On the Design, Implementation and Use of Laziness in R

Promises Made, Promises Forced

ANONYMOUS AUTHOR(S)

The R programming language has been lazy for twenty five years. We review the design and implementation of call-by-need in R, and conduct a data-driven study of how generations of programmers have put laziness to use in their code. We analyze 14,875 R packages and find little evidence that programmers use laziness to avoid unnecessary computation or to operate over infinite data structures. For the most part R code appears to have been written without reliance on delayed evaluation. The only, significant, exception is a small number of widely used packages and core libraries which leverage call-by-need for meta-programming.

1 INTRODUCTION

Since its inception, in 1993, R has had a call-by-need semantics [Ihaka and Gentleman 1996]. When a function is invoked, such as `f(a+b,d+d)`, its arguments are packaged up into *promises* which are evaluated on demand. The values obtained by evaluating promises are memoized to avoid the need for recomputation. Thus the following definition,

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

evaluates `a+b` once and never evaluates `d+d`. With an estimated two million users world-wide [Smith 2011], R is the most widely used lazy functional programming language in existence. The R community has developed an extensive body of reusable, documented, tested and maintained code. The CRAN open source repository, the main source for R libraries, hosts over 13,000 packages with an average of 6 new ones added every day [Ligges 2017].

The authors are fascinated that R's laziness is mostly a secret. The overwhelming majority of end-users are unaware of the semantics of the language they write code in. Anecdotally, this holds even for colleagues in the programming language community who use R casually. Moreover, we do not know of any studies of the design and efficacy of call-by-need in R. With twenty five years of practical experience with laziness, some lessons surely can surely be drawn.

Hudak [1989] defines lazy evaluation as the implementation of normal-order reduction in which recomputation is avoided. He goes on to enumerate two key benefits. (B1) Laziness frees programmers from having to worry about evaluation order. In particular, sub-computations that are not needed for the final results will not be performed. (B2) Laziness allows programs to compute with unbounded data structures. Parts of a data structure which are not needed to compute a result will not be computed. Haskell is an example of a language designed and implemented to support lazy evaluation. The Haskell compiler has analysis and optimizations passes designed to remove the overhead of delayed evaluation when it is not needed. The language's type system permits laziness and side effects to co-exist in an orderly manner.

The design and implementation of lazy evaluation in R differs from Haskell. The differences are due, in part, to the nature of the language and to the goals of its designers. Neither of the two above mentioned benefits was relevant to the creators of R. For (B1), as it happens that most users are unaware that R is lazy, and they write code rife with side effects, the language and the libraries try to avoid surprises in evaluation order. Thus R is more eager than it needs to be. Library code, often forces evaluation of arguments to maintain the illusion that things happen in the order the caller specified them. For (B2), neither the core libraries, nor any of the user-defined libraries in widespread use implement unbounded data structures. This is not entirely surprising. R is first and

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

foremost a language for accessing optimized statistical libraries written in C. Most data structures end up being manipulated by C code written without any knowledge of the high-level language calling it. This explains why most R data structures are strict and have a memory layout that precludes the kind of boxing that would be required for laziness.

Given the above, one may wonder *why bother being lazy?* In particular, when R's implementation does little to optimize its runtime costs. Personal communication with the creators of R reveals that the motivation for adding lazy evaluation to R was to support meta-programming, in general, and, in particular, user-defined control structures. Scheme was one of the inspirations for R, but the creators of R found macros too complex and too static. Instead, they chose to use a combination of call-by-need and reflection which allowed them to have more dynamic bindings. In the R world, one can redefine a function at any time whereas macros are frozen at compile time.

In this paper we set out to explain the semantics of call-by-need in R, we describe its implementation in the production virtual machine, and provide examples and data about its use in practice. For the latter we perform a dynamic analysis by instrumenting the production R interpreter and tracing the execution of a large corpus of programs obtained from the CRAN and Bioconductor software repositories. This analysis allows us to provide quantitative answers to questions related to the usage of laziness in R. We can measure, for example, how often promises need to be evaluated, and when they are, how often their result is used. One of our long term goals is to use our result to inform a redesign of the language. We claim the following contributions:

- We describe the design and implementation of call-by-need in R and give a small-step operational semantics for a subset of the language that includes promises, **`eval`**, **`substitute`** and **`delayedAssign`**.
- We release an open source, carefully optimized, dynamic analysis pipeline for call-by-need. The pipeline consists of an instrumented R interpreter and data analysis scripts (written in R). Our framework can be adapted for dynamic analyses, such as taint analysis.
- Results from the analysis of 181,443 scripts over 14,875 packages. These include information about the strictness of functions, their possible evaluation orders, as well as information about the life-cycle of promises.

Our results were obtained with version `3.1.0` of GNU R, and packages retrieved on March 1st, 2019 from CRAN and Bioconductor. Our software and data is available, anonymized, for download at:

http://some-anon-dropbox-or-other

As our results are based on dynamic analysis, our data is only a good as its coverage of the possible behaviors of packages we analyzed. This is threat to validity as we may underestimate laziness. Consider some deterministic function `f` that happens to be called only once in our workload. Clearly we will not observe laziness. Our hope is that the sheer number of runs, 181,443 programs reduces the likelihood that our data is biased. We are encouraged by the fact that previous work report code coverage of over 60% for a similar corpus [Krikava and Vitek 2018] (our is larger and we execute more tests). To further mitigate this threat, we perform manual inspection of a random sample of functions to confirm the validity of the reported results.

2 BACKGROUND ON R

The R Project is a tool for implementing sophisticated data analysis algorithms. At heart, R is a *vectorized, dynamic, lazy, functional, object-oriented* programming language with a rather unusual combination of features [Morandat et al. 2012] designed to ease learning by non-programmers and enable rapid development of new statistical methods. The language was designed in 1993 by Ihaka and Gentleman [1996] as a successor to S [Becker et al. 1988]. We focus on features relevant to this work.

In R, most data types are vectorized. Values are constructed by the `c(...)` function: `c("hi","ho")` constructs a vector of two strings. The language does not differentiate scalar values from vectors, thus `1==c(1)`. In order to enable equational reasoning, R will perform a copy when a value accessible through multiple variables is written to. Consider the following `swap` function which exchanges two elements in a vector and returns the modified vector:

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

The argument vector `v` is shared, as it is aliased by `x`. Thus, when the `swap` function first writes at offset `i`, the vector is copied, leaving `v` intact. It is the copy that is returned. Under the scenes a two bit reference count is maintained for all values. Aliasing a value increases the count. Any update of a value with a count larger than one triggers a copy of that value. One important motivation for this design was to allow users to write code that can update large vectors in place. It is possible to write a loop that updates an array with at most one copy.

Every linguistic construct is desugared to a function call, even including control flow, assignment, or bracketing. Furthermore, all functions can be redefined in libraries or in user code. This makes R quite flexible and challenging to compile. Function definitions can include default expressions for arguments which can refer to other arguments. R functions are higher-order. The following snippet declares a function `f` which takes a variable number of arguments (triple dots are a vararg), whose argument `x` and `y`, if missing, evaluate to `y` and `3*x`. The function returns a closure.

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

It can be called with no arguments `f()`, with a single argument `f(3)`, with named arguments `f(y=4,x=2)` and with a variable number of arguments, `f(1,2,3,4,y=5)`.

Values can be tagged by user-defined attributes. For example, one can attach to a vector, `x<-c(1,2,3,4)`, the attribute `dim` with `attr(x,"dim")<-c(2,2)`. From that point, arithmetic functions will treat `x` as a two by two matrix. Another attribute is `class` which can be bound to a list of names. For instance, `class(x)<-"human"`, sets the class of `x` to `human`. Attributes are for object-oriented dispatch. The "S3 object system" supports single dispatch on the class of the first argument of a function, whereas the "S4 object system" allows dispatch on all arguments. These names refer to the version of the S language which introduced the features. Popular data types, such as data frames, leverage attributes. A data frame is a list of vectors with a **class** and a `colname` attribute.

R supports reflection and meta-programming. The **substitute**`(expr,env)` function is used to obtain the parse tree of the expression `expr` after performing substitutions defined by the bindings in **env**.

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

R also allows programmatic manipulation of parse trees, which are themselves first class objects. They can be evaluated using the `eval(expr,env)` function. It is also possible to parse strings.

3 CALL-BY-NEED IN R

The combination of side effects, frequent interaction with C, and absence of types has pushed R to be more eager than other lazy functional languages. In R, all arguments to a user-defined function are bundled into thunks called *promises*. Logically, a promise combines executable code, an environment, as well as the representation of the source code of the expression. To access the value of a promise that has not been evaluated, one must *force* it. Forcing a promise triggers evaluation and captures the return value for future reference. The salient differences with, for example, Haskell are that promises capture the syntactic expression appearing at the call site, promises are forced more eagerly, and that R does not provide any lazy data structures.

The following snippet defines a function `f` that takes an arguments `x` and returns `x+x`. When called with an argument that has the side effect of printing a string to the console, we can observe that the side effect is performed only once. When the promise is accessed for the second time, the cached value is returned.

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

Promises for default argument expressions are evaluated in their function's environment. Thus, the expression have access to all variables in function scope, which includes other arguments. Promises cannot be forced recursively. Promises are mostly encapsulated and hidden from user code. R only provides a small interface for operating on promises:

- **`delayedAssign(x,exp,eenv,aenv)`**: create a promise with body **`exp`** and binds it to variable `x` (where `x` is a value of type symbol). The environment `eenv` is used to evaluate the promise, and the environment `aenv` is used to perform the assignment.
- **`force(v)`**: forces the promise bound to `x`. A common programming idiom is to write `x<-x`, i.e. to assign `x` to itself, this forces `x` too.
- **`substitute(v,env)`**: This function takes a promise `v`. extracts its code, substitutes variables found in environment **`env`**, and returns a value of type **`expression`** (essentially a parse tree).
- **`forceAndCall(n,f,...)`**: This functions calls `f` with the arguments specified in the varargs of which the first `n` are forced before calling the function.

While R does not provide built-in lazy data structures, they can be coded up. Figure 1 shows a lazy list that uses environments as structs. R provides syntactic sugar for looking up variables, functions for creating environment out of lists (`list2env`) and for capturing the current environment (**`environment`**). The singleton `nil` has a `tag` that is tested in the `empty` function. A new list is created by a call to `cons` with two arguments. The function returns its environment in which `h` and **`t`** are bound to promises. The `head` and `tail` functions retrieve the contents of `h` and **`t`** respectively. At that point the promises are forced and values are returned. This example illustrates the way for promises to survive computation unevaluated, namely by being protected by an environment.

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

Fig. 1. Lazy list in R

3.1 Implementation

A promise has four slots: **exp**, **env**, `val` and `seen`. The **exp** slot contains a reference to the code of the promise, the **env** refers to environment in which the promise was originally created. It is typically the caller environment, except for default argument expression where the environment is that of the callee. The `val` slot holds the result of evaluating the promise. The `seen` slot is a flag used to avoid recursive evaluation of a promise.

When the value of promise is requested. First, the `val` slot is inspected. If it is bound to a value, it is returned. Otherwise, the `seen` flag is checked. If set, an error is thrown to cancel the current computation. Otherwise, the flag is updated and the expression is evaluated in the specified environment. Once the evaluation finishes, the `val` slot is bound to the result, the **env** slot is cleared to allow the environment to be reclaimed, and the `seen` flag is unset.

The implementation does little to optimize promises. In some cases, a promise can be created pre-forced with a value pre-assigned. The GNU R implementation recently added a bytecode compiler, this compiler eliminates promises when they contain a literal value.

R evaluates promises fairly aggressively. The sequencing operator `a;b` will evaluate both `a` and `b`, assignment `x<-a` evaluates `a`, and returning a value from a function also triggers evaluation. In addition, many core functions are strict. In addition to user-defined closures, R has two kinds of functions that are treated specially:

- *Builtins*: There are 680 builtins in GNU-R. They provide efficient implementations of numerical methods and other mathematical functions. They are typically written in C. Builtins expect that their arguments are values, so R will evaluate the argument list of a builtin before calling it. Evaluation proceeds left to right.
- *Specials*: There are 46 specials. These functions are used to implement core language features such as loops, conditionals, bracketing, etc. These functions do not take promises, instead they get expressions (parse trees). They will evaluate them in the calling environment or in a specially constructed environment.

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

We would be remiss if we did not mention one of strangest features of R: its variable context-sensitive lookup rules. When looking up a variable `x`, R performs a normal lexically scoped lookup. But, if the variable is part of a functions call `x(...)` then a different lookup is used. For function names, R will find the first definition of `x` check if it is bound to a closure. If it is, then the closure is returned. Otherwise, that variable is skipped, and lookup continues in the enclosing scope. One of the corollaries of the function lookup rule is that it may have to force promises on the way.

4 SEMANTICS

This section describes a small-step operational semantics in the style of [Wright and Felleisen 1992] for a core R language with promises. We build upon the semantics of Core R [Morandat et al. 2012], but omit vectors, out-of-scope assignments and the peculiar function variable look-up rules. Instead, we add delayed assignment, default values for arguments, `substitute` and `eval`. Supporting these features requires the addition of strings as a base type and the ability to capture the environment.

The surface syntax of our calculus includes terms for strings, variables, string concatenation, assignment, function declaration, function invocation (we support one argument function and no argument function calls), environment capture, substitution, eval, and delayed assignment.

$$\texttt{s} \mid \texttt{x} \mid \texttt{e} \# \texttt{e} \mid \texttt{x} \leftarrow \texttt{e} \mid \texttt{fun(x=e) e} \mid \texttt{x(e)} \mid \texttt{x()} \mid \texttt{env} \mid \texttt{subst(x)} \mid \texttt{eval(e, e)} \mid \texttt{delay(x, e, e)}$$

This syntax is extended with a number of additional terms used during reduction where variables and expression can be replaced by values ranged over by meta-variable v which can be one of a string (`s`), a closure ($\lambda \texttt{x} = \texttt{e}.\texttt{e},\ E$), an environment ($\texttt{env}(l)$), or a promise ($\texttt{prom}(l)$). Mutable values are heap allocated and ranged over by meta-variable l. We use $\perp$ to denote an invalid reference.

The reduction relation is of the form $S\ E \rightarrow S'; E'$ where the stack S is a collection of expression-environment pairs ($\texttt{e}\ E$) and the heap maps variables to values. The following describes these domains in detail. Frames are mutable and can be shared between closures, so they are stack allocated.

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

Promises are quadruples $(v, \texttt{e}, E, R)$ where v is the cached result of evaluating the body `e` in environment E. R is a status flag where $\downarrow$ indicates we have started evaluating the promise.

Finally, we define deterministic evaluation contexts $\mathbb{C}$ as follows:

$$[] \mid \mid \mathbb{C} \# \texttt{e} \mid v \# \mathbb{C} \mid \texttt{x} \leftarrow \mathbb{C} \mid \mathbb{C}(\texttt{e}) \mid \mathbb{C}() \mid \texttt{eval}(\mathbb{C}, \texttt{e}) \mid \texttt{eval}(v, \mathbb{C}) \mid \texttt{delay}(\texttt{x}, \texttt{e}, \mathbb{C})$$

Evaluation contexts specify the strictness properties of our calculus. Following R, some builtin operations such as string concatenation are strict. But function calls are notably not strict in their argument, the expression in `x` (`e`) remains untouched by the context.

The semantics ensures that promises are stored in environments and, whenever they are accessed they will be forced. In particular, assignment is strict, so it always stores values in variables. Function return also always returns evaluated values. The only way for a promise to outlive the frame that created it is by being returned as part of an environment or a closure. Following R, it is possible to create a cycle in promise evaluation, the expression $(\texttt{fun}(\texttt{x} = \texttt{x})\ \texttt{x})()$ when evaluated creates a closure and invokes it. The function's body triggers evaluation of the promise bound to `x`. Since no argument was provided, the default expression is evaluated causing a cycle. Like in R, this result in a stuck state.

Figure 2 contains the reduction rules along with a couple of auxiliary functions. We omit the definition of the functions *parse* and *string* which, respectively, turn strings into expressions and conversely. The expression *fresh* is used to obtain new heap references.

Rules `Fun` and `Concat` deal with creating a closure in the current environment and concatenating string values.

Rule `Assign` will add a mapping from variable `x` to value v in the current frame. As frames are allocated in the heap, this updates the heap.

Rule `Delay` performs delayed assignment. For this it needs an expression to assign and an environment in which this expression will be evaluated. The rule creates a new unevaluated promise and binds it to `x`. Rule `Env` grabs the current environment and returns it as a value. Rule `Subst` looks up the promise, extracts its body and deparses it into a string. Rule `Eval` takes a string and an environment, parses that string into an expression and schedules it for execution. Rule `EvalRet` takes the result of that evaluation and replaces the call to eval with it.

Rule `Invk1` and `Invk0` handle user-defined function calls. Both rules expect v to be a closure. They allocate a new promise for the argument `x`. They differ on the body of that closure and the environment. `Invk0` has no argument and will use the default expression `e'` specified in the function declaration, it uses the environment where `x` is bound for evaluation. Rules `Ret1` and `Ret0` are the corresponding returns rules that replace the call with the computed value.

Rules Lookup and Lookup1 are used to read variables from the current environment. If the result is a promise it is scheduled for execution by Lookup1. Rule Force will actually evaluate a promise that has been pushed on the stack, if that promise has not yet been evaluated. It sets the flag to avoid recursive evaluation. Rule ReadVal retrieves the value of an already evaluated promise. Rule Memo stores the result of evaluation in the promise and discard its environment. Finally, rule RetProm returns from evaluating a promise by replacing the variable looked up with the result.

Fun

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

Concat

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

Assign

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

Delay

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

Env

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

Subst

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

Eval

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

EvalRet

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

Invk1

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

Invk0

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

Ret1

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

Ret0

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

Lookup

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

Lookup2

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

Force

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

ReadVal

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

Memo

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

RetProm

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

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

Fig. 2. Semantics

5 METHODOLOGY

We now explain the infrastructure that assembles the corpus and collects and analyzes traces.

5.1 Infrastructure

Our analysis pipeline starts with a set of scripts to download, extract and install open source packages. Next, an instrumented R virtual machine generates events from program runs. This is followed by an analyzer that processes the execution traces to produce tabular data files in a custom binary format. Another set of R scripts postprocess the data, compute statistics, and generate graphs. The entire pipeline is managed by R scripts that extract runnable code snippets from installed packages and run the other steps in parallel. Parallelization is achieved using GNU Parallel [Tange 2011]. This infrastructure can be used for different dynamic analyses of the behavior of R programs. We intend to submit it for artifact evaluation.

5.2 Instrumented R VM

The instrumented R Virtual Machine is based on GNU-R version `3.1.0`. It has 818 lines of C code, exposing an event data structure with fields to describe a variety of execution events that capture the internal interpreter state. The execution events we record are:

- **Fun-Entry/Exit**: type of function, arguments, environment, return value. exception code.
- **S3-Generic-Entry/Exit**: S3 method name, class name, dispatch object, return value.
- **S4-Generic-Entry/Exit**: S4 method name, environment, definition and return.
- **Eval-Entry**: entry in the interpreter eval function.
- **Ctxt-Entry/Exit**: Stack frame pushed/popped
- **Ctxt-Jump**: Stack frames jumped to by R due to explicit returns or longjumps in C.
- **Prom-Entry/Exit**: Promise that is being forced
- **Prom-Lookup**: Promise value lookup
- **Prom-Substitute**: Promise expression lookup
- **Alloc**: Object allocation
- **Free**: Object unmarked by the garbage collector
- **Var-Def**: Variable Definition: symbol, value and environment
- **Var-Ass**: Variable Assignment: symbol, value and environment
- **Var-Rem**: Variable Remove: symbol and environment
- **Var-Lkp**: Variable Lookup: symbol, value and environment

Events can be selectively disabled. This allows to the analyzer to ignore events generated by internal interpreter functions and avoid recursion. R objects captured in events are protected to prevent them from being reclaimed during tracing.

It was a challenge to modify an interpreter whose code is 103,962 LOC written over 25 years by many developers and outside contributors. The system has grown in complexity with an eclectic mix of ad-hoc features designed to support growing user requirements. It was particularly difficult identifying all the points that needed to be instrumented. For instance, the code to manage environments and variable bindings in `main/envir.c` is over 2932 LOC with 131 functions with a large number of identical code fragments for managing these datastructures duplictated in various files. This made it hard to identify the points where variables were defined, assigned, looked up or mutated.

5.3 Tracer

The tracer is an R package (73 LOC) that calls to C++ (4,884 LOC) for the heavy-lifting. The package is loaded on our modified R interpreter.

During program execution, the tracer maintains data structures that model various relevant aspects of the language such as functions, calls, promises, variables, environments, R stack frames and the R stack. The events generated by the program cause the tracer to update these objects. Table 3 gives an idea of the amount of state managed by the tracer. Many values are collections that hold additional objects.

Class	Members	Size (Bytes)
TracerState	27	464
ExecutionContextStack	1	24
ExecutionContext	3	24
Environment	3	72
Variable	5	64
Function	10	176
Argument	16	56
Call	13	128
Denoted Value	73	536

Table 1. Summary of tracing data structures

Some design decisions allowed the tracer to scale and be able to process 1.5 M events per second. Firstly, we avoid copying objects. Objects are created by a unique factory that implicitly caches objects in a global table. This is pays off because objects are big and relatively expensive to initialize (e.g. serializing R data to string). Secondly, to enable parallelization, we reduce tracing footprint by deallocating our data as soon as the R objects they are modeling are collected. Thirdly, all objects are linked to provide fast access. This pays off because events require sequences of state changes to multiple objects. The downside of this design is that the structures are circular and require reference counting to know when objects can be deleted. The prevalent use of longjump for non-local returns forced us to maintain a shadow stack that is kept in sync with R's stack at every function return or longjump.

The tracer stores large amounts of data. Our first prototype used Sqlite. However, we found the approach limiting. Firstly, during development we kept running into errors because the database schema and some part of the tracer were out of sync. Due to the iterative nature of data analysis, we were modifying the table schema quite frequently and this very quickly became a pain point. Secondly, our database was normalized, thus requiring join operations in the analysis. At our scale, these joins were so expensive that the database operations would run for days. Thirdly, database insertions ended being a bottleneck. With this setup, we were able to trace less than 2000 packages in 2 days filling up a 2 TB disk.

In the end, we chose to implement a custom format. As the event stream has substantial amounts of redundancy, we applied streaming compression on the fly. Compression yields an average 10x saving in space and 12x improvement loading time. As the data will be read by R, we also store a schema to speed up the subsequent loading.

5.4 Execution

For each R package to be analyzed our infrastructure must extract executable code from the package. Extraction invokes an R API which locates executable code snippets in the package documentation files and RMarkdown files. Files in the test directory are copied as is. All code snippets and test files are combined into one executable file that first sets up the tracer and initializes it with paths to input and output data. For each package, the tracer generates 9 data files and 4 status file. The status files denote the different possible states of the tracing. They allow the infrastructure to recover

from errors and re-execute interrupted executions. The R scripts responsible for generating traces do extensive logging of intermediate steps for debugging purposes.

5.5 Post-processing

Scale was our major challenge. We faced difficulties both due to execution time and data size. In the 181,443 programs that were traced, the tracer observed 534.3 B expression evaluations, 482.6 B calls to 586.8 K functions and 177.8 B promises. The raw data generated by the tracer is 2.8 TB but the reduced data is just 61 GB. In hindsight, it appears that incorporating analysis in the tracer, i.e., presummarizing data in the C++ code, would have been beneficial. However, this would require knowing ahead of time all the analysis that we would perform. Part of the challenge lies in the fact that the event of interest and attached analysis were not fixed ahead of time. Presummarized data makes it harder to pose new questions. It also makes it harder to detect bugs because summarized data resists correlation with actual code. This part of the pipeline analyzes the raw data. It is 4K lines of R code. There are multiple steps that are detailed next.

5.5.1 Prescan. This step scans the raw data directory and outputs a list of all the subdirectories that contain raw data. There is a directory per package and multiple files in each directory.

5.5.2 Reduce. Given a list of directories, this step uses GNU Parallel to reduce the raw data. This is the most expensive step in terms of size and speed. Since the data files are large, we must limit the degree of parallelism drastically to avoid running out of memory.

5.5.3 Scan. After all the reductions are done, this creates a list of all the files successfully reduced.

5.5.4 Combine. This step combines information from all the programs into a single data table per analysis question. This steps result in the generation of 27 large data tables.

5.5.5 Summarize. This step computes summaries of the merged data. The summaries of interest are: (1) the frequency of events, the frequency of the type of R objects, (3) the list of functions with their definitions. (4) information about function arguments, such as distribution of types of arguments, number of default and non-default arguments, (5) information about escaped arguments, such as the return type of functions from which arguments escape, force, lookup and meta-programming on escaped arguments before and after escape, (6) information for each formal parameter position, such as parameter position strictness, function strictness and package strictness, (7) information about promises, such as number of lookups, meta-programming.

5.5.6 Report. The last step generates graphs and tables from an RMarkdown notebook. It also generates LaTeX macros for inclusion in the paper.

5.6 Threats to Validity

We have mentioned in the introduction that code coverage is a worry for any dynamic approach. There is an additional point to consider.

C and C++ functions can bypass the R extension API and directly modify R objects' internals. For example, set a promise's value without going through the API thus obviating our hooks. Such behavior breaks the R semantics and is error prone as the R internals do change. We have not observed this behavior in practice, but given the large number of packages it may happen.

6 CORPUS

The corpus of programs used in this study was assembled on March 1st, 2019 and obtained from the two main R code repositories, namely the *Comprehensive R Archive Network* (CRAN) [Ligges 2017] and *Bioconductor* [Gentleman et al. 2004]. Both are curated repositories; to be admitted packages must conform to some well-formedness rules. In particular, they must contain use-cases and tests along with the data needed to run them.

Our snapshot of CRAN includes 13,936 packages, and for Bioconductor, 2,974 packages, out of which 1,649 contain software, 1,302 contain data and 23 are so-called workflows. All together this is 16,910 libraries. Our scripts downloaded and successfully installed 16,689 packages. The reasons why some packages did not install were varied, missing dependencies, compiler errors, etc. We believe that they may be fixable but are hard to automate. Out of the installed packages we were able to successfully trace the execution of 14,875 packages. The reason why some package did not trace vary. Some of them were empty programs, and the remaining either did not build or failed at runtime. In terms of lines of code, we traced 8.5 M lines of R and 4.9 M lines of C. Table 2 details these numbers. Examples account for the majority of what was run.

	Tests	Examples	Vignettes	
Scripts	39.8 K	209.1 K	9.2 K	Installed
	30.1 K	171.8 K	7.9 K	Traced
LOC	2.3 M	1.5 M	587.6 K	Installed
	1.5 M	1.1 M	380.4 K	Traced

Table 2. Corpus

For each package, our scripts gathered runnable code from three sources, test cases, examples and vignettes. Test cases are typically unit tests written to exercise individual functions while examples and vignettes demonstrate the expected use of the particular package by end-users. These use-cases may load other packages and access data shipped with the package or obtained from the internet. The recorded data is summarized in Table 3. The total size of our database is 2.8 TB.

We start with some general observations. Figure 3 shows the distribution of application-level objects and promises created in our corpus. The majority of objects created by applications consist of character strings, logical, integer, and double vectors. Environments are frequently observed, this is because one is created for each function call. They are also used as maps by user code.

We have observed 177.8 B promises, out of which 82.6% were created as part of argument lists. The remainder are used internally, e.g. for lazy loading of functions. The number of argument promises is smaller than the total number of user values created because some values are composite (e.g. data frames) and some values are stored in lists or returned without being bound to an argument. In the remainder, we will ignore the promises that do not correspond to function arguments.

We observed that 87.6% of promises were forced. The remainder were unused. Unused promises happen when a function does not use all of its arguments, a common occurrence in R where

Data	Tests	Examples	Vignettes	Total
Objects	1.1 MB	25.7 MB	30 MB	27.8 MB
Escaped Args.	23.2 MB	272.1 MB	63 MB	324.2 MB
Calls	96.3 MB	1.6 GB	209 MB	1.8 GB
Function Defs	502 MB	9 GB	800 GB	1.3 TB
Promises	82 GB	452 GB	168 GB	702 GB
Arguments	120 GB	735 GB	326 GB	1.2 TB

Table 3. Data (2.8 TB)

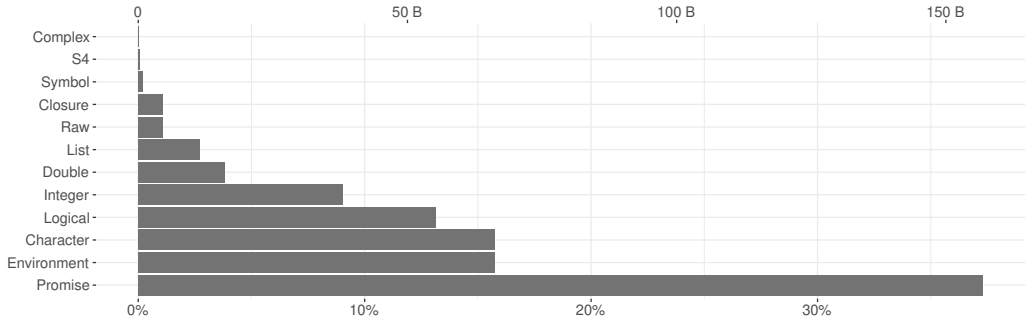

Fig. 3. Object Counts

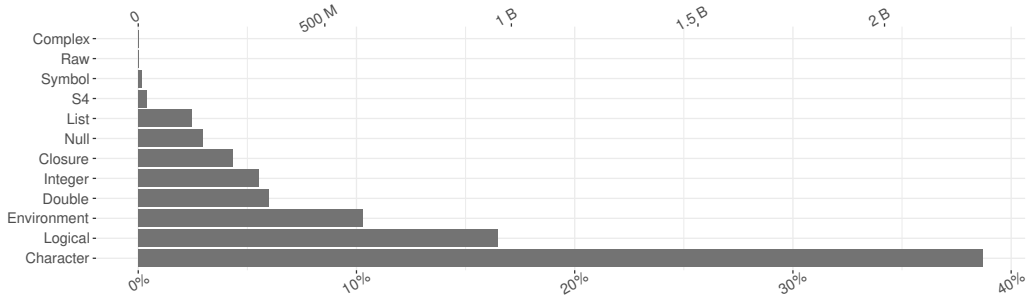

Fig. 4. Promise result types

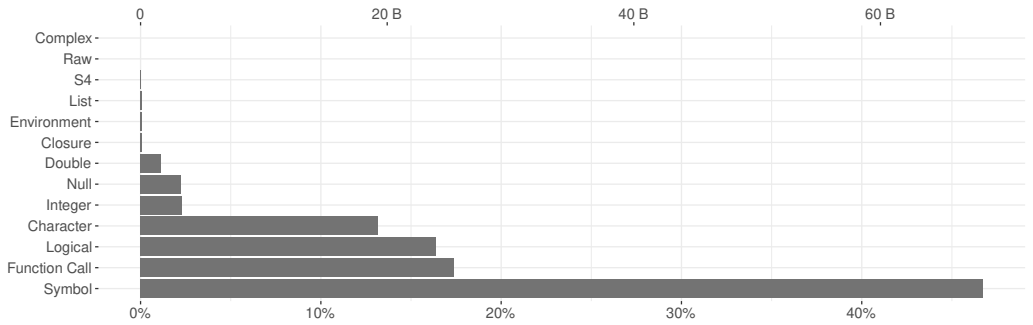

Fig. 5. Promise expression types

functions sometimes have 20+ arguments. One reason for eager forcing of promises is when they are used in object-oriented programming, namely they are dispatched upon by either the S3 or S4 object system. Overall only 1% of promises are forced due to dispatch. As this number is small, we will ignore dispatch for the remainder of the document. Figure 4 shows the content of forced promises. The most common types are character strings, logicals, environments, double, integer and closure. The presence of null values is explained by the fact that many default argument values are set to null. Figure 5 shows the content of the expression slot of promises, i.e. their code. less than 20% of promises contain a function call, e.g. `1+2` or `f(z)`. The majority contain a single symbol to be looked up, e.g. `x`. Of course that symbol may be bound to another promise. The remainder are inlined values (scalar constants).

7 QUANTITATIVE ANALYSIS

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

7.1 Strictness

A function is *strict* if it evaluates all of its arguments. In our corpus, out of a total of 329.9 K functions, 84.5% are strict. Figure 6 gives a histogram of function strictness ratios per package. The majority of packages mostly contain strict functions. The packages that are less than 75% strict account for only 2.1 K packages (15.7% of all packages) and 63.4 K functions (19.4% of all functions).

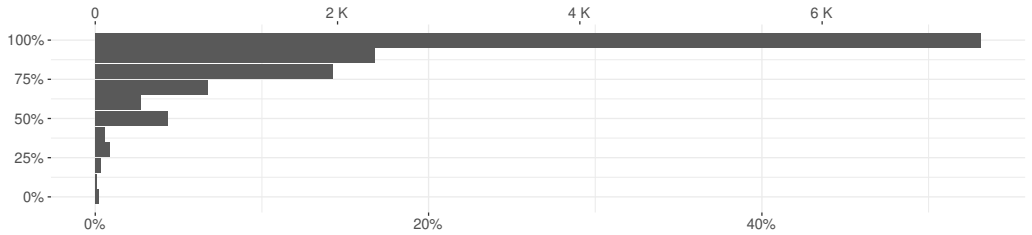

Fig. 6. Function Strictness per package (x-axis = packages; y-axis = strict function ratio)

The strictness of the 155.1 K individual parameters positions can also be measured. We distinguish three categories: *Always* is for parameters that are always evaluated, *Never* is for parameters that are never evaluated, and *Sometimes* is for parameters that we have observed being evaluated in some calls and not in others. Figure 8 illustrates the distribution of parameters across the categories. There are 6.7% parameters that are never evaluated, and 6.3% parameters that are inconsistently evaluated. A strict function has all of its parameters always evaluated.

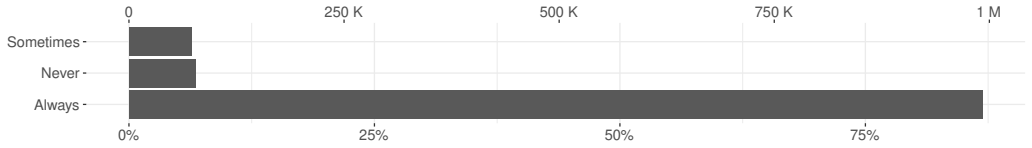

Fig. 7. Parameter strictness

7.2 Computation reuse and avoidance

One advantage of call-by-need is that it avoids multiple evaluations of the same expression. How often is this used in practice? Figure 8 shows, for each individual promise, how often its value was accessed. The majority of promises are used once. 9.7% of promises are accessed twice, and 3% are accessed three times. A small proportion of promises, 0.09%, are accessed over 1,000 times.

Another advantage of laziness is that it could help avoid unnecessary computation. But do programmers avail themselves of this feature? One way to test this hypothesis is to compare the execution time of arguments that are strict (promises passed to *Always* parameters) to those of lazy arguments (promises passed to *Sometimes* parameters) and see if the lazy arguments are more computationally intensive in any way.

Figure 9 shows the probability density of promises at different running times. Promises running for less than a millisecond are ignored. While there is a difference in the profiles, the promises that are passed to Sometimes arguments tend to be faster to evaluate. But given the large amount of promises being observed it is difficult to be conclusive.

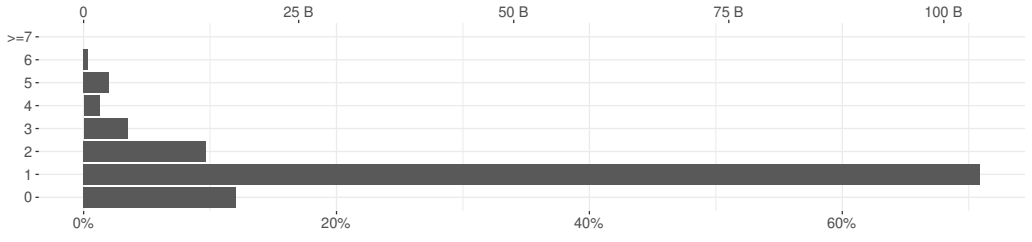

Fig. 8. Computation reuse

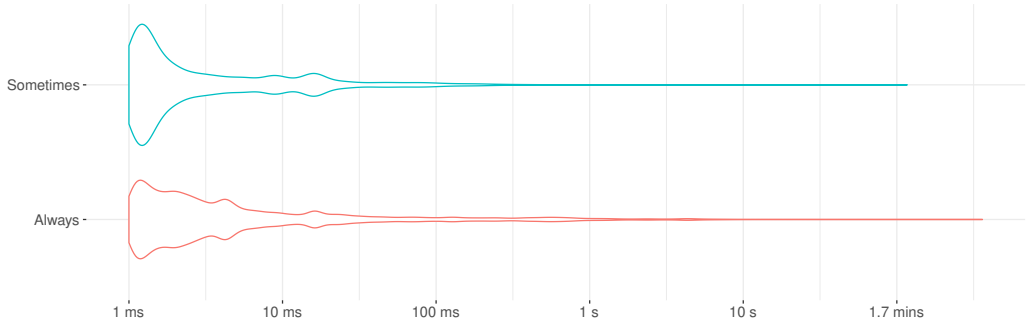

Fig. 9. Promise evaluation

7.3 Lazy data structures

Laziness makes it possible to compute over infinite data structures. While R does not provide any libraries with such data structures, it is conceivable that some programmers may have created some in their code. As we have shown earlier, one can use promises together with environments or closures to create unbounded data. How often does that occur? One thing that we can measure is *escaped promises*, the promises that outlive the function their are passed into. Of the 48.2 B promises we have observed only, 4.4 M escape. This is a rather small number, and we need to establish the reason why the promise escape. One thing we can measure is the return type of the functions for which promises escape.

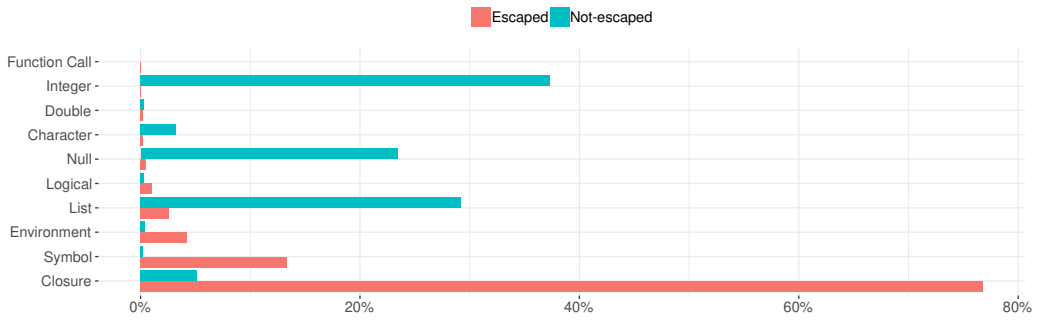

Fig. 10. Computation reuse

Figure 10 compares the return types of functions which have at least one of their promises escaping, and functions that do not have any promise that escape. The main difference between the two is that functions with escaping promises have a large number of closures as their return values. The next section performs a qualitative analysis to understand how those closures are used.

7.4 Meta-programming

We define meta-programming as the use of the **substitute** function to extract an abstract syntax tree from the body of a promise. We observed 923.5 M calls (2% of all calls) to **substitute**. Figure 11 shows the number of promises that were meta-programmed. The graph has four categories: promises that were created but never used, promises that were meta-programmed, promises that were both meta-programmed and accessed, and lastly, promises that were only accessed. The data shows that only 0.5% of promises were used purely for meta-programming purposes, while 0.2% were both forced to obtain a value and used for meta-programming.

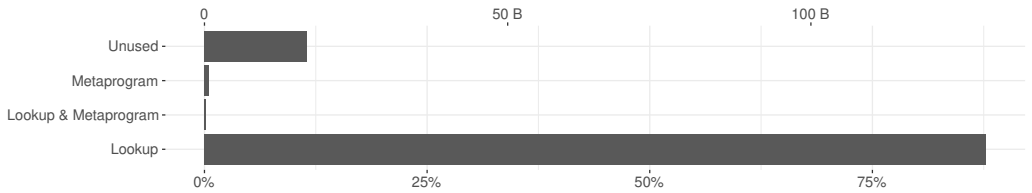

Fig. 11. Meta-programmed promises

While the percentage of meta-programming is small, the use of meta-programming is widespread and used in every package. There are 1.3 K (11.8%) packages that do meta-programming.

8 QUALITATIVE ANALYSIS

This section reports on manual inspection, conducted by the authors, of randomly selected functions from the corpus.

8.1 Strictness

We inspected 100 randomly selected functions that our dynamic analysis marked as strict. Out of those, 82 functions were found to be indeed strict. The remaining 18 were in fact lazy, but we did not observe the laziness. The majority of incorrectly labeled functions were either not using all of their arguments, only using them along some execution paths or returning early (16/18). We found a single function that was lazy because it called another function that was itself lazy in the particular argument and one function for which one of the arguments escaped.

In the code we inspected, the instances of functions that do not evaluate all of their arguments could be cases where computational effort is saved assuming that the arguments passed are complex expressions. We also found occurrences of explicit forcing of arguments. Programmers will write code such as `x<-x` or `force(x)` to ensure that function's argument are values. An example of such code is the `viridis_pal` function in the `scales` package. The function returns a closure, but forces all of its arguments to avoid capturing their environments:

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

Another related situation is when using higher-order functions such as **`apply`** or `reduce`. Following bugs and confusion from users around unwanted lazy evaluation and variable capture interactions, these functions were made strict. This is achieved by the addition of a call to the `forceAndCall` function. Looking for uses of `forceAndCall` in our corpus revealed an additional number of packages that enforce strictness for higher-order functions. The function is invoked 858.1 M times (1%) in our corpus.

The `force` function to force evaluation is also widely used. At runtime, 75.8 M calls (0.1% of all calls in the corpus) are made to this function. We found it in 60% of the packages we inspected. In all cases the argument would be captured in a closure and the author of the code seemed to want to avoid capturing an unevaluated promise.

In summary, our manual inspection suggests that our analysis overestimates strictness by approximately 20%, and the majority of cases where a function is not strict is when some path through the code does not require all arguments. Improving the precision of dynamic analysis in this case would take increasing code coverage. We also found numerous occasions where programmers require strictness to avoid unpredictable side-effects.

8.2 Computation reuse and avoidance

Our quantitative analysis has shown that result of a promise is reused in practice and the execution time of promises suggested that arguments which may not be evaluated are not substantially more computational than those that are always evaluated. What is hard to asses from the quantitative results is whether programmers benefit from laziness. Consider a call to a hypothetical function, `f(a+b,c)`, and imagine that depending on the value of booolean **`c`**, the first argument may or may not be evaluated. Now further consider the case where `a` and `b` are large matrices. Clearly, if **`c`** is very infrequently true and the programmer is aware of laziness, they would not worry about it. But without laziness, the API of the function would perhaps be changed so that instead of passing the result of the computation one would pass the individual arguments and let the function perform the addition if needed.

In our manual inspection, we have not found any occurrences of idioms suggesting that programmers are concerned about the cost of evaluating expressions are described above. This is likely partly due to the fact that R is lazy, and also, possibly, because many users are not performance sensitive.

The most promising use of laziness seems to be around the `delayedAssign` operation which creates a promise and binds it to a variable. At runtime, 51.1 M (0.08% of all calls) calls are made to this function. We inspected manually 36 packages that use it. We found several recurring usage patterns that are all about avoiding unnecessary computation. Examples are: the `AzureML` package uses delayed assignment to avoid loading unneeded parts of its workspace; `crunch` uses it to delay fetching data from a server; and (slightly surprisingly) `callCC` uses it to redefine the `throw` function to perform a return instead of throwing an exception.

Overall we found little evidence of programmers taking advantage of call-by-need, other than in cases where they explicitly called `delayedAssign`, in the sample of functions we inspected. We did find cases where the authors of the code seemed to want to enforce a consistent evaluation order and prevent argument-induced side-effects from happening in the midst of evaluation of the function.

8.3 Lazy data structures

To detect an infinite data structure we looked at occurrences of promises that outlive the function in which they were passed. We inspected 100 functions with escaping arguments and observed the following patterns: (1) arguments captured in closures, (2) arguments captured in S4 objects,

Closure:

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

S4 object:

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

Environment:

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

Finalizer:

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

Delayed assignment:

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

Formula:

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

Fig. 12. Escaping promises

(3) arguments stored in environments, (4) arguments passed into finalizers, (7) argument passed into delayed assignments, and (8) arguments passed into formulas. Figure 12 gives examples of each of these categories. In terms of linguistic mechanisms, all but the last two end up as variants of closure-captured promises. Delayed assignment is a builtin that installs a promise into the designated variable of a specific environment. Formula is interesting, because it is really a domain specific language that is interpreted with different semantics.

In our time spent working with R, we found a single package, `Rstackdeque` [O'Neil 2015] that advertised the use of lazy data structures, specifically fully persistent queues based on [Okasaki 1995]. The package, which depends on lazy lists, is the only use of lazy data structures we are aware of in the R ecosystem.

8.4 Meta-programming

We manually inspected 100 functions that meta-program their arguments, here we limit meta-programming to the use of the **`substitute`** function, and classified them based on the usage patterns.

One common pattern is to extract the source text of an argument. This is used by various plotting functions to give default names to the axes of a graph if none are provided. In the following definition `x` and `y` evaluate to the data that needs to be plotted while `nmx` and `nmy` are optional arguments giving names to the corresponding axes. The call to **`substitute`** returns the AST of the arguments, and **`deparse`** turns those into text. This pattern explains why we saw many promises that are both evaluated and meta-programmed.

```
function(x, y, nmx, nmy) {
  xAxisName <- if(missing(nmx)) deparse(substitute(xAxis)) else nmx
  yAxisName <- if(missing(nmy)) deparse(substitute(yAxis)) else nmy
  plot(x,y,type="n",xlab=xAxisName,ylab=yAxisName)
}
```

Another common pattern, that is a syntactic convenience, is to allow the use of symbols instead of strings. In R, the `::` operator is used prefix function names with their packages. It is implemented as a reflective function, and expects two strings. But programmers would rather write `base::log` to

select the **log** function from the base package, rather than `"base"::"log"`. To support this idiom, the arguments are left uninterpreted, instead the function deparses them to strings.

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

Meta-programming is used for better error reporting and logging. This is again an example where the code only retrieves the source text of the argument.

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

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

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

A combination of meta-programming, eval and first-class environments opens up the door to domain specific languages. The pipe operator heavily used in the `tidyverse` group of packages performs non-standard evaluation on its arguments. While the user writes code like this **df %>% mean**, what is actually executed is **mean(df)**. To achieve this the function turns both arguments into abstract syntax trees, and captures the calling environment.

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

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

9 LAZINESS RECONSIDERED

The previous sections have painted a nuanced picture of the importance of call-by-need in R. The traditional benefits of lazy evaluation do not seem to apply to R. We found only two broad categories of usages that benefited from it. The first is the creation of delayed bindings. These, in our experience, are always explicit. The second is for meta-programming. Within that category, uses are split between accessing the source text of an expression for debugging purposes and performing non-standard evaluation.

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

It is worth asking the question whether R could be converted to default to strict evaluation and how much changes this would entail. Any change to the semantics of a widely used language has

to be minimally invasive. We are considering the following combination of ideas. For the meta-programming uses that require source code we propose to offer a function `caller(x)` which returns the expression *at the call site* of argument `x`, this information is present in the debug meta-data of the interpreter. For functions that do non-standard evaluation, we propose to have annotations on the function definitions to request a promise to be generated, e.g. **`function`**`(x@lazy){...}`. This can be implemented by adding a runtime check before function calls. The results of the check can be cached. For all other function arguments the idea would be to set the default to not generate a promise.

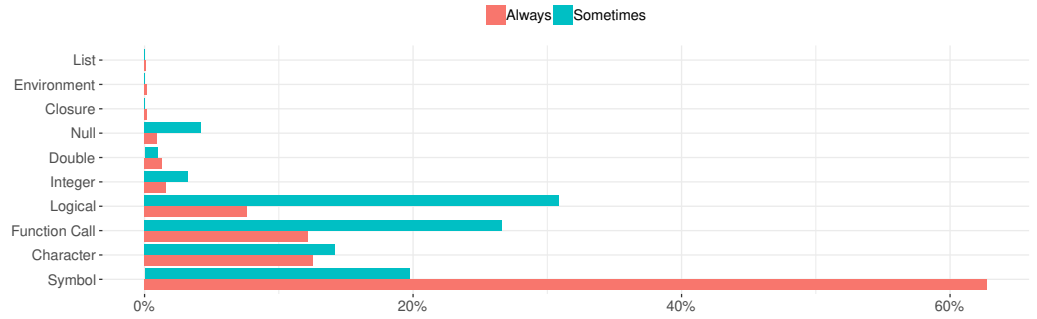

Fig. 13. Expression Types of Parameters

The one wrinkle is of course that changing the evaluation order for a language with side-effects is likely to cause massive breakage. There are some redeeming features that make R a better candidate than most languages. For one, it is mostly functional. Vector data types are copied-on-write when shared. Moreover, as Figure 13 shows the contents of expressions held in promises, contrasting promises that are passed to strict (Always) and lazy (Sometimes) argument positions. Only 25% of promises perform any computation. In our experience most of those computations are side-effect free.

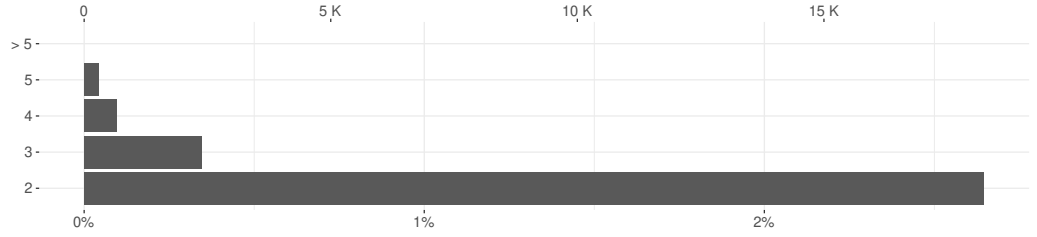

Fig. 14. Function force orders

Another way to look at the potential for evaluating promises earlier is to look at the order in which arguments are forced by functions. We call these force orders. The majority of functions (97%) have a single force order (regardless of strictness). About 2.5% of the functions have two force order, and very few functions have more than that. This means that for the majority of function we could extract a single force order and simply evaluate the function early. For the multi-force order functions it is reasonable to expect that they are not sensitive to side-effects.

So, while there may be a way to remove laziness there is also an argument for strengthening it. In many ways, R is only *weakly* lazy, it forces promises in many places where other languages would not. The work of Hadley Wickham on tidyverse [Wickham 2017] and Kalibera on FastR [Kalibera et al. 2014] suggest that more laziness can bring interesting optimization opportunities.

10 RELATED WORK

Lazy functional programming languages have a rich history [Turner 2012], The earliest lazy programming language was Algol 60 [Backus et al. 1963] which had a call-by-name evaluation strategy. This was followed by a series of purely functional lazy languages [Augustsson 1993; Turner 1979, 1981, 1985]. The motivations for the pursuit of lazyness were modularity, referential transparency and the ability to work with infinite data structures [Hughes 1989]. These languages inspired the design of Haskell [Ryder and Hailpern 2007].

The meta-programming support of R is reminiscent of fexprs [Wand 1998] in Lisp. Fexprs are first class functions with unevaluated arguments. In R, all functions always have access to their unevaluated and their evaluated arguments. Pitman [Pitman 1980] 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.

FastR [Kalibera et al. 2014] is an AST interpreter for R written in Java to explore the applicability of simple compiler optimization techniques, within the reach of scientific community lacking expertise in language runtimes. The authors implement an optimization technique that defers element wise operations on vectors by constructing expression trees called Views, which are evaluated on demand. This prevents the materialization of temporary vectors in a chain of vectorized mathematical operations. Like promises, views cache the result of evaluating the expression. However, unlike promises which are exposed to the user through meta-programming, views are completely transparent to the user. Promises are built by packaging arbitrary argument expressions but views are built incrementally by piling referentially transparent vector operations such as +, -, log, ceil, etc. Promises are evaluated very quickly due to the eager nature of most functions, but the expression trees of views are evaluated only when the entire result vector or its subset is demanded or a selected aggregate operation such as **`sum`** is applied.

Building upon the implicit argument quoting of promises is a data structure called quosure, short for quoted closure, that bundles an expression and its evaluation environment for explicit manipulation at the the language level. A quosure is thus an explicit promise object exposed to the user, with APIs to access the underlying expression and environment. Quosures are a central component of a collection of R packages for data manipulation, Tidyverse [Wickham 2017], that have a common design language and underlying data structures. Dplyr [Wickham et al. 2018], a package of Tidyverse, implements a DSL for performing SQL like data transformations on tabular data and ggplot2 [Wickham 2016] implements a declarative language for graphing data, inspired by the Grammar of Graphics [Bailey 2007]. These libraries automatically quote, unquote and quasiquote user supplied expressions and evaluate them in an appropriate environments. To facilitate this, these libraries also provide an evaluation function, eval_tidy that extends the base R **`eval`** function by supporting evaluation of quosures. This indicates that the community has found it useful to reify the promise object to make better APIs. This support is intimately tied to first class environment and existing laziness support in R.

11 CONCLUSION

This paper is a glimpse in the design, implementation and usage of call-by-need in the R programming language. Call-by-need is the default in R, but our data suggests that it is used less than one would expect. In part this is because in order to deal with side-effects and to manage programmers' expectation, many functions are stricter than they need to be. In fact we found little evidence of lazy data structure or that users leverage call-by-need to avoid unnecessary computation. Instead, the main use of laziness is for meta-programming – R's promises can return the unevaluated code of the expression.

Laziness is costly at runtime, promises must be allocated and the language implementation must deal with the possibility of any variable access causing side-effects. If it is little used, could it be eliminated? We believe that flipping the switch and making most arguments strict would be safe. This is because most lazy arguments are trivial and because the language is mostly functional so side-effects are rare to start with. In future work we plan to explore this direction.

REFERENCES

Lennart Augustsson. 1993. The Interactive Lazy ML System. *J. Funct. Program.* 3, 1 (1993), 77–92. https://doi.org/10.1017/S0956796800000617

J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. 1963. Revised Report on the Algorithm Language ALGOL 60. *Commun. ACM* 6, 1 (Jan. 1963), 1–17. https://doi.org/10.1145/366193.366201

Mark Bailey. 2007. The Grammar of Graphics. *Technometrics* 49, 1 (2007), 104. https://doi.org/10.1198/tech.2007.s456

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

R.C. Gentleman, V.J. Carey, D.M. Bates, B. Bolstad, M. Dettling, S. Dudoit, B. Ellis, Laurent Gautier, Y.C. Ge, J. Gentry, K. Hornik, T. Hothorn, W. Huber, S. Iacus, R. Irrizary, F. Leisch, C. Li, M. Maechler, A.J. Rossini, G. Sawitzki, C. Smith, G. Smyth, L. Tierney, J.Y.H. Yang, and J.H. Zhang. 2004. Bioconductor: open software development for computational biology and bioinformatics. *Genome Biology* 5 (2004). https://doi.org/10.1186/gb-2004-5-10-r80

Paul Hudak. 1989. Conception, Evolution, and Application of Functional Programming Languages. *ACM Comput. Surv.* 21, 3 (Sept. 1989). https://doi.org/10.1145/72551.72554

John Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (1989), 98–107. https://doi.org/10.1093/comjnl/32.2.98

Ross Ihaka and Robert Gentleman. 1996. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics* 5, 3 (1996), 299–314. http://www.amstat.org/publications/jcgs/

T. Kalibera, P. Maj, F. Morandat, and J. Vitek. 2014. A Fast Abstract Syntax Tree Interpreter for R. In *Conference on Virtual Execution Environments (VEE).*

Filip Krikava and Jan Vitek. 2018. Tests from traces: automated unit test extraction for R. In *International Symposium on Software Testing and Analysis (ISSTA).* https://doi.org/10.1145/3213846.3213863

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 (1995). Issue 4.

Shawn T. O'Neil. 2015. Implementing Persistent O(1) Stacks and Queues in R. *The R Journal* 7 (2015). Issue 1.

Kent M. Pitman. 1980. Special Forms in LISP. In *LISP Conference.* 179–187.

Barbara G. Ryder and Brent Hailpern (Eds.). 2007. *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007.* ACM. http://dl.acm.org/citation.cfm?id=1238844

David Smith. 2011. The R Ecosystem. In *The R User Conference 2011.*

O. Tange. 2011. GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine* 36, 1 (Feb 2011), 42–47. http://www.gnu.org/s/parallel

D. A. Turner. 1979. A New Implementation Technique for Applicative Languages. *Softw., Pract. Exper.* 9, 1 (1979), 31–49. https://doi.org/10.1002/spe.4380090105

D. A. Turner. 1981. The semantic elegance of applicative languages. In *Conference on Functional programming languages and computer architecture, FPCA.* https://doi.org/10.1145/800223.806766

D. A. Turner. 1985. Miranda: A Non-Strict Functional language with Polymorphic Types. In *Functional Programming Languages and Computer Architecture, FPCA*. https://doi.org/10.1007/3-540-15975-4_26

D. A. Turner. 2012. Some History of Functional Programming Languages. In *Trends in Functional Programming - 13th International Symposium, TFP*. 1–20. https://doi.org/10.1007/978-3-642-40447-4_1

Mitchell Wand. 1998. The Theory of Fexprs is Trivial. *Lisp and Symbolic Computation* 10, 3 (1998).

Hadley Wickham. 2016. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. http://ggplot2.org

Hadley Wickham. 2017. *tidyverse: Easily Install and Load the 'Tidyverse'*. https://CRAN.R-project.org/package=tidyverse R package version 1.2.1.

Hadley Wickham, Romain François, Lionel Henry, and Kirill Müller. 2018. *dplyr: A Grammar of Data Manipulation*. https://CRAN.R-project.org/package=dplyr R package version 0.7.8.

Andrew K. Wright and Matthias Felleisen. 1992. A Syntactic Approach to Type Soundness. *Information and Computation* 115 (1992), 38–94.