# DATA-DRIVEN ECOSYSTEM MIGRATION
Non-Intrusive Migration of R Ecosystem
from Lazy to Strict Semantics

AVIRAL GOEL

Doctor of Philosophy

Khoury College of Computer Sciences
Northeastern University
Boston, Massachusetts, USA
2023

Northeastern University
Khoury College of
Computer Sciences

## PhD Thesis Approval

**Thesis Title:** Data-Driven Ecosystem Migration: Non-Intrusive Migration of R Ecosystem from Lazy to Strict Semantics

**Author:** Aviral Goel

**PhD Program:** [✓ Computer Science] ____ Cybersecurity ____ Personal Health Informatics

PhD Thesis Approval to complete all degree requirements for the above PhD program.

| | |
|---|---|
| Dr. Jan Vitek [illegible] | 31 March 2023 |
| *Thesis Advisor* | *Date* |
| Dr. Arjun Guha Arjun Guha | 31 March 2023 |
| *Thesis Reader* | *Date* |
| Dr. Jonathan Bell [illegible] | 31 March 2023 |
| *Thesis Reader* | *Date* |
| Dr. Manu Sridharan [illegible] | 31 March 2023 |
| *Thesis Reader* | *Date* |
| Dr. Jeffrey Foster Jeffrey S Foster | 31 March 2023 |
| *Thesis Reader* | *Date* |

**KHOURY COLLEGE APPROVAL**:

| | |
|---|---|
| [illegible] | 4/13/2023 |
| *Associate Dean for Graduate Programs* | *Date* |

**COPY RECEIVED BY GRADUATE STUDENT SERVICES**:

| | |
|---|---|
| Pete Morency | 4/13/2023 |
| *Recipient's Signature* | *Date* |

Distribution: Once completed, this form should be attached as page 2, immediately following the title page of the dissertation document. An electronic version of the document can then be uploaded to

# ABSTRACT

*Once you factor in documentation, debuggers, editor support,*
*syntax highlighting, and all of the other trappings,*
*doing it yourself becomes a tall order.*

— Robert Nystrom, "Crafting Interpreters" [22]

Evolving a contemporary mainstream language ecosystem can be a formidable undertaking owing to huge package repositories and millions of active users. For example, as of this writing, the R ecosystem has 19,022 packages in CRAN and over 2 Million users worldwide, as estimated by the R Consortium. At this scale, even a modest language update can impact millions of active users by breaking a significant portion of otherwise functional code, discouraging adoption. If the updates do not offer clear incentives to the users, partial adoption ensues, leading to fragmentation of the ecosystem from incompatible library "islands." Nevertheless, language designers routinely roll out updates to programming languages to fix bugs and incorporate new features without systematic migration strategies in place, leading to undesirable but easily avertible situations, such as the Python 2 to 3 migration fiasco.

In this thesis, I assess the feasibility of migrating R to strict semantics by studying the use of laziness in legacy R code. To evolve a language like R, which has a substantial package ecosystem and a large user base, requires a strategy that can scale while minimizing the impact on its users. I propose and evaluate a strategy for *large-scale migration* of the R language ecosystem to strict semantics semantics with *minimal user-visible changes* and *good precision*.

# ACKNOWLEDGEMENTS

First and foremost, I thank my parents and wife for their constant love and support. Their infinite patience, positivity, wisdom, and maturity helped me immensely through this challenging journey. Without them, this would not have been possible.

Frank Tip deserves special recognition; his push for securing funding made this journey possible.

Matthias Felleisen shared valuable advice and prescient predictions in my first year as a graduate student. He likes doing that!

Heather Miller was my first mentor at Northeastern. She took the time to guide me, critique my research ideas, and help me navigate complex situations.

Amal Ahmed's "Intensive Principles of Programming Languages", Eli Barzilay's "Programming Languages", and Kapil Arya's "Computer Systems" are easily the best courses I have ever taken. Not only did I learn a lot, but their teaching style also left a profound pedagogical impact.

This dissertation has significantly benefited from the suggestions and criticisms of my thesis committee. Arjun Guha meticulously reviewed every detail of my proposal and thesis. I met him innumerable times to discuss ideas and experiments. Jon Bell suggested an interesting application of fuzzing. Manu Sridharan provided sage counsel at many steps along the way. I met him in the early days of graduate school and have kept running into him since then. Jeff Foster raised many thoughtful questions and provided valuable suggestions. His "Dynamic Inference of Static Types for Ruby" was the first paper I presented as part of my graduate coursework. Jan Vitek identified the research potential in this problem and ensured rigorous empirical validation.

Meg Barry, Sarah Gale, and Laura Adrien promptly resolved all graduation, travel, and immigration issues. They made dealing with bureaucracy and administration a breeze.

I owe much gratitude to my colleagues at PRL. Ben Chung, Artem Pelenitsyn, Julia Belyakova, Ming-Ho Yee, and Alexi Turcotte provided excellent critiques for my practice talks. Apart from creating an intellectually stimulating environment, they make for lively conversationalists. Ben, in particular, can be counted upon to discuss anything from language semantics, video production, cheese making, rocket landing, and Minecraft. The solitary lifestyle imposed by the pandemic made their constant virtual presence even more valuable. Celeste Hollenbeck provided counsel and support at a time when it was sorely needed. I will be forever reminded of her courage in the face of adversity. Petr Maj ensured the smooth operation of the Prague servers, without which my large-scale analyses would not be so large. I am leaving with many fond memories of our travel with Guido Chari in Greece. In Filip Křikava, I found a friend, mentor, and collaborator. Gabriel Scherer helped me understand various aspects of language semantics. He is a storehouse of knowledge and a very warm person. Ben Greenman provided a lot of help along the way, patiently listening to my problems and research ideas. Michael Ballantyne explained the inner workings of Racket macros and shared insights for rewriting R code. Andrew Cobb's and Justin Slepak's witty remarks still make me laugh. The name `rastr` was coined by Andrew Cobb as a joke; it has remained with me ever since. Dimitri Racordon's ideas on engineering abstract syntax trees made their way into the design of `rastr`. Thank you all for helping me in this journey.

# CONTENTS

# 1 INTRODUCTION

Software migration is a routine activity in modern-day software development. Migration is the process of performing syntactic changes to a codebase to conform to a new version of the language, runtime, or dependency while offering the same functionality. The new versions typically fix bugs, address design issues, and add new features. Migration can be a trivial task if the new version is backward-compatible or a formidable undertaking if a feature is redesigned.

From the perspective of migration, a language ecosystem can be viewed as a tiered structure. At the base are the language implementation and associated core packages maintained and developed by a small set of core developers who intimately understand the design and implementation of the language. These form the base for the next tier consisting of packages from official packages repositories written by expert language users. Standing upon this is the last tier: notebooks, scripts, blog posts, and books written by end-users with varying expertise. Language changes happen at the base tier by developers on a relatively small codebase with the deepest understanding of language internals. The expert package authors, assisted by their tests suites, absorb these changes by updating their packages. Finally, they are propagated to the end-users, least equipped with the knowledge of the language's internals. Thus, the migration process proceeds like a ripple: from a few thousand lines of code to millions of lines of code, from maximum to limited user expertise, from a controlled to an open-ended codebase, from implementation and packages with tests to end-user scripts, usually without tests. This migration ripple originates from a controlled, predictable setting and culminates with a potentially unforeseen impact on the end-users.

From the perspective of migration, large package repositories of language ecosystems are a deterrent. While they are primarily responsible for a language's popularity and adoption; they discourage, or at the very least, restrict, evolution of the language. If fixing design mistakes and retrofitting new features in a language breaks millions of lines of otherwise functional code, the changes will be met with much resistance from the users. An even worse predicament is when changes introduced in a language are adopted partially by the users, splitting the ecosystem into islands of incompatible package ecosystems.

The goal of this dissertation is to address an instance of the software migration problem, that of migrating the R ecosystem from lazy to strict-by-default and lazy-on-demand. R uses the call-by-need evaluation strategy; function argument evaluation in R is delayed by bundling it in a thunk called a *promise*. The promise is evaluated when the argument is used, and the computed value is captured for future reference. Lazy evaluation in R is the building block of its meta-programming facilities; argument text can be accessed reflectively from a promise as a first-class expression object, modified, and evaluated in any environment. This is used for extending the language and for creating embedded domain-specific languages. Unfortunately, laziness is error-prone, inconsistent, and costly, at least when combined with side-effects in a language without type annotations. When a function with multiple evaluation orders is provided side-effecting arguments, the order of effects is hard to predict, leading to subtle bugs. R's laziness is inconsistent as there are points where evaluation is arbitrarily forced, e.g., the right-hand side of assignments and function returns. Laziness is costly as each argument has to be boxed in a promise object that must be allocated and freed, and compiler optimizations are hindered due to the side-effects from evaluating arguments.

This dissertation will focus exclusively on the migration of packages found in CRAN, the official package repository of R. Changes to the R implementation and core packages are made by the language developers; hence they are excluded from the migration process. End-user scripts and notebooks are also excluded since they are often use-and-throw, do not have tests, and are scattered all over the internet, unlike the packages in official repositories. If required, they can be bundled with tests to migrate them like packages. Migrating blog posts, books, and social media posts is beyond the scope of this work. Finally, this work only proposes and evaluates a migration strategy; the actual migration would require polishing the tools and techniques for industrial adoption and convincing the package authors to incorporate strict semantics.

## 1.1 MIGRATION IN THE WILD

In this section, I briefly discuss a few languages that have dealt with the problem of migration in recent times. Understanding their failures and emulating their successes will help us avoid pitfalls in our migration journey.

**PYTHON** One of the most unsuccessful migrations is the transition from Python 2 to Python 3. Python 3.0 was released in 2008 with a plan to end Python 2 support by 2015. However, the transition was so slow that support for Python 2 was extended till 2020[1]. Python 3 introduced numerous backward-incompatible changes[2] to Python 2 but did little to incentivize the developers to migrate. The official migration program shipped with the Python distribution could not automate the migration process beyond simple syntactic transformations. Migration to Python 3 proceeded glacially, with most major open-source Python packages pledging to drop support for Python 2.7[3] after a decade. Many factors facilitated this slow migration: first, "visible" incentives introduced by subsequent Python releases, such as support for asynchronous programming, second, migration of popular libraries such as NumPy [10] and django which served as dependencies for a significant chunk of Python's ecosystem, and lastly, improvement in migration tooling support over time.

**SCALA** Scala 3, released in May 2021, introduces new features and restricts and drops a few Scala 2.13 constructs while retaining a significant chunk of the old syntax. To facilitate interoperability, both Scala 2.13 and Scala 3 share the same ABI, and Scala 3 source can be consumed as a dependency by a Scala 2.13 compiler after compilation. The Scala 3 compiler with appropriate flags can migrate parts of Scala 2.13 to the new syntax. The scalafix refactoring tool provides rewrite rules to fix some incompatibilities in Scala 2.13 source to facilitate migration. Libraries can be migrated incrementally: first migrate the dependencies, then the compiler options, and finally, the deprecated library syntax using scalafix. Formatting tools and IDE plugins are in the process of adding Scala 3 support. The migration tooling does not yet support automatic migration of `implicits` and macros. As of this writing, most Scala libraries are still stuck on older Scala versions. According to Scaladex, the Scala library index, there are 1,364 libraries on version 3, 3,358 on version 2.13, 5,570 on version 2.12, and 4,644 on version 2.11. Scala migration has a long way to go.

1 Sunsetting Python 2: https://www.python.org/doc/sunset-python-2/
2 Miscellaneous Python 3.0 Plans: https://www.python.org/dev/peps/pep-3100/
3 Python 3 Statement: https://python3statement.org/

**JAVASCRIPT** Because of its humble beginnings as a language for embedding short code snippets in web pages, JavaScript had many design issues related to its dynamic behavior and reluctance to throw runtime errors, hindering the development of large dynamic webpages. The *strict* mode, an opt-in dialect of JavaScript, sought to address some of these design oddities. It was designed to be subtractive: it didn't add new features, only eliminated problematic ones. For the most part many common coding errors were turned into runtime errors, such as assignment to an undeclared variable. This facilitated adoption by ensuring that the code exhibited the same runtime behavior when run under a browser that did not yet support *strict* mode. Furthermore, instead of introducing new syntax, `"use␣strict";`, a literal string constant followed by a semicolon, was chosen to opt into *strict* mode. Evaluating this constant has no side effects, so browsers that did not implement *strict* mode would ignore its presence. This enabled users to incrementally migrate their scripts to the new dialect without worrying about browser support. Today, all major browsers support *strict* mode, and JavaScript modules are in *strict* mode by default.

**TYPESCRIPT** TypeScript is a typed superset of JavaScript. It provides a structural type system for JavaScript for static type-checking. TypeScript only extends the JavaScript syntax to add support for type annotations; hence, JavaScript code remains valid TypeScript code. Types can also be provided through external declaration files. After typechecking, the TypeScript compiler translates the code to plain JavaScript by erasing the types. TypeScript neither changes the program behavior based on the inferred types nor does it add any additional runtime libraries to the program. This makes it trivial to migrate existing JavaScript code to TypeScript. Types can be added incrementally. Tight editor integration helps in identifying bugs based on type information. This has made TypeScript extremely popular among JavaScript developers.

**HACK** Hack is a dialect of PHP, developed and used at Facebook. The migration of PHP to Hack at Facebook has enjoyed the benefits of a closed feedback loop. As all Hack users share an employer and a source code repository, it is possible to develop language features targeted at relevant usage patterns and ensure rapid migration. Unfortunately, most mainstream language ecosystems don't enjoy such closed feedback loops. They are used by multiple corporations and open-source developers which makes it difficult to ensure Hack like rapid migration.

We can make a few valuable observations from these examples for software migration. It is not surprising in hindsight that forcing a new backwards-incompatible Python version with no tooling support and incentive for adoption, failed. Scala is providing well-integrated tooling support from the beginning to facilitate migration to the new version. JavaScript's *strict* was deliberately designed to be backwards-compatible, so that the migrated code could still be run on browsers that did not support it. Hack speaks to the advantage of having a controlled ecosystem, which is not usually the case for most mainstream languages.

## 1.2 THESIS

My thesis is:

> It is possible to *effectively* migrate R from a lazy to strict semantics with *minimal impact* on its legacy code.

A migration strategy is *effective* if it *minimizes user-visible changes* and *automates migration* with *provide good precision*. I will address this thesis in two steps. First, I will assess the impact of migration on the ecosystem. Impact analysis will help assess the suitability of strict semantics for R and identify the affected parts of the ecosystem. It will also help identify incentives for the users to migrate to strict semantics. Next, I will develop tools to migrate the ecosystem at scale. Automation is crucial for migrating R's large package repository since manual migration is time-consuming and error-prone and will inhibit adoption. The proposed tools will perform a minimal rewrite of R code. Finally, I will evaluate the effectiveness of these tools at scale and manually perform a migration of the tidyverse ecosystem.

## 1.3 CONTRIBUTIONS

I have published the following papers towards the implementation of my three-step migration strategy:

1. On the Design, Implementation, and Use of Laziness in R [8]
   This paper reviews the design and implementation of laziness in R and presents a data-driven study of how generations of programmers have put laziness to use in their code. Analysis of 16,707 R packages reveals that for the most part R code appears to have been written without reliance on, and in many cases even knowledge of, delayed argument evaluation. The only significant exception is a small number of packages which leverage call-by-need for meta-programming.

2. Promises Are Made to Be Broken [7]
   This paper explores how to evolve the semantics of R towards strictness-by-default and laziness-on-demand by providing tooling for developers to migrate libraries without introducing errors. It reports on a dynamic analysis that infers strictness signatures for functions to capture intentional and accidental laziness with over 99% accuracy.

Overall, I present the following contributions:

1. A description of the design and implementation of laziness in R and a small-step operational semantics for a subset of the language. This is covered in chapter 2.

2. An open-source, carefully optimized, dynamic analysis pipeline, consisting of an instrumented R interpreter and data analysis scripts for analyzing the use of laziness in R programs. This is covered in chapter 3.

3. Empirical evaluation of 232,290 scripts exercising code from 16,707 R packages on the use of laziness by programmers, the strictness of R functions and their possible evaluation orders, and the life cycle of promises. This is covered in chapter 4.

4. Tools that automate the migration of the R ecosystem by inferring strictness signatures for functions and rewriting them to introduce strict semantics. This is covered in chapter 5.

5. An evaluation of the automated migration of R to strict semantics. The tools infer strictness with good accuracy; over 99% of the inferred signatures were correct when tested against clients of the libraries. This is covered in chapter 6.

The following chapters provide a detailed discussion of these contributions.

# 2 R AND CALL-BY-NEED

The R project is a tool for implementing sophisticated data analysis algorithms. This chapter provides a brief primer on the design of R, followed by a detailed description of the interface, implementation, and small-step operational semantics of the call-by-need semantics of R. At heart, R is a vectorized, dynamic, lazy, functional, and object-oriented programming language with a rather unusual combination of features [20], designed to be easy to learn by non-programmers and enable rapid development of new statistical methods. It was created in 1993 by Ihaka and Gentleman [15] as a successor to an earlier language for statistics named S [3].

**DATA TYPES** In R, most data types are vectorized; this includes integers, doubles, strings, logicals (booleans), complex numbers, and raw bytes. Vectors are constructed by the `c(...)` function: `c("hi","ho")` creates a vector of two strings. The language does not differentiate scalars from vectors, thus `1==c(1)`. R provides a heterogeneous vector type, `list`, which also serves as a record type since the elements can be named. Environments are used as mutable maps. Native code pointers are wrapped as `externalptr` values for interoperability with R.

**ATTRIBUTES** Custom types can be constructed by tagging values with user-defined attributes. For instance, one can attach the attribute `dim` to the value `x<-c(1,2,3,4)` by evaluating `attr(x,"dim")<-c(2,2)`. Once done, arithmetic functions will treat `x` as a `2x2` matrix. Another attribute is `class` which can be bound to a list of names. For instance, `class(x)<-"human"`, sets the class of `x` to `human`. Attributes are used for object-oriented dispatch. The "S3 object system" supports single dispatch on the class of the first argument of a function, whereas the "S4 object system" allows dispatch on all arguments. These names refer to the version of the S language which introduced them. Popular data types, such as data frames, also leverage attributes. A data frame is a `list` of vectors with `class` and `colname` attributes.

**FUNCTIONS** R has lexically-scoped higher-order functions. In R, every linguistic construct is desugared to a function call, even control flow statements, assignments, and bracketing. Furthermore, all functions can be redefined. This makes R both flexible and challenging to compile. A function definition can include default expressions for parameters that can refer to other parameters. The following snippet declares a function `f` which takes a variable number of arguments, whose parameters x and y, if missing, have default expressions y and 3*x, and which are only evaluated when needed. The function returns a closure.

```
f <- function(x=y,...,y=3*x) { function(z) x+y+z }
```

This function can be called with a single argument matching x, as in f(3), with named arguments, as in f(y=4,x=2), with a variable number of arguments, for example f(1,2,3,4,y=5), with multiple arguments captured by ..., or with no arguments at all, f(), which creates a cyclic dependency between x and y and errors out when the returned function is used. Some functions are written to behave differently in the presence of missing arguments. To this end the missing(x) built-in can be used to check if parameter x was provided at the call site or not, even if it was later substituted by a default value. A vararg parameter, written ..., accepts an arbitrary number of arguments, including missing arguments. A vararg can be materialized into a list with list(...). Most frequently varargs are forwarded to a called function. This enables the function to expose its callee's interface to the callers without listing the callee's parameters and their default values.

**REFLECTION** R supports meta-programming.
The substitute(exp,env) function yields the AST of the expression exp after performing substitutions defined by the bindings in env. The env parameter can be a list of bindings or an environment. It defaults to the current environment if not explicitly supplied. Consider the following call that substitutes 1 for a in the expression a + b and returns the expression 1 + b.

```
> substitute(a + b, list(a = 1))
1 + b
```

Environments, used as scopes, are first-class mutable maps with a reference to their lexical environment. Code can always access its local environment, but it is also possible to reflectively extract the environment of any function currently on the call stack [9]. Such reflective code is brittle. The following example shows that code that looks up its call stack is sensitive to small implementation changes, such as the addition of a call to an identity function. A call to as.environment(-1) returns the environment of the caller. Positive argument to as.environment refers to package environments instead of call stack. In the first call to f, both x and y yield the global environment, i.e., the environment from which f is called. In the second call, argument y returns the environment of f since y is evaluated inside the id function which is called from f. Thus, the evaluation of x and y will yield different results as the latter is executing within the id function.

```
f <- function(x, y) { x; y }
f(as.environment(-1), as.environment(-1))
id <- function(a) a
f <- function(x, y) { x; id(y) }
f(as.environment(-1), as.environment(-1))
```

R provides other functions for reflective stack access, such as `parent.frame`. However, this function is less brittle as it accesses frames relative to the promise's creation environment.

**EFFECTS** While R strives to be functional, it has imperative features such as assignment to local variables `<-`, assignment to variables in an enclosing scope `<<-`, and assignment in a programmatically chosen scope `assign()`. R supports non-local returns either through exceptions or by delayed evaluation of a `return` statement. Of course, there are all sorts of external effects and no monads.

Logically, function arguments are passed by value to facilitate equational reasoning. To avoid unnecessary copying of values, the implementation performs a copy-on-write of aliased values. As long as an aliased value is not modified, the variables refer to the same object. On write, a *copy* of the value is modified and the corresponding binding is updated. Consider the `swap` function which exchanges two elements in a vector and returns the modified vector:

```
> swap <- function(x, i, j) { t<-x[i]; x[i]<-x[j]; x[j]<-t; x }
> v <- c(1,2,3)
> swap(v,1,3)
```

The argument vector `v` is shared, as it is aliased by `x` in the function. Thus, when `swap` first writes to `x` at offset `i`, the vector is copied, leaving `v` unchanged. It is the copy that is returned by `swap`. Behind the scenes, a reference count is maintained for all objects. Aliasing a value increases the count. Any update of a value with a count larger than one triggers a copy. One motivation for this design was to allow users to write iterative code that updates vectors in place. A loop that updates all elements of an array will copy at most once.

In a nutshell, R is a lexically-scoped pass-by-value dynamic language with higher-order functions, first-class environments, and rich support for reflection.

## 2.1 CALL-BY-NEED

Since its inception, in 1993, R has had a call-by-need semantics. When a function is invoked its arguments are packaged up into *promises* which are evaluated on demand. The values obtained by evaluating those promises are memoized to avoid the need for recomputation. Thus the following definition when called with `a+b` and `d+d` evaluates `a+b` and does not evaluate `d+d`.

```
f <- function(x,y) x + x
```

With an estimated two million users world-wide [28], R is the most widely used lazy functional programming language in existence. It is fascinating to observe that R's laziness mostly remains secret. The majority of end-users are unaware of the semantics of the language they write code in. Anecdotally, this holds even for colleagues in the programming language community who use R casually.

Hudak [12] defined lazy evaluation as the implementation of normal-order reduction in which recomputation is avoided. He went on to enumerate two key benefits for programmers: (1) Sub-computations are only performed if they are needed for the final result; (2) Unbounded data structures include elements which are never materialized. Haskell is a language designed and implemented to support lazy evaluation, its compiler has optimization passes that remove some of the overhead of delayed evaluation, and its type system allows laziness to co-exist with side effects in an orderly manner.

R differs from Haskell in its approach to lazy evaluation. The differences are due in part to the nature of the language and in part to the goals of its designers. As R frequently calls into legacy C and Fortran libraries, performance dictates that the memory layout of R objects be consistent with the expectations of those libraries. For statistical and mathematical codes, this mostly means array of primitive types, integer and floating point, should be laid out contiguously using machine representations for primitives. Interoperability is thus the reason for builtin datatypes being strict, and consequently for mostly giving up on the second benefit of laziness right out of the gate. As for the first benefit, it goes unfulfilled because R tries to only be as lazy as it needs. In numerous places, design choices limit its laziness in favor of a more predictable order of execution; this is compounded by a defensive programming style adopted in many packages where arguments are evaluated to obtain errors early.

Given the above, one may wonder *why bother being lazy?* In particular, when this implies run-time costs that can be significant as R does not optimize laziness. Communications with the creators of R suggests call-by-need was added to support meta-programming and, in particular, user-defined control structures. While R was inspired by Scheme, the latter's macro-based approach to meta-programming was not adopted; macros were deemed too complex for users. Instead, R offers a combination of call-by-need and reflection. Call-by-need postpones evaluation so that reflection can inspect and modify expressions, either changing their binding environment or their code. This approach was deemed sufficiently expressive for the envisioned use-cases and, unlike macros, it was not limited to compile-time redefinition—an important consideration for an interactive environment.

The combination of side effects, frequent interaction with C, and absence of types has pushed R to be more eager than other lazy languages. Let's look at the design and implementation of laziness in R.

### 2.1.1 Interface

In R, arguments to a user-defined function are bundled into a thunk called a *promise*. Logically, a promise combines an expression's code, its environment, and its value. To access the value of a promise, one must *force* it. Forcing a promise triggers evaluation and the computed value is captured for future reference. The following snippet defines a function `f` that takes argument `x` and returns `x+x`. When called with an argument that has the side effect of printing to the console, the side effect is performed once as the second access to the promise is cached.

```
> f <- function(x) x+x
> f( {print("Hi!");2} )
"Hi!"
4
```

Promises associated to parameters' default values have access to all variables in scope, including other parameters. Promises cannot be forced recursively, that is an error. Promises are mostly encapsulated and hidden from user code. R only provides a small interface for operating on promises:

- **`delayedAssign(x,exp,eenv,aenv)`**: create a promise with body `exp` and binds it to variable `x` (where `x` is a symbol). Environment `eenv` is used to evaluate the promise, and `aenv` is used to perform the assignment.
- **`substitute(e,env)`**: substitutes variables in `e` with their values found in environment `env`, returns an `expression` (a parse tree).
- **`force(x)`**: forces the promise `x`. This replaces a common programming idiom, `x<-x`, which forces `x` by assigning it to itself.
- **`forceAndCall(n,f,...)`**: call `f` with the arguments specified in the varargs of which the first `n` are forced before the call.

While R does not provide built-in lazy data structures, they can be encoded. Figure 2.1 shows a lazy list that uses environments as structs;

environments have reference semantics. R provides syntactic sugar for looking up variables ($), functions for creating environment out of lists (`list2env`) and for capturing the current environment (`environment`). The singleton `nil` has a `tag` that is tested in the `empty` function. A new list is created by `cons`; it returns its environment in which `h` and `t` are bound to promises. The `head` and `tail` functions retrieve the contents of `h` and `t` respectively. This example also illustrates how promises can be returned from their creation environment, namely by being protected by an environment.

```
nil  <- list2env(list(tag="nil"))
empty <- function(l) l$tag=="nil"
cons <- function(h,t) environment()
head <- function(l) l$h
tail <- function(l) l$t
```

**Figure 2.1:** Lazy list in R

R evaluates promises aggressively. The sequencing operator `a;b` will evaluate both `a` and `b`, assignment `x<-a` evaluates `a`, and `return` also triggers evaluation. In addition, many core functions are strict. R has two kinds of functions that are treated specially:

- *Builtins*: There are 680 builtins, typically written in C, providing efficient implementations of numerical methods and other mathematical functions. The argument lists of builtins are evaluated eagerly.
- *Specials*: There are 46 specials used to implement core language features such as loops, conditionals, bracketing, etc. These functions take expressions (parse trees) which are evaluated in the calling environment or in a specially constructed environment.

Builtins and specials are exposed as functions to the surface language either directly or through wrappers which perform pre-processing of arguments before passing them to these functions.

I would be remiss if I did not mention context-sensitive lookup, one of the unusual features of R. When looking up a variable `x`, in head position of a function call, e.g., `x(...)`, R finds the first definition of `x` in a lexically enclosing environment, if `x` is bound to a closure, that closure is returned. Otherwise, lookup continues in the enclosing scope. A corollary of function lookup is that it forces promises encountered along the way.

### 2.1.2 Implementation

A promise has four slots: `exp`, `env`, `val` and `forced`. The `exp` slot contains a reference to the code of the promise, the `env` refers to environment in which the promise was originally created. The `val` slot holds the result of evaluating the `exp`. The `forced` flag is used to avoid recursion. When a promise is accessed, the `val` slot is inspected first. If it is not empty, that value is returned. Otherwise, `forced` is checked, and if it is set an exception is thrown. The `forced` flag is updated and the expression is evaluated in the specified environment. Once the evaluation finishes, the `val` slot is bound to the result, the `env` slot is cleared to allow the environment to be reclaimed, and the `forced` flag is unset. The implementation does little to optimize promises. In some cases, a promise can be created pre-forced with a value pre-assigned. The GNU R implementation recently added a bytecode compiler, this compiler eliminates promises when they contain a literal [29].

### 2.1.3 Semantics

This section describes a small-step operational semantics in the style of Wright and Felleisen [38] for a core R language with promises. My goal is to provide an easy to follow—the entire semantics fits on a page—but precise—unlike the above prose description—account of R's call-by-need semantics. I build upon the semantics of Core R [20], but omit vectors and out-of-scope assignments. Instead, I add delayed assignment, default values for arguments, `substitute` and `eval`. To support these features I add strings as a base type and the ability to capture the current environment.

$$\begin{array}{lcl}
\mathtt{e} & ::= & \mathtt{s} \mid \mathtt{x} \mid \mathtt{e}\,\#\,\mathtt{e} \mid \mathtt{x} \leftarrow \mathtt{e} \mid \mathtt{fun}(\mathtt{x}=\mathtt{e})\,\mathtt{e} \mid \mathtt{x}(\mathtt{e}) \mid \mathtt{x}() \mid \mathtt{env} \\
& \mid & \mathtt{subst}(\mathtt{x}) \mid \mathtt{eval}(\mathtt{e},\, \mathtt{e}) \mid \mathtt{delay}(\mathtt{x},\, \mathtt{e},\, \mathtt{e})
\end{array}$$

$$\begin{array}{lcl}
\mathbb{C} & ::= & [] \mid \mid \mathbb{C}\,\#\,\mathtt{e} \mid v\,\#\,\mathbb{C} \mid \mathtt{x} \leftarrow \mathbb{C} \mid \mathbb{C}(\mathtt{e}) \mid \mathbb{C}() \mid \mathtt{eval}(\mathbb{C},\, \mathtt{e}) \\
& \mid & \mathtt{eval}(v,\, \mathbb{C}) \mid \mathtt{delay}(\mathtt{x},\, \mathtt{e},\, \mathbb{C})
\end{array}$$

$$\begin{array}{lcll}
F & ::= & \epsilon \mid F[\mathtt{x} \mapsto l] & \textit{frames} \\
E & ::= & \epsilon \mid l \cdot E & \textit{environments} \\
S & ::= & \epsilon \mid \mathtt{e}\, E \cdot S & \textit{stacks} \\
H & ::= & \epsilon \mid H[l \mapsto v] \mid H[l \mapsto F] \mid H[l \mapsto (v,\, \mathtt{e},\, E,\, R)] & \textit{heap} \\
R & ::= & \uparrow \mid \downarrow & \textit{forced flag}
\end{array}$$

$$\frac{E = l \cdot E' \quad H(l) = F \quad F(\mathtt{x}) = v}{get(H,\, E,\, \mathtt{x}) = v}$$

$$\frac{E = l \cdot E' \quad H(l) = F \quad \mathtt{x} \notin dom(F)}{get(H,\, E,\, \mathtt{x}) = get(H,\, E',\, \mathtt{x})}$$

[Fun]

$$\frac{v = (\lambda\texttt{x}{=}\texttt{e}.\texttt{e}',\, E)}{\mathbb{C}[\texttt{fun}(\texttt{x}{=}\texttt{e})\,\texttt{e}']\,E \cdot S;\, H \;\rightarrow\; \mathbb{C}[v]\,E \cdot S;\, H'}$$

[Concat]

$$\frac{}{\mathbb{C}[\texttt{s}\,\#\,\texttt{s}']\,E \cdot S;\, H \;\rightarrow\; \mathbb{C}[\texttt{ss}']\,E \cdot S;\, H}$$

[Assign]

$$\frac{\begin{array}{c} E = l' \cdot E' \qquad H(l') = F \\ F' = F[\texttt{x} \mapsto v] \qquad H' = H[l' \mapsto F'] \end{array}}{\mathbb{C}[\texttt{x} \leftarrow v]\,E \cdot S;\, H \;\rightarrow\; \mathbb{C}[v]\,E \cdot S;\, H'}$$

[Delay]

$$\frac{\begin{array}{c} H(l) = l' \cdot E' \qquad H(l') = F \qquad \mathit{fresh}\ l'' \\ F' = F[\texttt{x} \mapsto \texttt{prom}(l'')] \qquad H' = H[l'' \mapsto (\bot,\, \texttt{e},\, E,\, \downarrow)][l' \mapsto F'] \end{array}}{\mathbb{C}[\texttt{delay}(\texttt{x},\, \texttt{e},\, \texttt{env}(l))]\,E \cdot S;\, H \;\rightarrow\; \mathbb{C}[\texttt{env}(l)]\,E \cdot S;\, H'}$$

[Env]

$$\frac{\mathit{fresh}\ l \qquad H' = H[l \mapsto E]}{\mathbb{C}[\texttt{env}]\,E \cdot S;\, H \;\rightarrow\; \mathbb{C}[\texttt{env}(l)]\,E \cdot S;\, H'}$$

[Subst]

$$\frac{\mathit{get}(H,\, E,\, \texttt{x}) = v \qquad v = (\_,\, \texttt{e},\, \_,\, \_) \qquad \mathit{string}(\texttt{e}) = \texttt{s}}{\mathbb{C}[\texttt{subst}(\texttt{x})]\,E \cdot S;\, H \;\rightarrow\; \mathbb{C}[\texttt{s}]\,E \cdot S;\, H}$$

[Eval]

$$\frac{\texttt{e} = \texttt{eval}(\texttt{s},\, \texttt{env}(l)) \qquad \mathit{parse}(\texttt{s}) = \texttt{e}' \qquad H(l) = E'}{\mathbb{C}[\texttt{e}]\,E \cdot S;\, H \;\rightarrow\; \texttt{e}'\,E' \cdot \mathbb{C}[\texttt{e}]\,E \cdot S;\, H}$$

[EvalRet]

$$\frac{\texttt{e} = \texttt{eval}(\texttt{s},\, \texttt{env}(l'))}{v\,E' \cdot \mathbb{C}[\texttt{e}]\,E \cdot S;\, H \;\rightarrow\; \mathbb{C}[v]\,E \cdot S;\, H}$$

[Invk1]

$$\frac{\begin{array}{c} v = (\lambda\texttt{x}{=}\texttt{e}'.\texttt{e}'',\, E') \qquad \mathit{fresh}\ l, l' \\ E'' = l \cdot E' \qquad H' = H[l \mapsto F] \\ F = [\texttt{x} \mapsto l'] \qquad H'' = H'[l' \mapsto (\bot,\, \texttt{e},\, E,\, \downarrow)] \end{array}}{\mathbb{C}[v(\texttt{e})]\,E \cdot S;\, H \;\rightarrow\; \texttt{e}''\,E'' \cdot \mathbb{C}[v(\texttt{e})]\,E \cdot S;\, H''}$$

[Invk0]

$$\frac{\begin{array}{c} v = (\lambda\texttt{x}{=}\texttt{e}'.\texttt{e}'',\, E') \qquad \mathit{fresh}\ l, l' \\ E'' = l \cdot E' \qquad H' = H[l \mapsto F] \\ F = [\texttt{x} \mapsto l'] \qquad H'' = H'[l' \mapsto (\bot,\, \texttt{e}',\, E'',\, \downarrow)] \end{array}}{\mathbb{C}[v()]\,E \cdot S;\, H \;\rightarrow\; \texttt{e}''\,E'' \cdot \mathbb{C}[v()]\,E \cdot S;\, H''}$$

[Ret1]

$$\frac{}{v\,E' \cdot \mathbb{C}[v'(\mathtt{e})]\,E \cdot S;\,H \;\rightarrow\; \mathbb{C}[v]\,E \cdot S;\,H}$$

[Ret0]

$$\frac{}{v\,E' \cdot \mathbb{C}[v'()]\,E \cdot S;\,H \;\rightarrow\; \mathbb{C}[v]\,E \cdot S;\,H}$$

[Lookup]

$$\frac{get(H,\,E,\,\mathtt{x}) = v \qquad v \neq \mathtt{prom}(l)}{\mathbb{C}[\mathtt{x}]\,E \cdot S;\,H \;\rightarrow\; \mathbb{C}[v]\,E \cdot S;\,H}$$

[Lookup2]

$$\frac{get(H,\,E,\,\mathtt{x}) = \mathtt{prom}(l)}{\mathbb{C}[\mathtt{x}]\,E \cdot S;\,H \;\rightarrow\; \mathtt{prom}(l)\,E \cdot \mathbb{C}[\mathtt{x}]\,E \cdot S;\,H}$$

[Force]

$$\frac{H(l) = (\perp,\,\mathtt{e},\,E',\,\downarrow) \qquad H' = H[l \mapsto (\perp,\,\mathtt{e},\,E',\,\uparrow)]}{\mathtt{prom}(l)\,E \cdot S;\,H \;\rightarrow\; \mathtt{e}\,E' \cdot \mathtt{prom}(l)\,E \cdot S;\,H'}$$

[ReadVal]

$$\frac{H(l) = (v,\,\mathtt{e},\,\epsilon,\,\downarrow)}{\mathtt{prom}(l)\,E \cdot S;\,H \;\rightarrow\; v\,E \cdot S;\,H}$$

[Memo]

$$\frac{v \neq \mathtt{prom}(l'') \qquad H(l) = (\perp,\,\mathtt{e},\,E',\,\uparrow) \qquad H' = H[l \mapsto (v,\,\mathtt{e},\,\epsilon,\,\downarrow)]}{v\,E' \cdot \mathtt{prom}(l)\,E \cdot S;\,H \;\rightarrow\; v\,E \cdot S;\,H'}$$

[RetProm]

$$\frac{v \neq \mathtt{prom}(l)}{v\,E' \cdot \mathbb{C}[\mathtt{x}]\,E \cdot S;\,H \;\rightarrow\; \mathbb{C}[v]\,E \cdot S;\,H'}$$

**Figure 2.2**: Syntax and Semantics

Figure 2.2 gives the syntax and semantics of my calculus. The surface syntax includes terms for strings, variables, string concatenation, assignment, function declaration, function invocation (one and zero argument functions), environment capture, substitution, eval, and delayed assignment. The syntax is extended with additional terms used during reduction where variables and expressions can be replaced by values ranged over by meta-variable $v$ which can be one of a string (s), a closure ($\lambda\mathtt{x}{=}\mathtt{e}.\mathtt{e},\,E$), an environment ($\mathtt{env}(l)$), or a promise ($\mathtt{prom}(l)$). Mutable values are heap allocated and ranged over by meta-variable $l$. I use the $\perp$ value to denote an invalid reference.

The reduction relation is of the form $S\ E \rightarrow S'; E'$ where the stack $S$ is a collection of expression-environment pairs (e $E$) and the heap maps variables to values. Frames are mutable and can be shared between closures, so they are stack allocated. Promises are quadruples ($v$, e, $E$, $R$) where $v$ is the cached result of evaluating the body e in environment $E$. $R$ is a status flag where $\downarrow$ indicates that the promise has started evaluating. Evaluation contexts $\mathbb{C}$ are deterministic. Following R, some builtin operations such as string concatenation are strict. But function calls are lazy in their argument, the expression in x (e) remains untouched by the context. I omit the definition of the functions *parse* and *string* which, respectively, turn strings into expressions and vice-versa. The expression *fresh* is used to obtain new heap references.

The semantics is given by the following rules. Rules Fun and Concat deal with creating a closure in the current environment and concatenating string values. Rule Assign will add a mapping from variable x to value $v$ in the current frame. As frames are allocated in the heap, this updates the heap. Rule Delay performs a delayed assignment, that is to say, an assignment that does not force the right-hand side. It takes a variable, an expression and an environment in which this expression will be evaluated. The rule creates a new unevaluated promise and binds it to x. Rule Env grabs the current environment and returns it as a value. Rule Subst looks up the variable given as argument, obtains the promise bound to it, extracts its body and deparses it into a string. Rule Eval takes a string and an environment, parses that string into an expression and schedules it for execution. Rule EvalRet takes the result of that evaluation and replaces the call to eval with it. Rule Invk1 and Invk0 handle user-defined function calls. Both rules expect $v$ to be a closure. They allocate a new promise for x. They differ on the body of that closure and the environment. Invk0 has no argument and will use the default expression e′ specified in the function declaration, it uses the environment where x is bound for evaluation. Rules Ret1 and Ret0 are the corresponding returns rules that replace the call with the computed value. Rules Lookup and Lookup1 are used to read variables from the current environment. If the result is a promise it is scheduled for execution by Lookup1. Rule Force will actually evaluate a promise that has been pushed on the stack, if that promise has not yet been evaluated. It sets the flag to avoid recursive evaluation. Rule ReadVal retrieves the value of an already evaluated promise. Rule Memo stores the result of evaluation in the promise and discard its environment. Finally, rule RetProm returns from evaluating a promise by replacing the variable looked up with the result.

## 2.2 RELATED WORK

Lazy functional programming languages have a rich history. The earliest lazy programming language was Algol 60 [2] which had a call-by-name evaluation strategy. This was followed by a series of purely functional lazy languages [1, 31, 32]. The motivations for the pursuit of laziness were modularity, referential transparency and the ability to work with infinite data structures [14]. These languages inspired the design of Haskell [13].

I compare the small-step operational semantics to the work of Bodin, Diaz, and Tanter [4]. The semantics makes no claims of being correct (there is no specification of R) or of being faithful to the language. The semantics is useful in as much it provides a readable account of delayed evaluation in R. Bodin's work is more ambitious, it aims to provide an executable semantics. The benefits of executable semantics is that they can be tested against an implementation, in this case the GNU R virtual machine. The semantics consists of 28,026 lines of Coq and 1,689 lines of ML. Validation is done through testing and visual comparison between the GNU R's C code and Coq code. Unfortunately, in the current state Bodin's specification is still far from complete. Out of 20,976 tests, only 6,370 pass. Inspection of the specification reveals that key functions for laziness such as `force`, `forceAndCall`, and `delayedAssign` are not implemented. Only a handful of the provided tests deal with lazy evaluation (they check that promises are evaluated only when forced). Furthermore, package loading and interaction with C code is not supported, thus packages from my corpus cannot be tested. I tried to match my semantics to theirs but their paper Bodin, Diaz, and Tanter [4] does not describe their treatment of laziness. Due to the size of the Coq codebase and lack of documentation, it is unclear how to align the two artifacts.

## 2.3 CONCLUSIONS

R is rather strict for a lazy language. This manifests itself in the definition of evaluation contexts. Intuitively, any position where $\mathbb{C}$ appears is evaluated strictly in left-to-right order. The key place where R differs from other lazy languages is that the right-hand side of assignments is strict. The semantics does not show data structures, in R they are all strict. Strictness also shows up in the `Ret1` and `Ret0` rules which force the evaluation of the return expression. Lastly, strictness is enforced in the `Lookup2` rule which does function lookup. If a promise is returned, it must be evaluated. Another property that the semantics ensures is that promises are stored in environments and, whenever they are accessed they are forced. The only way for a promise to outlive the frame that created it is to be returned as part of an environment or closure. Following R, it is possible to create a cycle in promise evaluation, the expression (`fun(x = x) x`)() when evaluated creates a closure and invokes it. The function's body triggers evaluation of the promise bound to `x`. Since no argument was provided, the default expression is evaluated causing a cycle. Like in R, this results in a stuck state in the semantics.

# 3 DYNAMIC ANALYZER

In this chapter, I discuss the first step towards understanding the real-world use of laziness in R. I present an open-source, carefully optimized, dynamic analysis pipeline, published in Goel and Vitek [8]. This pipeline consists of an instrumented R interpreter and data analysis scripts to analyze the use of laziness in R packages, which is described in detail in the next chapter. The artifact has been validated as *Functional* and *Reusable* and is available from:

`https://doi.org/10.5281/zenodo.3369573`

The analysis pipeline starts with scripts to download, extract and install open source R packages. Next, an instrumented R virtual machine generates events from program runs. This is followed by an analyzer that processes the execution traces to generate tabular data files in a custom binary format. Other scripts post-process the data, compute statistics, and generate graphs. The entire pipeline is managed by a Makefile that invokes an R script to extract runnable code snippets from installed packages and runs the other steps in parallel. Parallelization is achieved using GNU Parallel. Figure 3.1 shows the main stages of the pipeline. The figure provides corpus size, number of files generated, size of data and time taken by each step. The remainder of this chapter details the various stages.

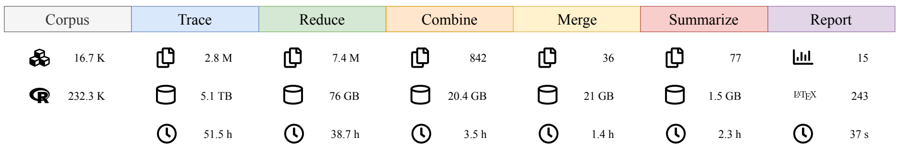

Figure 3.1: Tracing Pipeline

## 3.1 INSTRUMENTED R

The instrumented R Virtual Machine is based on GNU-R version 3.5.0. Its goal is to produce program execution traces with all the events required to answer the research questions. On the face of it, this is not a difficult task. And, in the end, I only had to add 1,886 lines of C code to expose an event data structure with fields to describe a variety of execution events that capture the internal interpreter state. The challenge was identifying where to insert those 1,886 lines in an interpreter whose code is 542,809 LOC written over twenty-five years by many developers and outside contributors. The system has grown in complexity with an eclectic mix of ad hoc features designed to support growing user requirements. For instance, the code to manage environments and variable bindings in `main/envir.c` is over 2,864 LOC with 131 functions with a large number of identical code fragments for managing these data structures duplicated in various files. I succeeded by a lengthy trial and error process. The events recorded by the instrumented virtual machine are:

- **`Call`, `Return`**: at each function call, records function's type, arguments, environment and return value.
- **`S3Dispatch`**: at each S3 dispatch, records method name and first argument.
- **`S4Dispatch`**: at each S4 dispatch, records method name, definition, environment and dispatch arguments.
- **`Eval`**: at each `eval`, records the evaluated expression and its environment.
- **`Substitute`**: at each `substitute`, records the arguments.
- **`ArgListEnter`, `ArgListExit`**: records expressions that are being promised.
- **`CtxtEnter`, `CtxtExit`**: records the address of a stack frame.
- **`CtxtJmp`**: records the popped stack frames at each non-local return.
- **`PromEnter`, `PromExit`**: records evaluated promise and when evaluation terminates.
- **`PromRead`**: when a promise's value is read, records the promise and its value.
- **`PromSubst`**: generated when a promise's expression is read.
- **`FunLoadStart`, `FunLoadEnd`**: generated when looking up a function.
- **`GC`**: generated at each garbage collection cycle.
- **`Alloc`**: generated when memory is allocated.
- **`Free`**: generated memory is reclaimed.
- **`Deserialize`**: generated when an object is deserialized.
- **`VarDef`**: when a variable is defined, record the symbol, value and environment.
- **`VarWrite`**: when a variable is updated, record a the symbol, value and environment.
- **`VarRem`**: when a variable is deleted, records the symbol and environment.
- **`VarRead`**: when a variable is read, record the symbol, value and environment.

Events can be disabled to ignore implementation details of the virtual machine and also to avoid recursion. R objects captured in events are protected from the garbage collector to prevent them from being reclaimed during analysis. Table 3.1 shows the number of times events are triggered.

Table 3.1: Events

| | | | | | |
|---|---|---|---|---|---|
| **Alloc** | 2.4 T | **CtxtEnter** | 143 B | **VarRem** | 8.4 B |
| **Free** | 2.3 T | **CtxtExit** | 143 B | **CtxtJmp** | 6 B |
| **Eval** | 1.7 T | **VarWrite** | 140 B | **S3Dispatch** | 2.6 B |
| **VarRead** | 1.6 T | **ArgListEnter** | 111 B | **PromSubst** | 1.9 B |
| **Call** | 831 B | **ArgListExit** | 111 B | **Substitute** | 1.7 B |
| **Return** | 807.4 B | **Deserialize** | 109 B | **S4Dispatch** | 936 M |
| **VarDef** | 365 B | **PromRead** | 102 B | **GC** | 4.8 M |
| **PromEnter** | 223 B | **FunLoadStart** | 78 B | | |
| **PromExit** | 223 B | **FunLoadEnd** | 78 B | | |

## 3.2 TRACER

The tracer is a small R package (73 LOC) that calls into a larger C++ library (6,080 LOC). It is loaded in the instrumented R virtual machine and, during program execution, it maintains objects that model various aspects of the program such as functions, calls, promises, variables, environments, stacks and stack frames. As events are generated, the tracer updates its model of the state. The tracer is able to process 803.1 K events per second on the benchmark machine.

Some design decisions allowed the tracer to scale. Firstly, copying model objects is avoided as much as possible. They are created by a singleton factory that caches them in a global table. This optimization pays off as model objects are large and costly to copy. But keeping these objects alive too long will increase footprint and hinder any attempt at running multiple tracers on the same machine in parallel. To reduce tracing footprint, the R garbage collector was modified so that model objects can be deallocated as soon as the R object they represent is freed. One slightly surprising design choice is to link all model objects together. This pays off when an event triggers a cascade of changes to model objects. This comes at a price of course, as lists of model objects are circular it is necessary to perform reference counting to reclaim them. One last implementation trick is the use of a shadow stack that mirrors the stack maintained by the R virtual machine. The shadow stack is used to look up data after a `longjump`.

The tracer generates large amounts of data. My first prototype used Sqlite to store the generated data. However, I found the approach limiting. During development I kept running into errors because the database schema and the tracer were out of sync. Due to the iterative nature of data analysis, I was modifying the schema frequently and this became a pain point. Furthermore, the database was normalized, thus requiring join operations in the analysis. At scale, these joins were expensive, causing database operations to run for days. Lastly, insertions ended being a bottleneck; I could trace fewer than 1000 packages per day and filled up a 1 TB disk. In the end, I implemented a custom format. As the event stream has substantial amounts of redundancy, I applied streaming compression on the fly. Compression yields an average 10x saving in space and 12x improvement in loading time.

## 3.3 EXECUTION

For each package to be analyzed the infrastructure extracts executable code from that package. Extraction invokes an R API which locates executable code snippets in the documentation and RMarkdown files. Files in the test directory are copied as is. All snippets and tests set up the tracer and initialize it with paths to input and output data before execution. For each package, the tracer generates 12 data files and 4 status files. These files denote the different possible states of the tracing. They allow the infrastructure to discard data from failing programs. The R scripts responsible for generating traces do extensive logging of intermediate steps for debugging purposes.

## 3.4 PROCESSING

This part of the pipeline analyzes the raw data. It is 4K lines of R code. Scale was the major challenge. I faced difficulties both due to execution time and data size. In the 232,290 programs that were traced, the tracer observed 1.7 T expression evaluations, 831 B calls to 698.4 K functions and 270.9 B promises. The raw data generated by the tracer is 5.1 TB but the reduced data is just 76 GB. In hindsight, it appears that incorporating analysis in the tracer, i.e., pre-summarizing data in C++ would have been beneficial. However, this would require knowing ahead of time all the analysis that I would perform. Part of the challenge lies in the fact that the event of interest and attached analyses were not fixed ahead of time. Pre-summarized data makes it harder to pose new questions. It also makes it harder to detect bugs because summarized data resists correlation with actual code.

The pipeline steps are detailed next.

- *Prescan*: Scan the raw data directory and output a list of all the subdirectories that contain raw data. There is a directory per package and multiple files in each directory.

- *Reduce*: Given a list of directories, this step uses GNU Parallel to partially summarize the raw data. This is the most expensive step in terms of size and speed. Since the data files are large, I limit the degree of parallelism drastically to avoid running out of memory.

- *Scan*: Create a list of all the files successfully reduced.
- *Combine*: Combine information from all the programs into a single data table per analysis question.
- *Summarize*: Compute summaries of the merged data for: (1) event frequency, (2) object frequency, (3) functions with their definitions, (4) argument information, (5) escaped arguments, (6) information about parameters, (7) information about promises.
- *Report*: Generate graphs and tables from an RMarkdown notebook as well as LaTeX macros for inclusion in the paper.

## 3.5 RELATED WORK

Morandat et al. [20] implemented a tool called TraceR for profiling R programs. The architecture of TraceR was similar to that of the pipeline presented here, but it did not target large scale data collection and has gone unmaintained for several years. My infrastructure is less invasive than TraceR.

## 3.6 CONCLUSIONS

This chapter discussed the design and implementation of a dynamic analyzer for R. While the idea of instrumenting the interpreter to collect runtime information is simple at heart, the complex design of R and the scale of data collection introduces a surprising amount of complexity. The following chapter will employ this analyzer for performing a large-scale data-driven study of how generations of programmers have put laziness to use in their code.

# 4 USE OF CALL-BY-NEED

In this chapter, I present an empirical evaluation of 16,707 R packages on the use of call-by-need by programmers, the strictness of functions and their possible evaluation orders, and the life cycle of promises. One might be tempted to question the choice of using dynamic analysis for this evaluation. Unfortunately, alternative approaches suffer from serious drawbacks. An obvious alternative is to modify the R VM to perform strict evaluation and observe how much code will break. However, built-in functions (e. g., conditionals and exception handling mechanism) require unevaluated terms, so all scripts will break. Another alternative is to annotate the arguments in the entire code base manually, but that is cumbersome, error-prone, and unscalable. The final alternative, static analysis, would fail to yield meaningful insights because of R's dynamic nature. However, dynamic analysis is not without shortcomings. It is limited to behaviors that it observes; arguments whose evaluation is not observed dynamically may be evaluated along code paths that were not exercised. Hence low code coverage will affect the quality of results. However, it is encouraging to note that Krikava and Vitek [18] reports line-level code coverage of over 60% for a similar corpus. To mitigate this threat to validity, I will also perform a qualitative analysis of a sample of the corpus.

For this evaluation, I have analyzed 16,707 packages hosted in the CRAN and Bioconductor software repositories, and observed the creation of 270.9 B promises. The results were obtained with version 3.5.0 of GNU R and packages retrieved on August 1st, 2019. The software and data was validated as Functional and Reusable and is available in open source from:

https://doi.org/10.5281/zenodo.3369573

The data suggests that there is little supporting evidence to assert that programmers use laziness to avoid unnecessary computation or to operate over infinite data structures. For the most part R code appears to have been written without reliance on, and in many cases even knowledge of, delayed argument evaluation. The only significant exception is a small number of packages which leverage call-by-need for meta-programming.

## 4.1 CORPUS

The corpus used in this study was assembled on August 1st, 2019 from the two main code repositories, namely the *Comprehensive R Archive Network* (CRAN) [19] and *Bioconductor* [11]. Both are curated repositories; to be admitted packages must conform to some well-formedness rules. In particular, they must contain use-cases and tests along with the data needed to run them. I believe this corpus is representative of sophisticated uses of the R language. Anecdotal evidence suggests that the majority of R code written is made up of small scripts, straight-line sequences of package calls, that read data, apply some models to it and then visualize the results. Most end-users neither define functions nor write loops, their code is simple. Without a source of end-user code it is not possible to validate this hypothesis, but if true then the corpus is representative of the interesting R code. R is also used in many industrial settings that do not publish their code to open source repositories. I have no information on those use-cases.

My snapshot of CRAN includes 14,762 packages, and for Bioconductor, 3,087 packages. Bioconductor is also used to store data, 1,741 packages contain software, 1,319 contain data and 27 are so-called workflows. Starting with 17,849 software packages, the scripts downloaded and successfully installed 17,479 of them. The reasons some packages did not install were varied, they included missing dependencies and compiler errors.

These installation errors may be fixable but automating those fixes would be hard. I chose to discard the packages that could not be installed. Out of the installed packages, I was able to successfully record execution traces for 16,707 packages. Some packages did not trace owing to run-time failures. Again, I discarded the failing packages on the grounds of having a sufficient number of running ones

For each package, the scripts gathered runnable code from three different sources: test cases, examples and vignettes. Test cases are typically unit tests written to exercise individual functions, while examples and vignettes demonstrate the expected end-user usage of the particular package. These use-cases may load other packages and access data shipped with the package or obtained from the internet.

**Table 4.1:** Corpus

| | **Tests** | **Examples** | **Vignettes** | |
|---|---|---|---|---|
| Scripts | 44.1 K | 220 K | 9.8 K | *Install* |
| | 23.3 K | 202 K | 6.6 K | *Trace* |
| LOC | 2.7 M | 1.6 M | 614 K | *Install* |
| | 1.3 M | 1.6 M | 327 K | *Trace* |

Table 4.1 gives the number of scripts of each kind that could be installed and the number of scripts that were successfully traced. In terms of lines of code, I exercised 25.6 M lines of R and 10.4 M lines of C. The total size of the database after analysis is 5.2 TB.

I observed 831 B calls to 698.4 K functions. 26.7% of the calls are made to 157 builtin functions, and 60.3% are made to 33 special functions. The remainder are calls to R functions. Of these, 2% are calls to 34.1 K different S3 methods and 0.8% are calls to 60.2 K S4 methods. There are 415.3 K plain R functions.

The number of times `delayedAssign` is called is 82 M, `force` is called 101.8 M times, `forceAndCall` is invoked 1.3 B times and `substitute` 1.7 B times. Functions exit due to lonjumps 23.7 B times (this marks explicit use of the `return` function).

Figure 4.1 shows how many functions were exercised in each test package. More than 9 K (59.2%) of the packages had over 10 functions called. On the other hand 2.7 K of the packages had a single function invoked (7.0%). These may either be small packages, or, more likely, the provided tests have low coverage.

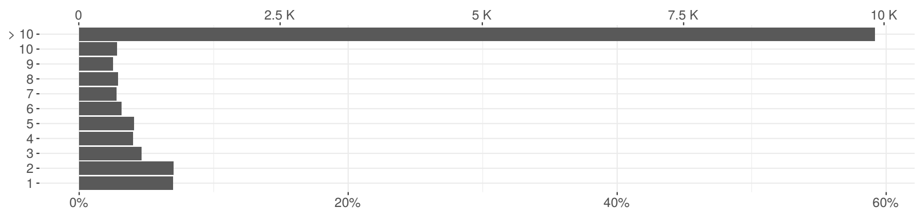

**Figure 4.1**: Functions per package

Figure 4.2 shows how many times functions were called. 45.0% of the exercised functions were called more than ten times and 18.9% of the functions were called only once. Clearly functions that are called only once may lack coverage.

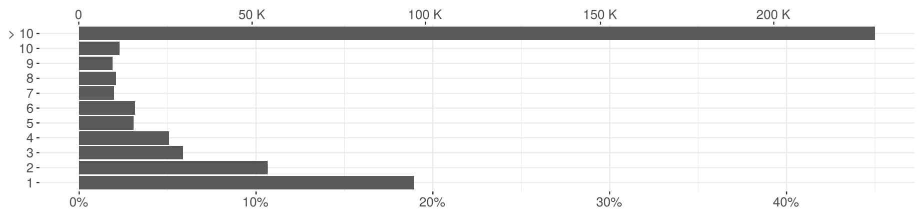

**Figure 4.2**: Function calls

Figure 4.3 shows the number of parameters per function; 43.7% have only one, while 9.8% have more than five. The function `meta::forest.meta` has the most with 199 parameters. There are 2.1 M distinct parameters.

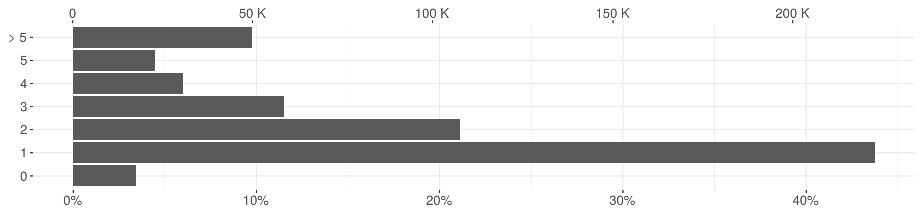

**Figure 4.3:** Formal Parameters

Of the 261 B arguments that were passed at run-time, 25.1% were default arguments, 72.1% were non-default arguments and the remaining 2.8% were missing arguments.

## 4.2 ANALYZING LAZINESS USAGE PATTERNS

This section presents the results of the empirical study of call-by-need in the R language.

### 4.2.1 Life Cycle of Promises

The first research question I address is how promises are created and used in the wild.

**RQ1**: *What is the life cycle of R promises in the corpus?*

In my corpus, and likely, all R programs, promises are the most frequently allocated object. I observed the creation of 270.9 B promises. For context, Figure 4.4 shows the distribution of application-level objects; most are vector of characters, logical, integers, and doubles, in that order. Lists are often used in package code and internal functions. Raw values hold uninterpreted byte strings. Closures represent functions and environments map names to value, symbols are language-level names and S4 are instance of classes. Environments are frequently observed because one is created for each function call, but they are also, albeit rarely, used as hashmaps in user code.

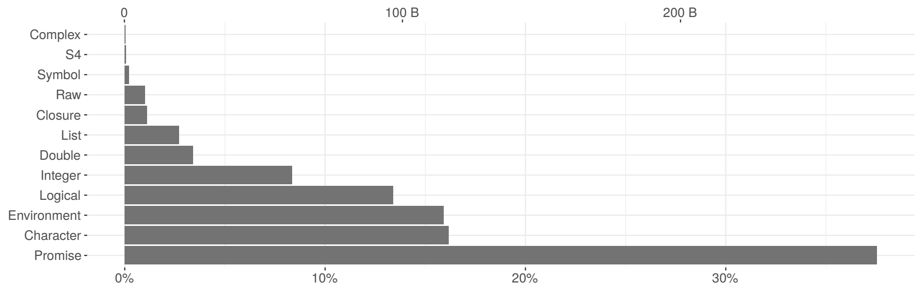

**Figure 4.4:** Object counts

**WHERE ARE PROMISES CREATED?** Argument lists account for 94.3% of promises. The remainder are used for lazy loading of functions, are created by calls to `delayedAssign`, or in internal functions of the R virtual machine. One could expect that there would be more promises than values since every operation in R is a function call. This is not the case. Some values are composite of multiple simpler elements (e.g. data frames) and these are wrapped in a single promise. Values can be returned without being bound to a promise. Lastly, calls to internal (builtin and special) functions do not pack arguments in promises.

**WHAT DO PROMISES YIELD?** Figure 4.5 shows the contents of promises that were forced. The most common types are character vector, logical vector, environment, closure, integer vector, double vector, null and list, in that order. The presence of null values is explained by the fact that many default parameter values are set to null and these default values are promised. S4 objects are rarely used in R programs outside of Bioconductor packages. Symbol, complex vector and raw vector are also quite rare in practice.

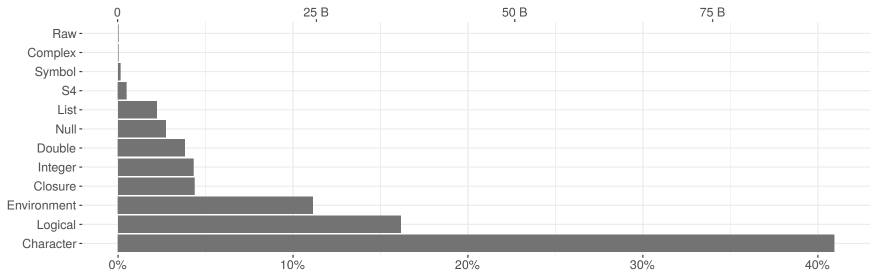

**Figure 4.5:** Promise results

Figure 4.6 shows the content of the expression slot of promises, i.e. their code. Only 17.3% of promises contain a function call, e.g. `1+2` or `f(z)`. The majority contain a single symbol to be looked up in the promise's defining environment, e.g. `x`. That symbol may be bound to another promise, in which case forcing the promise will be recursive. Promises can also hold inlined scalar constants, such as a single double `1.1`.

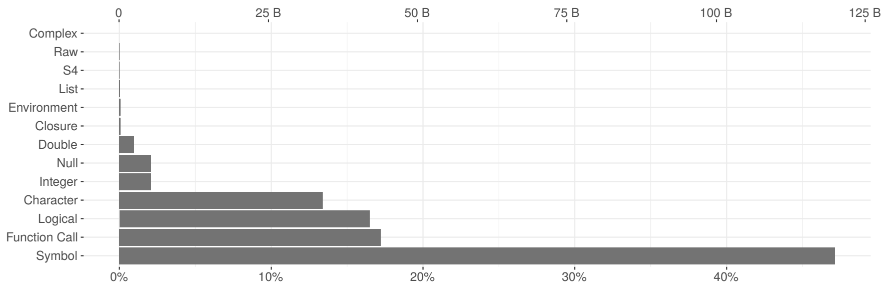

**Figure 4.6:** Promise expressions

**HOW OFTEN ARE PROMISES ACCESSED?** 87.3% of argument promises are forced; the remaining went unused. Unused arguments are not unusual, some functions have over twenty parameters, and many of these are only needed in special circumstances. Figure 4.7 shows, for each individual promise, the number of times its value was read. Most promises are used once (forced), 9.6% are accessed twice, and 3% are accessed three times.

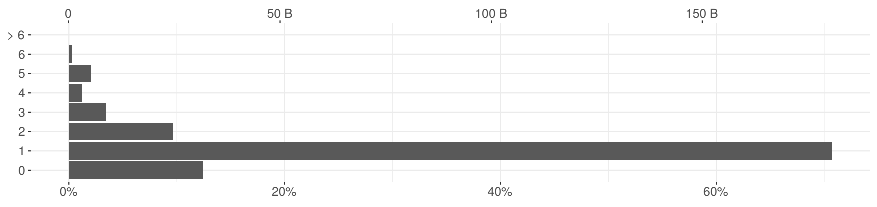

**Figure 4.7:** Reads

**HOW FAR ARE PROMISES FORCED FROM THEIR CREATION?** Promises can be passed from one function to the next, traveling down the call stack. Regardless of the distance from the frame that created them, promises evaluate in their creation environment. But the farther from creation, the harder it is for a compiler to optimize them. Figure 4.8 shows the distance between promise creation and forcing. 79.7% promises are evaluated in the callee, 17.1% promises are forced two level down, and the remaining 3.2% are evaluated deeper.

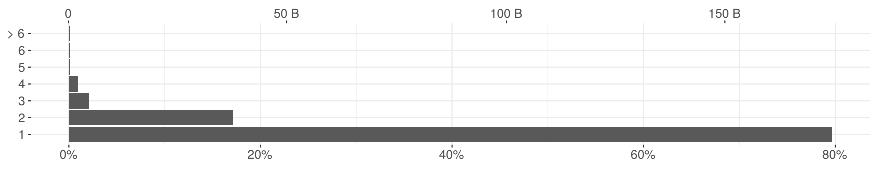

**Figure 4.8:** Force depth

**HOW LONG DO PROMISES LIVE?** Promises are short-lived, over 99.5% do not survive one garbage collection cycle. This confirms the folklore that most objects die young. It also means that promises are exerting pressure on the memory subsystem of the virtual machine. Only 0.5% of promises survive multiple cycles. Of those, 76.7% are non-argument promises and 0.08% are escaped arguments. As mentioned earlier, non-argument promises are created explicitly through `delayedAssign` and implicitly through lazy-loading of package code; both are expected to be long-lived. Escaped promises are promises that outlive their defining function. They may be long-lived as well. Of the long-lived promises, 23.2% are argument promises; their longevity is likely due to long running functions.

**Table 4.2:** Promise life cycle

| | | | | | |
|---|---|---|---|---|---|
| F | 70.5% | EF | 70.7% | A | 67.5% |
| – | 11.9% | FRER | 5.4% | – | 16.5% |
| FR | 9.5% | FER | 5% | AR | 11.2% |
| FRR | 3% | FRRER | 4% | AA | 2% |
| FRRRR | 2% | EFR | 3% | F | 2% |
| FRRR | 1% | FRRERR | 1% | S | 0.2% |
| M | 0.5% | EFRR | 1% | | |
| a. Argument | | b. Escaped | | c. Non-Argument | |

**WHAT ARE PROMISE LIFE CYCLES?** If we characterize the life of a promise by the events that affect it, promise life cycles can be summarized by sequences of events. Ignoring creation and reclamation, the events of interest are Force, Read, Meta-program, Escape, Assign, and deSerialize. Note that forcing a promise is an implicit read. I observed 28.1 K unique life cycles. Table 4.2 shows the most frequent sequences for (a) argument promises (19.2 K unique sequences), (b) escaped argument promises (7.3 K sequences) and (c) non-argument promises (6.2 K sequences). For argument promises the two most common sequence are `F` (a promise that is forced) and the empty sequence (unused promise). The next sequences are forces followed by a growing number of reads. Meta-programming occurs only infrequently. For escaped promises the most frequent sequence is `EF`, the promise escapes and is forced later. The second most frequent sequence is `FRER`, the promise is forced, read, escapes and is read again. Lastly, non-argument promises are most often created and assigned a value in the C code. About 0.2% of these promises are obtained from deserialization (`S`) owing to lazy-loading of packages.

**DOES CONTEXT SENSITIVE LOOKUP FORCE PROMISES?** Looking up a function name such as `f()` may force a promise if that name is bound to one in the environment. If the promise yields a closure, that closure will be invoked, otherwise lookup continues. This allows "harmless" shadowing of function names as seen in Figure 4.9 where `addToGList.grob` from package `grid` defines a parameter `gList` that shadows a function of the same name defined by the same package.

```
addToGList.grob <- function(x, gList)
   if (is.null(gList)) gList(x) else { gList[[length(gList)+1L
       ]]<-x; return(gList) }
```

**Figure 4.9:** Shadowing

I observed 651.7 K function lookups (out of a total of 831 B lookups) which caused a promise to be forced. Out of those, 86.3% yielded a closure. The 13.7% that did not yield a closure are cases where a function name was shadowed. Table 4.3 shows the 30 most commonly shadowed function names and the number of functions in which shadowing happens. Many of those names correspond to common R functions such as `c`, `names`, `print` and `max`.

Table 4.3: Context sensitive lookup

| | | | | | |
|---|---|---|---|---|---|
| `plot` | 971 | `round` | 130 | `max` | 67 |
| `log` | 821 | `title` | 129 | `format` | 66 |
| `c` | 461 | `order` | 83 | `sort` | 60 |
| `legend` | 334 | `drop` | 82 | `nrow` | 57 |
| `file` | 221 | `which` | 82 | `list` | 53 |
| `length` | 209 | `grid` | 76 | `rug` | 49 |
| `formula` | 204 | `class` | 74 | `matrix` | 48 |
| `scale` | 188 | `print` | 74 | `clean` | 47 |
| `t` | 184 | `ncol` | 71 | `start` | 45 |
| `names` | 182 | `dim` | 69 | `unique` | 43 |

**DO PROMISES FORCE EACH OTHER?** A parameter list can include parameters with default values that refer to each other. This is a semantic quirk that can lead to promises forcing one another. Consider Figure 4.10 and function `sample.int` from the `base` package. Parameter `useH` has a default value that depends on the other four arguments and `s` depends on `n`. Function `rmslash` has a recursive dependency between `center` and `Scatter`. The function expects at least one of those to be provided at each call site, if this is not the case an error will be reported.

```
sample.int <- function(n,s=n,r=F,p=NULL,useH=(!r && is.null(p)
    && s<=n/2 && n>1e+07))
   if (useH) .Internal(sample2(n,s)) else .Internal(sample(
       n,s,r,p))

rmslash <- function(center=rep(0,nrow(Scatter)), Scatter=diag(
    length(center))) {
   if (length(center) != nrow(Scatter)) stop("<error-message>")
...
```

Figure 4.10: Argument lists

I found 4.8 K default value expressions in 3.9 K functions. At run-time, there were 336.9 M default expressions forcing other parameters. This is 0.2% of the forced promises.

**DOES METHOD DISPATCH FORCE PROMISES?** The two widely used object systems, S3 and S4, have an impact on promises. In order to find which method to invoke, one or more arguments must be forced. Overall, only 1% of promises participate in method dispatch. 671.5 M promises are forced due to S3 dispatch and 536.2 M are forced due to S4 dispatch. As these numbers are small, I ignore dispatch for the remainder of the study.

**HOW OFTEN DOES NON-LOCAL RETURN HAPPEN?** The `return` function pops the call stack until it arrives at the frame where it originated from. A call such as `f(return(1))` will, when `f`'s argument is forced, return from `f` and its caller. So, when a promise containing `return` is forced, the current function stops executing and the stack is unwound; this is a non-local return. Only 297.4 M arguments to 16 functions performed non-local returns. One of the most common causes is the `base::tryCatch` function. The function sequentially attaches handlers specified in its vararg list and executes `exp` by wrapping it in a call to `return`. The return causes the control flow to exit `doTryCatch` and `tryCatch`.

```
tryCatch <- function(exp, ..., fin) {
...
   doTryCatch <- function(e, nm, penv, handler) {
      .Internal(.addCondHands(nm,handler), penv, environment(),
          F))
      e
   }
   value <- doTryCatch(return(exp), nm, penv, handler)
...
```

**TAKEWAYS.** Promises dominate the memory profile of R programs. They are short lived, 80% are evaluated in the called function and over 99% do not survive a single GC cycle. The vast majority of promises contain a value or a variable. Only 17.3% contain code that needs to be evaluated. Of those expression-carrying promises 80.7% are unused, 7.0% are evaluated in the called function and 12.3% are evaluated down the call-stack or meta-programmed. Overall most promises lead a rather mundane life that one would hope a compiler could optimize out of existence.

### 4.2.2 Strictness

My second research question concerns strictness. A function is said to be *strict* if it evaluates all of its arguments in a single pre-ordained order (*e.g.*, left to right).

**RQ2**: *What proportion of R functions are strict?*

To answer this question I start with individual parameters. I only consider plain R functions that are called more than once, have at least one parameter and have not abruptly stopped executing owing to non-local returns. There 2.1 M distinct parameters to such functions. For a given parameter and a given function, I aggregate all calls and all uses of that parameter into three categories: parameters that are *Always* evaluated, parameters that are *Never* evaluated, and *Sometimes* evaluated. Figure 4.11 summarizes this analysis. 87.6% of parameters are always evaluated, 6.0% parameters are evaluated in some calls and not others, 6.4% parameters are never used.

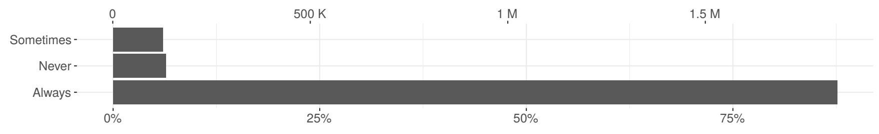

**Figure 4.11:** Parameter strictness

A function is strict if its parameters are always evaluated in the same order. Most functions, 93.6% to be precise, have a single order of evaluation. This means they are candidate for being strict. Functions with multiple orders of evaluation for their arguments are summarized in Figure 4.12. About 4.7% of the functions have two force orders, and very few functions have more.

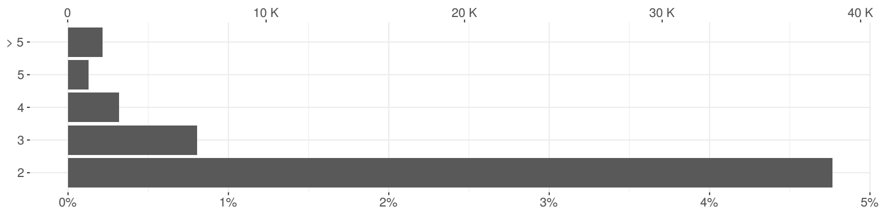

**Figure 4.12:** Function force orders

Based on the above data, out of a total of 388.3 K functions, 83.7% are strict. Figure 4.13 gives a histogram of function strictness ratios per package. The majority of packages contain only strict functions. The packages that are less than 75% strict account for only 2.6 K packages (16.9% of all packages) and 81.7 K functions (21.0% of all functions).

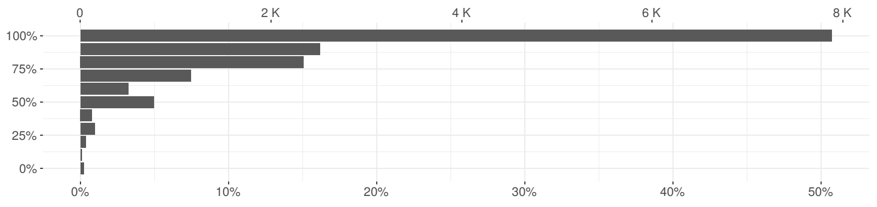

**Figure 4.13:** Strictness per package (x-axis=packages; y-axis=strict function ratio)

One could consider relaxing strictness to allow multiple orders of evaluation if those orders of evaluation could be shown to be semantically equivalent. Some features of R may help. First, all vectors have a copy-on-write semantics, thus many side effects are hidden from view. Moreover, as Figure 4.14 shows only 25% of Sometimes promises perform any computation. In my experience most of those computations are side effect free.

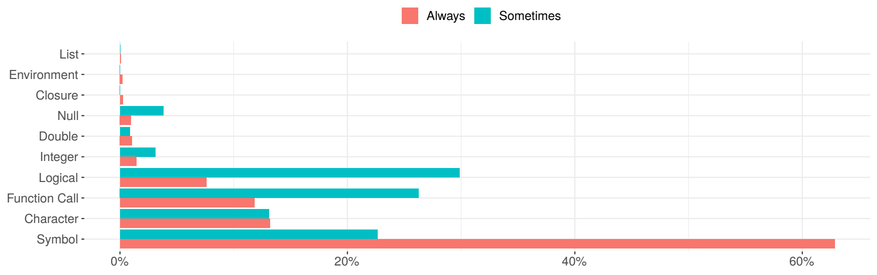

**Figure 4.14:** Promise expressions

I performed an additional analysis to get an upper bound on the side effects performed during promise evaluation. I considered only variable reads and writes, external side effects such as filesystem operations were not taken into account. A meager 16.5 M promises (out of 270.9 B) perform any side-effecting computation. There are several cases to consider, the simplest when the promise performs a side effect to its local environment. For example, consider the `stats::power` function:

```
power <- function() {
   linkinv <- function(eta) pmax(eta^(1/lambda),
       .Machine$double.eps)
   mu <- linkinv(eta <- eta + offset)
...
```

The call to `linkinv` takes, as argument, an expression that performs a side effect to the local variable `eta`. This could have been avoided by moving the assignment above the call to `linkinv` but the programmer likely wanted to save one line. This kind of local side effect can affect other promises coming from the same environment (which is not the case here). Of the side-effecting promises, 41.2% are local. Non-local effects can be performed, e.g., using the `<<-` operator to assign to a variable in the lexically enclosing environment. The following is snippet from a test script in `cliapp` package. The argument to `capt0` modifies `id` using `<<-`.

```
test_that("auto_closing", {
   id <- ""
   f <- function() capt0(id <<- cli_par(class = "xx"))
   capt0(f())
...
```

Out of the side-effecting promises, 0.5% affect a parent environment. Finally, 58.3% promises perform side effects in other environments. The `methods::callNextMethod` function is among the most common sources of mutation of other environments.

```
callNextMethod <- function (...) {
...
   callEnv <- parent.frame(1)
   assign(".nextMethod", nextMethod, envir = callEnv)
...
```

I further count how many of those side effects are performed directly by assignments occurring textually in the promise. Of the 6.8 M promises performing local side effects, 98% perform side effects directly. Of the 9.6 M promises performing side effects in other environments, 0.1% perform side effects directly. Of the 80.2 K promises performing side effects in the lexical parent environment, 0.08% perform side effects directly.

**QUALITATIVE ANALYSIS** I inspected 100 randomly selected functions that the dynamic analysis marked as strict. Out of those, 82 were indeed strict. The remaining 18 were not, but I did not observe their laziness. The majority (16 out of 18) of incorrectly labeled functions were not using all of their parameters, using them along some execution paths or returning early. I found a single function that was lazy because it called another function that was itself lazy in that particular argument and one function for which an argument escaped. The functions that do not evaluate all parameters could be cases where computational effort is saved if the arguments passed are complex expressions. I also found occurrences of explicit argument forcing. Programmers write code such as `x<-x` or `force(x)` to ensure that argument are values. An example of such code is the `scales::viridis_pal` function; it returns a closure, but forces all of its arguments to avoid capturing their environments:

```
function(alpha=1, begin=0, end=1, dir=1) {
   force_all(alpha,begin,end,dir)
   function(n) viridis(n,alpha,begin,end,dir)
}
```

The authors of higher-order functions such as `apply` or `reduce` often enforce strictness after receiving bug reports from end-users around unwanted lazy evaluation and variable capture interactions. The `forceAndCall` function has been introduced to mitigate this issue by forcing the arguments of a function prior to calling it. Looking for uses of `forceAndCall` revealed additional packages that enforce strictness for higher-order functions. The function is invoked 1.3 B times. The `force` function is also widely used to enforce strictness; it is called 101.8 M times and used in 60% of the packages I inspected. In summary, manual inspection suggests that I overestimate strictness. Improving precision of the analysis would require increasing code coverage. I also found numerous occasions where programmers require strictness to control side effects.

### 4.2.3 Meta-programming

The next research question pertains to the use of call-by-need to enable meta-programming.

**RQ3**: *How frequently are promises used for meta-programming?*

For the purposes of this discussion I define meta-programming as the manipulation of code through calls to `substitute` which lets programmers extract an abstract syntax tree from the body of a promise, modify it, and evaluate with `eval`. I observed 1.7 B calls (2% of all calls) to `substitute`. Figure 4.15 shows the number of promises that were meta-programmed. The graph has four categories: promises that were created but never used, promises that were meta-programmed, promises that were both meta-programmed and accessed, and lastly, promises that were only accessed. The data shows that 0.5% of promises were used purely for meta-programming purposes, while 0.2% were both forced to obtain a value and used for meta-programming.

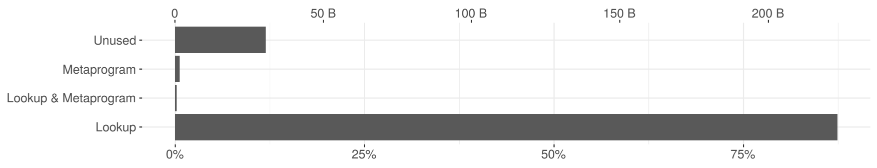

**Figure 4.15**: Meta-programmed promises

Meta-programming is widespread with 2 K (11.9%) packages using it. One feature of `substitute` is that programmers can specify the environment in which to resolve names occurring in its argument. It is also possible to access an environment up the call chain and invoke `substitute` on it. Over 99% of calls to `substitute` use the current environment. Only 0.7% of calls use a custom replacement list or a new environment, and 209.9 K use a parent frame. These are almost entirely due to `deparse`, `eval` and `do.call` which allow specifying their arguments' evaluation environment. For example, the `envnames::get_env_names` function uses `substitute` in different contexts to extract user-defined names of environments. The first call to `substitute` runs in the second frame from the top of the stack, the next call in the first frame, and the third call in the environment in which it is called.

```
get_env_names <- function(envir=NULL, include_functions=FALSE) {
   get_informative_environment_name <- function(envir)
      if (...) envir_name <- deparse(substitute(envir,
          parent.frame(n = 2)))
      else   envir_name <- deparse(substitute(envir, parent.frame
          (n = 1)))
   if (!is.null(envir) && !is.environment(envir))
      { error_NotValidEnvironment(deparse(substitute(envir)));
          return(NULL) }
   envir_name <- get_informative_environment_name(envir)
...
```

The `glmmTMB::makeOp` function constructs ASTs from replacement lists instead of an environment. Arguments `x`, `y` and `op` are evaluated and bound to `X`, `Y` and `op` respectively in the `list` which is then used for replacement in the AST by `substitute`.

```
makeOp <- function(x, y, op=NULL) {
   if (is.null(op) || missing(y)) {
      if (is.null(op)) substitute(OP(X), list(X = x, OP = y))
      else          substitute(OP(X), list(X = x, OP = op))
   } else substitute(OP(X, Y), list(X = x, OP = op, Y = y))
}
```

**QUALITATIVE ANALYSIS** I manually inspected 100 functions that meta-program their arguments, and classified them based on the usage patterns. One common pattern is to extract the source text of an argument. This is used by various plotting functions to give default names to the axes of a graph if none are provided. In the following definition `xAxis` and `yAxis` are parameters used that way. The call to `substitute` returns the AST of the arguments, and `deparse` turns those into text.

```
function(design, xAxis, yAxis) {
   designName <- deparse(substitute(design))
   xAxisName <- deparse(substitute(xAxis))
   yAxisName <- deparse(substitute(yAxis))
   plot(1, type = "n", main = designName, xlab = xAxisName, ylab
        = yAxisName,
...
```

The following pattern explains why many promises are both evaluated and meta-programmed. The call to `substitute` extracts the AST for `deg` and the next line evaluates `deg`.

```
function(deg) {
   degname <- deparse(substitute(deg))
   deg <- as.integer(deg)
   if (deg < 0 || deg > 1) stop(paste0("Error␣",degname))
   deg
}
```

Another common pattern, that is a syntactic convenience, is to allow the use of symbols instead of strings. The `::` operator is used to prefix function names with their packages. It is implemented as a reflective function that expects two strings. But programmers would rather write `base::log` than `"base"::"log"`. To this end, the arguments are left uninterpreted, instead the function deparses them to strings.

```
‘::‘ <- function(pkg, name) {
   pkg <- as.character(substitute(pkg))
   name <- as.character(substitute(name))
   get(name, envir=asNamespace(pkg), inherits=FALSE)
}
```

Meta-programming is also used for better error reporting and logging. This example shows code that only retrieves the source text of the argument.

```
function(arg)
   if (!is.numeric(arg)) stop(paste(deparse(substitute(arg)),"is
       ␣not␣numeric"))
```

I also found functions that leverage non-standard evaluation. The following definition is for `base::local` which provides limited form of sandboxing by evaluating code in a new environment. The argument is extracted and evaluated using `eval` in an empty or user-supplied environment.

```
function(arg, envir=new.env()) eval.parent(substitute(eval(quote
    (arg),envir)))
```

A combination of meta-programming, dynamic evaluation and first-class environments opens up the door for domain specific languages. The pipe operator heavily used in the `tidyverse` group of packages performs non-standard evaluation on its arguments. While the user writes code like this `df %>% mean`, what is actually executed is `mean(df )`. While the actual definition relies on more intricate non-standard evaluation techniques, a simple definition that achieves a similar effect turns both arguments into abstract syntax trees, and captures the calling environment.

```
function(lhs, rhs) {
   lhs <- substitute(lhs)
   rhs <- substitute(rhs)
   eval(call(pipe, rhs, lhs), parent.frame(), parent.frame())
}
```

Overall, the use of meta-programming is widespread and falls in two rough categories: access to the source text of an argument in the direct caller and non-standard evaluation of an argument. The latter is the source of much of the expressive power of the language and is critical to some of the most widely used packages such as `ggplot` and `dplyr`.

### 4.2.4 Revisiting the Traditional Benefits of Laziness

The next research question asks whether the benefits of lazy evaluation that were advocated by Hudak [12] are realized in the R ecosystem.

**RQ4**: *Are the traditional benefits of laziness realized in R?*

These benefits are that programmers need not worry about the cost of unused arguments and they are able to define and use unbounded data structures. I posit the following hypothesis, if programmers understand how call-by-need works, they will feel free to pass complex computations in non-strict positions. If true, one could hope to observe a difference in running time for arguments known to be strict (promises passed to *Always* parameters) and those that are not (promises passed to *Sometimes* parameters). Figure 4.16 shows the probability density of promise running times with times smaller than a millisecond discarded. The promises that are passed to *Sometimes* arguments tend to be less expensive to evaluate. While there is a difference in the profiles, the data does not allow us to confirm that programmers are taking advantage of laziness.

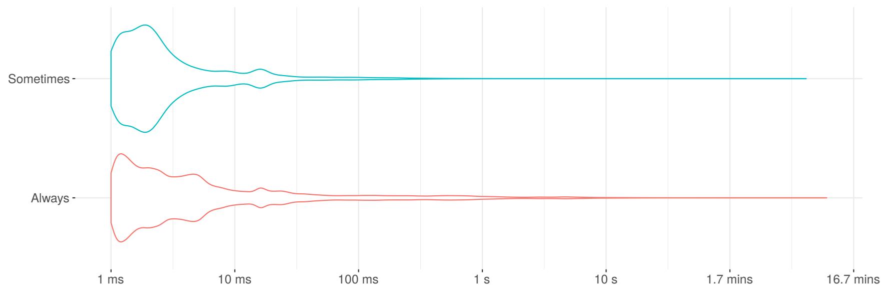

Figure 4.16: Promise evaluation

Laziness makes it possible to compute over infinite data structures. While R does not provide such data structures, it is conceivable that programmers created some in their code. As I have shown, one can use promises together with environments to create unbounded data structures. While it is difficult to measure this directly, we can measure *escaped promises*. These promises outlive the function they are passed into, and these would be a superset of lazy data structures. Of the 261 B promises I have observed, only 11.6 M escape. This is a rather small number. We need to establish the reason why they escape. Figure 4.17 compares the return types of functions which have at least one of their promises escaping, and functions that do not have escaping promises. The main difference between them is that functions with escaping promises have a large number of symbols and closures as their return values. The next section performs a qualitative analysis to understand how those closures are used.

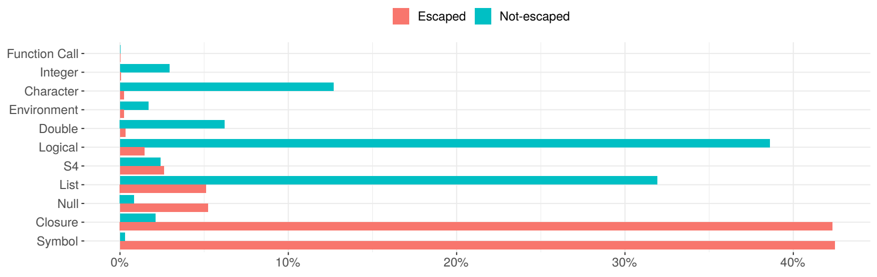

**Figure 4.17**: Function results

**QUALITATIVE ANALYSIS** It is not easy to assess, from the quantitative results alone, whether programmers benefit from laziness. Consider a call, `f(a+b,c)`, and imagine that depending on the value of `c`, the first argument may or may not be evaluated. If `a` and `b` are large matrices and `c` is infrequently true, a programmer aware of laziness would not worry about the performance of this code. Without laziness, the API of the function would likely have to change so that instead of passing the result of the computation one would pass the individual arguments and let the function perform the addition if needed. In my manual inspection, I have not found any code suggesting that programmers are concerned about the cost of evaluating expressions, but this is most likely due to the fact that many users are not performance sensitive.

One promising use of laziness is related to `delayedAssign`. I observed 82 M calls to this function. I inspected manually 36 packages that use it. I found several recurring patterns that aim to avoid unnecessary computation. Examples are: the `AzureML` package uses delayed assignment to avoid loading unneeded parts of its workspace; `crunch` uses it to delay fetching data from a server; and (slightly surprisingly) `callCC` uses it in conjunction with non-local return to implement the `call /cc` function. Overall I found little evidence of programmers taking advantage of call-by-need, other than in cases where they explicitly called `delayedAssign`, in the sample of functions I inspected. I did find cases where the authors of the code seemed to want to enforce a consistent evaluation order and prevent argument-induced side effects from happening in the midst of evaluation of the function.

To detect an infinite data structure I looked at occurrences of promises that outlive the function in which they were passed. I inspected 100 functions with escaping arguments and observed the following patterns: (1) arguments captured in closures, (2) arguments captured in S4 objects, (3) arguments stored in environments, (4) arguments passed into finalizers, (7) argument passed into delayed assignments, and (8) arguments passed into formulas. Figure 4.18 gives examples of each of these categories.

```
   function(arg, e=10^-5) {
     function(x) (arg(x+e)-arg(x-e))/(2*e)
   }
```

(a) Closure

```
function(arg) {
   new("FLXcomponent",df=arg$df)
}
```

(b) S4 object

```
function(arg) {
   env<-new.env();
   env$fn<-function(x) {
      ...
      out<-arg(x)
      ...
   }
   env
}
```

(c) Environment

```
function(arg) {
   reg.finalizer(environment(), function(...) dbDisconnect(arg))
}
```

(d) Finalizer

```
function(arg, e) {
   delayedAssign(x, get(from, arg), assign.env=as.environment(e))
       )
}
```

(e) Delayed assignment

```
function(arg, i) {
   as.formula(arg[, 1] ~ arg[, i])
}
```

(f) Formula

Figure 4.18: Escaping promises

In terms of linguistic mechanisms, all but the last two end up as variants of closure-captured promises. Formula is interesting, because it is really a domain specific language that is interpreted with different semantics. In my time spent working with R, I found a single package, `Rstackdeque` [23] that advertised the use of lazy data structures, specifically fully persistent queues based on [24]. This package, which depends on lazy lists, is the only use of lazy data structures I am aware of in the R ecosystem.

## 4.3 RELATED WORK

The meta-programming support of R is reminiscent of fexprs [33] in Lisp. Fexprs are first class functions with unevaluated arguments. In R, functions always have access to their unevaluated and evaluated arguments. Pitman [26] argued in favor of macros over fexprs. Macros are transparent, their definition can be understood by expanding them to primitive language forms before the evaluation phase. fexprs on the other hand perform code manipulation during evaluation. This makes it harder for compilers to statically optimize fexprs. Furthermore, expression manipulation such as substitution of an expression for all evaluable occurrences of some other expression can be performed correctly by macros because they expand before evaluation to primitive forms.

Purdue FastR [17] is an AST interpreter for R written in Java to explore the applicability of simple compiler optimization techniques, within the reach of scientific community lacking expertise in language run-times. The authors implement an optimization technique that defers element wise operations on vectors by constructing expression trees called Views, which are evaluated on demand. This prevents the materialization of temporary vectors in a chain of vectorized mathematical operations. Like promises, views cache the result of evaluating the expression. However, unlike promises which are exposed to the user through meta-programming, views are completely transparent to the user. Promises are built by packaging arbitrary argument expressions but views are built incrementally by piling referentially transparent vector operations such as `+`, `-`, `log`, `ceil`, etc. Promises are evaluated very quickly due to the eager nature of most functions, but the expression trees of views are evaluated only when the entire result vector or its subset is demanded or a selected aggregate operation such as `sum` is applied.

Building upon the implicit argument quoting of promises is a data structure called quosure, short for quoted closure, that bundles an expression and its evaluation environment for explicit manipulation at the the language level. A quosure is thus an explicit promise object exposed to the user, with APIs to access the underlying expression and environment. Quosures are a central component of a collection of R packages for data manipulation, Tidyverse [36], that have a common design language and underlying data structures. Dplyr [37], a package of Tidyverse, implements a DSL for performing SQL like data transformations on tabular data and ggplot2 [35] implements a declarative language for graphing data, inspired by the Grammar of Graphics. These packages quote, unquote and quasiquote user supplied expressions and evaluate them in appropriate environments. To facilitate this, these packages also provide an evaluation function, `eval_tidy` that extends the base `eval` to deal with quosures. This suggests that reifying promises can be useful.

Renjin is an implementation of R built on the Java virtual machine designed to analyze large data sets and facilitate integration with enterprise systems. Renjin supports delaying evaluation of side effect free computation [21]. Instead of returning the actual result of a computation, Renjin returns placeholders which look and behave exactly like the actual computation result, but will only calculate results if forced to. The difference between Renjin and FastR, both systems are more "lazy" than GNU R, lies in Renjin's support for relational-style optimization.

## 4.4 THREATS TO VALIDITY

Code coverage is a worry for any dynamic approach. There are two additional points to consider. Firstly, C and C++ functions can bypass the R extension API and directly modify R objects' internals. For example, set a promise's value without going through the API thus obviating the hooks. Such behavior breaks the R semantics and is error prone as the R internals do change. I have not observed this behavior in practice, but given the large number of packages, it may happen. Secondly, I disable the bytecode compiler for this study. Since the compiler eliminates promises for literal arguments, I observe more promises than actually created by the GNU R Virtual Machine under its default settings.

## 4.5 CONCLUSION

This chapter offers a glimpse into the use of call-by-need in the R programming language. Call-by-need is the default in R, but this data suggests that it is used less than one would expect. To deal with side effects and manage programmers' expectation, many functions are stricter than they need to be. I found little evidence of lazy data structures or that users leverage lazy evaluation to avoid unnecessary computation. I found only two broad categories of usages that benefited from it. The first is the creation of delayed bindings. These, in my experience, are always explicit. The second is for meta-programming. Within that category, uses are split between accessing the source text of an expression for debugging purposes and performing non-standard evaluation.

The costs of lazy evaluation in performance and memory use are substantial. Every argument to a function must be boxed in a promise, retaining a reference to the function's environment until evaluated. Every access to a variable must check if it is bound to a promise and either evaluate it or read the cached value. Lazy evaluation complicates the task of compilers and program analysis tools as they must deal with the possibility of any variable access causing side effects. Lastly, the majority of users do not expect arguments to be evaluated in a lazy fashion, thus leading to hard to understand bugs.

If laziness is mostly unused, could it be eliminated? Any change to the semantics of a widely used language has to be minimally invasive. The next chapter discusses a set of tools and techniques to remove undue laziness from R code.

# 5 STRICTNESS INFERENCE

In this chapter, I will describe three tools to migrate R code to strict semantics: LazR, a scalable infrastructure for inferring strictness annotations for function arguments by dynamic program analysis, StrictR, an R package that inserts strictness in R packages at runtime, and rastr, an R package that performs source rewriting to insert strictness information in R package code. The goals behind the design of these tools are minimal code changes and accurate strictness inference.

Tying these tools together is a strictness signature, a mechanism to specify function strictness. A strictness signature is of the form:

$$sig \quad ::= \quad \mathsf{strict}\ \text{‘}\mathsf{fun}\text{’}\ \ \langle i_1, i_2, i_3, \ldots \rangle$$

Here, fun is the name of the function. The sequence of integers specifies which argument positions are evaluated strictly. These signatures are stored in signature files; there is one signature file per package. These files are generated and consumed by the tools described in this chapter. The advantage of using external signature files is that the developers don't have to modify the source code of programs. It also enables easy experimentation by switching the signature files. A limitation of storing strictness signatures in external files is that they will inevitably diverge from the code over time. From the point of view of maintainability, it would be preferable to provide strictness information in the source code so they can evolve in tandem. Another drawback of specifying strictness through signatures is that anonymous and inner functions can not be annotated as they have no names. Only top-level functions have a canonical name. Additionally, some packages generate functions at runtime, which are also not handled since they don't have a canonical name. However, I have found these cases to be relatively rare.

LazR and StrictR have been validated as *Functional* and *Reusable* and are available from:

https://doi.org/10.5281/zenodo.5394235

rastr is available from:

https://doi.org/10.5281/zenodo.7803644

## 5.1 LAZR

LazR generates strictness signatures for R packages through dynamic analysis. It has two important components: a system for tracing the execution of R scripts and an infrastructure for extracting executables and running analyses that scale to thousands of packages. The goal behind the design of LazR is not to infer maximally strict signatures but rather to minimize the impact of semantic change on clients by considering both intrinsic and accidental laziness.

**INTRINSIC LAZINESS.** When should an argument be lazy? The empirical evaluation discussed in the previous chapter found barely any use of functional programming idioms related to call-by-need, such as infinite data structures. In fact, most code seems to be written as if R was strict. There is one significant exception: meta-programming. Consider the following:

```
f <- function(a,b) {
  print(deparse(substitute(a)))
  x <- eval(substitute(b))
  x+a
}
```

A call of `f(1+2,3+4)` creates two promises. The first is accessed by `substitute`, turned into a string by `deparse` and printed. The code of the second is accessed by `substitute` and evaluated by `eval`. Then expression `x+a` forces the first promise; the second is never forced. Both arguments are intrinsically lazy. Additionally, C code sometimes expects an argument to be a promise, when using the `PREXPR` macro to access its expression. Lastly, an argument that is not always evaluated may be marked as lazy. Though, this is rarely necessary.

**ACCIDENTAL LAZINESS.** In order to preserve the behavior of legacy code, some parameters will be labeled as lazy even if the called function does not require it. An argument that performs a side-effect is treated as lazy to retain semantics. For instance, in the call `f(g(),x<-1)`, function `f` is free to evaluate its arguments in any order. Enforcing one particular order may lead to observable differences in behavior, e. g.if the call to `g()` reads `x`. Writing such code is error-prone, as small changes to `f` may break it. R has a call-by-value semantics for vectors and lists. These are the most frequently used data types, so many updates will be locally contained. Errors and exceptions are another source of effects inside promises. Some reflective functions can make evaluation of a promise sensitive to its position on the call stack, for e.g., `as.environment` accesses specific frames on the stack by their index. Strict evaluation of such promises is observable if it changes the position of the promise on the call stack. It is worth noting, again, that such code is brittle as any change in the target function can change the frame returned by the reflective calls.

One special case is that of vararg arguments. Assigning a single strictness annotation to `...` is tricky because a function can have different strictness behaviors for each element. For example, object-oriented dispatch uses a vararg to forward all method arguments from the caller to the target function. The current choice is pragmatic but imprecise; all varargs remain lazy.

**ORDER OF EVALUATION** A design choice I faced was to select an evaluation order for strict arguments. My tool evaluate all arguments left-to-right in the order they appear in the function signature. This means that end-users need to know in what order arguments are defined. I elucidate this point in Fig. 5.1 which shows a `popular` function that takes six arguments; `a` is always evaluated, `b` is conditionally evaluated, `c` is forwarded to `substitute` for meta-programming, `d` and `e` are forwarded to a strict function, and `f` is evaluated if it is not missing. The client code has three invocations. In all of them, `f` is missing. This makes `f` lazy because there is no information available about its evaluation. For `r1`, variable `b` should not be evaluated, evaluating it strictly will change the output of the program; for `r2`, evaluating `c` will immediately terminate execution, departing from the original program behavior; and for `r3`, if `d` is evaluated before `e` a different result can be observed. To summarize, the expected output of the analysis of legacy code is to increase strictness while keeping the number of observable semantic differences low. For new code, I expect programmers to mostly use strict parameters.

```
 popular <- function(a,b,c,d,e,f) {
   if (a) b
   print(substitute(c))
   if(!missing(f)) print(f)
   return(e+d)
}
```

(a) *Library*

```
r1<-popular(FALSE,print('Hi'),3,4,5)
r2<-popular(TRUE,1+2,stop(),0,9)
r3<-popular(TRUE,1+2,3,r1<-4,r1+1)
```

(b) *Client code*

Figure 5.1: Inferring strictness signatures

**TRACING** The heart of LazR is a dynamic analysis tool built on the R-dyntrace package which extends the GNU R virtual machine version 4.0.2 [8]. When R-dyntrace executes a script, it generates a *trace* which is a sequence of low-level events that mirrors the operations performed by the interpreter. The trace exposes raw R objects being operated on as well as control flow. As traces can get large, rather than recording them, R-dyntrace exposes callbacks that are used to hook analysis-specific functionality to events. LazR intercepts these callbacks and provides a layer of abstraction by maintaining *model objects* to abstract from concrete R data structures. These objects represent function instances, environments, and stack frames. Model objects help in handling some of the complexities of R. For instance, they have unique identities, whereas R objects are identified by their memory address which can be reused. For scalability, LazR reclaims model objects when the corresponding raw R object is garbage collected and supports efficient indexing of these objects. Model objects are also used for bookkeeping of the relevant operations on the corresponding raw R objects. Model functions have names heuristically reconstructed by keeping track of their lexical scopes. The model stack can also deal with the use of `longjump` for non-local returns. Lastly, LazR maintains a notion of logical time, used to record when some events of interest happened. For instance, environments record the time of last read and last write. Similarly, code blocks record evaluation start and end.

**INFERENCE** LazR monitors traces looking for signals that parameters are lazy. These signals are relative to how the called function uses the parameter and what the provided argument does; I summarize them here:

**V**: A vararg parameter; I consider this to be a signal of intrinsic laziness. The reason for this particular choice is that it is unclear how to treat individual elements of the varag.

**M**: A parameter passed to `substitute` or `PREXPR` is *meta-programmed*; this is a signal of intrinsic laziness.

**G**: A parameter that did not receive an argument on any function invocation; this is a signal for laziness owing to a lack of information.

**U**: A parameter that remains *unevaluated*; this is a signal of intrinsic laziness as we do not know the contract of the function for that parameter.

**S**: An argument with *side-effects*; a signal of accidental laziness as some effects may be benign. I monitor three kinds of effects: variable reads, variable writes (definitions, updates, and removals of variables), and errors.

**R**: An argument that uses *reflection* to observe the state of the call stack; this is a signal of accidental laziness as strict evaluation may evaluate the promise with a different stack.

For any trace, LazR models each argument of each function. If a function is invoked, the tracer records all operations related to its arguments and correlates them with the parameters. For each operation performed by the interpreter, the analysis finds the responsible promises, i.e., all promises on the call stack. Since each promise is bound to an argument of a function, one can connect that promise to a corresponding parameter. When a side-effect or reflection occurs, the parameters corresponding to all responsible promises are marked S or R. Reads and writes are ignored in some cases. LazR keeps track of the last accessed and modified time of each binding. For a write from a promise, if the binding being modified was not accessed or modified after the promise was created and before it was executed, the write is ignored because evaluating the promise early will not introduce a conflict. Similarly, for reads, if the binding being read was not modified after the promise was created and before it was executed, then the read is ignored. For performance but at the cost of precision, only the last 10K read and write times for each variable are recorded.

The result of analyzing an R script is a summary for each function and each parameter of the signals that were observed during tracing. Multiple scripts can be straightforwardly merged as the union of signals for each parameter. Finally, from this summary, strictness signatures are generated. Given a set of signals the current implementation treats them all strong signals and conservatively makes a parameter lazy at the first signal.

Consider Fig. 5.1, for `a` and `e` no signals are observed, `b` is **U** due to `r1`, `c` is **M** from all invocations, `d` is **S** due to `r3` since its argument mutates `r1` after a read by `e`, and `f` is **G** since it is missing in all invocations. LazR combines this information to synthesize the signature, `strict popular<1,5>`, parameters `a` and `e` are strict, and others are lazy.

**LIMITATIONS** The strictness signatures generated by LazR are unsound. LazR does not handle IO. The R ecosystem has a rich collection of libraries for handling data, resulting in a vast API for reading and writing files to disk. These functions call arbitrary C/C++ libraries for IO. Tracking these calls would require tracing *syscalls* which is currently not supported by the dynamic analysis infrastructure. LazR also does not handle state changes in the native code of packages because there is no particular API that can be intercepted to track those changes. Finally, since LazR relies on dynamic analysis, code coverage directly affects the quality of strictness signatures.

## 5.2 STRICTR

StrictR is a tool that enforces strict semantics on R code as dictated by the strictness signatures produced by LazR. StrictR inserts `force` calls at function boundaries for arguments specified in the strictness signatures.

Consider the `str_to_upper` function from the stringr package and its strictness signature as shown below.

```
str_to_upper <- function(string, locale = "en") {
  stri_trans_toupper(string, locale = locale)
}
```

strict str_to_upper $\langle 1, 2 \rangle$

**Figure 5.2:** `stringr::str_to_upper`: definition and strictness signature

Based on the strictness signature, StrictR modifies `str_to_upper` as follows after stringr is loaded.

```
str_to_upper <- function(string, locale = "en") {
  force(string)
  force(locale)
  stri_trans_toupper(string, locale = locale)
}
```

**Figure 5.3:** `stringr::str_to_upper` definition updated by `strictr`

**IMPLEMENTATION** StrictR is implemented as an R package that works with an unmodified GNU R interpreter. It uses a feature of R that allows registering callbacks when packages are loaded. The callback reads the signature file for the loaded package from the current directory or from the load path and injects `force` calls in the functions accordingly. An important design decision was to copy functions as they were rewritten. An earlier implementation mutated function bodies without copying. This resulted in failures because the same function object can occur with different names and signatures. For instance, in the `rlang` package, `is_same_body` aliases `is_reference`. Mutating `is_reference` to make arguments strict inadvertently also made the function `is_same_body` strict if it was not copied before rewriting. To remedy this, StrictR copies functions as it rewrites them.

**LIMITATIONS** Dynamic strictness insertion through StrictR introduces an extra run time dependency on StrictR. Furthermore, a user interested in leveraging strictness would have to modify their program to manually import StrictR at the beginning of the file so that the functions in subsequently loaded packages are rewritten per their signatures. Dynamic code modification also complicates debugging since the modified code does not match the source.

## 5.3 RASTR

In this section, I will describe rastr, a tool that implements an extension to R syntax, sugr. sugr enables strictness annotations to be supplied as part of a function declaration. rastr also provides functions to transform this extended R syntax to standard R syntax and vice-versa. Conceptually, rastr performs the same task as StrictR in that it replaces strictness annotations with `force` calls. It differs in that it performs rewriting on the concrete source code instead of modifying abstract syntax at run time. Since annotations can be provided in source files, rastr can handle anonymous and inner functions.

To explain how rastr works, I will use the example of `str_to_upper` function from the `case.R` file of `stringr` package as shown in Figure 5.2. Running the `sugar` function from `rastr` on this code generates a `case.sugr` file with the following definition for `str_to_upper`.

```
str_to_upper <- function(strict string, strict locale = "en") {
  stri_trans_toupper(string, locale = locale)
}
```

**Figure 5.4**: Sugared version of `stringr::str_to_upper`

The `sugar` function introduces the strictness annotations inside the source code. This facilitates their co-evolution with code. Additionally, this makes it easy to identify and fix annotations that are inferred incorrectly on account of unsoundness. After ensuring the correctness of these annotations, the `desugar` function is invoked, which converts the `case.sugr` file back to `case.R` by replacing the annotations with `force` calls, as shown in Figure 5.3. rastr also provides an RStudio plugin to invoke the `desugar` function on the current file or project with the click of a button.

rastr is implemented as an R package consisting of 26K lines of C/C++ code and 1.5K lines of R code. Conceptually, the implementation has three parts: a hand-written R parser to read code into an AST, an exhaustive API for manipulating syntax tree nodes, and functions for rewriting ASTs.

**PARSER** The main component of rastr is a full-featured hand-written parser for R, written in a recursive-descent operator-precedence style as described by Pratt [27]. The parser reads the R source into an AST, which can be traversed to annotate function parameters with `strict` annotations. The parser keeps track of the concrete code in the file, including spaces and comments.

GNU R already includes an R parser which can be invoked via the `parse` function to obtain an AST. Why bother reimplementing it? The goal is to keep changes to the original code to a minimum; the R code written back to file after incorporating the `strict` annotations should be identical to the code originally written by the developer in every other way. Unfortunately, the abstract representation of R code used by GNU R ignores many source-level details. Spaces and comments are ignored; this means the indentation and documentation supplied by the developer are lost. Literals are parsed to values in a canonical form; this makes it impossible to know if a string is delimited using one or two quotes, or if a function is defined using the **`function`** keyword or the `\` operator, or the precision used to write floating point numbers. While R does provide a `srcref` mechanism that stores concrete representation of tokens, this mechanism is flaky at best and won't satisfy all the requirements. The APIs provided by GNU R to manipulate its AST have inconsistent behavior when handling missing arguments or literal `NULL` values in the source code. This makes it challenging to perform source rewritings. The GNU R parser is an LALR parser generated using yacc. It was difficult to extend it with optional strictness annotations without introducing new parsing conflicts. One limitation of my implementation is that it only supports UTF-8 encoding.

**AST** rastr provides APIs to inspect and manipulate ASTs both from R, for convenience and quick prototyping, and from C/C++, for performance. There are 390 APIs as of now. The AST nodes are modeled using a C `struct` with a type tag to disambiguate the different node types. To avoid the difficulty of managing memory for cyclic tree structures and nodes with multiple owners, the tree data structure is engineered to ensure unique ownership, i.e., a node can have at most one parent. Nodes are allocated from a custom allocator, which provides constant-time deletion and an amortized constant-time allocation. Instead of providing the address of the allocated node, the allocator returns a number that represents the index at which the node is stored in the allocator's internal store. These opaque node references force clients to use `rastr` APIs for all node operations. The APIs perform the necessary checks and clone nodes to satisfy the unique ownership constraint. Interoperability between R and native code requires significant boilerplate code. Currently, there are 42 types of nodes. For every node type, there is a constructor, and getters and setters for child nodes. For every API, there is a C function and an R function that invokes the C function via a mediator. The mediator function converts R values to their C equivalents and vice-versa. The implementation exploits the regular structure of these APIs by generating them from a template whose parameters are substituted from a YAML file containing a description of all the nodes. This affords two conveniences: bugs can be fixed by changing the relevant template and regenerating the APIs, and new node APIs can be generated by simply adding the node description to the YAML file.

**SOURCE REWRITING** The `desugar` and `sugar` functions are implemented using a generic `walk` function. `walk` performs a depth-first walk over the AST. It takes node-specific functions as arguments and invokes them when it encounters the corresponding nodes. These functions can inspect or modify the nodes. Modifications are performed in-place for performance reasons. The `clone` function can be used to copy the AST for functional updates.

**LIMITATIONS** In terms of its limitations, rastr complements StrictR. rastr cannot enforce strictness in dynamically generated functions. StrictR can handle functions that are generated during package loading. Neither of the two can handle functions that are generated while the program is being executed. However, rastr can add strictness to inner functions which StrictR cannot. Unlike StrictR, the use of rastr does not add a run time dependency because the package code is modified before it is even installed.

## 5.4 RELATED WORK

Functional languages with a call-by-need evaluation strategy must contend with memory pressure and associated performance issues due to the allocation of a substantial number of thunks (suspended computations) [6, 16, 25]. The Glasgow Haskell Compiler performs a strictness analysis pass to identify arguments that can be evaluated strictly. While most programs benefit from such a transformation, due to its conservative nature, this pass misses some opportunities for optimizations. To recover performance, programmers can manually insert strictness annotations to control evaluation; identifying where to put them, however, can be challenging. Wang, Nunez, and Fisher [34] proposed Autobahn, a tool that automatically infers strictness annotations using a genetic algorithm. This approach relies on dynamic analysis, which can be more precise than static analysis but does not guarantee termination on all inputs. As the annotations are based on a heuristic, developers must manually validate their soundness. The authors report an average 8.5% speedup (with a maximum speedup of 89%). Chang and Felleisen [5] solve the complementary problem of suggesting laziness annotations for call-by-value $\lambda$ calculus using dynamic analysis. They introduce the notion of laziness potential, a predictor of the benefit obtained by making an expression lazy. They use this as a guide to insert laziness annotations. They demonstrate benefits on Racket implementations of Okasaki's purely functional data structures [24], monadic parser combinators, and an AI game. My work is similar to Autobahn in that I infer annotations dynamically and I do not guarantee soundness. I depart from Autobahn in that I use dynamic analysis of execution traces to determine strictness. Furthermore, I must deal with side-effects and reflective operations which adds extra complexity to the inference algorithm.

The concept of sugr is inspired by TypeScript, but unlike TypeScript, the strictness annotations change the semantics of the code. Similar to TypeScript, I intend to extend sugr to support type annotations and a C++-style class declaration syntax for R's object systems.

## 5.5 CONCLUSIONS

I discussed the implementation of three tools in this chapter, LazR, StrictR, and rastr, to remove laziness with high accuracy and minimal modifications to existing code. LazR uses dynamic analysis to observe argument usage and synthesizes strictness signature: a sequence of argument positions that can be evaluated eagerly. LazR considers almost all features of R that interact with laziness; arguments that were not evaluated at least once, varargs, missing arguments, arguments used for meta-programming, and arguments performing a non-local side-effect or a reflective operation on the call-stack are all marked lazy. LazR does not handle IO and state changes in the native code, which makes the signatures unsound. Since it relies on dynamic analysis, which depends on code coverage, LazR underestimates laziness, which further adds to the unsoundness of signatures. The signatures generated by LazR are used by StrictR and rastr to add strictness to R code by calling `force` on the arguments at the function boundary. While StrictR inserts these calls at run time, rastr inserts them in the source code itself. The next chapter goes on to remove laziness from R code and, in the process, evaluate the impact of unsoundness of the strictness signatures.

# 6 ECOSYSTEM MIGRATION

In this chapter, I will use LazR, StrictR, and rastr to assess the impact of automated migration on a subset of the R ecosystem. First, I will show that the automated migration breaks only a small number clients. Next, I will focus on the tidyverse packages and discuss the errors arising from the automated inference.

## 6.1 CORPUS

For this experiment, I selected a corpus of 500 packages with most client packages from CRAN [19].have a total of 13,308 clients ranging from `ggplot2` with 2,382 clients all the way to `factoextra` with only 14. The packages have 2.1M lines of R code and 2.8M lines of native code.

Table 6.1 summarizes the runnable code extracted from these 500 packages. Each test is run as a separate script. Examples and vignettes are snippets of code embedded in the documentation. LazR extracts them into self-standing scripts. Typically, vignettes are longer examples with input data, while examples are smaller code fragments.

**Table 6.1**: Corpus

| | **Tests** | **Examples** | **Vignettes** |
|---|---:|---:|---:|
| Scripts | 4.7K | 18.9K | 581 |
| LOC | 342.9K | 195.6K | 44.4K |

Given the large corpus size, the amount of data to process can reach multi-terabyte sizes. LazR adopts a simple map-reduce style for scalability. It splits the analysis in phases as shown in Fig. 6.1.

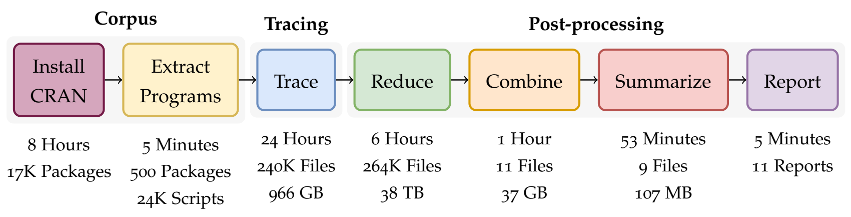

**Figure 6.1**: Analysis Pipeline

The reduce maps a function on the output of one trace to get a partial summary. The combine phase concatenates partial summaries. Then, the summarize phase aggregates summaries into a result table. Finally, the report phase creates graphs and tables for inclusion in the paper. For reproducibility, the LazR pipeline is set up with a container image that includes all the dependencies for installing analysis code and R packages. To run it, I mirrored the repositories, installed their packages, and executed the script to generate traces. These traces are analyzed to output tabular data files and strictness signatures. Whenever possible, I parallelized the steps. The experiments were performed on two Intel Xeon 6140, 2.30GHz machines with 72 cores and 256GB of RAM each.

The tracer encounters 51.5K top-level functions; 161 packages have 25 functions or less, and 13 packages with more than 500 functions. The largest package, `spatstat.geom`, has 889 functions. We observe 130M calls to these functions, their distribution per function is in Fig. 6.2; 49% of functions are called more than ten times, while 17% are called only once.

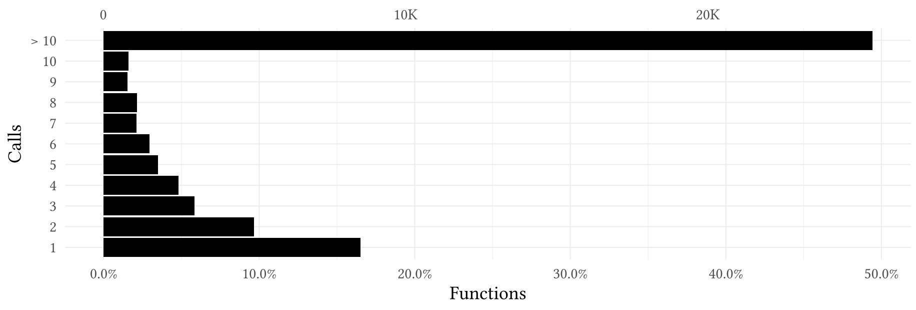

**Figure 6.2:** Call Distribution

The traces record 288M arguments, of those, 3.7M are missing, 20K are varargs, and the remaining are promises. These arguments correspond to 204K parameter positions, their distribution per function is in Fig. 6.3; 2% of functions have no parameters, 20% have 1, 6% have over 10, and 13 functions have over 50. Function `ergm::control.ergm` takes the cake with `150` parameters.

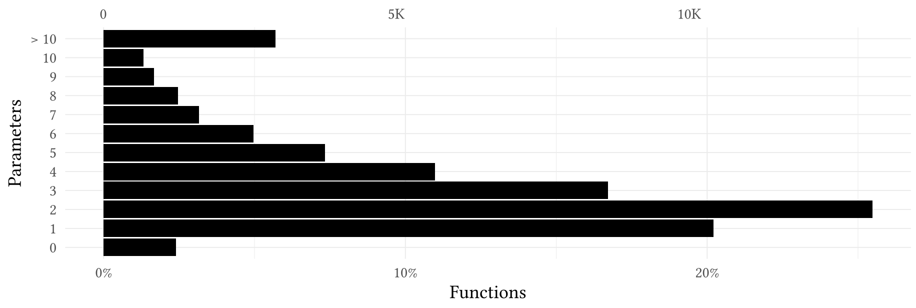

**Figure 6.3:** Parameter Distribution

## 6.2 STRICTNESS SIGNATURES

There are three signals I consider strong, parameters marked by either one of them are considered lazy. LazR recorded 1.3K parameters being meta-programmed (marked **M**), furthermore 3.2K were missing (marked **G**), and 20K were varargs (marked **V**). All these remain lazy. For other combinations of signals Table 6.2 summarizes their distribution. One function can be counted in multiple rows as its different parameters may have different combinations. Rows are interpreted as follows: the fourth row, for instance, indicates that no signals were observed for 148.4K parameters coming from 49.3K functions and 489 packages.

**Table 6.2**: Signature Summary

| V | M | G | U | S | R | Parameters | | Functions | | Packages |
|---|---|---|---|---|---|---|---|---|---|---|
| ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | 20.0K | 9.8% | 20.0K | 38.8% | 430 |
| ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | 1.3K | 0.6% | 825 | 1.6% | 118 |
| ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | 3.2K | 1.6% | 1.7K | 3.2% | 240 |
| ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | 148.4K | 72.9% | 49.3K | 95.7% | 489 |
| ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | 529 | 0.3% | 509 | 1% | 119 |
| ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | 1.3K | 0.6% | 950 | 1.8% | 207 |
| ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | 76 | 0% | 74 | 0.1% | 22 |
| ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | 28.5K | 14% | 12.4K | 24.1% | 450 |
| ✗ | ✗ | ✗ | ✓ | ✗ | ✓ | 33 | 0% | 32 | 0.1% | 24 |
| ✗ | ✗ | ✗ | ✓ | ✓ | ✗ | 314 | 0.2% | 226 | 0.4% | 98 |
| ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | 9 | 0% | 9 | 0% | 5 |

The majority of parameters (72.9%) are always evaluated, and the corresponding arguments do not perform side-effects or reflective operations. These parameters can safely be evaluated strictly. Unevaluated parameters account for 14.2% of observations, a significant source of potential laziness. Variable lengths arguments and missing arguments account for 9.8% and 1.6% of the data respectively. The remaining configurations are in the noise. It is noteworthy that parameters marked **R** come from very few functions and packages. I now discuss the different sources of laziness in more detail.

**[V] VARARGS** Some 20K parameters from 20K functions are marked **V** for *vararg*.

**[M] META-PROGRAMMING** Table 6.3 counts arguments, parameters and functions using the meta-programming facilities of R. Numbers are also provided for C code that uses PREXPR. Arguments count all of the invocations of a function. PREXPR was not addressed there: it is a C macro used to extract a promise's expression for ad-hoc evaluation. Packages such as `lazyeval` and `rlang` use it. `PREXPR` is also used by the builtins of the GNU R interpreter, a canonical example is the `missing` function which checks if an argument is provided. Unlike user packages, these uses of `PREXPR` do not require the corresponding arguments to be lazily evaluated. Hence, uses of `PREXPR` from the interpreter are not signals.

**Table 6.3**: Meta-programming

| | **R** | **Native** | **Total** |
|---|---|---|---|
| Arguments | 1.6M | 922.2K | 2.5M |
| Parameters | 613 | 680 | 1.3K |
| Functions | 387 | 438 | 825 |

**[G] MISSING ARGUMENTS** Some 3.2K parameters never receive arguments, default, or explicitly supplied; they are classified as *Always*. This is in contrast to the 3.3K *Sometimes* parameters which sometimes receive arguments.

**Table 6.4**: Missing

| | **Sometimes** | **Always** | **Total** |
|---|---|---|---|
| Argments | 2.8M | 864.7K | 3.7M |
| Parameters | 3.3K | 3.2K | 6.5K |
| Functions | 1.8K | 1.7K | 3.1K |

Table 6.4 gives numbers of missing arguments and parameters. *Always* parameters are a strong signal, whereas *Sometimes* missing are only lazy if also marked **U**. In the following definition the `names` argument was missing in all calls to this function from `rlang`, it is thus an *Always* parameter. That fact is not surprising when inspecting the code of the function: the argument is not used! It is presumably only there for backwards compatibility.

```
new_environments <- function(envs, names) {
  stopifnot(is_list(envs))
  structure(envs,names=map_chr(unname(envs),env_name),class="
      rlang_envs")
}
```

For a *Sometimes* parameter, consider the `linewidth` argument which is missing in some calls to this function which assigns a default value of `0L` to `linewidth`.

```
base64encode <- function(what, linewidth, newline) {
  linewidth <- if (missing(linewidth) || !is.numeric(linewidth))
       0L
              else as.integer(linewidth[1L])
```

**[U] UNEVALUATED ARGUMENTS** Some parameters are *Sometimes* evaluated and others are *Never* evaluated. The latter may correspond to code paths not exercised or to dummy parameters. Table 6.5 has numbers for both categories.

Table 6.5: Unevaluated

| | Sometimes | Never | Total |
|---|---|---|---|
| Parameters | 17.3K | 11.6K | 28.9K |
| Functions | 7.4K | 6.4K | 12.5K |

A common pattern is to evaluate one argument only if another has a given value.

```
%||% <- function(x,y) if(is.null(x)) y else x
```

Another pattern is to delay evaluation: `expr` argument is delayed until `pkg` is loaded in the example below.

```
glue::on_package_load <- function(pkg, expr) {
  if (isNamespaceLoaded(pkg)) { expr } else {
    thunk <- function(...) expr
    setHook(packageEvent(pkg, "onLoad"), thunk)
} }
```

S3 generic methods are functions dynamically dispatched: here, `x` is always evaluated in order to dispatch, the evaluation of others depends on where the call dispatches to.

```
abind::acorn <- function(x,n=6,m=5,r=1,...) UseMethod('acorn')
```

Some functions implement a common interface, the `proxy` package defines over a dozen methods with interface `function(a,b,c,d,n)` to compute different proximity metrics with a subset of the arguments. Arguments can also be *Never* by design; `tail` is not defined for `tbl_lazy` objects, so it terminates the program with an error message without using its arguments.

```
dbplyr::tail.tbl_lazy <- function(x, n = 6L, ...)
  stop("tail()␣is␣not␣supported␣by␣sql␣sources", call.=FALSE)
```

**[S] SIDE-EFFECTS** Few promises have side-effects and many of those are benign for our purposes as they happen to local variables of the promise. Only 455.7K arguments out of 288M are made lazy owing to effects. This corresponds to 1.7K lazy parameters from 1.2K functions. A typical example of this category is the `along` parameter of `abind` function from the `abind` package. Its default value is the parameter `N` which is computed internally in the body of the function before `along` is used. Clearly, the evaluation of `along` has to be delayed.

```
abind <- function(..., along=N, ...) {
   N <- max(1, sapply(arg.list, function(x) length(dim(x))))
   if (!is.null(rev.along))
      along <- N + 1 - rev.along
}
```

Another example is the `expr` parameter of `withPrivateSeed` function from the `shiny` package. This function evaluates `expr` after setting the global random number generator seed to its own private value. The seed is set back to its initial value on function exit. The `expr` parameter should not be evaluated until the seed has been changed, hence it is made lazy by LazR.

```
withPrivateSeed <- function(expr) {
  origSeed <- .GlobalEnv$.Random.seed
  GlobalEnv$.Random.seed <- .globals$ownSeed
  on.exit({.GlobalEnv$.Random.seed <- origSeed})
  expr
}
```

**[R] REFLECTION** Very few promises look up the parent environments. In the whole corpus, 647 parameters in 614 functions probed the call stack. But in all cases, this was because they invoked either of two functions, `.getFunctionByName`, or `backports:::get0`. To give an example, the first function searches for a function by name in different scopes, and its second argument probes the stack by default.

```
 R.oo:::.getFunctionByName <- function(..., callEnvir=
     as.environment(-1L)) {
   envirT <- callEnvir
   #...
```

In Goel et al. [7], I developed tools to migrate the R ecosystem to strict semantics. This contribution addresses the second step of the migration strategy. The tools have been validated as *Functional* and *Reusable* and are available from:

https://doi.org/10.5281/zenodo.5394235

**Table 6.6:** Client Corpus

| | Tests | Examples | Vignettes |
|---|---|---|---|
| Scripts | 7.6K | 42.6K | 1.8K |
| LOC | 663.6K | 366.1K | 118.5K |

To investigate whether this unsound approach to strictness is viable, I conducted an experiment to synthesize and validate strictness signatures for R packages. I obtained 500 most widely used packages (corpus) in the R ecosystem and, using LazR, leveraged their regression tests to infer strictness signatures. LazR synthesized signatures for 51.5K top-level functions with 204K parameter positions from these packages. Overall, 27.1% of the parameters were marked lazy, and a majority, 72.9% of parameters, were marked strict. Then, I assessed the robustness of strictness signatures using the client packages of the corpus for which strictness signatures were generated. These client packages import the corpus packages and call their functions. From the 13,308 clients of the corpus, I selected 2000 packages for this experiment. From these packages, I extracted 51.5 K runnable programs and executed them twice to filter 45.1K deterministic programs – programs with the same output on both runs, which I then executed by applying strictness signatures using StrictR. I then compared the output of this strict run with the lazy run; a difference in outputs was attributed to the modified semantics. I observed that 3,139 scripts produced erroneous output. I narrowed down the cause of these errors to a handful of packages: R.oo, R.utils, rlang, vctrs, ggplot2, Matrix, and spam. Making these packages lazy decreased the number of failures to 358, a meager 0.79% of all the scripts with deterministic output. The differences in output originated from many sources, such as errors in native code or different startup messages printed by some packages. This experiment shows that it is possible to automatically infer strictness signature for legacy R code with reasonable accuracy for migration to the strict semantics.

## 6.3 ROBUSTNESS OF INFERRED SIGNATURES

This section reports on the robustness of strictness signatures generated by LazR. Each package in the corpus is widely used, there are many packages in CRAN which import it and call its functions, the idea is to use the tests of these clients for validation. From the 13,308 clients of the corpus, I select 2000 packages for this experiment.

Fig. 6.4 shows the steps performed in the experiment. First, I extract runnable code from the clients. Table 6.6 counts the various scripts and the lines of code they represent. Next, GNU R evaluates each script twice. Scripts that do not have the same output are considered non-deterministic and filtered out. This leaves us with 45.1K scripts. They are then run with StrictR after applying the strictness signatures obtained by LazR. LazR generates signatures for 51.5K functions with 204K parameters of which 148.4K (72.9%) parameters are strict. When the output of this strict run differs from the one obtained by GNU R, the difference is attributed to the modified semantics. Comparing the output of scripts is standard practice in the R community for detecting regressions. There may, of course, be differences in execution that do not manifest in the output; so the results reported here are a lower bound.

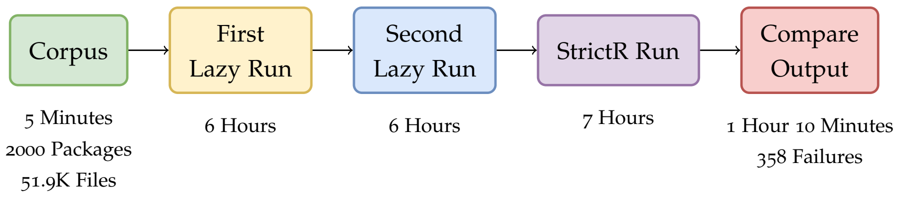

**Figure 6.4:** Validation Pipeline

With all the signatures applied, I observed 3,139 scripts produce erroneous output. I narrowed down the cause of these errors to a handful of packages: `R.oo`, `R.utils`, `rlang`, `vctrs`, `ggplot2`, `Matrix`, and `spam`. Making these packages lazy decreased the number of failures to 358 which is only 0.79% of all the scripts with deterministic output. I detail some of the errors next.

Loading package `R.oo`, and `R.utils` terminates execution with an error as strict semantics interfere with the initialization code.

```
call: getStaticInstance.Object(this, envir = ...envir)
error: Cannot get static instance.
Failed to locate Class object for class 'Package'.
Execution halted
```

Loading package `vctrs` fails non-deterministic error originating from native code. I believe it to be a memory bug as the error disappears after a couple of attempts.

```
Error: 'rho' must be an environment not NULL:
    detected in C-level eval
```

Loading `Matrix` and `spam` changes the message on package loading. This does not affect execution of the package itself.

These results suggest that the semi-automated inference algorithm works well for the most part. Errors in signatures could be located easily and do not require much effort to fix. It should be noted that one could increase the strictness by turning some of the parameters that are lazy for accidental reasons into strict ones, but this would require notifying clients and ensuring that their code is changed.

## 6.4 MIGRATING TIDYVERSE

The tidyverse is an influential collection of R packages for data science with a common underlying design philosophy. On account of being widely used, these packages are the most common dependencies in the ecosystem. *Effective* migration of the R ecosystem rests heavily on the successful migration of tidyverse packages. Furthermore, on account of being well-tested, these packages provide an excellent opportunity to identify bugs in the automated migration. For example, the `dplyr` package has 432 top-level functions and 2,266 tests. I will focus on the migration of eight tidyverse packages: `ggplot2`, `dplyr`, `purrr`, `tibble`, `readr`, `tidyr`, `stringr`, and `forcats`. Table 6.7 gives the size of these packages in terms of lines of code, number of top-level functions, and number of parameters.

**Table 6.7:** Tidyverse Corpus

| **Package** | **LOC** | | **Functions** | **Parameters** | | |
|---|---|---|---|---|---|---|
| | R | C/C++ | | Total | Strict | % |
| `ggplot2` | 25.2K | 0 | 619 | 2,839 | 2,110 | 74.3% |
| `dplyr` | 14.7K | 1.2K | 432 | 1,359 | 763 | 56.1% |
| `purrr` | 4.5K | 1.1K | 229 | 680 | 454 | 66.8% |
| `tibble` | 9.7K | 315 | 205 | 485 | 321 | 66.2% |
| `readr` | 4.4K | 6.2K | 170 | 802 | 580 | 72.3% |
| `tidyr` | 5K | 436 | 116 | 415 | 207 | 49.9% |
| `stringr` | 1.3K | 0 | 66 | 162 | 133 | 82.1% |
| `forcats` | 1.1K | 0 | 46 | 110 | 73 | 66.4% |

To migrate these packages, I perform the following steps.

1. I run LazR to generate strictness signatures for these packages.
2. I use `rastr::sugar` to introduce the strictness annotations in these packages, followed by a call to `rastr::desugar` to transform these annotations into `force` calls.
3. I compare and report the differences in test results between the migrated and the original packages.

Table 6.8 gives the number of test failures for every package.

Table 6.8: Tidyverse Migration

| Package | Total Tests | Failed Tests |
|---|---|---|
| ggplot2 | 1,648 | 0 |
| dplyr | 2,266 | 0 |
| purrr | 762 | 0 |
| tibble | 1,487 | 0 |
| readr | 746 | 3 |
| tidyr | 638 | 1 |
| stringr | 236 | 0 |
| forcats | 163 | 0 |

The results are encouraging; Out of 7,946 tests only four fail. These tests come from `readr` and `tidyr` packages. The tidyverse packages make heavy use of meta-programming, dynamic evaluation, dynamic code generation, and native code to implement domain-specific languages. Additionally, the size and complexity of GNU R make it unlikely for the inference algorithm to ever be exhaustive. Despite these limitations, the automated migration is very precise. Next, I will discuss the reasons for failing tests and fix them.

**`readr`** The three failing tests from `readr` throw the same error:

```
object 'n_max' not found
```

Manual inspection reveals that the error arises from dynamic code generation. I will take the example of `melt_csv_chunked` function to explain the cause of this error. The figure below shows how it is generated dynamically from the definition of `melt_csv`.

```
melt_csv_chunked <- generate_melt_chunked_fun(melt_csv)

melt_csv <- function(..., strict n_max = Inf, ...) {
   ...
}

generate_melt_chunked_fun <- function(x) {
   args <- formals(x)
   # Remove n_max argument
   args <- args[names(args) != "n_max"]
   ....
}
```

Since `melt_csv` is strict in parameter `n_max`, the desugaring inserts `force (n_max)` in the body of `melt_csv`. At run time, `generate_melt_chunked_fun` generates a definition for `melt_csv_chunked` by removing `n_max` from the parameter list. However, the body of the function still contains `force(n_max)`. Upon invocation, `melt_csv_chunked` calls force on n_max, which is no longer defined, leading to the error: object 'n_max' not found.

Interestingly, this error is a consequence of static source rewriting: the body of melt_csv is modified statically and reused at run time for melt_csv_chunked. Run time source rewriting using StrictR avoids this problem because strictness insertion will happen directly in the body of melt_csv_chunked after its definition is generated.

The error can be fixed by removing the strict annotation on n_max. This has the consequence of adding superfluous laziness to melt_csv but prevents melt_csv_chunked from raising the error.

The same fix was applied to melt_csv2, melt_delim, and melt_tsv. These functions forward n_max to melt_delimited, so melt_delimited was also made lazy in n_max. After applying these changes, all tests passed.

**tidyr** The test in tidyr fails with the following message:

```
Names must be unique.
x These names are duplicated:
  * "a" at locations 1 and 2.
```

The source of the error is a call to the wrap_error_names function, whose definition is shown below.

```
wrap_error_names <- function(strict code) {
   tryCatch(code, ...)
}
```

Figure 6.5: tidyr::wrap_error_names: definition and strictness signature

The signature causes code to be evaluated before entering the tryCatch block. The fix is easy; delay the evaluation of code so that any error raised by it can be caught by the surrounding tryCatch function. The test passes after removing the strict annotation on code.

Identifying and fixing the source of the errors from failing tests was relatively straightforward since failing tests clearly indicated the top-level functions from which to start the debugging process. In a program, such as an example or a vignette, it is challenging to identify the top-level function responsible for the error. The situation is even more complex for clients since they typically invoke functions indirectly, sometimes via dynamically generated code invoked from native code. Sometimes, intervening exception handlers capture the original error and raise a different error , masking the source of the error. Hence, I restricted this manual migration to just the package tests. In the real world, I expect developers to use tests to infer strictness, fix broken signatures manually, and require the clients to rewrite code that breaks from unexpected reliance on laziness, or enrich their test suite per client usage patterns. The goal should be to minimize the breakage of clients. Not all clients can be accommodated; there are many subtle ways in which they can introduce laziness. Satisfying all of them may create unnecessary technical debt on the package maintainers and incentivize undesirable API usage.

## 6.5 RELATED WORK

Turcotte et al. [30] empirically inferred type signatures for functions by observing the type of arguments and return values. These signatures were validated by inserting type checking code and monitoring failures on client programs. This approach inspired the strictness inference; however, types are easier to check than strictness. Types are checked by validating that if an argument is evaluated, it has the expected type. For strictness, I have to worry about the interplay of side-effects and changes to the order of evaluation of arguments.

## 6.6 CONCLUSIONS

This chapter used the migration tools to infer strictness signatures for R functions to capture both intentional and accidental laziness. The large-scale evaluation showed that over 99% of the inferred signatures were correct when tested against clients of the libraries. The package authors can subsequently refine these signatures. To elucidate the process, I perform this refinement manually for `tidyverse` packages and discuss the errors arising from the unsoundness of the tooling. The results shown in this chapter are encouraging. Automated migration of R to strict semantics can be performed with reasonably good accuracy.

# 7 CONCLUSIONS

In R, function arguments are not evaluated at the call-site, instead, the evaluation is suspended until the callee needs them. The definition of *need* is quite liberal here as, for example, local re-binding, returning, and many builtin functions are strict. As this dissertation shows, this leads to many programs being on the strict side of the spectrum for a lazy language. Why is R lazy at all? It turns out that allowing users to reflectively alter argument expressions, before evaluating them, is a very expressive and powerful meta-programming technique, enjoyed by many package authors in the R ecosystem to build, e. g.embedded domain-specific languages. It is part of what makes R appealing to its users, even if they do not realize that the language they use has a lazy core. However, the joy is limited when it comes to writing robust R code — as both the caller and the callee co-determine what a function actually does — and also when implementing the language itself. Taking everything into account, I believe that R should be strict by default, giving package authors the option to opt-in to laziness.

In this dissertation, I propose and evaluate a strategy for evolving R as an ecosystem to strict semantics. First, I provide strictness signatures as a non-invasive R extension to avoid changes to legacy code. Second, I automatically infer robust strictness signatures for package code by capturing the desired and accidental laziness of arguments passed to R functions, thereby allowing most of the client code to run unchanged — in my experiments, only 0.79% of all depending packages' tests failed. Such automatically generated strictness signatures can be subsequently refined by the package authors and users. I elucidate this by fixing the strictness signatures for `tidyverse`.

Changing R to a strict language would be beneficial in several ways. Implementations would become faster, compilers and program analyses would be easier to perform, users would be presented with a more commonly expected call semantics, and it would open up the path for further evolution. Currently, many standard techniques such as gradually typed function signatures and efficient just-in-time optimizations are difficult to apply to R because of laziness.

While the dissertation focuses exclusively on making R strict, it is worth mentioning that there is also an argument for strengthening laziness. In many ways, R is only *weakly* lazy, it forces promises in many places where other languages would not. The works of Wickham [36], Mühleisen, Bertram, and Kallen [21] and Kalibera et al. [17] suggest that more laziness can bring interesting optimization opportunities, especially when performing operation on large data objects.

Through this dissertation, I have addressed an instance of software migration. In most cases, this problem has been addressed in ad-hoc ways. My approach for migrating the R ecosystem to strict semantics can be generalized in the form of a recipe for language migration:

- Analyze code to assess the feasibility of changing the language feature.
- Identify incentives to encourage language users to adopt the change.
- Design tools to automate migration with good accuracy.
- Migrate popular packages to encourage adoption.

I believe these steps generally apply to a wide array of language migration problems. For instance, Python's migration would have benefited greatly if changes were introduced in Python 3 after assessing developers' needs so they could be incentivized to migrate. Automating the conversion and migrating popular packages such as NumPy and django would have significantly sped up Python migration. In stark contrast is the migration to TypeScript, which enjoyed much-needed additions to JavaScript and tooling support from the beginning.

# BIBLIOGRAPHY

[1] Lennart Augustsson. "The Interactive Lazy ML System". In: *Journal of Functional Programming* 3.1 (1993). DOI: 10.1017/S0956796800000617.

[2] J. W. Backus et al. "Revised Report on the Algorithm Language ALGOL 60". In: *Communications of the ACM* 6.1 (1963). DOI: 10.1145/366193.366201.

[3] Richard A. Becker, John M. Chambers, and Allan R. Wilks. *The New S Language*. Chapman & Hall, 1988.

[4] Martin Bodin, Tomás Diaz, and Éric Tanter. "A trustworthy mechanized formalization of R". In: *International Symposium on Dynamic Languages (DLS)*. 2018. DOI: 10.1145/3276945.3276946.

[5] Stephen Chang and Matthias Felleisen. "Profiling for Laziness". In: POPL '14 49.1 (2014). DOI: 10.1145/2578855.2535887.

[6] Robert Ennals and Simon Peyton Jones. "Optimistic Evaluation: An Adaptive Evaluation Strategy for Non-Strict Programs". In: *ICFP '03* 38.9 (2003). DOI: 10.1145/944746.944731.

[7] Aviral Goel, Jan Ječmen, Sebastián Krynski, Olivier Flückiger, and Jan Vitek. "Promises Are Made to Be Broken: Migrating R to Strict Semantics". In: *PACMPL* 5.Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA) (2021). DOI: 10.1145/3485478.

[8] Aviral Goel and Jan Vitek. "On the Design, Implementation, and Use of Laziness in R". In: *PACMPL* 3.Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA) (2019). DOI: 10.1145/3360579.

[9] Aviral Goel and Jan Vitek. "First-Class Environments in R". In: *International Symposium on Dynamic Languages (DLS)*. 2021. DOI: 10.1145/3486602.3486768.

[10] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585 (2020). DOI: 10.1038/s41586-020-2649-2.

[11] W. Huber et al. "Orchestrating high-throughput genomic analysis with Bioconductor". In: *Nature Methods* 12.2 (2015), pp. 115–121. DOI: 10.1038/nmeth.3252. URL: http://www.nature.com/nmeth/journal/v12/n2/full/nmeth.3252.html.

[12] Paul Hudak. "Conception, Evolution, and Application of Functional Programming Languages". In: *ACM Comput. Surv.* 21.3 (1989). DOI: 10.1145/72551.72554.

[13] Paul Hudak, John Hughes, Simon L. Peyton Jones, and Philip Wadler. "A history of Haskell: being lazy with class". In: *History of Programming Languages Conference (HOPL-III)*. 2007. DOI: 10.1145/1238844.1238856.

[14] John Hughes. "Why Functional Programming Matters". In: *The Computer Journal* 32.2 (1989). DOI: 10.1093/comjnl/32.2.98.

[15] Ross Ihaka and Robert Gentleman. "R: A Language for Data Analysis and Graphics". In: *Journal of Computational and Graphical Statistics* 5.3 (1996). DOI: 10.2307/1390807. URL: http://www.amstat.org/publications/jcgs/.

[16] Simon Peyton Jones and Will Partain. "Measuring the effectiveness of a simple strictness analyser". In: *Proceedings of the 1993 Glasgow Workshop on Functional Programming*. Workshops in Computing. 1993. DOI: 10.1007/978-1-4471-3236-3_17.

[17] Tomas Kalibera, Petr Maj, Floreal Morandat, and Jan Vitek. "A Fast Abstract Syntax Tree Interpreter for R". In: *Conference on Virtual Execution Environments (VEE)*. 2014. DOI: 10.1145/2576195.2576205.

[18] Filip Krikava and Jan Vitek. "Tests from traces: automated unit test extraction for R". In: *International Symposium on Software Testing and Analysis (ISSTA)*. 2018. DOI: 10.1145/3213846.3213863.

[19] Uwe Ligges. "20 Years of CRAN (Video on Channel9)". In: *UseR! Conference*. 2017.

[20] Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. "Evaluating the Design of the R Language: Objects and Functions for Data Analysis". In: *European Conference on Object-Oriented Programming (ECOOP)*. 2012. DOI: 10.1007/978-3-642-31057-7_6.

[21] Hannes Mühleisen, Alexander Bertram, and Maarten-Jan Kallen. "Database-Inspired Optimizations for Statistical Analysis". In: *Journal of Statistical Software* 87.4 (2018). DOI: 10.18637/jss.v087.i04.

[22] Robert Nystrom. *Crafting Interpreters*. Genever Benning, 2021. ISBN: 9780990582939. URL: https://craftinginterpreters.com/.

[23] Shawn T. O'Neil. "Implementing Persistent O(1) Stacks and Queues in R". In: *The R Journal* 7 (1 2015). DOI: 10.32614/RJ-2015-009.

[24] Chris Okasaki. "Simple and efficient purely functional queues and deques". In: *Journal of Functional Programming* 5.4 (1995). DOI: 10.1017/S0956796800001489.

[25] Simon L. Peyton Jones and André L. M. Santos. “A Transformation-Based Optimiser for Haskell”. In: *Sci. Comput. Program.* 32.1–3 (1998). DOI: 10.1016/S0167-6423(97)00029-4.

[26] Kent M. Pitman. “Special Forms in LISP”. In: *LISP Conference*. 1980. DOI: 10.1145/800087.802804.

[27] Vaughan R. Pratt. “Top down Operator Precedence”. In: POPL ’73. 1973. DOI: 10.1145/512927.512931.

[28] David Smith. “The R Ecosystem”. In: *The R User Conference 2011*. 2011.

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

[30] Alexi Turcotte, Aviral Goel, Filip Křikava, and Jan Vitek. “Designing Types for R, Empirically”. In: *Proceedings of the ACM on Programming Languages (PACMPL)* 4.Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA) (2020). DOI: 10.1145/3428249.

[31] David A. Turner. “A New Implementation Technique for Applicative Languages”. In: *Softw., Pract. Exper.* 9.1 (1979). DOI: 10.1002/spe.4380090105.

[32] David A. Turner. “Miranda: A Non-Strict Functional language with Polymorphic Types”. In: *Functional Programming Languages and Computer Architecture (FPCA)*. 1985. DOI: 10.1007/3-540-15975-4_26.

[33] Mitchell Wand. “The Theory of Fexprs is Trivial”. In: *Lisp and Symbolic Computation* 10.3 (1998). DOI: 10.1023/A:1007720632734.

[34] Yisu Remy Wang, Diogenes Nunez, and Kathleen Fisher. “Autobahn: Using Genetic Algorithms to Infer Strictness Annotations”. In: *SIGPLAN Not.* 51.12 (2016). DOI: 10.1145/3241625.2976009.

[35] Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag, 2016. URL: http://ggplot2.org.

[36] Hadley Wickham. *tidyverse: Easily Install and Load the 'Tidyverse'*. 2017. URL: https://CRAN.R-project.org/package=tidyverse.

[37] Hadley Wickham, Romain Francois, Lionel Henry, and Kirill Müller. *dplyr: A Grammar of Data Manipulation*. 2018. URL: https://CRAN.R-project.org/package=dplyr.

[38] Andrew K. Wright and Matthias Felleisen. “A Syntactic Approach to Type Soundness”. In: *Information and Computation* 115 (1992). DOI: 10.1006/inco.1994.1093.