# Promises Are Made to Be Broken

Migrating R to Strict Semantics

AVIRAL GOEL, Northeastern University, USA
JAN JEČMEN, Czech Technical University, Czechia
SEBASTIÁN KRYNSKI, Czech Technical University, Czechia
OLIVIER FLÜCKIGER, Northeastern University, USA
JAN VITEK, Czech Technical University and Northeastern University, USA

Function calls in the R language do not evaluate their arguments, these are passed to the callee as suspended computations and evaluated if needed. After 25 years of experience with the language, there are very few cases where programmers leverage delayed evaluation intentionally and laziness comes at a price in performance and complexity. This paper explores how to evolve the semantics of a lazy language towards strictness-by-default and laziness-on-demand. To provide a migration path, it is necessary to provide tooling for developers to migrate libraries without introducing errors. This paper reports on a dynamic analysis that infers strictness signatures for functions to capture both intentional and accidental laziness. Over 99% of the inferred signatures were correct when tested against clients of the libraries.

CCS Concepts: • **General and reference** → **Empirical studies**; • **Software and its engineering** → **General programming languages**; **Scripting languages**; *Semantics.*

Additional Key Words and Phrases: R language, delayed or lazy evaluation

**ACM Reference Format:**
Aviral Goel, Jan Ječmen, Sebastián Krynski, Olivier Flückiger, and Jan Vitek. 2021. Promises Are Made to Be Broken: Migrating R to Strict Semantics. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 101 (October 2021), 20 pages. https://doi.org/10.1145/3485478

## 1 INTRODUCTION

The R programming language is widely used in data science. Yet many users are unaware that function calls have lazy semantics: arguments are suspended computations that are evaluated if and when they are needed. Goel and Vitek [2019] provided a thorough observational study of R packages. Most packages are written without reliance on laziness with the exception of meta-programming. This paper argues that laziness should be the exception in R. We propose to migrate R programs to a *strict-by-default, lazy-on-demand* semantics. We investigate whether one can switch the semantics of a language without causing undue breakage in legacy code that is in daily use. Even if programmers do not avail themselves of laziness, their code may accidentally depend on it.

Authors' addresses: Aviral Goel, Northeastern University, USA; Jan Ječmen, Czech Technical University, Czechia; Sebastián Krynski, Czech Technical University, Czechia; Olivier Flückiger, Northeastern University, USA; Jan Vitek, Czech Technical University and Northeastern University, USA.

2475-1421/2021/10-ART101
https://doi.org/10.1145/3485478

*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 effectful arguments, the order of effects is hard to predict. Pure functional languages prevent this by controlling effects from their type system. While R users do not have types, they often force evaluation at function boundaries. 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. Furthermore, object-oriented multiple dispatch evaluates arguments strictly to obtain their class. The costs of laziness come from having to box each argument in data structure that must be allocated and freed, and the hindering of compiler optimizations due to the side-effects from evaluating arguments.

*The case for laziness.* The success of a programming language comes down to the strength of its ecosystem. With tens of millions of lines of library code, any change risks breaking something. Preserving the status quo is a pragmatic choice to protect an ecosystem. There is also a genuine need for lazy evaluation: it is the building block of R's meta-programming facilities. Unevaluated arguments can be coerced back to their source code, that code can be modified and evaluated in an environment of the programmer's choice. Meta-programming is used for extending the language and to create embedded domain-specific languages. While one could imagine using macros instead, the number of libraries that would have to be refactored would be significant.

*A pragmatic path forward.* How does one migrate an entire ecosystem? Our viable migration path abides by four tenets: (1) minimal legacy code changes, (2) semi-automated migration, (3) testable outcomes, and (4) measurable benefits. We envision a migration path with the following characteristics. Changes to legacy code are avoided by non-invasive extensions to the language's syntax and semantics and limited to library functions that need lazy semantics. Strictness signatures are inferred from the tests associated with every R package. For popular libraries, client code is used as an oracle to check the correctness of inferred signatures. Performance and memory footprint are improved by compiler optimizations that leverage the new semantics. This paper aims to evaluate the following components of this migration path:

— **StrictR:** a prototype R interpreter with strict-by-default semantics; only functions that demand lazy evaluation need to be annotated.
— **LazR:** a scalable infrastructure for inferring strictness annotations for function arguments by dynamic program analysis.
— **$\check{\mathbf{R}}_s$:** an optimizing just-in-time compiler for R modified to take advantage of strict-by-default semantics.

Specifically, it should be the case that changes to the code are small, that inference is accurate, and that there are performance improvements. We conduct two experiments that aim to investigate whether such a semi-automated path is viable. First, we obtain the most widely used packages in the R ecosystem and, with LazR, leverage their regression tests to infer strictness signatures. Since StrictR can load signatures from external files, no changes to code are required. We then use clients of those packages to test the accuracy of the inference results. Only 0.79% of the client programs executed by StrictR exhibit observable differences from the reference semantics. Our second experiment is independent of the first. We take a suite of small computational tasks, manually annotate them for maximal strictness, and run the benchmarks in strict and lazy modes using $\check{\mathrm{R}}_s$ . We observe speedups ranging from 0.92× to 1.95× for the strict version.

Our conclusions are that the approach is viable as user-visible changes are small and the precision of the inference is promising. For community review, we intend to further reduce the error rate by improving the precision of our analysis and using all available clients to infer signatures rather than only the tests bundled with a package. The performance numbers should be viewed as a limit study as the baseline is a compiler that already performs aggressive optimizations. We have not evaluated the benefits of StrictR on the GNU R reference implementation. It is conceivable that the relative speedups could be more impressive than in our native compiler, but in the long run, we expect users to move away from bytecode interpretation.

*Availability.* Our software is at: https://doi.org/10.5281/zenodo.5394235

## 2 BACKGROUND

This section sets the stage by introducing related work and giving a brief overview of R.

### 2.1 Related Work

There are three clusters of related results: research on adding and removing laziness in functional languages, research on the R language, and approaches to language migration.

*Call by need.* 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) [Ennals and Jones 2003; Jones and Partain 1993; Peyton Jones and Santos 1998]. 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 et al. [2016] 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 [2014] 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 [Okasaki 1995], monadic parser combinators, and an AI game. Our work is similar to Autobahn in that we infer annotations dynamically and we do not guarantee soundness. We depart from Autobahn in that we use dynamic analysis of execution traces to determine strictness. Furthermore, we must deal with side-effects and reflective operations which adds extra complexity to our inference algorithm.

*The R language.* Goel and Vitek [2019] investigated the design and use of laziness in R. They provide a detailed account of the language's evaluation strategy with a small-step operational semantics and an empirical evaluation of laziness in 16,707 packages. Their study shows that most of the R code is written without reliance on, or awareness of, laziness. Out of 388K functions, 83% evaluate all of their arguments in a single order across all calls. The authors found that programmers sometimes force evaluation of arguments at the beginning of a function to protect their code from non-deterministic effects, this is done by adding calls to `force` to every argument. Only a single instance of a lazy data structure could be found in all the programs the authors inspected. The main *raison d'être* for delayed evaluation seems to be meta-programming which requires delayed arguments that can be manipulated symbolically. In that work, a function was deemed to be strict if it had a single evaluation order for its arguments. Our approach is inspired by that work, but we do not require a single evaluation order for a function. Instead, we choose to look at clients and declare a parameter lazy if some clients call the function with effectful arguments. Turcotte et al. [2020] 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 our 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, we have to worry about the interplay of side-effects and changes to the order of evaluation of arguments.

*Language migration.* Changing a language with a large codebase is challenging. No migration has been more fraught than that of Python. Python 3 was released in 2008 with many changes that broke backward compatibility and no automated upgrade path. Aggarwal et al. [2015] attempted to use statistical machine learning to convert Python 2 to 3. Pradel et al. [2020] described a tool for discovering types in Python programs as a combination of probabilistic type prediction and search-based refinement. Another popular migration is the adoption of *strict mode* for JavaScript [Wirfs-Brock and Eich 2020], a dialect of JavaScript that fixed some historical mistakes in the language. To encourage adoption, *strict mode* was designed to be subtractive; it only removed problematic features and did not add new features. This enabled *strict mode* code to run on browsers that did not support it. Migration was also studied in the context of Java libraries [Xu et al. 2019], Android apps [Fazzini et al. 2020] and C++ applications [Collie et al. 2020]. A more successful experience is the migration from PHP to Hack at Facebook. The key to success was a close feedback loop between language changes and their impact on the ecosystem at large. As all Hack users share an employer and a source code repository, it was possible to test changes and develop tools targeted at relevant usage patterns.[1] One last relevant thread of work is the migratory typing of Tobin-Hochstadt and Felleisen [2006] where a gradual type system is added to a variant of the Scheme programming language to enable gradual migration from untyped to typed code.

## 2.2 Laziness and the R Language

The R language is widely used in data science. R is a vectorized, dynamic, lazy, functional, and object-oriented programming language [Morandat et al. 2012], designed to be easy to learn by non-programmers and to enable rapid development of new statistical methods. The language was created in 1993 by Ihaka and Gentleman [1996] as a successor to an earlier language for statistics named S [Becker et al. 1988].

*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, these can refer to other parameters. R functions are higher-order. 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

[1]Private communication with the developers of Hack.

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 (environments in R are first-class objects). 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
```

R allows programmatic manipulation of ASTs, which are themselves first-class objects. They are evaluated using the `eval(exp,env)` function. In R, the local scope of a closure is a first-class mutable map.

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. 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.* Logically function arguments are passed by value to facilitate equational reasoning; the implementation optimizes this with a simple 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. 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.

*Delayed Evaluation.* The combination of side effects, frequent interaction with C, and absence of types has pushed R to be more strict than other lazy languages. Strictly speaking, R is not lazy as it evaluates some arguments that it does not need to. Let us review its design. Arguments to a 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 that takes an 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 only once, the second access to the promise uses the memoized result.

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

A promise associated to the default value of a parameter has access to all variables in scope, including other parameters. Promises cannot be forced recursively, if this were to happen the R interpreter terminates execution of the expression. Promises are mostly encapsulated and hidden from user code. R only provides a small interface for operating on them:

**`delayedAssign(x,exp,env,aenv)`**: creates a promise with body `exp` and binds it to variable `x` (here `x` is a symbol) in the environment `aenv`, evaluation of the promise will take place in environment `env`.

**`substitute(e,env)`**: substitutes variables in the expression `e` with their values found in environment `env`, returns an `expression`.

**`force(x)`**: forces evaluation of its argument. This replaces a common programming idiom, `x<-x`, which forces `x` by assigning it to itself.

**`forceAndCall(n,f,...)`**: calls `f` with the arguments specified in the varargs, of which the first `n` are forced before the call.

There is implicit forcing of values returned by functions and on the right-hand side of assignments. In addition, many core functions are strict. R code can include extensions written in C, these are able to freely access and mutate the expression stored in a promise, as well as the memoized result of promises. Legacy C code, such as mathematical functions, typically expects unwrapped values and not promises. While R does not provide built-in lazy data structures, they can be encoded. For instance, to return an unevaluated promise from a function, it can be wrapped in a newly created environment.

## 3 STRICTR: AN R INTERPRETER WITH STRICTNESS SIGNATURES

StrictR is a prototype implementation of R with a strict-by-default semantics and strictness signatures to specify which arguments should remain lazy. Our implementation is a prototype in the sense that it is not looking to improve performance, instead, its purpose is to allow us to experiment with the semantics of the language. The implementation uses source-to-source rewriting, and is, itself, loaded as a package in an unmodified GNU R interpreter. Strictness information is provided by external signature files to avoid having to modify the source code of programs and also enable easy experimentation with the signatures.

*Strictness Signatures.* A signature file contains strictness signatures for functions of one or more packages. The format of signatures is of the form

$$sig \ ::= \ \text{strict `pack :: fun`} \ \langle i_1, i_2, i_3, \dots \rangle$$

Here, pack::fun is the name of the package and the function. The sequence of integers specifies which argument positions are evaluated strictly. This format is meant to be generated by our tools. A drawback of signature files is that inner functions can not be annotated as they have no names. Only functions exported to a package namespace have a canonical name. We considered heuristics, but some packages dynamically create code in package load hooks, this makes it hard to identify the correct name. Luckily, these cases are relatively rare.

*Execution.* Functions are source-to-source rewritten after the package is loaded. The rewriting is simple; each function must `force` every argument that was not declared to be lazy. Parameters with missing arguments, i.e., no default and supplied arguments, are not forced. The implementation uses a feature of R that allows registering callbacks when packages are loaded. StrictR registers a callback that reads the signature file in the current directory or on the load path. Then, after the package is loaded, StrictR injects code in its functions in accordance with their signature. An earlier implementation mutated function bodies. This resulted in failures as the same function object can occur with different names and different signatures. For instance, in the `rlang` package, `is_same_body` aliases `is_reference`. Mutating `is_reference` to make arguments strict inadvertently also makes the function `is_same_body` strict. Further discussion will clarify why this is undesirable. To avoid this, StrictR copies functions as it rewrites them.

*Intrinsic laziness.* When should an argument be lazy? We have 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. Our current choice is pragmatic but imprecise; all varargs remain lazy.

*Order of evaluation.* A design choice we faced was to select an evaluation order for strict arguments. One can retain the evaluation order observed in the function if unique, but this is obscure to programmers. Another reasonable choice is to evaluate strict arguments left-to-right at the call site and evaluate strict default values left-to-right in the function prologue. This may be natural to users who wrote the call and had control over the order of arguments and can reason about their potential side effects. For ease of implementation, StrictR picks a third alternative. We 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, something that can be awkward as functions can have upward of 20 arguments. This is convenient as StrictR rewrites function bodies and not call sites.[2]

## 4 LAZR: A SCALABLE INFRASTRUCTURE FOR INFERRING STRICTNESS

LazR is a pipeline for inferring strictness signatures for legacy R libraries at scale. LazR 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 of this analysis is not to infer maximally strict signatures but rather minimize the impact of semantic change on clients by considering both intrinsic and accidental laziness. We elucidate this point in Fig. 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, we 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)
}
```

*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)
```

*Client code*

Fig. 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 [Goel and Vitek 2019]. 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

[2]There is a way to implement the second alternative as GNU R keeps a PROMARGS list which stores promises in the same order as at the call site.

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; we summarize them here:

- **V**: A vararg parameter; we 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. We 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 our current implementation treats them all strong signals and conservatively makes a parameter lazy at the first signal. Our design for StrictR allows us to easily explore different merging strategies.

Consider Fig. 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.* 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 our 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.

## 5 Ř$_s$: A STRICT-BY-DEFAULT COMPILER

We considered several approaches to evaluate the performance of a change of semantics. While one could expect that StrictR would be extended to optimize strict functions, this was not straightforward in our proof of concept implementation which does source-level rewriting. To achieve performance in the GNU R implementation would have required changing the bytecode compiler and the associated bytecode interpreter. While possible, the benefits one would measure with such an approach would be bounded by the speed of interpreted code. In the long run, R implementations should generate native code. The alternative was to use an experimental native compiler. There are two available in open source, Oracle's Truffle implementation of R [Stadler et al. 2016], and our own Ř [Flückiger et al. 2019]. Both systems have their drawbacks. Truffle is not yet a complete implementation of the language and it is unfamiliar to the authors. Ř is less mature, in terms of optimizations, but runs all R programs. For convenience, we picked Ř as a starting point.

One advantage of Ř is that it is based on the GNU R reference implementation and can always defer to GNU R when it hits codes that is not supported. It introduces an additional two-tiered compilation strategy. The first tier is realized by a bytecode interpreter, the second tier by an optimizing native compiler that relies on LLVM for code generation. The compiler employs, among many other optimizations, speculative inlining of R closures and promises [Flückiger et al. 2019, 2020]. Choosing Ř allows us to better evaluate the impact of laziness. In Ř we can both measure the impact on an interpreter with few optimizations applied to the bytecode, and an optimizing compiler that already does its best to elide promises when the semantics of the language allows it.

We build Ř$_s$ as a minimal adaptation of Ř that lets us evaluate performance benefits. We changed the first-tier compiler that generates the Ř bytecode (which differs from the GNU R bytecode) to strictly evaluate arguments, unless they are marked as lazy. This strict bytecode also serves as the source code for the optimizing tier. The changes in the lower tier were minimal. The benefits we show with Ř$_s$ are a lower bound on improvements as we do not factor in strictness in the analysis passes and other optimizations of the compiler.

## 6 INFERENCE EXPERIMENT

The goal of the inference experiment is to gauge the actual need for laziness in legacy code and the robustness of our inference algorithm. LazR is scalable, it can handle all the packages in CRAN, but the amount of data to process can reach multi-terabyte sizes. LazR adopts a simple map-reduce style. Analysis is split in phases shown in Fig. 2.

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, we mirror repositories, install their packages, run the script to generate traces. These traces are analyzed to output tabular data files and strictness signatures. Whenever possible, we parallelize the steps. Our experiments were performed on two Intel Xeon 6140, 2.30GHz machines with 72 cores and 256GB of RAM each.

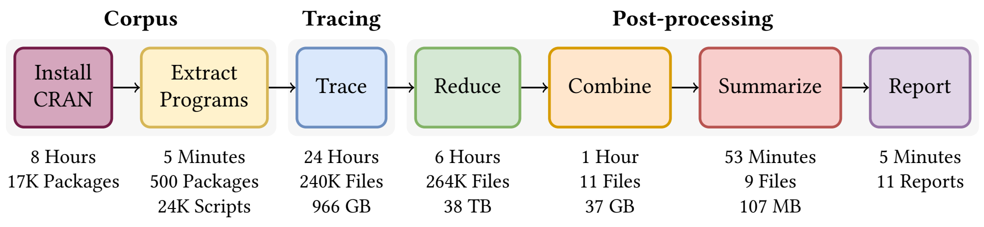

Fig. 2. Analysis Pipeline

## 6.1 Corpus

We selected a corpus of 500 packages with most client packages from CRAN [Ligges 2017].The packages 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 1. Corpus

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

Table 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.

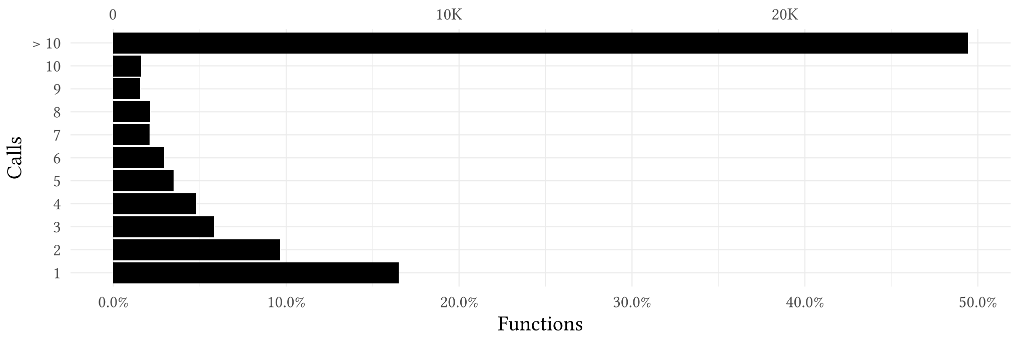

Fig. 3. Call Distribution

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. 3; 49% of functions are called more than ten times, while 17% are called only once. 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. 4; 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.

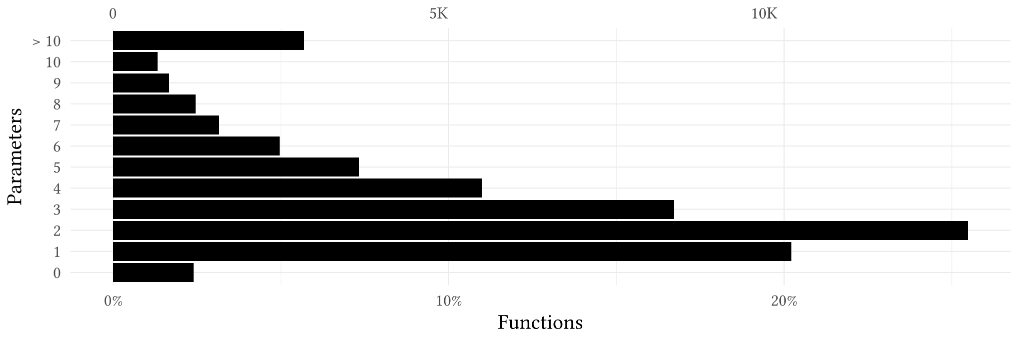

Fig. 4. Parameter Distribution

## 6.2 Inferred strictness signatures

There are three signals we 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 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 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 |

We observe that 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. We 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 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. For details about substitute, we refer the reader to [Goel and Vitek 2019]. 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 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 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.

Table 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 |

```
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 5 has numbers for both categories. 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.

Table 5. Unevaluated

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

```
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 our 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
   #...
```

## 6.3 Robustness of inferred signatures

This section reports on the robustness of strictness signatures generated by LazR. The intuition is that while we used the tests included in our corpus for inference, we need other tests to validate that the results are not limited to the code we trained on. Each package in our 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 our corpus, we select 2000 packages for this experiment.

Table 6. Client Corpus

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

Fig. 5 shows the steps performed in the experiment. First, we extract runnable code from the clients. Table 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.

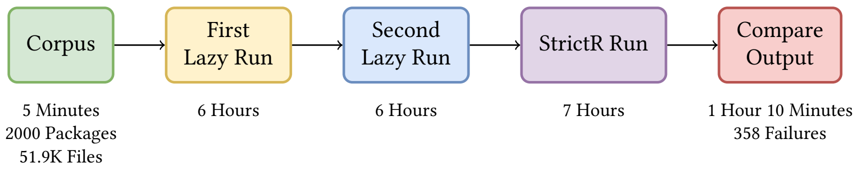

Fig. 5. Validation Pipeline

With all the signatures applied, we observed 3,139 scripts produce erroneous output. We 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. We 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. We 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.

## 7 PERFORMANCE EXPERIMENT

Our hypothesis is that strict semantics lead to faster programs. This might surprise readers who expect call-by-need to avoid unnecessary computation. Delaying computations is complex and hinders performance in two ways. First, it leads to more allocation. Second, it obstructs compiler analyses and optimizations. We expect the hypothesis to hold for a just-in-time compiler that uses advanced optimizations like Ř. In particular, we expect: (1) strict semantics to improve the performance of most benchmarks, for both tiers of Ř; (2) a portion of the speedup to come from reduced garbage collection; (3) and an additional speedup due to improved compiler optimizations. We present our evidence in support of (1) and (3), and partial evidence for (2).

*Methodology.* We conducted a limit-study experiment to assess the highest achievable performance given the current state of the compiler. We manually picked the strictest, semantically-correct, signatures for our programs. Only a handful of functions, such as `tryCatch`, had to be kept lazy. Table 7 gives our benchmarks. We picked a subset of the Ř benchmark suite which includes one variant of each benchmark. We acquired one real-world program, `flx`, and ported three programs from the *Are we fast* suite to R. All of these are available with our artifact.

Experiments are run on a dedicated benchmarking machine, with all background tasks disabled. The system features an Intel i7-6700K CPU, stepping 3, microcode 0xe2 with 4 cores and 8 threads. The system has 32 GB of RAM and runs Ubuntu 18.04 on a 4.15.0-136 Linux kernel. For ease of use, experiments are built as containers, based on Ubuntu 20.04, and executed on the Docker runtime 20.10.5, build 55c4c88. We verified the overhead introduced by the containerization to be uniform. All measurements are recorded repeatedly and we keep a historical record to spot unstable behavior. This led us to exclude the `convolution` benchmark as it appears to have a bi-modal performance profile, likely caused by the LLVM backend. Performance measurements are gathered by running $t_e$ invocations of Ř on each benchmark. Within each invocation we measure the execution time of $t_i$ in-process invocations. For each invocation, the first 5 in-process iterations are discarded to exclude warmup behavior. Aggregate numbers are reported as the speedup over the arithmetic mean of the execution times. Multiple speedup numbers are averaged using a geometric mean. As a baseline we use Ř which already has a 1.89× mean speedup over GNU R on our subsetted benchmark suite, with parameters $t_e = 1, t_i = 15$. Speedup ranges between 0.64× and 56×.

Table 7. Benchmarks

| Id | Benchmark | Suite |
|---|---|---|
| bnc | Bounce | Are we fast |
| mnd | Mandelbrot | Are we fast |
| sto | Storage | Are we fast |
| flx | Flexclust | Real thing |
| bin | Binarytrees | Shootout |
| fst | Fasta | Shootout |
| far | Fastaredux | Shootout |
| fnk | Fannkuchredux | Shootout |
| knu | Knucleotide | Shootout |
| nbo | Nbody | Shootout |
| pdg | Pidigits | Shootout |
| rgx | Regexdna | Shootout |
| rev | Reversecomplement | Shootout |
| spn | Spectralnorm | Shootout |

*Speedup.* First, we compare Ř against $Ř_s$ . This experiment estimates the end-to-end improvement on performance that a change to strict semantics in the R language would have on Ř. The execution times were measured with $t_e = 4, t_i = 25$. Figure 6 shows a boxplot for the speedups of $Ř_s$ along the X-axis, normalized with respect to Ř (lazy). Outliers are represented by black dots. Overall, we

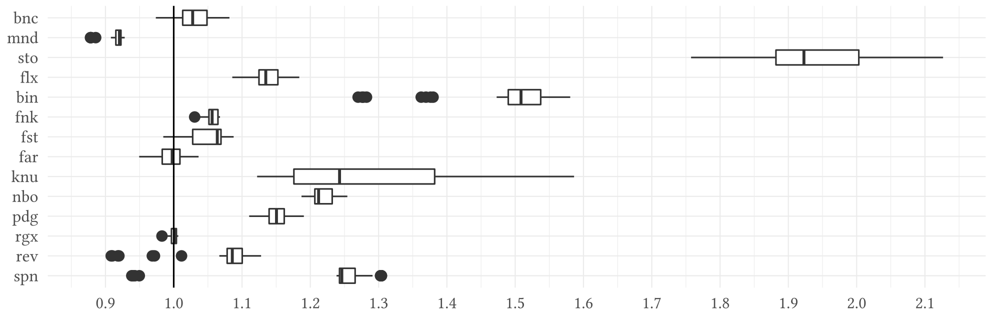

Fig. 6. Speedup

observe a mean speedup of 1.17×, ranging from 0.92× to 1.95×. For 9 out of 14 benchmarks we measure a significant increase in performance.

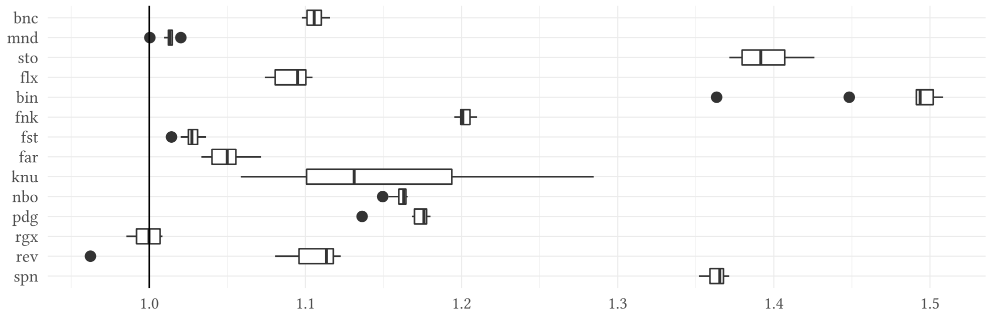

Fig. 7. Speedup without optimizations

Then, we repeated the performance experiment, but with the second tier optimizer completely disabled. In other words, we compare non-optimized variants of Ř$_s$ vs. Ř. Since this variant executes overall 0.38× slower, we chose to only run with $t_e = 1, t_i = 15$. The results are presented in Figure 7. We found that the bytecode interpreter also gains a speedup of 1.16×, ranging from 1.00× to 1.48×. Thus, we conclude that in our benchmark suite both a naive interpreter and a speculatively optimizing native compiler achieve better performance on a strict dialect of R. Even though the speedup is very similar in numbers, the reasons seem to be different at times. We will get back to this point in the discussion at the end of this section.

*Garbage Collection.* We measure the number of promises allocated per iteration for GNU R, Ř, and Ř$_s$ . The results are shown in Figure 8. We report on the geometric mean of the reductions in each benchmark. The first jump from GNU R's bytecode interpreter to the optimizing just-in-time compiler Ř, leads to a 91% reduction in allocations. The number is reached by taking the geometric mean of light-gray bars, yielding 9%. Therefore, the reduction is 91%. Note that for some benchmarks the light-gray bar is too small to be seen and replaced by the actual number. The second step from lazy to strict semantics leads to 2 benchmarks not allocating any promises at all. On the remainder, we see an additional 96% reduction (Ř$_s$ normalized to Ř).

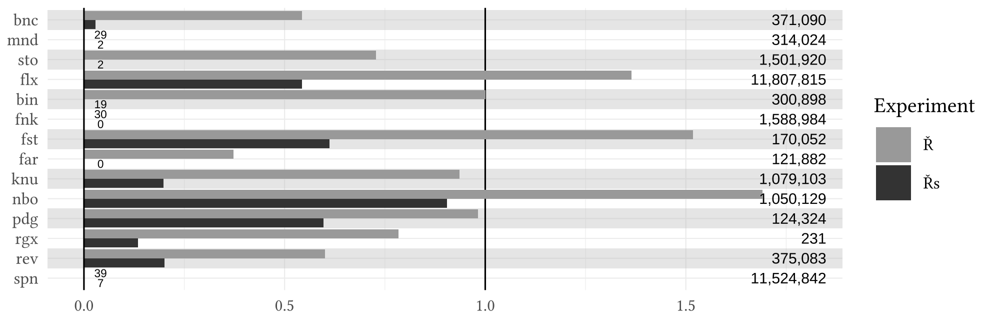

Fig. 8. Promises allocated, normalized to GNU R. The right column shows GNU R's actual promises. If the bar is too small to display, we show the actual number of promises in the respective experiment.

Overall, 2 benchmarks reduce to zero allocations, the rest reduce on average by 99%. To reach this number, once again we compute the geometric mean of dark bars, yielding 1%. Therefore, the reduction is 99%. The lowest reduction observed is 9.5%. Surprisingly the number of remaining promises is still relatively high in some cases. As far as we were able to observe, they originate largely from special forms, *i.e.*, R builtins with a custom evaluation strategy, that are not yet natively supported in the Ř bytecode.

It turns out that even an optimizing compiler has to allocate many promises for R code, as oftentimes, they cannot be eliminated entirely. Ř$_s$ allows for a larger reduction in the number of promises allocated. Thus, we expected a significant portion of the overall speedup to originate from the reduced allocation rate. We measured the differences in garbage collection time and it ranged from 50% to 190%, but found the contribution to the overall speedup to be smaller than expected. The GNU R garbage collector, which is reused in Ř, has a fairly slow allocation path, which includes mutating a doubly-linked list. Therefore, some portion of the speedup could be due to the saved time in the allocation of promises, which is not counted in the GC time. Therefore, we conducted an additional experiment, where we evaluated arguments to calls strictly, but additionally allocated a promise only to be subsequently discarded. This configuration led to an overall speedup of 1.1× instead of 1.16×, suggesting that about 38% of the performance improvement in the bytecode interpreter is due to the reduced allocation. Unfortunately, a similar experiment cannot be as easily conducted for the optimizing tier of Ř, since its compiler would just optimize away all unnecessary allocations. We conjecture that the contribution should be even smaller in that case, because it allocates much fewer promises to start with.

*Discussion.* In both tiers, we observed a proportional reduction of time spent tracing the heap and allocating promises. Surprisingly, that was not the main reason for the speedups. We investigated the remaining difference using the `perf` profiler and found that the overheads of lazy evaluation are to be found in setting up the execution context for promise evaluation. This includes marking it as under evaluation, to detect recursive evaluation dependencies, and either calling the interpreter's main eval-loop on the code of the promise or calling the compiled function. In the case of the native backend, having the promises inlined at the call-site instead of in a separate native function invoked by the callee, resulted in fewer instruction cache misses. We also found some instances where both tiers sped up similarly, however, the underlying reasons were very different. Take for instance the `bin` benchmark, which showed in both interpreter and compiler a speedup of about 1.5× in the strict mode. In native, the execution time decreased from 79ms to 53ms. In the interpreter, time went from 143ms to 97ms per iteration. In the former case, the speedup comes from the effects described above; for the latter, part of the speedup is due to better optimizations. Previously the local environment of the innermost function was live. Thanks to strict evaluation of the arguments, the argument promises do not leak the environment and therefore Ř is able to elide it.

## 8 CONCLUSION

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 was reported in previous work, 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, we believe that R should be strict by default, giving package authors the option to opt-in to laziness.

We propose and evaluate a strategy for evolving R as an ecosystem to strict semantics. First, we provide strictness signatures as a non-invasive R extension to avoid changes to legacy code. Second, we 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 our 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. Finally, by turning R into a mostly strict language, the Ř just-in-time compiler ran 1.17× faster through our benchmarks without any further changes, incentivizing the users to adopt the new semantics.

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.

## ACKNOWLEDGMENTS

This work has received funding from the National Science Foundation awards 1759736, 1544542, 1925644, and 1910850, the Czech Ministry of Education, Youth and Sports from the Czech Operational Programme Research, Development, and Education, under grant agreement No. CZ.02.1.01/0.0/0.0/-15_003/0000421, and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme, under grant agreement No. 695412.

## REFERENCES

Karan Aggarwal, Mohammad Salameh, and A. Hindle. 2015. Using machine translation for converting Python 2 to Python 3 code. *PeerJ Prepr.* 3 (2015). https://doi.org/10.7287/PEERJ.PREPRINTS.1459V1

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

Stephen Chang and Matthias Felleisen. 2014. Profiling for Laziness. 49, 1 (2014). https://doi.org/10.1145/2578855.2535887

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

Robert Ennals and Simon Peyton Jones. 2003. Optimistic Evaluation: An Adaptive Evaluation Strategy for Non-Strict Programs. *ICFP '03* 38, 9 (2003). https://doi.org/10.1145/944746.944731

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

Olivier Flückiger, Guido Chari, Jan Jecmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek. 2019. R melts brains: an IR for first-class environments and lazy effectful arguments. In *International Symposium on Dynamic Languages (DLS)*. https://doi.org/10.1145/3359619.3359744

Olivier Flückiger, Guido Chari, Ming-Ho Yee, Jan Jecmen, Jakob Hain, and Jan Vitek. 2020. Contextual Dispatch for Function Specialization. *Proc. ACM Program. Lang.* 4, OOPSLA (2020). https://doi.org/10.1145/3428288

Aviral Goel and Jan Vitek. 2019. On the design, implementation, and use of laziness in R. *Proc. ACM Program. Lang.* 3, OOPSLA (2019). https://doi.org/10.1145/3360579

Ross Ihaka and Robert Gentleman. 1996. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics* 5, 3 (1996). http://www.amstat.org/publications/jcgs/

Simon Peyton Jones and Will Partain. 1993. Measuring the effectiveness of a simple strictness analyser. In *Proceedings of the 1993 Glasgow Workshop on Functional Programming (Workshops in Computing)*. https://doi.org/10.1007/978-1-4471-3236-3_17

Uwe Ligges. 2017. 20 Years of CRAN (Video on Channel9). In *UseR! Conference*.

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

Chris Okasaki. 1995. Simple and efficient purely functional queues and deques. *Journal of Functional Programming* 5, 4 (1995). https://doi.org/10.1017/S0956796800001489

Simon L. Peyton Jones and André L. M. Santos. 1998. A Transformation-Based Optimiser for Haskell. *Sci. Comput. Program.* 32, 1–3 (1998). https://doi.org/10.1016/S0167-6423(97)00029-4

Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. *TypeWriter: Neural Type Prediction with Search-Based Validation.* https://doi.org/10.1145/3368089.3409715

Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. 2016. Optimizing R language execution via aggressive speculation. In *Symposium on Dynamic Languages (DLS)*. https://doi.org/10.1145/2989225.2989236

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

Alexi Turcotte, Aviral Goel, Filip Krikava, and Jan Vitek. 2020. Designing Types for R, Empirically. *Proc. ACM Program. Lang.* 4, OOPSLA (2020). https://doi.org/10.1145/3428249

Yisu Remy Wang, Diogenes Nunez, and Kathleen Fisher. 2016. Autobahn: Using Genetic Algorithms to Infer Strictness Annotations. *SIGPLAN Not.* 51, 12 (2016). https://doi.org/10.1145/3241625.2976009

Allen Wirfs-Brock and Brendan Eich. 2020. JavaScript: The First 20 Years. 4, HOPL (2020). https://doi.org/10.1145/3386327

S. Xu, Z. Dong, and N. Meng. 2019. Meditor: Inference and Application of API Migration Edits. In *International Conference on Program Comprehension (ICPC)*. https://doi.org/10.1109/ICPC.2019.00052