DATA-DRIVEN ECOSYSTEM MIGRATION

Non-Intrusive Migration of R Ecosystem from Lazy to Strict Semantics

A thesis proposal by

AVIRAL GOEL

Northeastern University
Khoury College of Computer Sciences
Boston, Massachusetts, USA
May 2022

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" [19]

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.

To evolve a language with a substantial package ecosystem and a large user base requires a strategy that can scale while minimizing the impact on its users. In this thesis, I propose the following three-part data-driven strategy for *large-scale migration* of the R language ecosystem from lazy to strict-by-default and lazy-on-demand semantics with *minimal user-visible changes* and *good precision*:

1. Assess the feasibility of migrating to strict-by-default semantics on the R ecosystem by studying the use of laziness in legacy code.

2. Migrate legacy code at scale by developing tooling to automate transition to strict-by-default semantics with minimal impact on existing code.

3. Encourage adoption by identifying incentives to motivate R package developers to migrate their code.

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.

Large package ecosystems of contemporary mainstream languages are a double-edged sword. On the one hand, they are majorly responsible for languages' popularity and adoption; on the other hand, they deter experimentation and evolution. If fixing design mistakes and retrofitting new features in a language breaks millions of lines of otherwise functional code, they will be met with much resistance from the users. An even worse predicament is when partial adoption of changes splits the ecosystem into islands of incompatible package ecosystems.

I intend to migrate the R ecosystem from lazy to strict-by-default and lazy-on-demand in this work. I will focus exclusively on the migration of packages found in CRAN, the official package repository of R. Changes to the implementation and core packages are made by

the language developers; hence I exclude them from the migration process. I also exclude end-user scripts and notebooks 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.

1.1 R AND LAZINESS

This section provides a brief primer on the R language and the motivation for switching R's semantics from lazy to strict-by-default.

R is a vectorized, dynamic, lazy, functional, and object-oriented programming language with an unusual combination of features [18], 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 [16] as a successor to an earlier language for statistics named S [4]. Today, R is widely used in scientific computing domains such as data science.

In R, most data types are vectorized; it does not differentiate scalars from vectors. Environments, used as scopes, are first-class mutable maps with a reference to their lexical environment. Code can always access its local and lexical environments, but it is also possible to reflectively extract the environment of any function currently on the call stack [11]. Expressions are first-class objects that can be evaluated in any programmatically accessible environment using `eval` [8].

R has lexically-scoped higher-order functions. All functions, including the ones in loaded packages, can be redefined. This makes R flexible but challenging to analyze. In R, every linguistic construct is desugared to a function call, even control flow statements, assignments, and bracketing.

R uses the call-by-need evaluation strategy; function argument evaluation in R is delayed by bundling it in 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.

While R strives to be functional, it allows assignment to variables in any programmatically chosen scope. Apart from this, there are all sorts of external effects and no monads.

WHY BOTHER BEING LAZY? 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.

THE CASE FOR STRICTNESS 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.

1.2 RELATED WORK

Changing a language with a large codebase is challenging. I briefly discuss a few languages that have dealt with this problem in recent times.

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 [12] 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.

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/

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.

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.

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. Un-

fortunately, 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.

The problem of migration has also been addressed by the research community. Aggarwal, Salameh, and Hindle [1] attempted to use statistical machine learning to convert Python 2 to 3. Pradel et al. [21] described a tool for discovering types in Python programs as a combination of probabilistic type prediction and search-based refinement. Migration was also studied in the context of Java libraries [29], Android apps [7] and C++ applications [6]. One last relevant thread of work is the migratory typing of Tobin-Hochstadt and Felleisen [22] where a gradual type system is added to a variant of the Scheme programming language to enable gradual migration from untyped to typed code.

2 THESIS QUESTION

This thesis addresses the following question:

Thesis Question

How to *effectively* migrate R from a lazy to strict-by-default and lazy-on-demand language with *minimal impact* on its legacy code?

The answer to this question lies in the solution to these subproblems:

Q1 HOW WILL THE ECOSYSTEM BE IMPACTED?
Impact analysis helps assess the suitability of strict semantics for R and identify the affected parts of its ecosystem. An empirical analysis of the R package ecosystem can help determine the impact of switching to strict semantics.

Q2 HOW 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. An automation tool should perform a minimal rewrite of the affected code identified in the previous step.

Q3 HOW TO ENCOURAGE ADOPTION?
Clear incentives should be identified from the impact analysis of step one for the users to consider migration.The automation tool built in the second step should be used to migrate the popular packages to accelerate adoption.

These solutions lead to the following thesis proposal:

Thesis Proposal

I propose the following three-part data-driven strategy for an effective *large-scale migration* of the R language ecosystem from lazy to strict-by-default and lazy-on-demand semantics with *minimal user-visible changes* and *good precision*:

1. **Assess the impact of migration** to strict semantics by studying the use of laziness in legacy code.
2. **Migrate legacy code at scale** automatically using tooling with minimal changes to existing code.
3. **Encourage adoption** by identifying incentives for migration to new semantics and migrating popular packages.

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 [10]
   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 little supporting evidence to assert that programmers use laziness to avoid unnecessary computation or to operate over infinite data structures. by a small number of packages.

2. Promises Are Made to Be Broken [9]
   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.

I claim the following intellectual contributions from these papers:

Thesis 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.
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.
3. An 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.
4. An open-source, dynamic analysis pipeline, that infers strictness signatures for R functions to migrate them from lazy to strict-by-default and lazy-on-demand semantics and validates the robustness of these signatures against client code.

The following sections discuss these intellectual contributions vis-à-vis the three-step migration strategy.

3.1 DESIGN AND IMPLEMENTATION OF LAZINESS

In Goel and Vitek [10], I discuss the implementation of laziness through promises in GNU R, the official R VM, and a small-step operational semantics of R's call-by-need semantics. This contribution relates to the first step of my migration strategy – assessment of laziness in R.

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 the environment in which the promise was originally created, the `val` slot holds the result of evaluating the `exp`, and 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; else, `forced` is checked. If the `forced` flag is set, an exception is thrown to avoid recursive promise evaluation. If it is not set, it is set, and `exp` is evaluated in `env`. 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.

SEMANTICS The small-step operational semantics is described in the style of [28] for a core R language with promises. It provides an easy-to-follow account of R's call-by-need semantics. The entire semantics fits on a page, yet it is precise, unlike the prose description. I build upon the semantics of Core R [18], 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. 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 semantics has not been validated by testing.

INSIGHTS The key insight from this contribution is that the design of laziness in R is inconsistent. There are numerous randomly-enforced strictness points: **(i)** the right-hand side of assignments is strict, **(ii)** sequencing operator `a;b` evaluates both expressions `a` and `b`, **(iii)** the data structures are all strict, **(iv)** function returns are strict, **(v)** many core functions in R are strict, **(vi)** there are no lazy data structures, and **(vii)** object-oriented dispatch evaluates arguments strictly to obtain their class . On the one hand, this makes R a rather strict lazy language; most end-users perceive R as a strict language, utterly unaware of its lazy semantics. On the other hand, developers force argument strictness at function boundaries to regain predictability manually because

of inconsistently enforced strictness. This suggests that laziness, at least the way it is implemented, is a wrong design choice for R.

3.1.1 Related Work

I compare my semantics to the work of Bodin, Diaz, and Tanter [5]. My 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 the DLS'18 paper does not describe their treatment of laziness. Due to the size of the Coq codebase and lack of documentation, it was unclear how to align the two artifacts.

3.2 TOOL FOR ANALYZING THE USE OF LAZINESS

In Goel and Vitek [10], I release an open-source, carefully optimized, dynamic analysis pipeline consisting of an instrumented R interpreter and data analysis scripts to analyze the use of laziness in R packages. 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.

INSTRUMENTED R VM 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 related to laziness. On the face of it, this is not a difficult task. And in the end, I only need 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 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 many identical code fragments for managing these data structures duplicated in various files. I succeeded by a lengthy trial and error process.

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. 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. However, keeping these objects alive too long will increase the footprint and hinder any attempt at running multiple tracers on the same machine in parallel. To reduce the 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. However, 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 using 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, expensive join operations and synchronization issues between the database schema and the tracer forced me to abandon this approach and implement a custom format. As the event stream has substantial redundancy, I applied streaming compression on the fly. Compression yielded an average 10x saving in space and 12x improvement in loading time.

POST-PROCESSING This part of the pipeline analyzes the raw data. It is 4K lines of R code. This pipeline uses parallelism and map-reduce style analysis to handle a large volume of data in a reasonable time. The pipeline steps are:

1. **Prescan**: Scans the raw data directory to produce an index for the next step.
2. **Reduce**: Partially summarizes the raw data in parallel.
3. **Scan**: Creates a list of all the files successfully reduced.
4. **Combine**: Combines information from all the programs into a single data table per analysis question.
5. **Summarize**: Computes summaries of the merged data.
6. **Report**: Generate graphs and tables from an RMarkdown notebook.

3.2.1 Related Work

Like this dynamic analyzer, Morandat et al. [18] 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.

3.3 ANALYSIS OF THE USE OF LAZINESS

In Goel et al. [9], I present an empirical evaluation of 16,707 R packages on the use of laziness by programmers, the strictness of functions and their possible evaluation orders, and the life cycle of promises. This contribution relates to the first step of my migration strategy – assessment of laziness in R. I chose dynamic analysis for this assessment since alternative approaches suffer from serious drawbacks. One 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. It is important to note that the result of dynamic analysis depends on code coverage; the data reflects only the code paths that were executed. Because of this, the analysis may underestimate the use of laziness.

CORPUS The corpus used in this evaluation was assembled from the two main code repositories, namely CRAN [17] and Bioconductor [13]. Both are curated repositories; to be admitted, packages must contain use-cases and tests and the data needed to run them. The corpus consists of 14,762 CRAN packages and 3,087 Bioconductor packages. From this, the analysis pipeline extracted 232K scripts and 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. This corpus only includes code written by package authors, not end-user code. Anecdotal evidence suggests that end-users typically write straight-line calls to package functions for data analysis and visualization. They do not define functions; hence they are less likely to leverage laziness.

3.3.1 Laziness Usage Patterns

This section presents the results of the empirical study of laziness in the R language.

3.3.1.1 Life Cycle of Promises

Promises dominate the memory profile of R programs. In this corpus, they were the most frequently allocated object – 270.9 B promises accounting for 38% of the allocated objects were observed. They are short-lived; 80% were evaluated in the called function, and over 99% did not survive a single GC cycle. Only 17% of promises contained expressions, i. e.code that required evaluation. The vast majority of promises, 48%, contain a single symbol, e. g.x, to be looked up in the promise's defining environment, and the remaining promises contain constants. Most argument promises, 87.3% to be precise, were used only once; the remaining were forced and read multiple times. 79.7% of promises were evaluated in the immediate callee, and the rest were evaluated indirectly through symbols contained inside other promises. Overall most promises lead a rather mundane life.

3.3.1.2 Strictness

Of the 2.1 M distinct parameters to 388.3 K functions in the corpus, 87.6% were evaluated across all the calls to their corresponding functions, 6.0% parameters were evaluated in some calls and not others, and 6.4% parameters were never used. Out of the total 388.3 K functions, 83.7% were strict – these functions evaluate all their arguments in a single pre-ordained order across all calls. Furthermore, the majority of packages, 51%, had only strict functions; only 17% of packages had less than 75% strict functions. For the most part, R code appears to have been written without reliance on, and in many cases, even knowledge of, delayed argument evaluation.

3.3.1.3 Traditional Benefits of Laziness

The first traditional benefit of laziness is that programmers need not worry about the cost of unused arguments since they will not be evaluated under the call-by-need evaluation strategy. I compared the execution time of promises passed to parameters that were always evaluated against those that were only evaluated across some calls. While there was a slight difference in their execution profiles, the data did not confirm that programmers are taking advantage of laziness.

The second traditional use of laziness is that it makes it possible to compute over infinite data structures. While R does not provide such data structures, programmers can create them explicitly by leveraging first-class environments and call-by-need evaluation strategy. From a qualitative analysis of potential candidates, I found only a single instance of the explicit creation and use of a lazy data structure.

3.3.1.4 Meta-programming

Meta-programming in R is accomplished 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`. Overall, 0.7% of promises were meta-programmed in 2 K packages. Meta-programming is used for extending the language. The core R library implements some operators using meta-programming. Many programmers use the combination of meta-programming, first-class environments, and `eval` to design domain-specific languages for plotting and data analysis.

3.3.1.5 Laziness and Side-Effects

Laziness and side-effects lead to unpredictable code. Function arguments can be evaluated in different orders across different paths, effectively randomizing the order of side-effects. This problem is further exacerbated by the fact that most R users do not realize that it is a lazy language, leading to hard-to-understand bugs. R programmers work around this problem by explicitly evaluating the arguments on function entry in a particular order using the `force` function. Even the built-in higher-order functions in R use a similar function, `forceAndCall`. The `force` function was widely used to enforce strictness; it was called 101.8 M times and was used in 60% of the inspected packages.

While the call-by-need evaluation strategy is the default in R, this analysis suggests that it is used far less than one would expect. To deal with side-effects and manage programmers' expectation, many functions are stricter than they need to be. There is little evidence of

lazy data structures or that users leverage lazy evaluation to avoid unnecessary computation. The only major use of laziness is for meta-programming. These results make a compelling case for eliminating laziness from R.

3.3.2 Related Work

Lazy functional programming languages have a rich history. The earliest lazy programming language was Algol 60 [3] which had a call-by-name evaluation strategy. This was followed by a series of purely functional lazy languages [2, 24, 25]. The motivations for the pursuit of laziness were modularity, referential transparency and the ability to work with infinite data structures [15]. These languages inspired the design of Haskell [14].

The meta-programming support of R is reminiscent of fexprs [26] in Lisp. Fexprs are first class functions with unevaluated arguments. In R, functions always have access to their unevaluated and evaluated arguments. Pitman [20] 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.

Building upon the implicit argument quoting of promises is a data structure called `quosure`, 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. They are a central component of a collection of R packages for data manipulation, tidyverse, that have a common design language and underlying data structures. dplyr, a package of tidyverse, implements a DSL for performing SQL like data transformations on tabular data and ggplot2 implements a declarative language for graphing data, inspired by The Grammar of Graphics [27]. 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.

3.4 TOOL FOR INFERRING STRICTNESS SIGNATURES

In Goel et al. [9], 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

The first tool, LazR, uses dynamic analysis to observe argument usage and synthesizes strictness signatures for functions. The strictness signature of a function is a sequence of argument positions of that function that can be evaluated strictly. The second tool, StrictR, runs R code with strict semantics dictated by the signatures. It loads signatures from external files and applies them to R functions by inserting calls to the `force` function when packages are loaded. 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 also adds to the unsoundness of signatures.

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.

3.4.1 Related Work

Turcotte et al. [23] 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.

4 PROPOSED WORK

This chapter will discuss the incomplete parts of my migration strategy and my plan to address them. I plan to submit this work to PLDI'23.

4.1 AUTOMATING MIGRATION

Section 3.4 discussed the tools published in Goel et al. [9] to migrate the R ecosystem to strict semantics. While the signature inference has good accuracy for use in the real world, they are not "production-ready" yet. I intend to make the following improvements to the migration tooling to address the third step of my migration strategy thoroughly.

REFACTORING LazR outputs partially processed data in data files processed by external scripts to synthesize strictness signatures. It also keeps track of extraneous program state, which is irrelevant for generating signatures. I intend to refactor LazR to address these issues.

PACKAGING LazR is designed as an R package that extends my generic dynamic analysis framework, instrumentr, which builds on top of a modified GNU R VM, R-dyntrace. To ease distribution and setup, I want to package this with Docker and integrate it with continuous-integration services such as Github Actions.

UNSOUNDNESS To address LazR's unsoundness, I would like to include a mechanism for manually overriding the strictness signatures. I will leverage roxygen2 to enable developers to specify strictness information as specially formatted comments placed next to the corresponding function, as shown in the example below.

```
#' This function wraps base::if
#' @strict cond
my_if <- function(cond, yes, no) {
   if(cond) yes else no
}
```

SYNTAX Strictness signatures placed in external files can diverge from code over time. As shown below, I want to introduce an annotation mechanism to enable developers to specify strictness information as part of function declaration syntax.

```
my_if <- function(cond: strict, yes, no) {
   if(cond) yes else no
}
```

To accomplish this, I am developing a tool, rastr, that will preprocess this enhanced R code to produce regular R code and move the strictness annotations to the package's signature file. This approach is similar to TypeScript, but unlike TypeScript, the annotations may change the semantics of the code. More generally, rastr will support arbitrary annotations to code to enable other use-cases such as the addition of type signatures and C++-style class declaration syntax for R's object systems.

4.2 MIGRATING POPULAR R PACKAGES

The next piece of proposed work aims to address the third step of the migration strategy. While I have identified two incentives, predictability [10] and performance [9], to encourage developers to migrate to strict semantics, I have not yet migrated the popular packages of the R package ecosystem. For this, I intend to focus on tidyverse, an extremely popular collection of 29 R packages used for data science. I have discussed this prospect with the developers of tidyverse, an influential group of R package authors; they support my vision for R and are interested in using my tool to migrate the tidyverse packages. On account of being widely used, these packages are the most common dependencies in the ecosystem. Hence, the R ecosystem's *effective* migration rests mainly on the successful migration of tidyverse packages. Furthermore, on account of being well-tested, these packages will likely help identify bugs in the automated migration. Focusing only on the 29 tidyverse packages will allow me to investigate the cases where automated migration fails and derive heuristics and guidelines for debugging and fixing them. This should lead to a better understanding of the impact of unsoundness on migration and improvements in the design of the tool.

5 CONCLUSIONS

This thesis presents a strategy for migrating the R ecosystem at scale from lazy to strict semantics. The strategy builds on three questions:

Q1 HOW WILL THE ECOSYSTEM BE IMPACTED?
Through a large-scale analysis of the use of laziness in Goel and Vitek [10], I discovered that much of R code does not rely upon laziness, with meta-programming being the only significant use of laziness in R packages. In fact, due to unrestricted side-effects, R programmers enforce argument strictness at function boundary to regain a predictable argument evaluation order. With the exception of meta-programming, the R ecosystem will benefit from the removal of laziness.

Q2 HOW TO MIGRATE THE ECOSYSTEM AT SCALE?
I developed tools to migrate R packages to strict semantics in Goel et al. [9] with good accuracy. LazR generates strictness signatures in external files, requiring no modifications to package code, which are read off at runtime by StrictR to evaluate arguments at the function boundary. Polishing these tools to prepare them for use by end-users is future work.

Q3 HOW TO ENCOURAGE ADOPTION?
I have identified two clear incentives for the developers to consider migration to strict semantics: predictable argument evaluation [10] and performance improvement [9]. Migrating popular R packages from tidyverse is future work.

5.1 SCHEDULE

I intend to finish the proposed future work by October and submit it to PLDI'23. After that, I will focus on the thesis, which I expect to be completed by the end of January 2023.

Month	**Task**
June-October	Migration
November	Paper
December-January	Thesis

BIBLIOGRAPHY

[1] Karan Aggarwal, Mohammad Salameh, and A. Hindle. "Using machine translation for converting Python 2 to Python 3 code". In: *PeerJ Prepr.* 3 (2015). DOI: 10.7287/PEERJ.PREPRINTS.1459V1.

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

[3] 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.

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

[5] 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.

[6] B. Collie, P. Ginsbach, J. Woodruff, A. Rajan, and M. F. P. O'Boyle. "M3: Semantic API Migrations". In: *Conference on Automated Software Engineering (ASE)*. 2020. DOI: 10.1145/3324884.3416618.

[7] Mattia Fazzini, Qi Xin, and Alessandro Orso. "APIMigrator: An API-Usage Migration Tool for Android Apps". In: *International Conference on Mobile Software Engineering and Systems (MobileSoft)*. 2020. DOI: 10.1145/3387905.3388608.

[8] Aviral Goel, Pierre Donat-Bouillud, Filip Křikava Christoph M. Kirsch, and Jan Vitek. "What We Eval in the Shadows: A Large-Scale Study of Eval in R Programs". In: *Proceedings of the ACM on Programming Languages (PACMPL)* 5.Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA) (2021). DOI: 10.1145/3485502.

[9] 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.

[10] 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.

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

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

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

[14] 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.

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

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

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

[18] 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.

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

[20] Kent M. Pitman. "Special Forms in LISP". In: *LISP Conference*. 1980. DOI: 10.1145/800087.802804.

[21] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. "TypeWriter: Neural Type Prediction with Search-Based Validation". In: *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESE/FSE)*. 2020. DOI: 10.1145/3368089.3409715.

[22] Sam Tobin-Hochstadt and Matthias Felleisen. "Interlanguage Migration: From Scripts to Programs". In: *Companion to the Symposium on Object-Oriented Programming Systems, Languages, and Applications*. 2006. DOI: 10.1145/1176617.1176755.

[23] 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.

[24] David A. Turner. "A New Implementation Technique for Applicative Languages". In: *Software: Practice and Experience* 9.1 (1979). DOI: 10.1002/spe.4380090105.

[25] 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.

[26] Mitchell Wand. "The Theory of Fexprs is Trivial". In: *Lisp and Symbolic Computation* 10.3 (1998). DOI: 10.1023/A:100772063.

[27] Leland Wilkinson. *The Grammar of Graphics (Statistics and Computing)*. Berlin, Heidelberg: Springer-Verlag, 2005. ISBN: 0387245448.

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

[29] S. Xu, Z. Dong, and N. Meng. "Meditor: Inference and Application of API Migration Edits". In: *International Conference on Program Comprehension (ICPC)*. 2019. DOI: 10.1109/ICPC.2019.00052.