observe an event connected with the same closure, we reuse the `FunRecording` by indexing with the `*_to_recoding_index` fields. Events only need to hold a single index into the `functions` field.

We use thoroughly that the GNU-R garbage collector is non-moving, as we can then index by the address of the objects and we can be sure that they are valid. There is still a possibility that an object whose address we have captured gets collected by GC and in its place a new object will be placed. To prevent that, we protect the objects by calling the `R_PreserveObject` function exported from GNU-R, marking the object as alive until it is released by a call to `R_ReleaseObject`. A drawback of this approach is the possibility of different runtime properties of some programs.

2.3.3 Events

An `Event` is an abstract class from which all other events inherit. Every event has an index of a `FunRecording` to which it is connected.

Compilation start and *compilation end* events denote the start and end of a *compilation session*, a single call to the compiler during which multiple closures might be lowered to native code. These events act as brackets of sorts, everything in between these is connected to the one compilation session. They record the heuristics used for triggering the compilation, its duration, and if it at any point failed. For each of the closures we then register a *compilation event*, where we reference the closure that was compiled, the type feedback it used, the PIR code that it resulted in and the LLVM bitcode to which it was lowered.

Every time we invoke a function, we register an *invoke event*. Due to contextual dispatch, there can be multiple destinations where a call can be dispatched. Thus, we capture the *call context* from the arguments and if we are dispatching to native code. There are also multiple control flow paths inside the interpreter that lead to dispatching, which is also captured.

A recording to the feedback vector is registered as *speculative context event*. We capture on which feedback slot the recording occurred, what is the updated information in it, if the recorded information is new, and if the change has occurred because of a deoptimization.

When a deoptimization occurs, the *deopt event* is triggered. It captures the deoptimization reason and value that triggered it, and the function and its feedback slot to which is the failed speculation connected.

There is also a definition of a *custom event*, which is a user-defined message that can be emitted with the R API.

2.3.4 Serialization

In order to observe the recorded data, we have to serialize it from the memory. For that, we transform the events into an R value as a named list with two

```
template <typename T>
struct Serializer;

template <>
struct Serializer<bool> {
    static SEXP to_sexp_(bool flag) {
        return flag ? R_TrueValue : R_FalseValue;
    }
    static bool from_sexp_(SEXP sexp) { return sexp == R_TrueValue; }
};
```

Code listing 2.4 Definition of the `Serializer` struct and its usage for type `bool`

fields, `functions` containing the recorded closures of a same-named field in the `Recorder`, and `events`, containing all of the recorded events. Each event is then a named list with the same fields as the corresponding C++ class.

To analyze the event log outside of R, there is a script `replayer.r` that can transform the representation into a CSV file, where one event corresponds to one line. The documentation for the CSV fields can be found in the Ř repository[8, documentation/recording.md].

The logic for serializing the event log is in file `recording_serialization.h` under the namespace `rir::recording::serialization`. The main type definition is an incomplete templated struct `Serializer` working as a typeclass, defined as in the listing 2.4. To specify how to serialize a type, we need to explicitly instantiate the `Serializer` with the template argument of the type we want to serialize and two methods `to_sexp_` and `from_sexp_`, as seen for the type `bool` in the listing. This allows us to compose the logic for more complex types from the more basic ones. For example, the serialization of a C++ vector container is generic for any type as it is internally using the `Serializer` struct to delegate the serialization of its elements.

For serialization complex structures, there are helper function called `fields_from_vec` and `fields_to_vec`, with their signatures defined as in the listing 2.5. The `Derived` template parameter specifies the current structure that we are serializing, which needs to have two static members—`className`, a C string that uniquely identifies this class, and `fieldNames`, a vector of C strings naming all individual fields. To use the serialization to and from fields, we need to pass in the fields into the helper functions in the same order in both calls. This can be seen in listing 2.6. Note how we only need to specify which fields to serialize, but the method of how they are serialized is managed through template resolving.

```
template <typename Derived, typename... Ts>
SEXP fields_to_sexp(const Ts&... fields);

template <typename Derived, typename... Ts>
void fields_from_sexp(SEXP sexp, Ts&... fields);
```

Code listing 2.5 Function definition of field serialization functions

```
constexpr const char* CompilationStartEvent::className = "event_compile_start";

const std::vector<const char*> CompilationStartEvent::fieldNames = {
    "funIdx", "name", "compile_reason_heuristic", "compile_reason_condition",
    "compile_reason_osr"};

SEXP CompilationStartEvent::toSEXP() const {
    return serialization::fields_to_sexp<CompilationStartEvent>(
        funRecIndex_, compileName, compile_reasons.heuristic,
        compile_reasons.condition, compile_reasons.osr);
}

void CompilationStartEvent::fromSEXP(SEXP sexp) {
    serialization::fields_from_sexp<CompilationStartEvent>(
        sexp, funRecIndex_, compileName, compile_reasons.heuristic,
        compile_reasons.condition, compile_reasons.osr);
}
```

Code listing 2.6 Example of using the fields serialization functions defined in 2.5

2.3.5 Interface

Currently, there are two ways to interact with the recording—environment variables passed to the program, and exported R functions.

Environment Variables

With the environment variables, it is possible to record the whole run of a program. This is done by setting `RIR_RECORD` to the path to which the recording data should be serialized, using the RDS serialization format for R objects[5, 1.8 Serialization Formats]. With the `RIR_RECORD_FILTER` variable, we can control which events should be considered, while ignoring the rest. The available values are `Compile`, `Deopt`, `TypeFeedback` and `Invoke` and multiple can be specified when separated by a comma. Custom events cannot be filtered out.

As an example, by calling

```
RIR_RECORD=output.rds RIR_RECORD_FILTER=Compile,TypeFeedback \
  R -f test.R
```

we record all compilation and type feedback events that were generated while running the script `test.R` into the file `output.rds`.

There is also a `RIR_RECORD_SERIALIZE` environment variable which if it is set to nonzero integer enables the serialization of closures and deopt events.

R API

The functions available in R are[8, documentation/recording.md]:

- `recordings.start()` starts or resumes the recording,
- `recordings.stop()` pauses the recording,
- `recordings.get()` returns the object with recorded functions and events,
- `recordings.save(filename)` saves the recording as an RDS to the given file,
- `recordings.load(filename)` loads the recording from the given file and returns the object representing it,
- `recordings.reset()` clears all of the recording informations,
- `recordings.enabled()` returns a boolean representing if we are recording right now,
- `recordings.setFilter(compile, deoptimize, type_feedback, invocation)` sets the filtering of individual events,
- and `recordings.customEvent(message)` creates a custom event with the associated message.

2.4 Assessment

The recording tool is implemented in almost 1700 lines of code, excluding blank lines and comments. Apart from the implementation files and around 40 calls to the recording hooks through the rest of the compiler, no other changes needed to be made in the compiler.

While running programs with the recording tool turned on and capturing all available events, we have observed around 20% decrease in performance. However, the final serialization of the events has increased the overall runtime by up to 40 times in programs with a lot of function calls. This is still acceptable for usage as an analysis tool, as it is usually not needed to capture all of the events, but just a subset.

The final serialized recording varies in size depending on the number of observed events, ranging from a little over 200 kilobytes for 18 events to 2.8 gigabytes for 8.5 million events.

When the tool is not turned on, no performance change was observed, as expected.

Chapter 3

Feedback Pollution

Over a run of a virtual machine, the collected feedback information used for speculative optimizations is prone to loss of precision, resulting in less optimized compilations, negatively impacting performance. We call this trend feedback pollution. In this chapter, we analyze the pollution as it happens in the Ř compiler. First, we introduce the feedback pollution on an example. Next, we present the analysis of feedback pollution on the Ř JIT compiler. Lastly, we discuss possible ways to reduce the pollution.

This chapter is based on a paper we presented at the VMIL 2024 (Virtual Machines and Intermediate Languages) conference[21]. As one of the authors, we contributed to the research and analysis described in the paper. This was the first use case for the developed recording tool, and in this chapter we describe the outcomes that were a direct result of using the tool.

3.1 Motivation

Let's take the example in the listing 3.1 and without considerid the Ř contextual dispatch. The observed events of the listing are demonstrated in the figure 3.1.

At first, we will execute the function `sum` with a vector of doubles. The type feedback will reflect that, and after few invocations, a compilation will get triggered, assuming on the types of arguments being double. This will significantly speed up execution because the arithmetic operations can be specialized for the double type.

Next, when we run the function with integers, we fail the assumption on doubles, trigger a deoptimization, and fallback to the baseline interpreted version. We also update the feedback information of `sum`, now reflecting both double and integer types. Because of this, the next function compilation cannot speculate on one specific type of argument and instead uses a very general

```
1  sum <- function(vec, init) {
2    s <- init
3    for (i in 1:length(vec))
4      s <- s + vec[[i]]
5    s
6  }
7
8  for (x in 1:1000) sum(doubles, 0.0)
9  for (x in 1:1000) sum(integers, 0L)
10 for (x in 1:1000) sum(doubles, 0.0)
```

Code listing 3.1 Motivating example for feedback pollution

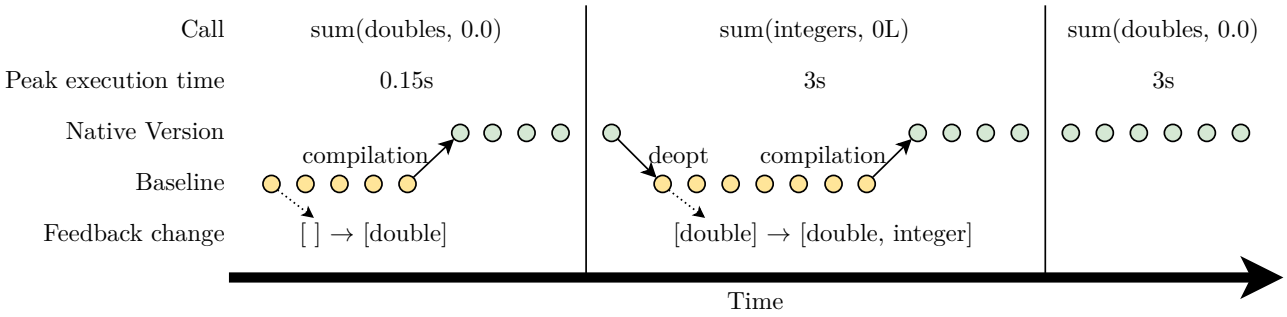

Figure 3.1 Event log of listing 3.1 without contextual dispatch, each point represents a function invocation

type (the only assumption made is that the value does not have any attributes). This makes the newly compiled native function about an order of a magnitude slower.

This is where we say that in the second compilation the *type feedback slots are polluted*. They contain too imprecise of an information, thus we specialize for a more general context and lose performance.

If we consider contextual dispatch, the performance is better, but not ideal (illustrated in 3.2). Same as without the contextual dispatch, we first observe and speculate on the type double. The final version is compiled under a first context.

When we call with integers, we do not dispatch into the already compiled version, because the call context is disjunct with the context of the first version. Instead, we run in the bytecode interpreter, updating the type feedback to also include integer. After a few invocations, we compile for the second context, and again we have to speculate on a more general type, resulting in a slower execution. But contrary to the non-contextual dispatch, when we call `sum` again with the `doubles` argument, it is dispatched to the first compiled version, and thus its execution is as fast as the first time we called it.

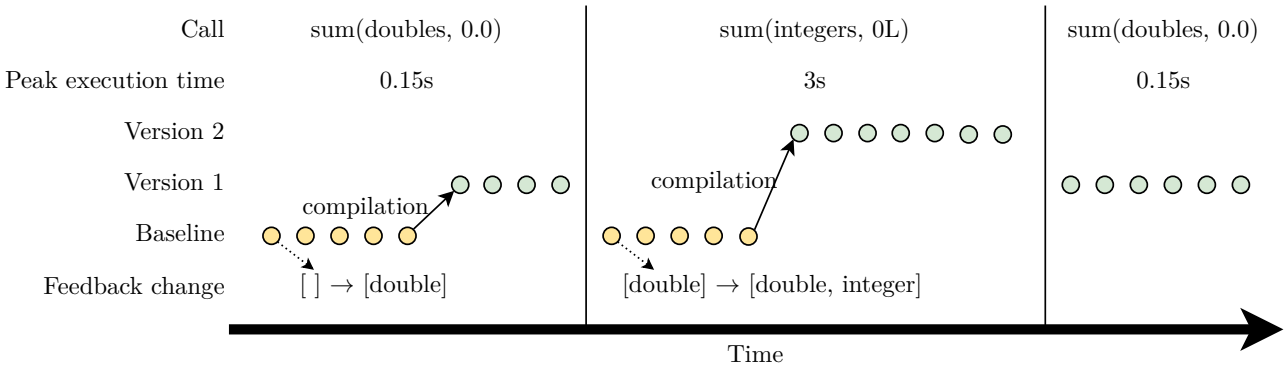

Figure 3.2 Event log of the listing 3.1 with contextual dispatch, each point represents a function invocation

3.2 Methodology

Our main goal is to quantify the pollution of the feedback vector. Pollution happens when between individual compilations the feedback vector changes, either because an interpreter has observed a new value or a native version fails on a speculation and deoptimizes. This implies that the first compilation cannot be polluted, hence we are interested in *subsequent compilations* (or also *recompilations*).

Formally, we define

- *polluted feedback slot* as a slot whose value at the point of compilation has changed from previous compilation,
- *feedback pollution* as a ratio of the number of modified feedback slots to the total number of feedback slots,
- *polluted compilation* as a compilation, where the feedback pollution is greater than zero,
- and *function pollution* as a ratio between the polluted compilations and the total number of recompilations.

Since the state of the feedback cannot go to a previous state (i.e. on an update, the feedback either stays the same or it is in a never before visited state), we can simply observe the state of the feedback slots when a compilation is triggered and compare it to the previous compilation.

To collect data, we used the recording tool introduced in the chapter 2. As input to the experiment, we used 16 benchmarks from the Ř benchmarks collection containing nearly 1300 lines of code, and the Kaggle competition program, both outlined in chapter 1.5. The Ř compiler is run with the default parameters, which means that a function is compiled after 100 invocations. Compilations triggered by loop iterations are ignored.

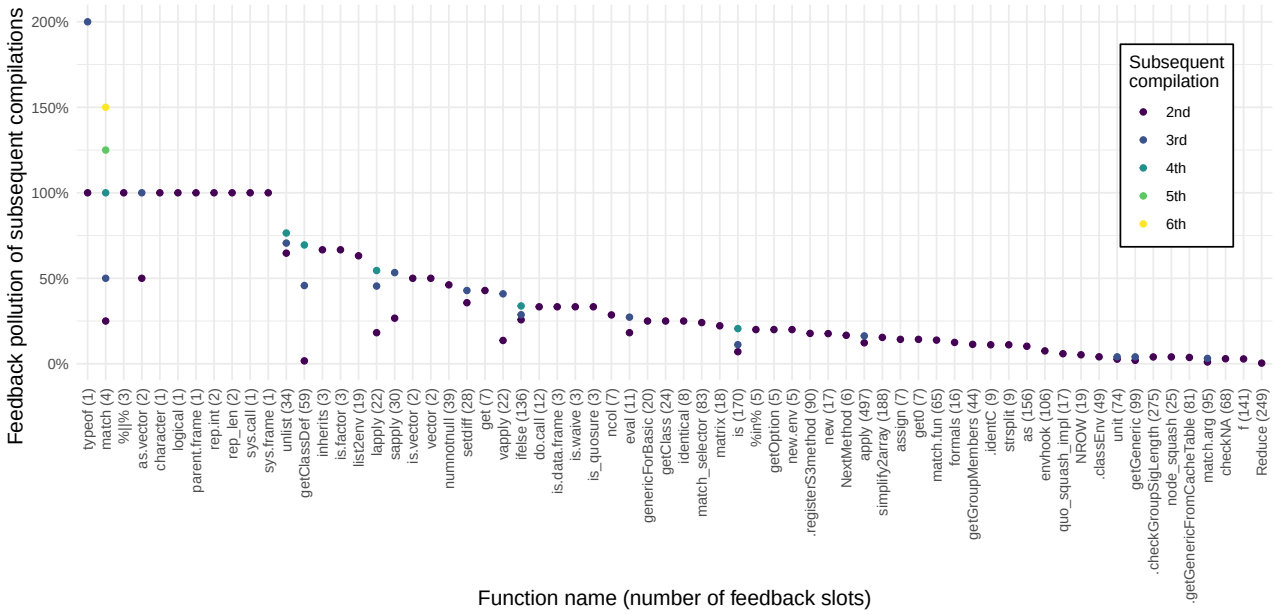

Figure 3.3 Function pollution in Kaggle script, each point represents a compilation[21]

3.3 Analysis

First, we will observe the Kaggle code. Running the script, 315 functions are compiled and 146 of them are compiled more than once. A function compilation is triggered 970 times, and out of these 824 are recompilations (2.6 recompilations on average per function). Overall, we have observed 90 polluted recompilations (10.9%), where 19 recompilations have more than half of the slots polluted and 10 have all of the slots polluted.

Figure 3.3 shows the function pollution in the Kaggle script. Each point represents a compilation, where the y-axis is showing an *accumulated pollution*, i.e. the summed pollution of all subsequent compilations up to that point. On the x-axis, we show the functions that were compiled more than once.

When we take as an example the function `typeof`, we have three compilations. The second compilation has a 100% accumulated pollution and the third a 200% accumulated pollution, meaning that both recompilations use all slots as polluted. The function is a small wrapper around a C function that returns the type of the argument. It has a single parameter to which the slot is connected, and since it can be called with any type, it will always pollute when a new type is observed.

Most functions compiled multiple times have about a quarter of slots polluted after the second compilation. The ones that change more than half of their feedback in the second compilation are small functions with very few

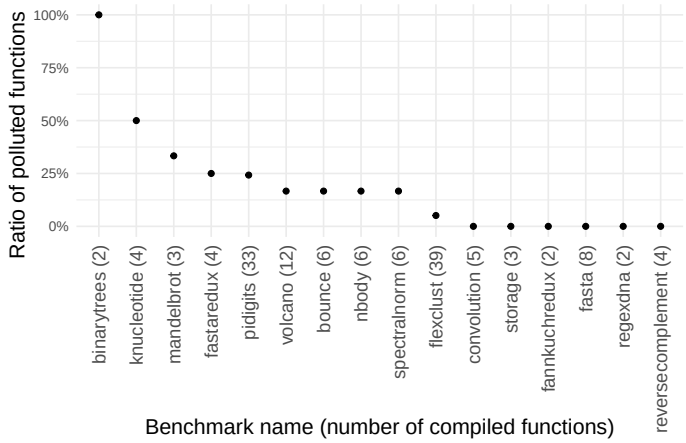

Figure 3.4 Pollution of functions in benchmarks[21]

feedback slots. Examining these functions, we have identified that most of them are polymorphic, with some of the slots being connected to the parameters that cause the pollution. But there are also larger functions, for example the `getClassDef` function has more than 50 feedback slots. It has a complex control flow, looking up the definition of a class in many different places. By calling this function with different classes, different paths get executed, filling up the feedback. This represent a function which is polluted because of a global state.

Looking at the benchmarks, we have observed much fewer compilations than in the Kaggle code. This is to be expected, as the benchmarks are mostly small numerical programs. In figure 3.4 we can see the *benchmark pollution*, which is the ratio of polluted recompilations out of all recompilations. Of the 16 selected benchmarks, 10 of them have at least one polluted compilation. The overall ratio between polluted compilations is 8.2%, which is very similar to the 10.9% observed in the Kaggle code. But when looking at function pollution, we have observed that out of 139 compiled functions there are 21 polluted functions (15.1%). This is very likely due to the nature of the benchmarks, as they are numeric programs that mostly use very few types. Still, a pollution can be observed and should be prevented.

Splitting the pollution by a feedback slot type, the observed values are the cause of most of the pollution. Out of the 11,199 slots in the Kaggle code, 0.5% of observed calls, 2.7% of observed tests, and 5.7% of observed values are polluted. This is not surprising, as the type feedback slots are the slots with the most variability.

Summary

In table 3.1 is a summary of the feedback compilation in the benchmarks and in the Kaggle code. We can see that feedback pollution happens in both the Kaggle code as well as in the benchmarks, although the benchmarks have lower pollution rates, most likely due to the stable nature of the code. We have observed that pollution is most likely caused by polymorphic functions that are often called with different types, but other causes for pollution can be for example a global state. Most of the polluted slots are the observed values.

	Kaggle	Benchmarks
Lines of code	108	1268
Compilations	970	257
Polluted compilations	90 (10.9%)	21 (8.2%)
Compiled functions	315	139
Polluted functions	66 (21%)	21 (15.1%)

Table 3.1 Summary of the feedback pollution in the corpus[21]

The code of the analysis is freely available on GitLab[1] as part of the VMIL paper[21] aritfact.

3.4 Pollution Prevention

After observing that feedback pollution is a real phenomenon, we might think of ways to reduce it.

One way would be to split the feedback vector into multiple. Since Ř already employs a contextual dispatch, the constructed context could be reused by feedback, constructing a unique vector for each call context with which the function is invoked. A proof-of-concept was implemented by Michal Štěpánek as a part of his master thesis[22]. This leads to a decrease of around 30% in the number of polluted compilations and a decrease of around 37% in function pollution. However, this solution brought new problems, since the observed information is much sparser and needs to be merged from multiple vectors. Splitting the feedback also negatively impacts the interpreter performance and complicates function compilation.

Another approach would be to implement a *feedback decay*. The idea is that the feedback information in each slot would slowly *decay* as new information is observed. This will need to be finely tuned, as very quickly the JIT could be stuck in a *deopt loop*, compiling a function just to trigger a deoptimization next time it is invoked.

Perhaps the most important question is whether feedback pollution is a real problem. We have demonstrated that it indeed does happen and that

[1] `https://gitlab.com/rirvm/splitfeedback-experiments/-/tree/artifact`

in synthetic examples it degrades performance. However, it remains unclear whether feedback pollution affects real programs and to what extent. This is not easy to answer as we would need to have an efficient implementation of at least one of the approaches to be able to draw any conclusion. Although interesting, it is beyond the scope of this thesis.

Chapter 4

Feedback Usage

After observing that the feedback vector can be polluted with information, the next step is to understand how the feedback is used in the compilation. Ultimately, we want to categorize slots in how they are used and quantify how much of the recorded information influences the compilation, as well as identifying the reason for why slot was not used in a compilation. First, we define the categories for how a slot can be used or not used. Next, we analyze the usage of feedback in Ř. Lastly, we acknowledge the limitations of the analysis.

In this chapter, we focus only on the *type slots* because, as observed in the feedback pollution analysis, these suffer from pollution the most. They are also the most variable parts, and having a much larger state space they allow for a richer analysis.

4.1 Definitions

- We define a *non-empty slot* as a slot that has at least one observation. An *empty* slot means that the instruction was never executed.
- We say that a slot is *referenced* if it is part of a function that is compiled to native, including slots inside inlined functions.
- A slot is then *read* if during a compilation the information in the slot is observed.
- A *used slot* is a type feedback slot that has an assumption connected to it in the final version of PIR (after all optimizations finished running). A used slot is always non-empty.

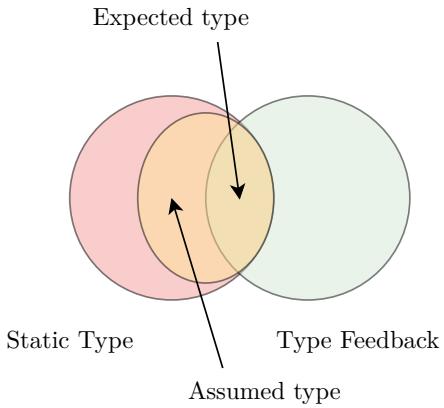

Figure 4.1 Illustration of the relationship between static, feedback, expected and assumed types

Used Slots

When an assumption on a type is emitted, it has a structure outlined in the listing 4.1. The value we speculate on is in the register `%1` with a *static type*, which is inferred from the call context, known types of builtins, and preceding assumptions. The instruction also has an associated *feedback type*, which is the union of all types observed in the interpreter. The speculated value is an input to an `IsType` instruction, which checks the actual runtime type against the *assumed type*. The result of the type check is then an input to the `Assume` instruction, along with the corresponding `Checkpoint`, resulting in a deoptimization if the type check fails.

```
[static type]   %1 = [instruction]   args... <feedback type>
lgl$#-          %2 = IsType          %1 isA [assumed type]
void                 Assume          %2, [checkpoint]
```

Code listing 4.1 PIR code structure of type assumption

The construction of the assumed type is not as straightforward as it might seem. We illustrate the process in the figure 4.1. Note that a larger area means a more general type.

First, we construct an *expected type* by intersecting the static and feedback types. If this intersection is empty, no assumption is emitted, otherwise we proceed to create the assumed type.

It can happen that the static type is more precise than the feedback type (in the figure 4.1 when the green area is non-empty). We call this a *narrowing* of the feedback type, since the static type narrows the feedback information to a more specific one. For example, this happens when we statically know that the value we are speculating on is a scalar, but we have also observed

non-scalar values.

If the R type of the expected type is not integer, float, or logical vector, we do not speculate on it as it is. The optimizations are not able to use this precise of an information, thus the compiler tries to relax the assumption first. We call this *widening*, as it widens the information of the feedback type (and also the expected type) while still staying in the bounds of the static type. This results in the final assumed type. If we cannot reasonably relax the type, we give up on assuming. A type can be both widened and narrowed at the same time.

If the assumed type is equal to the feedback type, we say that the feedback was used as an *exact match*. This implies that it has not been narrowed, nor widened.

Unused Slots

For the unused slots, we want to understand why they are not used and how this plays with the slot being polluted. We consider only non-empty slots, as empty type feedback slot cannot be speculated on.

The first reason we have identified is that the *slot is optimized away*. This happens when the instruction to which the type feedback is attached is not present in the final optimized PIR code. There are many reasons for removing instruction during optimizations, but the most common ones are *constant folding* and *dead code elimination*.

Another reason for not using a slot is that it contains *redundant information*. Formally we say that an *unused slot is redundant* if its type information is equivalent to any other slot, used or unused. This might be a broad definition, but it can still reveal us data about slot usage. Note that redundant slots and slots that are optimized away are not disjunct categories.

There are more reasons why a slot might not be used, like the information in the feedback is not useful for speculation, or the information is overridden by static information. We do not categorize these further, as this would require a much deeper analysis, possibly needing a rewrite of parts of the compiler in order to observe.

Polymorphic Slots

A *polymorphic slot* is a slot that has observed more than one distinct type. This is a superset of the *polluted slots*, i.e. a polluted slot is always polymorphic, but a polymorphic slot is not always polluted. A non-polymorphic slot is *monomorphic*.

Observing polymorphic slots allows us to outline the usage of polluted slots, while also observing the behavior of slots that have (potentially) too general of an information even before the first compilation.

Program name	Benchmark suite	Lines of code	Compilations	Referenced slots
bounce_nonames_simple	Are We Fast Yet	58	11	264
mandelbrot	Are We Fast Yet	65	14	358
flexclust_no_s4	Real Thing	163	144	5402
volcano	Real Thing	63	23	2037
binarytrees_naive	Shootout	31	22	1070
fannkuchredux	Shootout	63	6	251
fannkuchredux_naive	Shootout	62	5	244
fasta_naive_2	Shootout	88	17	598
knucleotide	Shootout	72	59	1493
pidigits/pidigits	Shootout	333	93	5710
titanic	-	108	2046	67477

Table 4.1 Overview of analyzed programs

4.2 Methodology

In order to collect all the information needed, we had to directly instrument the Ř compiler. We inspect the PIR code of closures and collect information about all of the assumptions, including the corresponding type test and cast instructions. We use the rich Ř APIs for traversing code and inspecting instructions. All of the code for collecting data is in the main repository in the branch `feedback-in-jits`[1].

Our unit for collecting information is a *closure version compilation*, one lowering of a closure from PIR to native code. For each compilation, we define its *universe* as the compiled closure and all of its inlinees. All counts are then in reference to this universe, so, for example, the number of referenced slots of one compilation is the sum of all slots in the universe. We ignore multiple inlinings of the same closure in one compilation, as we have observed that in most cases the slots are used in the same way across all inlinings.

The final data are aggregated over these closure version compilations. Thus when we say that there were two slots used, we mean that over all compilations a slot was used two times, it might even be the same slot. The reason for this was that between the individual compilations, the state of the slots can change, and there is no reasonable way to reference and quantify all slots after the program terminates.

We ran the analysis on a selection of ten benchmarks from the Ř benchmark suite and the Titanic Kaggle notebook, outlined in table 4.1. We can see that the Kaggle notebook has more compilations by an order of magnitude when compared to the benchmarks, and thus also many more referenced slots. All of

[1] `https://github.com/reactorlabs/rir/tree/feedback-in-jits`

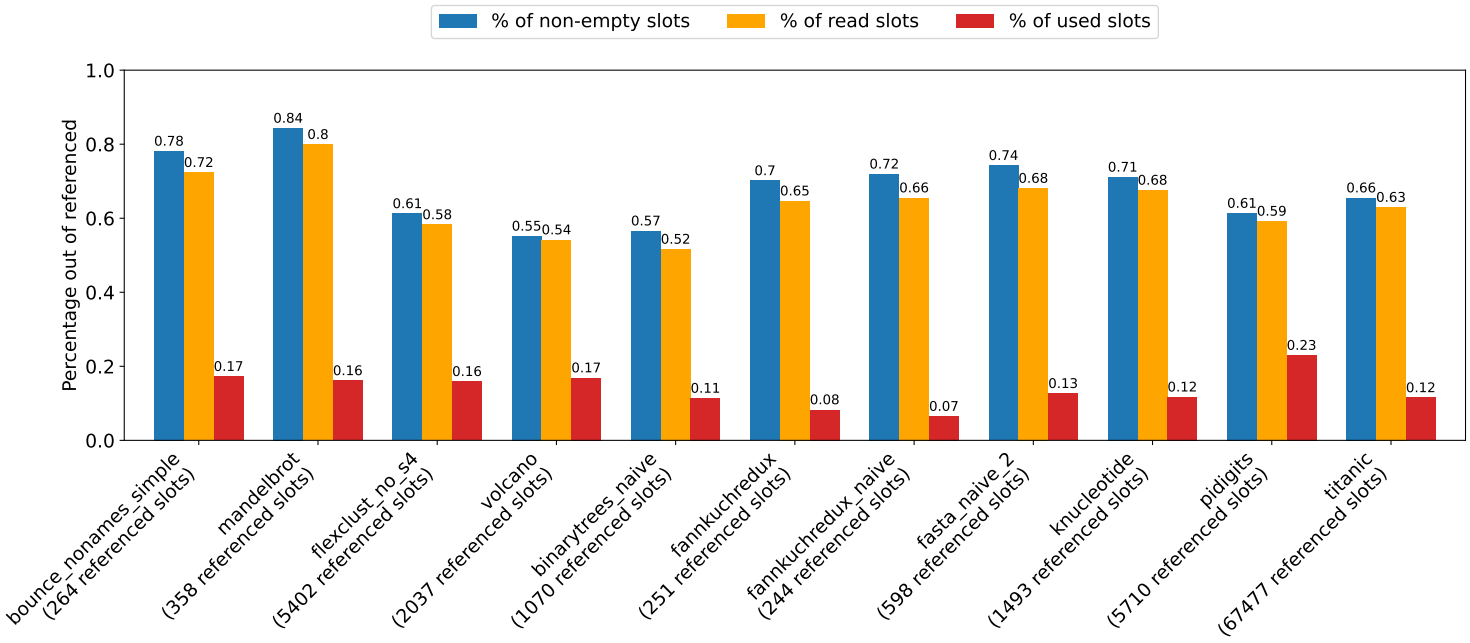

Figure 4.2 Usage of slots across closure compilations

the experiments were run with the invocation threshold for native compilation set to 10, as this allowed us to observe more compilations while still preserving the characteristics.

4.3 Observations

In contrast to the pollution analysis, while categorizing the slots we have observed that the Kaggle script behaves very much in the same way as the benchmark programs do. This is interesting because they are written in very different ways and are doing different kinds of computations. This might point to a deeper issue related to the behavior of the Ř compiler.

In figure 4.2 is the categorization of the used slots. We can see that most of the slots are not empty (68% on average), and most of the non-empty slots are read (93% on average), thus are considered for speculation. But on average only 14% of the slots are used (21% of non-empty slots). This is surprising, as the recording of information is impacting the speed of the bytecode interpreter, and yet the recorded information is used quite sparsely.

A hypothesis we had was that a polymorphic slot is less likely to be used due to having less information. Out of the polymorphic slots, 26% of them are used, compared to the non-empty monomorphic slots where 18% of them are used, contradicting the hypothesis. The observed numbers are probably due to the fact that by their nature polymorphic slots are on more frequently

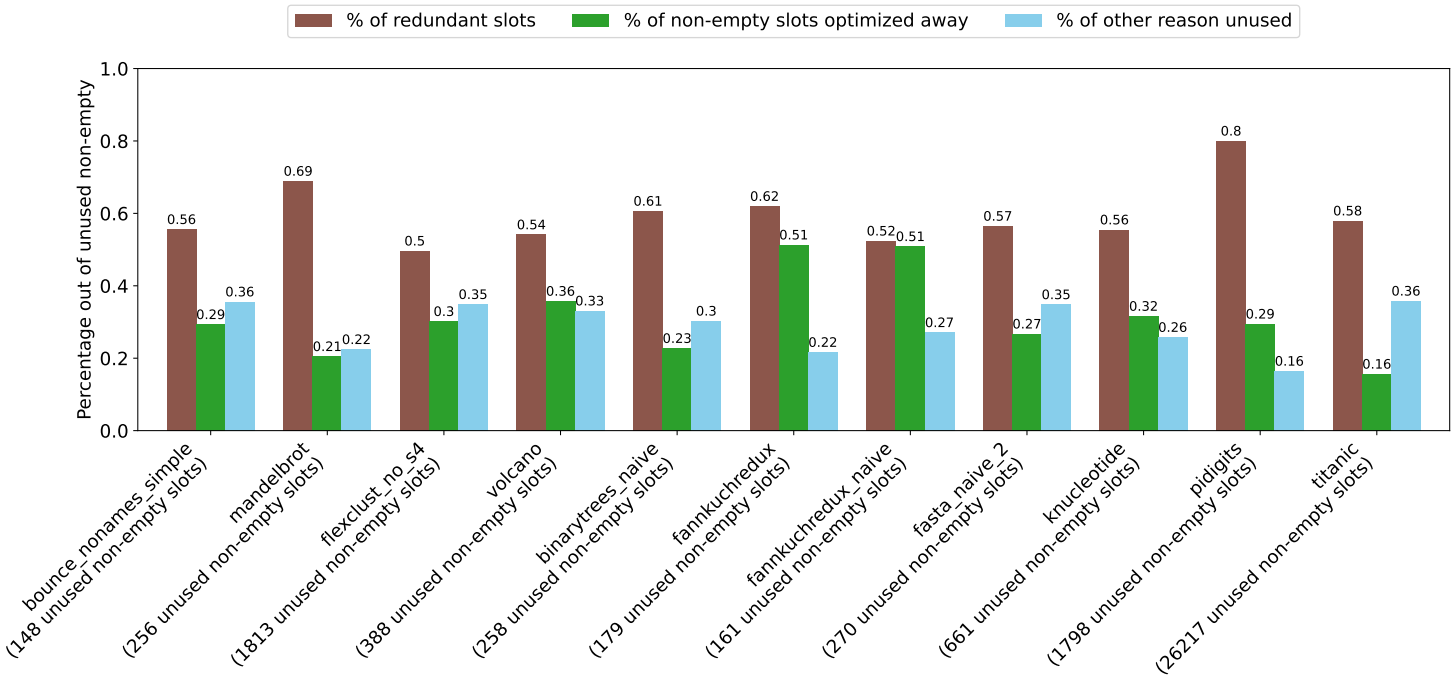

Figure 4.3 Categorization of unused slots across closure compilations

executed paths, making them more likely to not be optimized away and later speculated on.

When we consider the non-empty unused slots shown in figure 4.3, we can see that the dominating cause for a slot not being used is redundancy, on average 59% of all non-empty unused slots. This is expected in the benchmarks because they usually only use one numeric type across the whole program. But the Kaggle script uses many different types, yet 58% of the unused slots are deemed redundant, which is below the average. This leads us to believe that there is a deeper cause for a redundant slot in the way the information is collected in the interpreter, leaving room for optimizations.

Observing the slots that are optimized away, they take on average 31% of the non-empty unused slots. Separating the monomorphic and polymorphic slots, we see that 33% of the unused monomorphic slots are optimized always, whereas only 27% of unused polymorphic are optimized away. This might come from the same reasons as the comparison between monomorphic and polymorphic used slots, where the polymorphic ones are more likely to be on more frequently executed paths.

29% of all slots are neither redundant nor optimized away. This might include some actually redundant slots that are not caught by our approximate analysis, but there might also be other reasons for not using a slot, overlapping with the identified reasons. These need to be further analyzed.

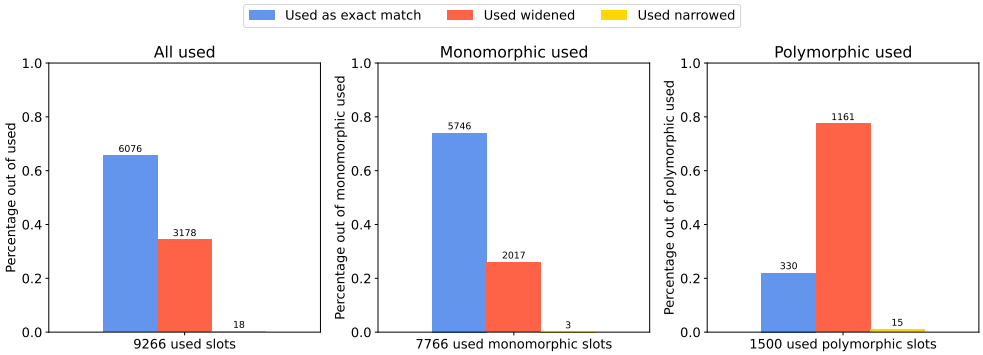

Figure 4.4 Categorization of used slots

When we look at the graph 4.4, we can see that more than half of the slots are used as an exact match. If we split the monomorphic and polymorphic used slots, we can see an even bigger distinction. 73% of used monomorphic slots are used as exact match, which means that most of the time when a slot is not polymorphic it contains an information precise enough to be used as is. On the other hand, 77% of used polymorphic slots are widened. This is to be expected, as the polymorphic slots have a more general type that gets widened before an assumption is made.

For the polymorphic slots used as exact match, we have observed two different kinds of slots. First is a single R type (integer, real, or logical) which might not be a scalar and which is either not an object, or it does not have any attributes. This is a precise enough type that we can speculate on it. The polymorphism comes from first observing scalarand then non-scalar values. The second kind has observed too many types and resolves to the most general R type (any R type which might be missing), but it still has a type flag either that it is not an object or that it does not have attributes. Although we cannot speculate on the R type, the type flag is still interesting enough that Ř speculates on it. This shows that the compiler is able to salvage information to speculate on even from a polymorphic slot.

Interestingly, there are very few slots that are narrowed by the static information, 18 to be precise.

- 3 of them have added information about the type not being NA. Since we do not record this information[2], it is trivial for the static type to narrow it in this way. These are the only monomorphic slots that are narrowed.
- 4 slots are narrowed to a scalar type. This is due to the slot observing also non-scalar types, but the static type can specialize the observation.

[2]In order to observe that a vector is not NA, we would need to inspect all elements of it and this is very costly for large vectors

- 2 slots have their type narrowed to a double and the only information used from the slot is that the value does not have attributes.
- In 9 cases, the static type removes a "might be missing" flag from the feedback type. In these cases, we have observed too many values and the type feedback falls back to the most generic type, which has the flag that it might be a missing value. But this speculation is on a `Force` instruction, which when forcing a value that is missing result in an error, thus the result of `Force` is always not missing or it diverges.

Conclusion

The main takeaway of the analysis is that the compiler is using very low numbers of feedback slots, and the main reason for that is redundancy. This leads to wasted time spent in the interpreter by recording observations. If we are able to detect which slots are dependent on each other, we could reduce the number of recordings and thus improve the interpreter performance.

Another key point is that a polymorphism, and by extend pollution, does not impact *if* a slot is used in compilation, but it influences *how* the feedback is used. Since the information in a polymorphic slot is more generic, we observe that it usually has to be widened before it is used.

The analysis is freely available online on GitHub [3].

4.4 Limitations

The biggest obstacle we have encountered during the analysis is the unknown origin of the type information. Due to the architecture of the compiler, we are unable to precisely track how a type is constructed or how a type information is propagated. This leads to a broad definition of redundant slots, which might catch unrelated slots or miss dependent slots recording different information due to type coercions. Still, the presented results give us an idea about the behavior of individual type feedback slots.

Another drawback is the tracking of inlined functions. A function may be inlined multiple times within a single closure compilation, making it difficult to summarize usage without skewing the results. A given slot might be used differently across inlinings or used in one and not the other. This can lead to double-counting across categories or missed usage patterns. We acknowledge these limitations and are looking for ways to mitigate them.

The key issue is how the conclusion relates to the performance of the executed code. While we have observed that very few feedback slots are used, leading to inefficiencies in the interpreter, we cannot determine the full impact on overall execution. To assess this, we would need to quantify the time spent

[3] `https://github.com/rihafilip/masters-thesis-analysis`

recording, not just in relation to the total interpreter time, but to the entire process. This falls outside the scope of the thesis.

While running the experiments, we have observed multiple runs of some of the benchmarks (namely flexclust_no_s4 and pidigits) and the Kaggle script result in different resulting numbers. However, the final characteristics are the same and the trends do not change. Therefore, the numbers presented in this chapter are from a single run of the experiments.

Chapter 5

Conclusion

The main goal of this thesis was to implement a tool which would help us to look under the lid of the JIT compiler pot, to increase its observability. Such a tool should then allow us to study different phenomena that occur along the dynamic compilations the JIT performs. Concretely, we were motivated to understand the behavior of the feedback that the compiler uses to generate code.

We implemented this tool as described in chapter 2, and as of now it is part of the Ř compiler codebase. We were able to minimize the impact on the rest of the compiler to a minimum by using conditional compilation and a series of hooks, simple functions that capture the state of the compiler at various points of execution and compilation.

Based on this tool, we were able to perform an analysis of feedback pollution in Ř, as described in the chapter 3 and VMIL paper[21]. We observed that feedback pollution occurs in approximately 19% of compilations. We also present ways in which we could implement a reduction of the pollution, either by splitting the feedback vector by context, or by introducing a feedback decay.

Continuing the observations about the feedback vector, we evaluated how the recorded type information is used during compilation. We observed that a very small number of recorded information is used (21% of feedback vector slots on average). We also conclude that if a feedback vector slot is polymorphic, and by extension if it is polluted, it does not affect whether it is going to be used, but it weakens the speculations that the compiler can assume on. Furthermore, we identified different reasons why feedback information is not used, namely that it contains duplicate information, or the instruction for which we record the feedback is optimized away.

5.1 Future Work

The main question we are currently unable to answer is *how much feedback pollution and the low usage of feedback information affect the performance of real programs*. This is a very hard question to answer as at this point we do not have information about performance impacts of the various components. We are unsure about how much time is spent in the interpreter, recording feedback information, in the JIT compiled code, in the builtin functions of GNU-R, or in the native extensions of libraries. Based on these metrics, we would be able to assess the real-world impact of the observations and prioritize optimizations for the most affected parts of the compiler.

Nevertheless, the observations made as part of this thesis unlock for us multiple ways in which we could advance the JIT compiler.

Reducing Pollution

In order to properly analyze how a polluted slot affects the runtime performance of JIT compiled code, we would need an *oracle* that at a point of compilation would be able to correctly return an *ideal feedback vector* such that the compilation produces as optimized code as possible. Since we want to observe the behavior of real-world programs, it is not feasible to hand-write this oracle for every compilation.

We could achieve at least an approximate oracle by extending the recording tool by its counterpart that would be able to *replay* the recorded information, i.e. influence a compilation based on previous observations. Iteratively, we would run the program with the trace of the previous run as an additional input from which it would approximate the ideal feedback vector.

Reducing Recoding

Another angle to take is reducing the time spent recording the feedback information in the interpreter.

If we are able to statically find redundant feedback slots and, therefore, eliminate some number of recording instructions, we should be able to speed up the interpreter, but this should not be to the detriment of JIT compilation. Another angle would be to dynamically observe which slots are being used and which are not and, based on this trace, conditionally turn off recording of certain slots.

Relaxing Assumptions

The last way we could improve the JIT is by relaxing the assumptions.

The Ř compiler does *eager speculations* on the observed values, which means that it tries to assume on the information whenever possible in hopes that this unlocks some optimizations later. This is contrary to how most other

JIT compilers do speculations, such as JavaScript V8 VM[13], where they only emit an assumption on a type at the point where the type is used. Eager speculation has the advantage that if an optimization is based on many complex speculations, it will be applied. The disadvantage is that we might restrict the type more than is necessary, e.g. we might speculate on a more specific type than is needed.

As an example, consider a function that has observed a double scalar type in one of its slots and based on the scalarness is able to do some significant optimizations, but the fact that the value is a double type is never used. Currently, the compiler still emits a guard on a double scalar. This means that if the observed value changes to an integer scalar, we fail the assumption even though the native code is still correct.

By carefully observing the usage of feedback information, we would be able to relax the assumptions in the native code if not all of the information is used. This relaxation could even extend to contextual dispatch. If we are compiling a function for a certain call context but we never use part of the context information, we could make the context more general, leading to more invocations of the function ending up in a native version.

Appendix A

Bytecode Examples

```
Code:
  1 LDCONST 1
  3 STARTFOR 4 3 30
  7 GETVAR 3
  9 LDCONST 5
 11 ADD 6
 13 LDCONST 7
 15 GT 8
 17 BRIFNOT 9 28
 20 GETFUN 10
 22 MAKEPROM 12
 24 CALL 11
 26 GOTO 29
 28 LDNULL
 29 POP
 30 STEPFOR 7
 32 ENDFOR
 33 POP
 34 GETFUN 10
 36 MAKEPROM 12
 38 CALL 11
 40 RETURN
Constant pool:
0:
 language {  for (i in 1:10) {; i..
1:
 int [1:10] 1 2 3 4 5 6 7 8 9 10
2:
 language 1:10
3:
 symbol i
4:
 language for (i in 1:10) {     i..
5:
 num 2
6:
 language i + 2
7:
 num 1
8:
 language i + 2 > 1
9:
 language if (i + 2 > 1) {     g(..
10:
 symbol g
11:
 language g(x)
12:
 Promise 0:
  Code:
    1 GETVAR 0
    3 RETURN
  Constant pool:
  0:
   symbol x
  1:
   language g(x)
  2:
   'expressionsIndex' int [1:4] N..
13:
 'expressionsIndex' int [1:41] NA..
```

Code listing A.1 GNU-R bytecode for listing 1.7.1, generated using A.3

```
0:
     0   push_  1
     5   visible_
     6   force_
     7   push_  10
    12   visible_
    13   force_
    14   ; :(1, 10)
         colon_input_effects_
    15   pop_
    16   swap_
    17   colon_cast_lhs_
    18   [ <?> ] Type#0
    23   ensure_named_
    24   swap_
    25   colon_cast_rhs_
    26   ensure_named_
    27   [ <?> ] Type#1
    32   dup2_
    33   ; NULL
         le_
    34   [ _ ] Test#0
    39   brfalse_  1
    44   push_  1L
    49   br_  2
1:
    54   push_  -1L
2:
    59   swap_
    60   pick_  2
    65   dup2_
    66   ; NULL
         ne_
    67   [ _ ] Test#1
    72   brfalse_  7
    77   dup_
    78   stvar_cached_  i{0}
    87   pull_  2
    92   ensure_named_
    93   ; NULL
         add_
    94   ldvar_cached_  i{0}
   103   [ <?> ] Type#2
   108   push_  2
   113   visible_
   114   ; +(i, 2)
         add_
   115   [ <?> ] Type#3
   120   push_  1
   125   visible_
   126   ; >(+(i, 2), 1)
         gt_
   127   [ <?> ] Type#4
   132   asbool_
   133   brtrue_  3
   138   br_  4
3:
   143   ldfun_  g
   148   [ 0, <0>, valid  ] Call#0
   153   mk_promise_  0
   158   ; g(x)
         call_  1
   175   pop_
4:
   176   dup2_
   177   ; NULL
         ne_
   178   brfalse_  7
   183   dup_
   184   stvar_cached_  i{0}
   193   pull_  2
   198   ensure_named_
   199   ; NULL
         add_
   200   ldvar_cached_  i{0}
   209   [ <?> ] Type#6
   214   push_  2
   219   visible_
   220   ; +(i, 2)
         add_
   221   [ <?> ] Type#7
   226   push_  1
   231   visible_
   232   ; >(+(i, 2), 1)
         gt_
   233   [ <?> ] Type#8
   238   asbool_
   239   brtrue_  5
   244   br_  6
5:
   249   ldfun_  g
   254   [ 0, <0>, valid  ] Call#1
   259   mk_promise_  1
   264   ; g(x)
         call_  1
   281   pop_
6:
   282   br_  4
7:
   287   popn_  3
   292   ldfun_  g
   297   [ 0, <0>, valid  ] Call#2
   302   mk_promise_  2
   307   ; g(x)
         call_  1
   324   [ <?> ] Type#11
   329   ret_

[Prom (index 0)]
0:
     0   ldvar_  x
     5   [ <?> ] Type#5
    10   ret_

[Prom (index 1)]
0:
     0   ldvar_  x
     5   [ <?> ] Type#9
    10   ret_

[Prom (index 2)]
0:
     0   ldvar_  x
     5   [ <?> ] Type#10
    10   ret_
```

Code listing A.2 RIR bytecode for listing 1.7.1

```
.Code <- as.symbol(".Code")

cat0 <- function(...) cat(..., sep = "")

pp.bytecode <- function(f, promise = FALSE) {
  offset <- if (promise) "  " else ""

  if (f[[1]] != .Code) {
    stop("Not a code")
  }

  code <- f[[2]]
  consts <- f[[3]]

  cat0("\n", offset, "Code:")
  for (i in 2:length(code)) {
    c <- code[[i]]

    if (is.numeric(c)) {
      cat0(" ", as.character(c))
    } else {
      opc <- as.character(c)
      opc <- substr(opc, 1, nchar(opc) - 3)

      cat0(
        "\n",
        offset,
        sprintf("%3i ", i - 1),
        opc
      )
    }
  }

  promise_idx <- 0
  cat0("\n", offset, "Constant pool:\n", offset)
  for (i in seq_along(consts)) {
    cat0(as.character(i - 1), ":", "\n", offset)
    c <- consts[[i]]
    if (is.list(c) && length(c) >= 1 && c[[1]] == .Code) {
      cat0(" Promise ", as.character(promise_idx), ":")
      pp.bytecode(c, TRUE)

      promise_idx <- promise_idx + 1
    } else {
      w <- if (promise) 33 else 35
      str(c, width = w, strict.width = "cut")
      if (i != length(consts)) {
        cat0(offset)
      }
    }
  }

  if (!promise) {
    cat0("\n")
  }
  invisible(NULL)
}
```

Code listing A.3 Code used for formating disassembled GNU-R bytecode

Bibliography

1. THE R FOUNDATION. *The R Project for Statistical Computing* [online]. [N.d.]. [visited on 2025-05-03]. Available from: `https://www.r-project.org/`.
2. *R Crash Course: Creating Publication-Quality Graphics* [online]. [N.d.]. [visited on 2025-05-03]. Available from: `https://r-crash-course.github.io/08-plot-ggplot2/`.
3. JENNYBC. *gapminder* [online]. GitHub, [n.d.] [visited on 2025-05-07]. Available from: `https://github.com/jennybc/gapminder`.
4. FLÜCKIGER, Olivier; CHARI, Guido; JEČMEN, Jan; YEE, Ming-Ho; HAIN, Jakob; VITEK, Jan. R melts brains: an IR for first-class environments and lazy effectful arguments. In: *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*. Athens, Greece: Association for Computing Machinery, 2019, pp. 55–66. DLS 2019. ISBN 9781450369961. Available from DOI: `10.1145/3359619.3359744`.
5. R CORE TEAM. *R Internals* [online]. 2025. [visited on 2025-04-14]. Available from: `https://cran.r-project.org/doc/manuals/r-release/R-ints.html`.
6. TIERNEY, Luke. *Notes on the Generational Generational GC for R* [online]. [N.d.]. [visited on 2025-04-26]. Available from: `https://homepage.stat.uiowa.edu/~luke/R/gengcnotes.html`.
7. THE R FOUNDATION. *The Comprehensive R Archive Network* [online]. [N.d.]. [visited on 2025-05-04]. Available from: `https://cran.r-project.org/`.
8. REACTORLABS. *rir* [online]. GitHub, [n.d.] [visited on 2025-04-28]. Available from: `https://github.com/reactorlabs/rir`.

9. LATTNER, Chris; ADVE, Vikram. LLVM: a compilation framework for lifelong program analysis & transformation. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, pp. 75–86. Available from DOI: 10.1109/CGO.2004.1281665.

10. MEHTA, Meetesh Kalpesh; KRYNSKI, Sebastián; GUALANDI, Hugo Musso; THAKUR, Manas; VITEK, Jan. Reusing Just-in-Time Compiled Code. *Proc. ACM Program. Lang.* 2023, vol. 7, no. OOPSLA2. Available from DOI: 10.1145/3622839.

11. GNU. *Flexible Array Fields (GNU C Language Manual)* [online]. 2023. [visited on 2025-05-04]. Available from: https://www.gnu.org/software/c-intro-and-ref/manual/html_node/Flexible-Array-Fields.html#Flexible-Array-Fields.

12. FLÜCKIGER, Olivier; CHARI, Guido; YEE, Ming-Ho; JEČMEN, Jan; HAIN, Jakob; VITEK, Jan. Contextual dispatch for function specialization. *Proc. ACM Program. Lang.* 2020, vol. 4, no. OOPSLA. Available from DOI: 10.1145/3428288.

13. *V8 JavaScript engine* [online]. [N.d.]. [visited on 2025-05-08]. Available from: https://v8.dev/.

14. STANTON, Michael. *V8 and How It Listens to You* [online]. 2016. [visited on 2025-05-08]. Available from: https://www.youtube.com/watch?v=u7zRSm8jzvA.

15. RICHARDS, Gregor; LEBRESNE, Sylvain; BURG, Brian; VITEK, Jan. An analysis of the dynamic behavior of JavaScript programs. *SIGPLAN Not.* 2010, vol. 45, no. 6, pp. 1–12. ISSN 0362-1340. Available from DOI: 10.1145/1809028.1806598.

16. ORACLE [online]. 2025. [visited on 2025-05-08]. Available from: https://www.oracle.com/java/technologies/whitepaper.html.

17. WINTERHALTER, Rafael. *An Introduction to JVM Performance* [online]. 2020. [visited on 2025-05-08]. Available from: https://www.youtube.com/watch?v=wgJWs14YcEs.

18. MARR, Stefan; DALOZE, Benoit; MÖSSENBÖCK, Hanspeter. Cross-language compiler benchmarking: are we fast yet? *SIGPLAN Not.* 2016, vol. 52, no. 2, pp. 120–131. ISSN 0362-1340. Available from DOI: 10.1145/3093334.2989232.

19. *Measured : Which programming language is fastest? (Benchmarks Game)* [online]. [N.d.]. [visited on 2025-05-04]. Available from: https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html.

20. RISDAL, Meg. *Exploring Survival on the Titanic* [online]. Kaggle, 2017 [visited on 2025-05-04]. Available from: https://www.kaggle.com/code/mrisdal/exploring-survival-on-the-titanic.

21. KRYNSKI, Sebastián; ŠTĚPÁNEK, Michal; ŘÍHA, Filip; KŘIKAVA, Filip; VITEK, Jan. Reducing Feedback Pollution. In: *Proceedings of the 16th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. Pasadena, CA, USA: Association for Computing Machinery, 2024, pp. 65–74. VMIL '24. ISBN 9798400712135. Available from DOI: 10.1145/3689490.3690404.

22. MICHAL, Štěpánek. *Obohacený kontextový dispatch pro Ř*. 2025. MA thesis. České vysoké učení technické v Praze. Vypočetní a informační centrum.

Contents of the Attachment

/
- `README.md` ..... Description of the contents
- `thesis` .... The source files for the PDF version of the Master's Thesis
- `rir-master` . Source of the Ř compiler with the implemented recording tool
- `rir-analysis` Source of the modified Ř compiler, used for the analysis of slot usage
- `used-analysis` ..... The analysis used for chapter 4 in the thesis
- `recorder-evaluate` .. The script and its sources used for evaluation of the recording tool