R Melts Brains

An IR for First-Class Environments and Lazy Effectful Arguments

Olivier Flückiger
Northeastern University

Guido Chari
Czech Technical University

Jan Ječmen
Czech Technical University

Ming-Ho Yee
Northeastern University

Jakob Hain
Northeastern University

Jan Vitek
Northeastern / Czech Technical U.

Abstract

The R programming language combines a number of features considered hard to analyze and implement efficiently: dynamic typing, reflection, lazy evaluation, vectorized primitive types, first-class closures, and extensive use of native code. Additionally, variable scopes are reified at runtime as first-class environments. The combination of these features renders most static program analysis techniques impractical, and thus, compiler optimizations based on them ineffective. We present our work on PIR, an intermediate representation with explicit support for first-class environments and effectful lazy evaluation. We describe two dataflow analyses on PIR: the first enables reasoning about variables and their environments, and the second infers where arguments are evaluated. Leveraging their results, we show how to elide environment creation and inline functions.

CCS Concepts • **Software and its engineering → Compilers**.

Keywords IR, first-class environment, lazy evaluation, R

ACM Reference Format:
Olivier Flückiger, Guido Chari, Jan Ječmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek. 2019. 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 (DLS '19), October 20, 2019, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3359619.3359744

1 Introduction

The R language [11] presents interesting challenges for implementers. R is a dynamic imperative language with vectorized operations, copy-on-write of shared data, a call-by-need evaluation strategy, context-sensitive lookup rules, multiple dispatch, and first-class closures. A rich reflective interface and a permissive native interface allow programs to inspect and modify most of R's runtime structures. This paper focuses on the interplay of first-class, mutable environments and lazy evaluation. In particular, we focus on their impact on compiler optimizations.

One might see the presence of `eval` as the biggest obstacle for static reasoning. With `eval`, text can be turned to code and perform arbitrary effects. However, the expressive power of `eval` can be constrained by careful language design. Julia, for instance, has a reflective interface that does not hamper efficient compilation [1]. Even an unconstrained `eval` is bound by what the language allows; for example, most programming languages do not allow code to delete a variable. Not so in R. Consider one of the most straightforward expressions in any language, variable lookup:

```
f <- function(x) x
```

In most languages, it is compiled to a memory or register access. From the point of view of a static analyzer, this expression usually leaves the program state intact. Not so in R. Consider a function doubling its argument:

```
g <- function(x) x+x
```

In most languages, a compiler can assume it is equivalent to `2*x` and generate whichever code is most efficient. At the very least, one could expect that both lookups of `x` resolve to the same variable. Not so in R.

Difficulties come from two directions at once. R variables are bound in environments, which are first-class values that can be modified. In addition, arguments are evaluated lazily; whenever an argument is accessed for the first time, it may trigger a side-effecting computation – which could modify any environment. Consequently, to optimize the body of a function, a compiler must reason about effects of the functions that it calls, as well as the effects from evaluating its arguments. In the above example, `` `+` `` could walk up the call stack and delete the binding for variable `x`. One could also call `g` with an expression that deletes `x` and causes the second lookup of `x` to fail. While unlikely, a compiler must be ready for it. Considering these examples in combination with `eval`, it is impossible to statically resolve the binding structure of

DLS '19, October 20, 2019, Athens, Greece

ACM ISBN 978-1-4503-6996-1/19/10…$15.00
https://doi.org/10.1145/3359619.3359744

R programs. Unsurprisingly, existing implementations resort to dynamic techniques to optimize code [6, 14, 16, 19].

The contribution of this paper is the design of PIR, an intermediate representation (IR) for R programs with explicit support for environments and lazy evaluation. PIR is a static single assignment (SSA) [12] code format inspired by our experience with the bytecode of the GNU R reference implementation, earlier work on FastR [6], the sourir IR we developed to model speculative optimizations [4], and an earlier attempt to optimize R using LLVM. In our experience, some of the most impactful optimizations are high-level ones that require understanding how values are used across function boundaries. We found that the GNU R bytecode [17] was too high level; it left too many of the operations implicit. In contrast, we found LLVM's IR [7] too low level for easily expressing some of our target optimizations.

PIR is part of **Ř**, a new just-in-time compiler for the R language. To motivate its need, we start with background on R and on related efforts in section 2. We give an informal overview of PIR in section 3. Then, section 4 details PIR and presents two transformation passes. The first, scope resolution, statically resolves bindings, and the second, promise inlining, removes lazy argument evaluation. Finally, section 5 illustrates how PIR helps **Ř**[1] reduce overheads. Our compiler is not complete and we are not yet able to run at competitive speed, so the results should be considered preliminary. **Ř** is available at https://github.com/reactorlabs/rir.

2 Background

This section describes key properties of environments and promises, and discusses work that deals with similar issues.

2.1 Environments in R

Inspired by Scheme and departing from its predecessor S, R adopted a lexical scoping discipline [5]. Variables are looked up in a list of environments. Consider this snippet:

```
g <- function(a) {
    f <- function() x+y
    if (a) x <- 2
    f()
}
y <- 1
```

The evaluation of `x+y` requires finding `x` in the enclosing environment of the closure `f`, and `y` at the top level. It is worth pointing out that, while R is lexically scoped, the scope of a free variable cannot be resolved statically. For instance, `x` will only be in scope in `g` if the argument `a` evaluates to true.

R uses a single namespace for functions and variables. Environments are used to hold symbols like `+`. While primarily used for variables, environments can also be created explicitly, *e.g.*, to be used as hashmaps. Libraries are loaded by the `attach()` function that adds an environment to the list of environments. A number of operations allow interaction with environments: `environment()` accesses the current environment; `ls(...)` lists bound variables; `assign(...)` adds or modifies a binding; and `rm(...)` removes variables from an environment. R has functions to walk the environment chain: `parent.frame()` returns the environment associated with the caller's call frame and `sys.frame(...)` provides access to the environment of any frame on the call stack. In R, frames represent function invocations and they have references to environments. Consider this code:

```
f <- function() get("x", envir=parent.frame())
g <- function() {x <- "secret"; f()}
```

Function `f` uses reflection to indirectly access `g`'s environment. This illustrates that any callee may access (and change) the caller environment.

2.2 Laziness in R

Since its inception, R has adopted a call-by-need evaluation strategy (also called lazy evaluation). Each expression passed as argument to a function is wrapped in a *promise*, a thunk that packages the expression, its environment, and a slot to memoize the result of evaluating the expression. A promise is only evaluated when its value is needed. Consider a function that branches on its second argument:

```
f <- function(a, b) if(b) a
```

A call `f(x<-TRUE,x)` creates two promises, one for the assignment `x<-TRUE`, and one to read `x`. One could expect this call to return `TRUE`, but this is not so. The condition is evaluated before variable `x` is defined, causing an error to be reported. Combined with promises, the `sys.frame` function allows non-local access to environments during promise evaluation:

```
f <- function() sys.frame(-1)
g <- function(x) x
g(f())
```

Here `g` receives promise `f()` as argument. When the promise is forced, there will be three frames on the stack: frame 0 is the global scope, frame 1 is `g`'s, and frame 2 is `f`'s frame.

`0: g(f())`
`1: x`
`2: sys.frame(-1)`

During promise evaluation, `parent.frame` refers to the frame where the promise was created (frame 0 in this example, as promise `f()` occurs at the top level). But, `sys.frame(-1)` accesses a frame by index, ignoring lexical nesting, thus extracting the environment of the forcing context, *i.e.*, the local environment of `g` at frame 1.

[1] Pronounced like a trilled "r", the sound one makes upon realizing that arguments can modify the environment of the function they are given to.

We leave the reader with a rather amusing brain twister. R has context-sensitive lookup rules for variables in call position. Variables that are not bound to functions are skipped:

```
f <- function(c) {c(1, 2) + c}
f(3)
```

The lookup of `c` in `c(1,2)` skips the argument `c`, since it is not a function. Instead, primitive `c()` is called to construct a vector. The second read of `c` is not in call position, thus it returns argument `c`, `3` in this case. The result is the vector `[4,5]` as addition is vectorized. Now, consider the following variation:

```
bad <- function() rm(list="c", envir=sys.frame(-1))
f(bad())
```

This time evaluation ends with an error as we try to add a vector and a function. Evaluation of `c(1,2)` succeeds and returns a vector. But, during the lookup of `c` for that call, R first encounters the argument `c`. In order to check if `c` is bound to a closure, it evaluates the promise, causing `bad()` to delete the argument from the environment. On the second use of `c`, the argument has been removed and a function object, `c`, is returned.

2.3 Related Work

R has one reference implementation, GNU R, and several alternative implementations. GNU R includes a bytecode compiler with a small number of carefully tuned optimizations [17]. Unlike ours, GNU R's bytecode implicitly assumes the presence of an environment for every function application. Variable lookup, in the worst case, requires inspecting all bindings of each environment in scope. To mitigate the lookup cost, GNU R caches bindings when safe. FastR's first version featured a type-specializing tree interpreter that outperformed GNU R [6]. It split environments into a statically known part (represented by arrays with constant-time accesses) and extensions that could grow and shrink at runtime. Environments were marked dirty whenever a reflective operation modified them. The second version of FastR uses Truffle for specialization and Graal for code generation [14, 20]. Graal's intermediate representation is general purpose [3]. FastR speculatively specializes the code based on profile-driven global assumptions. For instance, functions exhibiting a runtime stable binding structure are compiled under that assumption. The compiler elides environments and stores variables on the stack. Code is added to detect violation of assumptions and trigger deoptimization. Type specialization was also used in the ORBIT project, an attempt at extending GNU R with a type specializing bytecode interpreter [19]. On the other hand, the Riposte compiler tried to speed up R by recording execution traces for vector operations [16]. Riposte performed liveness analysis on the recorded traces to avoid unnecessary vector creations and parallelize code. None of these alternatives provides any special treatment for environment bindings. Our work departs from all these efforts in that we provide explicit support for environments and promises in the compiler IR. This allows us to combine static reasoning (when feasible) with speculative optimizations (when needed).

Other languages have some of the same features R has but, usually, are more amenable to compilation. Julia resembles R in that it is dynamically typed, reflective, and targets scientific computing. But, as shown by Bezanson et al. [1], it exhibits much better performance. This is due to a combination of careful language design and an implementation strategy that focuses on type specialization, inlining, and unboxing. Julia does not have lazy evaluation, it restricts `eval` to execute at the top level, and limits reflection. Another example is JavaScript. While it is also dynamic, the only way to add variables to a scope is using `eval`, which can only do so locally. Serrano [13] performs static reasoning on JavaScript by relying on type specialization and occurrence typing [18], as well as rapid atomic type analysis [8]. Whenever types cannot be statically determined, the compiler assumes the most likely structures ahead of time and relies on speculative guards for soundness. Smalltalk also features first-class contexts, although adding bindings at runtime is not supported. The Cog VM [10] maps context objects to the native stack and materializes contexts on demand when they are reflectively accessed.

3 An Intermediate Representation for R

We provide an example-driven explanation of PIR before the formal introduction. For readers who prefer a bottom-up explanation, we suggest starting with section 4. We distinguish between source-level R variables, which we call *variables*, and PIR local variables, called *registers*. Variables are stored in environments while the implementation of registers is left up to the compiler, and reflective access is not provided.

3.1 Scope Resolution to Lower Variables

We start with an example to illustrate how R variables are modeled, and if possible lowered to registers. We use the following simple function definition:

```
function() { answer <- 42; answer }
```

The function defines a local variable and returns its value. It translates to the following PIR instructions:

```
e0  =  MkEnv ( : G)
%1  =  LdConst [1] 42
       StVar (answer, %1, e0)
%3  =  LdVar (answer, e0)
%4  =  Force (%3) e0
       Return (%4)
```

First, `MkEnv` creates an empty environment nested in G, the global environment. As all values are vectorized, `42` is loaded as a vector of length 1. `StVar` updates environment `e0` with

a binding for variable `answer`. Then, `LdVar` loads variable `answer` again. As the examples in section 2 have conveyed, the compiler cannot assume much about the loaded value. Because returns are strict in R, the compiler inserts a `Force` instruction to evaluate promises. It refers to environment `e0` because a promise could reflectively access it. We record this fact as a data dependency. In general, we use a notation where actual arguments are inside parentheses and data dependencies outside. When `Force` is passed a value, rather than a promise, it does nothing.

After translation, the compiler runs a scope resolution pass to lower variables to registers. This requires combining an analysis and a transformation step. The analysis computes the reaching stores at each program point. Its results are then used to remove loads. In the previous example, the analysis proves that the value referenced by variable `answer` in instruction `%3` originates from `StVar (answer, %1, e0)`. Thus, `%3` can be substituted with `%1`. In case of multiple dominating stores, we insert a `Phi` instruction to combine them into a single register. Once this load is resolved, the environment is not used anymore, except for a dead store. Standard compiler optimizations, such as escape analysis of the environment and dead store elimination, can now transform this function into:

```
%1  =  LdConst [1] 42
       Return (%1)
```

This version has no loads, stores, or environment and does not require speculation.

3.2 Promise Elision

Promises consume heap memory and hinder analysis, since they might have side effects. Therefore, we statically elide them when possible with the following three steps: first, inline the callee; next, identify where the promise is evaluated; and last, inline the body of the promise at that location. To preserve observable behavior, inlining must ensure that side effects happen in the correct order. Consider the following code snippet:

```
f <- function(b) b
f(x)
```

This snippet translates to the following PIR instructions. First we show the creation of closure `f` and its invocation with one promise argument `x`:

```
  %1  =  MkClosure (f, G)
  %2  =  MkArg (pr0, G)
  %3  =  Call %1 (%2) G
pr0
  %4  =  LdVar (x, G)
  %5  =  Force (%4) G
         Return (%5)
```

The closure is explicitly created by `MkClosure`. Similarly, the promise `%2` is created by `MkArg` from `pr0`. Analogous to `Force`, `Call` has a data dependency on the environment because the callee can potentially access it. The translation of `pr0` does not optimize the read of `x` as this would require the equivalent to an interprocedural analysis.
Then, the function `f` translates to the following PIR:

```
f
  %6  =  LdArg (0)
  e7  =  MkEnv (b = %6 : G)
  %8  =  Force (%6) e7
         Return (%8)
```

The translation of `f` illustrates the calling convention chosen for PIR: it requires environments to be callee-created, *i.e.*, callees initialize environments with arguments. Accordingly, the `LdArg` in `f` loads an argument by position and `MkEnv` binds it to variable `b`.

We now walk through promise inlining. First, the callee must be inlined. Performing inlining at the source level in R is not sound as this would mix variables defined in different environments. However, this is not an issue in PIR; since environments are modeled explicitly, the inlinee keeps its local environment as `MkEnv` is also inlined. Therefore, after inlining `f`, we get:

```
%2  =  MkArg (pr0, G)
# inlinee
e7  =  MkEnv (b = %2 : G)
%8  =  Force (%2) e7
```

The next step is to elide the promise by inlining it where it is evaluated. We identify the `Force` instruction which dominates all other uses of a `MkArg` instruction. If such a dominating `Force` exists, it follows that the promise must be evaluated at that position. We inline `pr0` to replace `%8`:

```
%2  =  MkArg (pr0, G)
# inlinee
e6  =  MkEnv (b = %2 : G)
# inlined promise
%4  =  LdVar (x, G)
%5  =  Force (%4) G
```

We have succeeded in tracking a variable captured by a promise through a call and evaluation of that promise. The `%2` and *e*6 instructions are dead code and can be removed, leaving only the load and force of `x`.

4 PIR in Depth

This section describes PIR in detail. The IR is introduced in subsection 4.1. Scope resolution, the analysis that tracks R variables, is presented in subsection 4.2. Analysis precision is discussed in subsection 4.3. A technique to delay the creation of environments is presented in subsection 4.4. Lastly, promise inlining, which builds upon scope resolution, is presented in subsection 4.5.

4.1 Syntax and Semantics

Figure 1 shows the structure of programs. As in sourir [4], each function is versioned. The versions are compiled with different assumptions (A) and have different levels of optimization applied. Assumptions are predicates that may hold only for some executions of a function, *e.g.*, that arguments are already evaluated. **Ř** takes care of calling versions for which all assumptions are satisfied. A program is thus a set of functions, each with one or more versions with a function body and the promises it creates. Promise and function bodies are sets of basic blocks. Functions, promises, and basic blocks are labeled by names. All labels (id) are unique. Promises and functions in Figure 1 should not be confused with values that represent closures and promises; those are shown in Figure 2. A closure is a pair with a function and its environment, while a promise value is a triple with code, its environment, and a result. An environment is a sequence of bindings from variables to values.

Figure 3 shows the remainder of the PIR grammar. PIR is in SSA form: each statement (st) is constructed such that its result is assigned to a unique register. While there is only one kind of register in PIR, to help readability, our convention is to use (en) for registers that hold environments (or environment literals) and (%n) for everything else. PIR has instructions for the following operations: performing arithmetic; branching; deoptimizing a function; applying a closure; jumping to a basic block; loading arguments, constants, functions, and variables; creating promises, environments, and closures; forcing a promise; phi merges; returning values; and storing variables. Most of the instructions are unsurprising (and some have been elided for brevity). We focus our explanation on `MkEnv`, `MkArg`, and `Force`.

MkEnv. This instruction takes initial variables and a parent environment as arguments:

$$\texttt{MkEnv}\ ((x = a)^*\ : env)$$

The resulting environment contains the bindings $(x = a)^*$ and is scoped inside env. By default functions start out with an environment that contains all their declared arguments. Thus, a function defined at the top level with an argument called `a` has the following body:

```
%0  =  LdArg (0)
e1  =  MkEnv (a = %0 : G)
       ...
```

Variables can be added or updated with `StVar` and read with `LdVar`. The latter returns the value of the first binding for the variable in the stack of environments. That value can be a promise and may or may not be evaluated. When searching for a function, `LdFun` is used instead. The instruction evaluates promises and skips over non-function bindings. One optimization converts `LdFun` into `LdVar` when possible.

Function	$F ::= id : V^*$	labeled list of versions
Version	$V ::= A\ :\ C\ \ P^*$	assumptions, body, and promises
Promise	$P ::= id : C$	labeled piece of code
Code	$C ::= B^*$	list of basic blocks
Basic Block	$B ::= L : st^*$	labeled, with a list of statements
Label	$L ::= \texttt{BB}_n$	basic block number n

Figure 1. Programs

v	::=		
	|	v_e	environment
	|	v_p	promise
	|	v_c	closure
	|	c	constant
v_e	::=		
	|	$(x \mapsto v)^* : v_e$	variables + enclosing env.
v_p	::=		
	|	$<C, v_e, \_>$	unevaluated promise
	|	$<C, v_e, v>$	evaluated promise
v_c	::=		
	|	$<F, v_e>$	closure

Figure 2. Values

instr	::=		
	|	*Binop* $(a_1,\ a_2)$ *env*	binary op.
	|	`Branch` L	jump
	|	`Branch` $(a,\ L_1,\ L_2)$	branch
	|	`Call` $a_0\ (a^*)$ *env*	apply closure
	|	`Deopt` $(id,\ a^*,\ env)$	deoptimization
	|	`Force` (a) *env*	force promise
	|	`LdArg` (n)	load argument
	|	`LdConst` c	load constant
	|	`LdFun` $(x,\ env)$	load function
	|	`LdVar` $(x,\ env)$	load variable
	|	`MkArg` $(id,\ env)$	create promise
	|	`MkEnv` $((x = a)^*\ : env)$	create env.
	|	`MkClosure` $(id,\ env)$	create closure
	|	`Phi` $((L : v)^*)$	ϕ function
	|	`Return` (a)	return
	|	`StVar` $(x,\ a,\ env)$	store variable
st	::=		statements
	|	(%n | en) = *instr*	non-void instruction
	|	*instr*	void instruction
a, *env*	::=		argument
	|	(%n | en)	register
	|	*lit*	literal
Binop	::=		
	|	`Add`	
	|	...	
lit	::=		literals
	|	G	global env.
	|	O	placeholder env.
	|	_	no value
	|	*true*	true
	|	...	

Figure 3. Instructions

MkArg. This instruction creates a promise from an expression in the source program and an environment:

$$\texttt{MkArg}\ (id,\ env)$$

The instruction is mainly used to create promises for function arguments. A call such as `f(a+b)` translates to a load of a function `f`, the creation of a promise with body `p1` and a call:

```
  %0  =  LdFun (f, en)
  %1  =  MkArg (p1, en)
  %2  =  Call %0 (%1) en
p1
  %0  =  LdVar (a, en)
  %1  =  LdVar (b, en)
  %2  =  Force (%0) en
  %3  =  Force (%1) en
  %4  =  Add (%2, %3) en
         Return (%4)
```

The body of the promise contains two reads for `a` and `b` whose results get forced, a binary addition, and a return. The code is known statically, while the environment in which it is evaluated is a runtime value.

Force. This instruction takes a promise as input, evaluates it (recursively if needed), and returns its value:

$$\texttt{Force}\ (a)\ env$$

Note that *env* is a synthetic argument that is not needed for evaluation but describes a data dependency. The promise could access the current environment using reflective operations. If a is not a promise then a is returned intact. If $a =< C, v_e, \_ >$, then C is evaluated in v_e, and the result is stored in the data structure and returned. Otherwise, if $a =< C, v_e, v >$, then v is returned.

Typed Instructions. PIR instructions are typed, which allows more precise register types. The types include environments, vectors, scalars, closures, lists, etc. The type system also distinguishes between values and both evaluated and unevaluated promises. We omit additional details as the types are not relevant for the optimizations presented in this paper.

4.2 Scope Resolution

Scope resolution is an abstract interpretation over stores. The transformation draws inspiration from the *mem2reg* pass in *LLVM*. **Ř** first compiles variables to environment loads and stores and later lowers them to registers. The domain s of the analysis consists of sequences of abstract environments. Assume that we have environments $\texttt{e}1, \ldots, \texttt{e}n$ and variables $x_1 \ldots x_m$. Then, an abstract state s is a $n * m$ vector of sets of locations. A location is either a program point l or ϵ if the variable is undefined. We write $s_{i,j}$ to denote the abstract value of variable x_j in the environment accessed through register $\texttt{e}i$. The value $\top$ denotes the set of all locations—it represents the case where we do not know anything about a particular variable. The bottom value is represented by a vector where each element is the empty set. The analysis is defined by a transition function over statements and a merge function over states.

Transition Function. The transition function takes three arguments: a program point l, a statement st, and an abstract state s. The result is a new abstract state s'. We discuss the three interesting cases. Let st be the creation of a new environment stored in register $\texttt{e}i$ with some values for variables $x_1, \ldots, x_j$:

$$\texttt{e}i = \texttt{MkEnv}\ (x_1 = a_1,\ \ldots, x_j = a_j\ :\texttt{e}k)$$

Then, the resulting state s' is initialized with location l for variables $x_1, \ldots, x_j$ in the environment $\texttt{e}i$. Other variables in that environment are set to ϵ to denote that they are undefined.

$$(s')_{p,q} = \begin{cases} \{l\} & p = i,\ x_q \in x_1, \ldots, x_j \\ \{\epsilon\} & p = i,\ x_q \notin x_1, \ldots, x_j \\ s_{p,q} & \text{otherwise} \end{cases}$$

For the second interesting case let st be the store instruction which defines or updates variable x_i to a value held in register $\%j$ for environment $\texttt{e}k$:

$$\texttt{StVar}\ (x_i,\ \%j,\ \texttt{e}k)$$

This operation simply overwrites the state for that variable x_i with the current location.

$$(s')_{p,q} = \begin{cases} \{l\} & p = k,\ x_q = x_i \\ s_{p,q} & \text{otherwise} \end{cases}$$

The last case we describe is when an instruction taints the environment, *i.e.*, any instruction that may perform reflective manipulation; this includes `Call`, `Force`, and `LdFun`. For example, let st be a call instruction:

$$\texttt{Call}\ a_0\ (a_1,\ ..,\ a_n)\ \texttt{e}k$$

To be safe, the defined parts of the abstract state are set to $\top$, *i.e.*, we know nothing after this point.

$$(s')_{p,q} = \begin{cases} \top & s_{p,q} \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

We can improve precision by tracking parent relations between environments to avoid tainting them all. Also, the analysis can be extended to be interprocedural across `Call` or `Force` instructions. The state of a scope resolution in progress can be queried to resolve the target of a `Call` or `Force` instruction. Other mitigations to avoid tainting the state, such as speculative stub environments or special treatment for non-reflective promises, are discussed in subsection 4.3.

Merge. States are merged at control-flow joins. The merge operation is pointwise set union.

Transformation. Scope resolution computes the reachable stores. Based on its results, some `LdVar` instructions can be removed. Given a load instruction $\%i$ = `LdVar` (x_j, `e`k) with an abstract state s, there are three possible cases. First, when $s_{k,j} = \{l\}$, *i.e.*, the only observable modification to x_j is by the instruction at l, we can simply replace $\%i$ with the register stored by the instruction at that location. The second case is $s_{k,j} = \{l_1, \ldots, l_n\}$, *i.e.*, depending on the flow of control any one of the n instructions could have caused the last store. We use an SSA construction algorithm [2] to combine all stored registers in a phi congruence class. We replace $\%i$ with the `Phi` instruction produced by the SSA construction. Finally, the third case occurs if $s_{k,j} = \top$ or $\epsilon \in s_{k,j}$, *i.e.*, the load cannot be resolved and no optimization is applied.

Example. We conclude with an annotated example of a load that has two flow-dependent dominating stores:

```
function () {
  if (...) x <- 1
  else     x <- 2
  x
}
```

The translation starts by creating an empty environment. After some branching condition either `1` or `2` is stored in `x`. Finally, the value of `x` is loaded, forced, and returned.

```
BB0 :  e1   =  MkEnv ( : G)
       %2   =  ...
               Branch (%2, BB1, BB2)
BB1 :  %4   =  LdConst [1] 1
               StVar (x, %4, e1)
               Branch BB3
BB2 :  %7   =  LdConst [1] 2
               StVar (x, %7, e1)
               Branch BB3
BB3 :  %10  =  LdVar (x, e1)
       %11  =  Force (%10) e1
               Return (%11)
```

This function has one environment (`e1`) and one variable (`x`), thus it is represented as vector of length one, starting empty $\langle\{\}\rangle$. The scope analysis derives an abstract state $\langle\{5, 8\}\rangle$ (where `5` and `8` are the locations of both stores). Therefore we place a `Phi` instruction in BB_3 to join those two writes. We can replace the load `%10` with this phi.

```
BB0 :  %1   =  ...
               Branch (%2, BB1, BB2)
BB1 :  %4   =  LdConst [1] 1
               Branch BB3
BB2 :  %7   =  LdConst [1] 2
               Branch BB3
BB3 :  %10  =  Phi (BB1 : %4, BB2 : %7)
               Return (%10)
```

Since the load is statically resolved, dead store elimination is able to remove both `StVar` instructions. Combined with an escape analysis, the environment is also elided. The `Force` instruction is removed, since we know that the value is either of the two constants and not a promise.

4.3 Improving Precision

The problem with the analysis presented so far is that, because of R's semantics, any instruction that evaluates potentially effectful code taints the abstract environment. There are two kinds of non-local effects: callees may affect functions upwards the call stack by accessing the caller's environment, and callers pass promises that may affect functions downwards the call stack, when those functions force the promises. To make scope resolution useful in practice, the impact of these non-local effects should be somewhat mitigated. For instance, we rely on inlining to reduce the number of `Call` and `Force` instructions. Below we explain a special treatment for non-reflective promises and a speculative optimization assuming calls do not change the environment.

Contextual Assumptions. We leverage the fact that PIR functions can have multiple versions optimized under different assumptions to treat some promises specially by making contextual assumptions. For instance, if all arguments to a call are values, it is safe to invoke a version of the function that ignores the dangers of promise evaluation. Further specializations are possible for pure promises, or non-reflective ones. This specialization trades performance for code size, since functions must be compiled multiple times. **Ř** dynamically picks the optimal version of a function to invoke.

Stub Environments. If an environment is locally resolved, but could be tainted during a call using reflection, then we speculatively elide that environment and replace it by a stub. At runtime a stub environment has the same structure as a normal environment shown in Figure 2, but a more compact representation, since it does not need to support updates. If the stub environment is modified then it is transparently converted into a full environment. In PIR, stub environments are created by a structurally identical variant of the `MkEnv` instruction. After a call we check if the stub was materialized, in which case we deoptimize the current function. Consequently, analyses on PIR can assume stub environments to not experience any non-local modifications.

4.4 Delaying Environments

Ř has a deoptimization mechanism that transfers control back to the unoptimized version of a function. A deoptimization point includes the following instructions:

```
BB0 :  e1   =  MkEnv (foo = %i : e0)
               ...
       %2   =  ...
               Branch (%2, BB2, BB1)
BB1 :          Deopt (baselinePc, %1, ..., %n, e1)
```

An assumption (`%2`) is checked by the `Branch` instruction. Any boolean instruction can be used as an assumption. In

case it holds, we continue to BB_2, otherwise the deoptimization branch BB_1 is entered. The unconditional deoptimization instruction, `Deopt`, contains all the metadata needed to transfer control back to the baseline version of the function. The arguments *%1*, ..., *%n* are the values that the target code expects on the operand stack. Finally, since the baseline version in **Ř** requires the environment of a function to be always present, the `Deopt` instruction requires it to be present.

To avoid creating an environment at runtime whenever a compiled function performs any kind of speculative optimization, creation of environments should be delayed as much as possible. Optimizations are allowed to move `MkEnv` instructions into branches and even over writes to that environment. When that happens, the `StVar` is removed and the value is added to the initialization list. When an environment is used by multiple deoptimization points, then this is not sufficient, since each deoptimization branch will require the environment in a different state.

Partial escape analysis [15] intends to delay an allocation to only those branches where the object escapes. Similarly, in PIR we aim to materialize an up-to-date environment in each deoptimization branch, allowing us to elide the environment in the main path. This requires replaying stores between the original environment creation and the `Deopt` instruction. We use the output of scope analysis to determine the state of the environment in the deoptimization branch. A sufficient condition is that at the `Deopt` instruction none of the variables in the abstract state s (see sec. 4.2) is $\top$.

Assume there is an `StVar (bar, %j, e1)` instruction between `MkEnv` and `Deopt` in the previous example. We now proceed to assemble a synthetic environment to replace `e1` in the deoptimization branch. In this case the abstract environment `e1` according to scope resolution is such that `foo` is defined by the `MkEnv` instruction and `bar` is defined by the `StVar` instruction. In subsection 4.2 we presented our technique to replace a `LdVar` instruction with a PIR register. We now reuse the same technique to capture the current values of `foo` and `bar` as registers and then include them in a fresh `MkEnv` instruction:

```
BB0 :  e1  =  MkEnv (foo = %i : e0)
              ...
              StVar (bar, %j, e1)
       %2  =  ...
              Branch (%2, BB2, BB1)
BB1 :  e3  =  MkEnv (foo = %i, bar = %j : e0)
              Deopt (baselinePc, %1, ..., %n, e3)
```

Assuming we are able to materialize a copy of the environment in every deoptimization branch, it is then possible to remove the original `MkEnv`. This transformation duplicates variables for each deoptimization branch. They can later be cleaned up using some form of redundancy elimination, such as global value numbering.

Contrary to replacing `LdVar`s, it is possible to materialize environments even when the analysis results contain ϵ. For those cases a runtime marker is used to indicate the absence of a binding. `MkEnv` will simply skip a particular binding if its input value is equal to this marker value.

4.5 Promise Inlining

The analysis needed for promise inlining is a simple dataflow analysis. The values of interest are promises created by `MkArg`. The analysis uses a lattice for the state of a promise that starts at bottom, $\bot$, and can either be forced at program point l or leaked ($\triangledown$), and tops at $\top$. There is one such state per promise-creating instruction, thus the abstract state is a vector of length n where n is the number of `MkArg` instructions in the function. We present the abstract interpretation by discussing the transition function that takes a statement and an abstract state, and returns a new abstract state, and the merge function that combines two states.

Transition Function. The abstract state is initialized to $\bot^*$. There are three interesting cases.

First, given an instruction `Force (%i) ej` at location l, where *%i* is a `MkArg` instruction, we update the abstract state of the promise *%i* as follows: if the state is $\bot$, then it is set to l, indicating that this is the dominating `Force`. If the state is $\triangledown$ the result is $\top$, otherwise it stays unchanged.

Second, given an instruction `MkEnv` ($x_1 = a_1, \ldots, x_j = a_j$: `ep`), for any input $a_1, \ldots, a_j$ that refers to a promise, the state of that promise is set to leaked ($\triangledown$) if it is $\bot$, otherwise it stays unchanged.

Third, given any instruction which could evaluate promises, such as `Call`, `Force`, or `LdFun`, all escaped ($\triangledown$) promises are updated to $\top$.

Therefore, promises used first in a `MkEnv` and then in a `Force` instructions will end up at $\top$ and not be inlined. If we were to inline such a promise it would cause the result slots of the promise in the environment to be out of sync with the result of the inlined expression. It is possible to support some of those edge cases with an instruction to update the result slot of a promise.

Merge. When merging abstract states, identical states remain and disagreeing states become $\top$. The latter can happen for example in

```
function(a) {
  if(...) a
  a
}
```

where `a` is forced depends on a condition; it could be either line 1 or 2. While it would be possible to track those cases more accurately, we did not need it in practice yet.

Code Transformation. The promise inlining pass uses the analysis to inject promises at their dominating force instruction. As a precondition, we need a `MkArg` and the corresponding `Force` instruction to be in the same function. This only

happens after inlining, since initially creation and evaluation of promises is always separated by a call. The promise inliner will inline the promise body at the location of the dominating force, update all uses with the result of the inlinee, and remove both the `MkArg` and the `Force`.

If this promise originates from a `LdArg` instruction, then the promise originates from an argument passed to the current function. We do not know its code and therefore cannot inline it. On the other hand we can still replace all uses of the dominated `Force` instruction with the dominating `Force` instruction.

Example. We now present an example that combines scope resolution and promise inlining, shown in Figure 4. An inner closure `f` is called with `2` as an argument. `f` captures the binding of `a` from its parent environment. This translates (after scope resolution) to the PIR code shown in Figure 5. The parent environment O denotes the environment supplied by `MkClosure`. Since `f` is an inner function, it needs to be closed over the environment at its definition.

The first step necessary to get the promise creation and evaluation into the same PIR function, is to inline the inner function `f`. After this transformation we obtain the code in Figure 6. The open environment O is replaced with `e8`. And the argument `LdArg (0)` of the callee is replaced by the `MkArg (pr0, e8)` instruction of the caller.

Now, we can identify the dominating `Force` instruction at `%3`. Therefore, the promise inliner replaces the `Force` instruction with the body of the promise, yielding the result in Figure 7 (after another scope resolution pass).

Only after these steps finish can traditional compiler optimizations, such as escape analysis on the environment, dead code elimination, and constant folding, reduce the code to a single `LdConst` instruction.

5 Results

In this section, we assess scope resolution and promise inlining as means to statically resolve bindings and reduce the number of environments and promises needed by R programs. To do so, we present three experiments,each designed to answer one of the following questions:

RQ1 *What proportion of function definitions do not require an environment after optimizations?*

RQ2 *What proportion of function invocations do not require an environment after optimizations?*

RQ3 *What is the performance impact of scope resolution and promise inlining on the rest of the optimizations?*

Methodology. Measurements are gathered using **Ř**. Our first and second experiments rely on instrumenting the compiler to record information about code being compiled and also dynamic counters of events happening at runtime. For the last experiment, which looks at the impact of optimizations, we selected programs whose performance was impacted

```
g <- function() {
  a <- 1
  f <- function(b) b+a
  f(2)
}
```

Figure 4. An example with promises to be inlined.

```
g
  %7   =  LdConst [1] 1
  e8   =  MkEnv (a = %7 : G)
  %9   =  MkClosure (f, e8)
  %10  =  MkArg (pr0, e8)
  %11  =  Call %9 (%10) e8
  %12  =  Force (%11) e8
          Return (%12)
pr0
  %13  =  LdConst [1] 2
          Return (%13)
f
  %1   =  LdArg (0)
  e2   =  MkEnv (b = %1 : O)
  %3   =  Force (%1) e2
  %4   =  LdVar (a, e2)
  %5   =  Force (%4) e2
  %6   =  Add (%3, %5) e2
          Return (%6)
```

Figure 5. PIR translation of the function from Figure 4.

```
g
  %7   =  LdConst [1] 1
  e8   =  MkEnv (a = %7 : G)
  %10  =  MkArg (pr0, e8)
  # inlinee begins
  e2   =  MkEnv (b = %10 : e8)
  %3   =  Force (%10) e2
  %4   =  LdVar (a, e2)
  %5   =  Force (%4) e2
  %6   =  Add (%3, %5) e2
  # inlinee ends
  %12  =  Force (%6) e8
          Return (%12)
```

Figure 6. After inlining function `f` in Figure 5.

```
g
  %7   =  LdConst [1] 1
  e8   =  MkEnv (a = %7 : G)
  %10  =  MkArg (pr0, e8)
  # inlinee begins
  e2   =  MkEnv (b = %10 : e8)
  # inlined promise begins
  %13  =  LdConst [1] 2
  # inlined promise ends
  %6   =  Add (%13, %7) e2
  # inlinee ends
          Return (%6)
```

Figure 7. After inlining promise `p r0` in Figure 6.

Code usage is `check_code_usage_in_packages`, an analysis function shipped with GNU R. **Demos** runs all demos from the base packages. **Tcltk**, **Stats**, and **Utils** run all examples included, respectively, in the `tcltk`, `stats` and `utils` packages. **Pidigits** is from the language shootout benchmarks. **Mandelbrot** is from the *are-we-fast* benchmarks [9].

Figure 8. Description of benchmarks for **RQ1** and **RQ2**.

by **Ř**. To gather the measurements, we ran 5 invocations with 15 iterations of each benchmark on an Intel i7-3520M CPU, stepping 9, microcode version 0x21, a clock pinned at 1.2 GHz, on a Fedora 28 running Linux 5.0.16-100, with SpeedStep and lower C-States disabled. Our compiler has not been written for speed or optimized, so we discard the first 5 iterations to amortize compilation time.[2] We do not provide results for real-world applications, as our system is not ready to compete with mature R implementations.

5.1 Static Environment Reduction

For **RQ1**, we count the number of `MkEnv`s in compiled code. We distinguish between stubs and standard environments. Note that we ignore the environments created in deoptimization branches to allow transferring from optimized code to the baseline version. Figure 8 lists the programs analyzed. Figure 9 lists the number of closures that are compiled (Closures), the percentage of closures which have environments (Env), the percentage of closures that use stubs (Stub), and the percentage of closures that have no `MkEnv` instruction (No env). Stubs are inserted when our analysis determines that an environment is only accessible through reflection. Adding Stub and No Env, between 12% and 65% of environments are elided.

Program	Closures	Env	Stub	No env
Code usage	971	82%	1%	16%
Demos	381	81%	16%	3%
Tcltk	18	83%	11%	5%
Stats	871	85%	13%	2%
Utils	471	88%	10%	2%
Pidigits	139	79%	4%	17%
Mandelbrot	14	36%	29%	36%

Figure 9. Ratio of statically elided environments

5.2 Dynamic Environment Reduction

To answer **RQ2**, we counted `MkEnv`s executed at runtime with and without optimizations. Figure 10 lists the number of environments allocated in the non-optimized version (Baseline), the percentage reduction in allocated environments (Reduction), and the percentage of the reduction that is due to stubs (Stubbed). For instance, consider a program where the baseline allocates 100 environments and the optimized version only 50, with 10 stubs. This means we reduced the number of environments by 50%, and 20% of that reduction is achieved by stubbing. The data suggest that, for our benchmarks, 28% to 87% fewer environments were created.

Program	Baseline	Reduction	Stubbed
Code usage	2445996	36%	2%
Demos	192772	28%	5%
Tcltk	1271	43%	3%
Stats	3046614	28%	7%
Utils	2792534	33%	2%
Pidigits	9032031	31%	2%
Mandelbrot	8460641	87%	8%

Figure 10. Reduction in environments allocated

5.3 Effects on Optimizations

Finally, we address **RQ3** by providing slowdowns between **Ř** with and without optimizations. **Ř** performs traditional optimizations, such as constant folding, dead code elimination, and global value numbering. We posit that those optimization are helped by scope resolution and promise inlining. To test this hypothesis, we measure the impact of disabling those passes on running time. We run in four configurations: (1) default, (2) without promise inlining, (3) without scope resolution, and (4) with no optimizations. Figure 11 shows slowdowns relative to configuration (1). Disabling all optimizations slows down code by a range

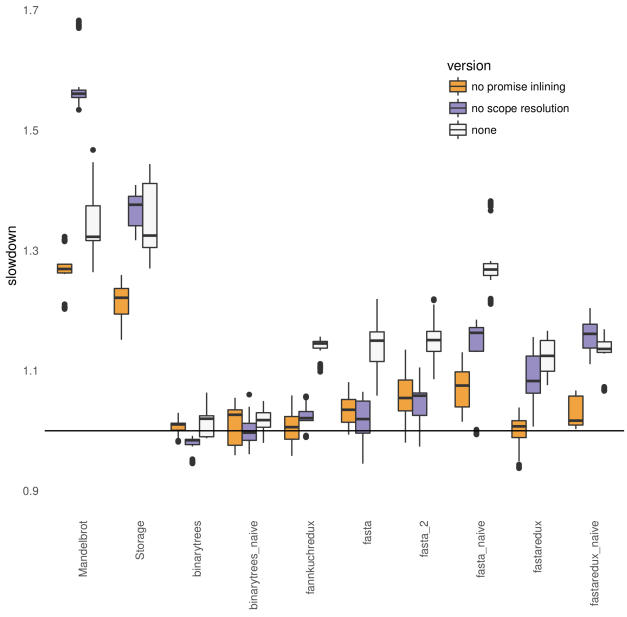

Figure 11. Slowdown with promise inlining (orange), scope resolution (purple), or all optimizations (white) turned off.

[2]The experiments are published in runnable form at gitlab.com/rirvm/rir_experiments/container_registry. All reported numbers are for revision: dba88e9bc417325a29c91acb088df7fe8109ca39e427e03931114e0715513bfafcd59a267812dcb1.

from 2% to 32%. Much of this slowdown can be attributed to the scope resolution pass, which when turned off, slows down execution from -1% to 55%. Disabling promise inlining has a smaller effect, ranging from a 0% to a 35% slowdown. Looking at the programs individually, we observe that `Mandelbrot` is surprising in that turning off scope resolution generates code that is slower than with no optimizations at all. A reason is that in this configuration, we still create stub environments and guards, while scope resolution would be the main consumer of this speculation. Here they only add overheads for the additional deoptimization points. Disabling scope resolution contributes significantly to the slowdown in `Fastaredux_naive`. `Storage`, `fannkuch_redux`, `fasta`, `fasta_2`, `fasta_naive` all have the expected behavior. As for `binarytrees` and `binarytrees_naive`, they show little slowdown in any configuration.

6 Conclusion

> *«Working on the thing can drive you mad. That's what happened to this friend of mine. So he had a lobotomy. Now he's well again.»* – Repo Man

Designing an intermediate representation for R has been a surprisingly difficult endeavor. Our goal was to arrive at a code format that captures the intricacies of the language while enabling compiler optimizations. Our explicit goals were to distinguish between arguments that need lazy evaluation and ones that do not, to distinguish between variables that are truly local and can be optimized and variables that must be allocated in environments and may be exposed through reflection, to allow for elision of environments when they are known to not be needed. To achieve this, we designed PIR, the intermediate representation of the **Ř** compiler. It has explicit instructions for creating environments, creating promises, and evaluating promises. Explicit modeling of constructs that are to be optimized away is a key design ingredient. For example, explicit environments allow functions to be inlined without fully resolving all R variables upfront.

The challenge presented by R is that it requires solving many problems at once. To get rid of laziness, one must track the flow of arguments and understand where they may be forced. To track arguments, one has to reason about environments and how they are manipulated. To discover if environments change, one has to analyze promises. This paper lays out our current strategy for dealing with this particular mix of dynamic features.

We illustrate the benefits of PIR with two optimizations, scope resolution and promise inlining. Scope resolution statically resolves bindings to reduce the issue of tracking loads and stores in environments. When successful, this pass lowers R variables to PIR registers. Explicit creation and evaluation of promises facilitates the inlining of promises. In combination, these transformations produce code with fewer promises and first-class environments. Evaluation shows encouraging results: up to 65% of functions do not need an R environment, resulting in up to 87% fewer environments created at runtime.

While PIR was designed for **Ř**, the design principles generalize to other implementations of the language. Other languages which have either lazy evaluation or first-class environments could adopt similar ideas in their intermediate representations.

Acknowledgments

We thank the anonymous reviewers and Tomáš Kalibera, for their insightful comments and suggestions to improve this paper. This work has received funding from the Office of Naval Research (ONR) award 503353, the National Science Foundation awards 1544542 and 1618732, the Czech Ministry of Education, Youth and Sports from the Czech Operational Programme Research, Development, and Education, under grant agreement No. CZ.02.1.01/0.0/0.0/15_003/0000421, and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme, under grant agreement No. 695412.

References

[1] Jeff Bezanson, Jiahao Chen, Ben Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. 2018. Julia: Dynamism and Performance Reconciled by Design. *Proc. ACM Program. Lang.* 2, OOPSLA (2018). https://doi.org/10.1145/3276490

[2] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1991). https://doi.org/10.1145/115372.115320

[3] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Workshop on Virtual Machines and Intermediate Languages (VMIL)*. https://doi.org/10.1145/2542142.2542143

[4] Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. 2018. Correctness of speculative optimizations with dynamic deoptimization. *Proc. ACM Program. Lang.* 2, POPL (2018). https://doi.org/10.1145/3158137

[5] Robert Gentleman and Ross Ihaka. 2000. Lexical scope and statistical computing. *Journal of Computational and Graphical Statistics* 9, 3 (2000). https://doi.org/10.1080/10618600.2000.10474895

[6] Toms Kalibera, Petr Maj, Floreal Morandat, and Jan Vitek. 2014. A Fast Abstract Syntax Tree Interpreter for R. In *Conference on Virtual Execution Environments (VEE)*. https://doi.org/10.1145/2576195.2576205

[7] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO.2004.1281665

[8] Francesco Logozzo and Herman Venter. 2010. RATA: Rapid Atomic Type Analysis by Abstract Interpretation – Application to JavaScript Optimization. In *International Conference on Compiler Construction*. https://doi.org/10.1007/978-3-642-11970-5_5

[9] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-language Compiler Benchmarking: Are We Fast Yet?. In *Symposium on Dynamic Languages (DLS)*. https://doi.org/10.1145/2989225.2989232

[10] Eliot Miranda. 2011. The Cog Smalltalk Virtual Machine. In *Workshop on Virtual machines and intermediate languages for emerging modularization mechanisms (VMIL)*.

[11] R Core Team. 2019. *R: A Language and Environment for Statistical Computing*. https://www.R-project.org

[12] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/73560.73562

[13] Manuel Serrano. 2018. JavaScript AOT Compilation. In *Symposium on Dynamic Languages (DLS)*. https://doi.org/10.1145/3276945.3276950

[14] Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. 2016. Optimizing R Language Execution via Aggressive Speculation. In *Symposium on Dynamic Languages (DLS)*. https://doi.org/10.1145/2989225.2989236

[15] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1145/2581122.2544157

[16] Justin Talbot, Zachary DeVito, and Pat Hanrahan. 2012. Riposte: A Trace-driven Compiler and Parallel VM for Vector Code in R. In *Conference on Parallel Architectures and Compilation Techniques (PACT)*. https://doi.org/10.1145/2370816.2370825

[17] Luke Tierney. 2019. *A Byte Code Compiler for R*. www.stat.uiowa.edu/~luke/R/compiler/compiler.pdf

[18] Sam Tobin-Hochstadt and Matthias Felleisen. 2010. Logical Types for Untyped Languages. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/1863543.1863561

[19] Haichuan Wang, Peng Wu, and David Padua. 2014. Optimizing R VM: Allocation Removal and Path Length Reduction via Interpreter-level Specialization. In *Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1145/2581122.2544153

[20] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Symposium on New Ideas in Programming and Reflections on Software (Onward!)*. https://doi.org/10.1145/2509578.2509581