On the Design, Implementation and Use of Laziness in R

Promises Made, Promises Forced

AVIRAL GOEL, Northeastern University, USA
JAN VITEK, Czech Technical University and Northeastern University, USA

The R programming language has been lazy for over twenty five years. This paper presents a review of the design and implementation of call-by-need in R, and a data-driven study of how generations of programmers have put laziness to use in their code. We analyze 16,707 libraries and observe the creation of 270.9 B promises. Our data suggests that there is little supporting evidence to assert that programmers use laziness to avoid unnecessary computation or to operate over infinite data structures. For the most part R code appears to have been written without reliance on, and in many cases even knowledge of, delayed argument evaluation. The only significant exception is a small number of packages and core libraries which leverage call-by-need for meta-programming.

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 and Jan Vitek. 2018. On the Design, Implementation and Use of Laziness in R: *Promises Made, Promises Forced. Proc. ACM Program. Lang. 1, OOPSLA, Article 1 (January 2018), 28 pages.*

1 INTRODUCTION

Since its inception, in 1993, R has had a call-by-need semantics. When a function is invoked its arguments are packaged up into *promises* which are evaluated on demand. The values obtained by evaluating those promises are memoized to avoid the need for recomputation. Thus the following definition when called with `a+b` and `d+d`

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

evaluates `a+b` and does not evaluate `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 16,000 packages with an average of 6 new ones added every day [Ligges 2017]. Bioconductor [Gentleman et al. 2004], another open source R package repository, hosts over 1,700 libraries for high-throughput genomic data analysis.

It is fascinating to observe that R's laziness mostly remains secret. The majority of end-users are unaware of the semantics of the language they write code in. Anecdotally, this holds even for colleagues in the programming language community who use R casually. 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 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 for programmers: (1) Sub-computations are only performed if they are needed for the final result; (2) Unbounded data structures include elements which are never materialized. Haskell is a language designed and implemented to support lazy evaluation. Its compiler has optimizations passes that remove some of

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

the overhead of delayed evaluation, and its type system allows laziness to co-exist with side effects in an orderly manner.

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

Given the above, one may wonder *why bother being lazy?* In particular, when this implies run-time costs that are bound to be significant as R does not optimize laziness. Personal communication with the creators of R reveals call-by-need was added to support meta-programming in general and, in particular, to support user-defined control structures. While Scheme provided inspiration for the design of R, its macro-based approach to meta-programming was not adopted. Macros were deemed too complex for R's intended users. Instead, R offers a combination of call-by-need and reflection. Call-by-need postpones evaluation so that reflection can inspect and modify expressions, either changing their binding environment or modifying their code. This approach was deemed to be sufficiently expressive for the envisioned use-cases and, unlike macros, it was not limited to compile-time redefinitions—an important consideration for an interactive environment. Evaluating how that choice worked out in the R ecosystem is an interesting but separate question left to future work.

This paper explains the semantics of call-by-need, describes its implementation in the production R virtual machine, and provides data about its use in practice. The latter is obtained by dynamic analysis of a corpus of programs obtained from the CRAN and Bioconductor software repositories. The analysis provides quantitative answers to questions related to the usage of laziness. Dynamic analysis is limited to behaviors that it observes, low coverage of the control flow of analyzed functions may overestimate the degree of strictness in those functions. We are encouraged by the fact that Krikava and Vitek [2018] reports line-level code coverage of over 60% for a similar corpus. To mitigate this threat to validity and explain some of the patterns in the quantitative data, we perform a qualitative analysis of a sample of our corpus. 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 232,290 scripts over 16,707 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 August 1st, 2019 from CRAN and Bioconductor. Our software and data was validated as Functional and Reusable and is available in open source from:

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

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 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]. 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")` creates a vector of two strings. The language does not differentiate scalars from vectors, thus `1==c(1)`. In order to enable equational reasoning, R copies most values accessible through multiple variables if they are written to. Consider the `swap` function which exchanges two elements in a vector and returns the modified vector:

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

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

Every linguistic construct is desugared to a function call, even control flow statements, assignments, and bracketing. Furthermore, all functions can be redefined in libraries or user code. This makes R both flexible and challenging to compile [Flückiger et al. 2019]. 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 (triple dots are a vararg), whose parameters `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 }
```

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

R supports reflection and meta-programming. The `substitute(e,envir)` function yields the parse tree of the expression `e` after performing substitutions defined by the bindings in `envir`.

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

R allows programmatic manipulation of parse trees, which are themselves first class objects. They are evaluated using the `eval(e,envir)` function. Environment on the call stack can also be accessed.

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. This section overviews the design and implementation of laziness. 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 Haskell are that promises capture the syntactic expression appearing at the call site, promises are forced more eagerly, and the lack of lazy data structures.

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

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

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

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

Some additional operations can be performed in C code through the native API for promises. The only relevant operation is the ability to assign a value to a promise without evaluating it.

```
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

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 how promises can be returned from their creation environment, namely by being protected by an environment.

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. 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 evaluates the argument list left to right before a call.
- *Specials*: There are 46 specials. They 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 context-sensitive lookup, one of the most unusual features of R. When looking up a variable `x`, R performs a normal lexically scoped lookup. But, if the variable is in head position of a function call, e.g., `x(...)`, 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 is that lookup will force promises encountered on the way.

3.1 Implementing Promises

A promise has four slots: `exp`, `env`, `val` and `forced`. The `exp` slot contains a reference to the code of the promise, the `env` refers to environment in which the promise was originally created. It is typically the caller environment, except for default parameter values where the environment is that of the callee. The `val` slot holds the result of evaluating the promise. The `forced` slot is a flag used to avoid recursive evaluation of a promise. When the value of a promise is requested, the `val` slot is inspected first. If it is bound to a value, that value is returned. Otherwise, the `forced` 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 `forced` flag is unset. The implementation does little to optimize promises. In some cases, a promise can be created pre-forced with a value pre-assigned. The GNU R implementation recently added a bytecode compiler, this compiler eliminates promises when they contain a literal value [Tierney 2019].

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 and out-of-scope assignments. Instead, we add delayed assignment, default values for arguments, `substitute` and `eval`. To support these features we add strings as a base type and the ability to capture the current environment.

Figure 2 gives the syntax and semantics of our calculus. The surface syntax includes terms for strings, variables, string concatenation, assignment, function declaration, function invocation (one and zero argument functions), environment capture, substitution, eval, and delayed assignment.

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

$$\mathbb{C} \;::=\; [] \mid \mid \mid \mathbb{C}\,\#\,\mathtt{e} \mid v\,\#\,\mathbb{C} \mid \mathtt{x} \leftarrow \mathbb{C} \mid \mathbb{C}(\mathtt{e}) \mid \mathbb{C}() \mid \mathtt{eval}(\mathbb{C},\,\mathtt{e}) \mid \mathtt{eval}(v,\,\mathbb{C}) \mid \mathtt{delay}(\mathtt{x},\,\mathtt{e},\,\mathbb{C})$$

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

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

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

Assign
$$\frac{E = l' \cdot E' \qquad H(l') = F \\ F' = F[\mathtt{x} \mapsto v] \qquad H' = H[l' \mapsto F']}{\mathbb{C}[\mathtt{x} \leftarrow v]\,E \cdot S;\ H \;\to\; \mathbb{C}[v]\,E \cdot S;\ H'}$$

Delay
$$\frac{H(l) = l' \cdot E' \qquad H(l') = F \qquad \textit{fresh } l'' \\ F' = F[\mathtt{x} \mapsto \mathtt{prom}(l'')] \quad H' = H[l'' \mapsto (\bot,\,\mathtt{e},\,E,\,\downarrow)][l' \mapsto F']}{\mathbb{C}[\mathtt{delay}(\mathtt{x},\,\mathtt{e},\,\mathtt{env}(l))]\,E \cdot S;\ H \;\to\; \mathbb{C}[\mathtt{env}(l)]\,E \cdot S;\ H'}$$

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

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

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

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

Invk1
$$\frac{v = (\lambda \mathtt{x}{=}\mathtt{e}'.\mathtt{e}'',\,E') \qquad \textit{fresh } l, l' \\ E'' = l \cdot E' \qquad H' = H[l \mapsto F] \\ F = [\mathtt{x} \mapsto l'] \qquad H'' = H'[l' \mapsto (\bot,\,\mathtt{e},\,E,\,\downarrow)]}{\mathbb{C}[v(\mathtt{e})]\,E \cdot S;\ H \;\to\; \mathtt{e}''\,E'' \cdot \mathbb{C}[v(\mathtt{e})]\,E \cdot S;\ H''}$$

Invk0
$$\frac{v = (\lambda \mathtt{x}{=}\mathtt{e}'.\mathtt{e}'',\,E') \qquad \textit{fresh } l, l' \\ E'' = l \cdot E' \qquad H' = H[l \mapsto F] \\ F = [\mathtt{x} \mapsto l'] \qquad H'' = H'[l' \mapsto (\bot,\,\mathtt{e}',\,E'',\,\downarrow)]}{\mathbb{C}[v()]\,E \cdot S;\ H \;\to\; \mathtt{e}''\,E'' \cdot \mathbb{C}[v()]\,E \cdot S;\ H''}$$

Ret1
$$\frac{}{v\,E' \cdot \mathbb{C}[v'(\mathtt{e})]\,E \cdot S;\ H \;\to\; \mathbb{C}[v]\,E \cdot S;\ H}$$

Ret0
$$\frac{}{v\,E' \cdot \mathbb{C}[v'()]\,E \cdot S;\ H \;\to\; \mathbb{C}[v]\,E \cdot S;\ H}$$

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

Lookup2
$$\frac{\textit{get}(H,\,E,\,\mathtt{x}) = \mathtt{prom}(l)}{\mathbb{C}[\mathtt{x}]\,E \cdot S;\ H \;\to\; \mathtt{prom}(l)\,E \cdot \mathbb{C}[\mathtt{x}]\,E \cdot S;\ H}$$

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

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

Memo
$$\frac{v \neq \mathtt{prom}(l'') \quad H(l) = (\bot,\,\mathtt{e},\,E',\,\uparrow) \quad H' = H[l \mapsto (v,\,\mathtt{e},\,\epsilon,\,\downarrow)]}{v\,E' \cdot \mathtt{prom}(l)\,E \cdot S;\ H \;\to\; v\,E \cdot S;\ H'}$$

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

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

Fig. 2. Syntax and Semantics

The syntax is extended with 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 $(\lambda \texttt{x}{=}\texttt{e.e},\ E)$, 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 the $\perp$ value to denote an invalid reference.

The reduction relation is of the form $S\ E \rightarrow S'; E'$ where the stack S is a collection of expression-environment pairs ($\texttt{e}\ E$) and the heap maps variables to values. Frames are mutable and can be shared between closures, so they are stack allocated. Promises are quadruples $(v,\ \texttt{e},\ E,\ R)$ 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. Evaluation contexts $\mathbb{C}$ are deterministic. Following R, some builtin operations such as string concatenation are strict. But function calls are lazy in their argument, the expression in `x (e)` remains untouched by the context. 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.

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

Takeaways. R is rather strict for a lazy language. This manifests itself in the definition of evaluation contexts. Intuitively, any position where $\mathbb{C}$ appears is evaluated strictly in left-to-right order. The key place where R differs from other lazy languages is that the right-hand side of assignments is strict. Our semantics does not show data structures, in R they are all strict. Strictness also shows up in the `Ret1` and `Ret0` rules which force the evaluation of the return expression. Lastly, strictness is enforced in the `Lookup2` rule which does function lookup. If a promise is returned, it must be evaluated. Another property that the semantics ensures is that promises are stored in environments and, whenever they are accessed they are forced. The only way for a promise to outlive the frame that created it is to be returned as part of an environment or closure. Following R, it is possible to create a cycle in promise evaluation, the expression $(\texttt{fun}(\texttt{x}{=}\texttt{x})\,\texttt{x})()$ 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 results in a stuck state in the semantics.

5 ANALYSIS INFRASTRUCTURE

We now explain the infrastructure that assembles the corpus and collects and analyzes traces. Our analysis pipeline starts with scripts to download, extract and install open source R packages. Next, an instrumented R virtual machine generates events from program runs. This is followed by an analyzer that processes the execution traces to generate tabular data files in a custom binary format. Another set 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].

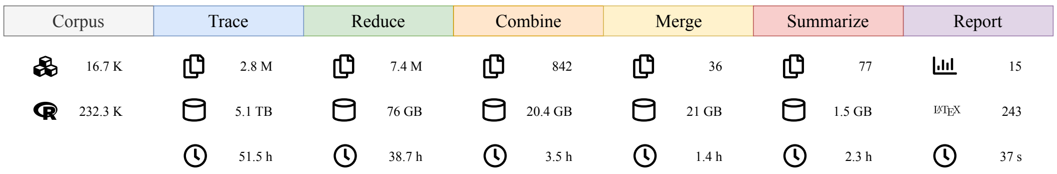

Fig. 3. Tracing Pipeline

Figure 3 shows the six stages of our pipeline. we give corpus size, number of files generated, size of data and time taken by each step. The remainder of this section details the various stages.

5.1 Instrumented R

The instrumented R Virtual Machine is based on GNU-R version 3.1.0. Its goal is to produce program execution traces with all the events required to answer our research questions. On the face of it, this is not a difficult task. And, in the end, we only need to add 818 lines of C code to expose an event data structure with fields to describe a variety of execution events that capture the internal interpreter state. The challenge was identifying where to insert those 818 lines in 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. 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 data structures duplicated in various files. We succeeded by a lengthy trial and error process. The events record by the instrumented virtual machine are:

- **`Call`, `Return`**: these events are generated on function calls and carry information about the type of function, its arguments, its environment and return value or exception code.
- **`S3Call`, `S3Return`**: the S3 system is a single dispatch object system, the events record the method name, the receiver's class name, the object on which dispatch occurred and return value or error code.
- **`S4Entry`, `S4Exit`**: the S4 system is a multi-dispatched object system, we record method name, environment, method definition, arguments and return.
- **`Eval`**: entry in the interpreter's `eval` function.
- **`CtxtEnter`, `CtxtExit`**: these events record when stack frames are pushed and popped with their address.
- **`CtxtJmp`**: this event is triggered upon various uses of `longjump` for non-local returns.
- **`PromEntry`, `PromExit`**: these events record which promise is being evaluated and when evaluation terminates.
- **`PromRead`**: this event occurs when the value of a promise is read, it records the promise and the returned value.
- **`PromSubst`**: this event occurs when the expression stored in a promise is accessed.

- `Alloc`: this event records which object was allocated.
- `Free`: this event occurs when the garbage collector marks an object as free.
- `VarDef`: this event records the definition of a variable, including its symbol, value and environment.
- `VarWrite`: this event records a write to a variable, including its symbol, value and environment.
- `VarRem`: this event records the deletion of a variable, including its symbol and environment.
- `VarRead`: this event records a variable read including its symbol, value and environment.

Events can be disabled to ignore events generated by the internals of the R virtual machine and also avoid recursion. R objects captured in events are protected from the garbage collector to prevent them from being reclaimed during analysis.

5.2 Tracer

The tracer is a small R package (73 LOC) that calls into a larger C++ library (6,080 LOC). It is loaded in the instrumented R virtual machine and, during program execution, it maintains objects that model various aspects of the program such as functions, calls, promises, variables, environments, stacks and stack frames. As events are generated, the tracer update its model of the state. Table 1 is representative of the size of the various data structures used by the tracer (it gives number of members of the C++ structs and they size in bytes). The tracer is able to process 803.1 K events per second on our benchmark machine.

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

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 model objects

The tracer generates large amounts of data. Our first prototype used Sqlite to store the generated data. However, we found the approach limiting. Firstly, during development we kept running into errors because the database schema and the tracer were out of sync. Due to the iterative nature of data analysis, we were modifying the schema frequently. This 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 database operations could run for days. Thirdly, database insertions ended being a bottleneck. We were able to trace fewer than 2000 packages in two days and filled up a 2 TB disk. In the end, we implemented 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.

5.3 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 set up the tracer and initialize it with paths to input and output data before execution. For each package, the tracer generates 12 data files and 4 status file. The status files denote the different possible states of the tracing. They allow the infrastructure to discard data from failing programs. The R scripts responsible for generating traces do extensive logging of intermediate steps for debugging purposes.

5.4 Post-processing

Scale was our major challenge. We faced difficulties both due to execution time and data size. In the 232,290 programs that were traced, the tracer observed 1.7 T expression evaluations, 831 B calls to 698.4 K functions and 270.9 B promises. The raw data generated by the tracer is 5.1 TB but the reduced data is just 76 GB. In hindsight, it appears that incorporating analysis in the tracer, i.e., presummarizing data in C++ 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 analyses 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. The steps are detailed next.

- *Prescan*: Scan the raw data directory and output a list of all the subdirectories that contain raw data. There is a directory per package and multiple files in each directory.
- *Reduce*: Given a list of directories, this step uses GNU Parallel to partially summarize the raw data. This is the most expensive step in terms of size and speed. Since the data files are large, we must limit the degree of parallelism drastically to avoid running out of memory.
- *Scan*: Create a list of all the files successfully reduced.
- *Combine*: Combine information from all the programs into a single data table per analysis question.
- *Summarize*: Compute summaries of the merged data for: (1) event frequency, (2) object frequency, (3) functions with their definitions, (4) argument information, (5) escaped arguments, (6) information about parameters, (7) information about promises.
- *Report*: Generate graphs and tables from an RMarkdown notebook as well as LaTeX macros for inclusion in the paper.

5.5 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 August 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. We believe this corpus is representative of sophisticated uses of the R language. Anecdotal evidence suggests that the majority of R code written is made up of small scripts, straight-line sequences of library calls, that read data, apply some models to it and then visualize the results. Most end-users neither define functions nor write loops, their code is quite simple. Without a source of end-user code it is not possible to validate this hypothesis, but if true then our corpus is representative of the interesting R code. We also mention that R is used in many industrial settings that do not publish their code to open source repositories. We have no information on those use-cases.

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

We believe that these errors may be fixable but automating those fixes would be harder. So we simply choose to ignore the packages that could be installed. Out of the installed packages, we were able to successfully record execution traces for 16,707 packages. Some packages did not trace owing to runtime failures. Again, we ignore the failing packages on the grounds of having a sufficient number of running ones

	Tests	**Examples**	**Vignettes**	
Scripts	44.1 K	220.6 K	9.8 K	*Install*
	23.3 K	202.4 K	6.6 K	*Trace*
LOC	2.7 M	1.6 M	614.6 K	*Install*
	1.3 M	1.6 M	327.2 K	*Trace*

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. Table 2 gives the number of scripts of each kind that could be installed and the number of scripts that were successfully traced. In terms of lines of code, we exercised 25.6 M lines of R and 10.4 M lines of C. The total size of our database is 5.2 TB.

We observed 831 B calls to 698.4 K functions. Out of these, 23.7 B calls exit early due to longjumps (explicit use of the `return` function). Of the remaining 807.4 B (97%) calls to 509.8 K functions, 215.2 B (26.7%) calls are made to 157 builtin functions, and 486.5 B (60.3%) calls are made to 33 special functions. Of the calls to R functions 2.4 B (2%) calls are made to 34.1 K (6.7%) S3 methods, 834.5 M (0.8%) calls are made to 60.2 K (11.8%) S4 methods and 102.4 B (97%) calls are made to 415.3 K (81.5%) ordinary R functions. 82 M (0.08%) calls are made to e `delayedAssign`, 101.8 M (0.1%) calls are made to `force`, 1.3 B (1%) calls are made to `forceAndCall` and 1.7 B (2%) to `substitute`.

Figure 4 shows how many functions were exercised in each test package. More than 9K (59.2%) of the packages had over 10 functions called in our corpus. On the other hand 2.7K of the packages had a single function invoked (7.0%). These may either be small packages, or, more likely, the provided tests have low coverage.

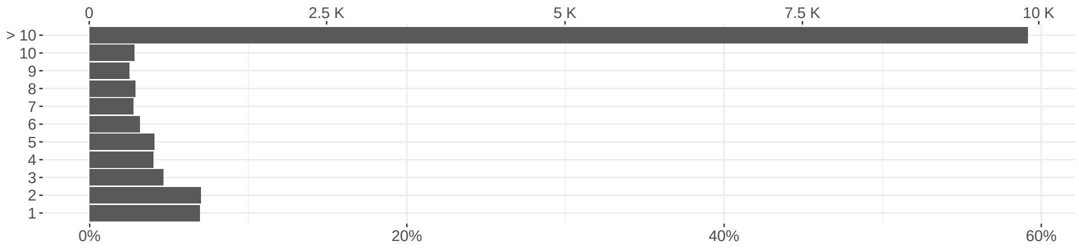

Fig. 4. Functions per package

Figure 5 shows how many times functions were called. 45.0% of the exercised functions were called more than ten times and 18.9% of the functions were called only once. Clearly functions that are called only once are likely to not have sufficient coverage.

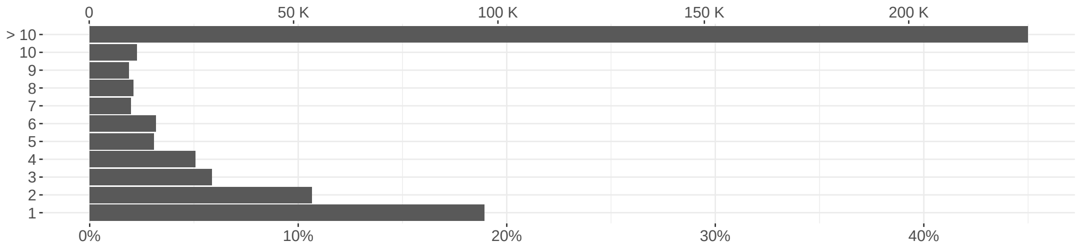

Fig. 5. Function calls

Figure 6 shows how many parameters exercised functions have. 43.7% have have a single parameter and 9.8% have more than five parameters. Function `meta::forest.meta` has the maximum, 199 parameters! We observed 2.1 M distinct parameters. Of the 261 B arguments 72.9 B (27.9%) were default arguments and the remaining 188.1 B (72.1%) were non-default arguments. 7.4 B (3%) arguments were missing.

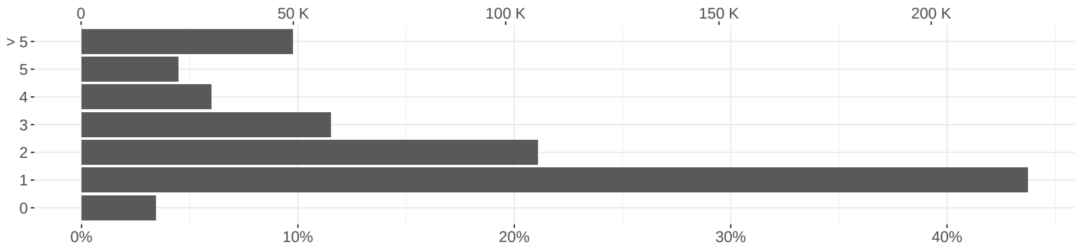

Fig. 6. Formal Parameters

7 ANALYZING LAZINESS USAGE PATTERNS

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

7.1 Life Cycle of Promises

The first research question we address aims to shine a light on how promises are created and used in the wild.

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

In our corpus, and likely, all R programs, promises are the most frequently allocated object. We observed the creation of 270.9 B promises. To provide context, Figure 7 shows the distribution of application-level objects created in the corpus. Most end-user data are vector of characters, logical, integers, doubles or complex values, in that order. Lists are polymorphic data structures that are usually in library code and internal functions. Raw values are used to hold uninterpreted byte strings. Closures represent functions and environments are sets of name-value bindings, symbols are language-level names and S4 are instance of classes. Environments are frequently observed because one is created for each function call, but they are also used, albeit rarely, as hashmaps in user code.

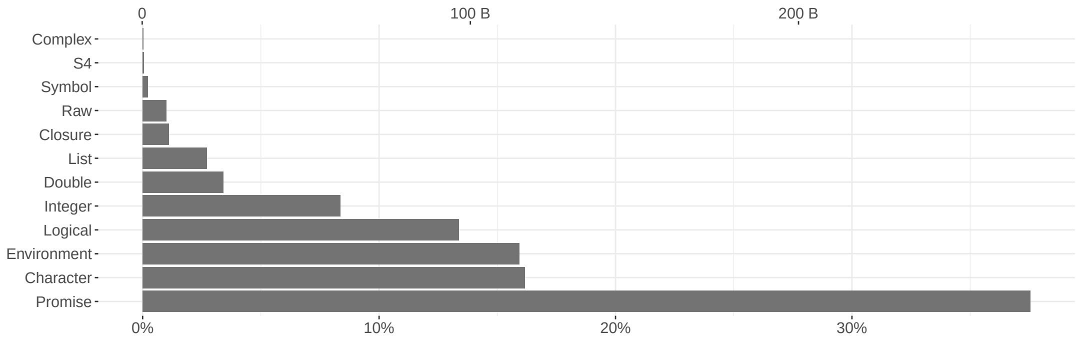

Fig. 7. Object counts

Where are promises created? Argument lists account for 94.3% of promises. The remainder are used internally, e.g. for lazy loading of functions. One could expect that there would be more promises than values since every operation in R is a function call. This is not the case because some values are composite of multiple simpler elements (e.g. data frames) and these are wrapped in a single promise. Furthermore, some values are returned without being bound to a promise. Lastly, some internal functions do not take promises. The promises that are not created in argument lists can also be created by calls to `delayedAssign` or in internal functions of the R virtual machine.

What do promises yield? Figure 8 shows the contents of forced promises. The most common types are character, logical, environment, closure, integer, double and null, in that order. The presence of null values is explained by the fact that many default parameter values are set to null and these default values are promised. S4 objects are rarely used in R programs outside of Bioconductor. Complex and Raw values are also quite rare in practice.

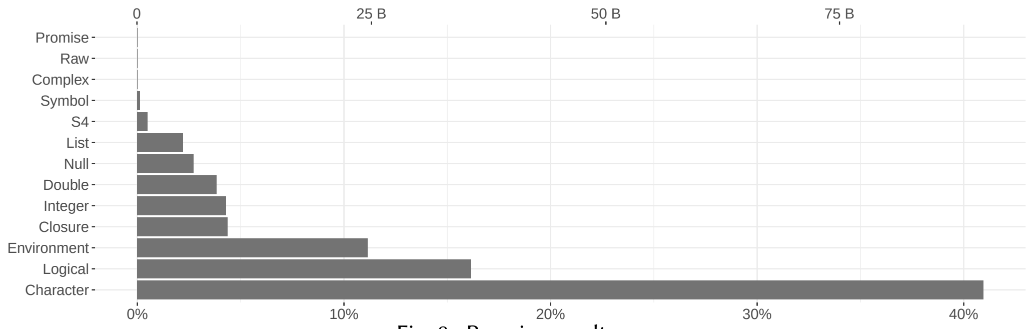

Fig. 8. Promise results

Figure 9 shows the content of the expression slot of promises, i.e. their code. Only 17.3% of promises contain a function call, e.g. `1+2` or `f(z)`. The majority contain a single symbol to be looked up in the promise's defining environment, e.g. `x`. Of course that symbol may be bound to another promise, in which case forcing the promise will be recursive. Promises can also hold inlined scalar constants, such as a single double `1.1`, etc.

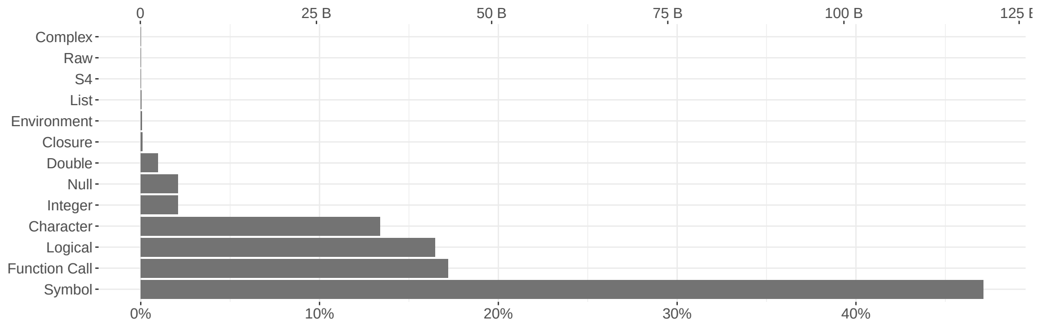

Fig. 9. Promise expressions

How often are promises accessed? We observed that 87.3% of argument promises are forced. The remaining went unused. This is not unusual in R where some functions have over twenty parameters, and many of these are only needed in special circumstances. Figure 16 shows, for each individual promise, the number of times its value was read. Most promises are used once, 9.6% are accessed twice, and 3% are accessed three times.

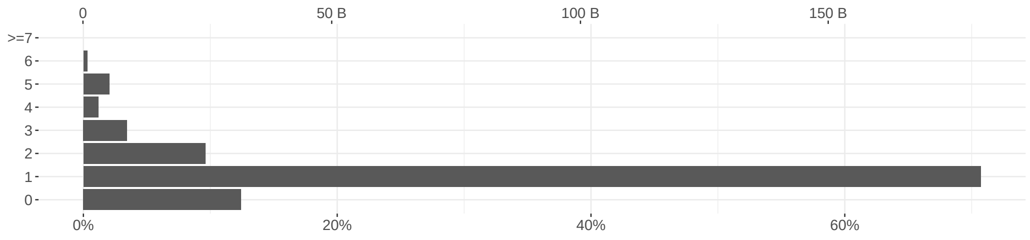

Fig. 10. Reads

How far are promises forced from their creation? Promises can be passed from one function to the next, traveling down the call stack. Regardless of the distance from the frame that created them, promises evaluate in the environment in which they were created. But the further from their creation point, the harder it is, e.g., for a compiler to optimize their evaluation.

Figure 11 aggregates forced argument promises by the number of function calls between their point of creation and forcing. 79.7% promises are evaluated in the call to which they are passed as arguments. 17.1% promises are forced by the immediate callee. The remaining 3% promises are evaluated deeper in the call-chain.

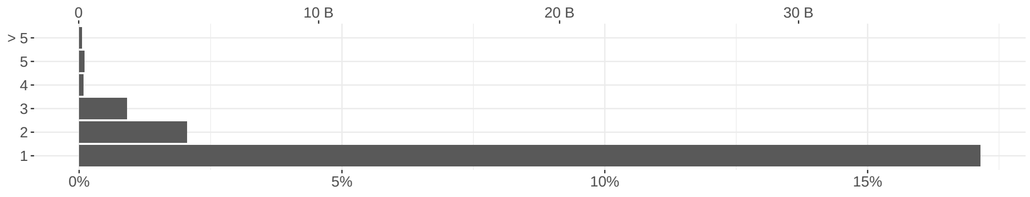

Fig. 11. Force depth

How long do promises live? Promises are short-lived, over 99.5% do not survive past the first garbage collection cycle following their creation. This follows the folklore in memory management that most objects die young. But it also means that promises are exerting pressure on the memory subsystem of the R virtual machine. Only 0.5% of promises survive multiple collection cycles. Of those, 76.7% are non-argument promises and 0.08% are escaped arguments. As mentioned earlier, non-argument promises are created explicitly through `delayedAssign` and implicitly through lazy-loading of package code; both are expected to be long lived. Escaped promises are promises that outlive their defining function. They may be long lived as well. Of the long lived promises, 23.2% are argument promises; their longevity is due to long running functions.

What are promise lifecycles? If we characterize the life of a promise by the events that affect it, we can summarize promise lifecycles by sequences of those events. Ignoring creation and reclamation, the events of interest are Force, Read, Meta-program, Escape, Assign, and deSerialize. In the corpus, we observed 28.1 K unique lifecycles. Figure 12 shows the most frequent sequences of promise life events differentiating between (a) argument promises (19.2 K unique sequences), (b) escaped argument promises (7.3 K sequences) and (c) non-argument promises (6.2 K sequences). For argument promises the two most common sequence are F (a promise that is forced) and the empty sequence (unused promise). The next sequences are forces followed by a growing number of reads. Meta-programming occurs only infrequently. For escaped promises the most frequent sequence is EF, the promise escapes and is forced later. The second most frequent sequence is FRER, the promise is forced, read, escapes and is read again. Lastly, non-argument promises are most often created and assigned a value in the C code. About 0.2% of these promises are obtained from deserialization (S) owing to lazy-loading of packages.

F	70.5%
	11.9%
FR	9.5%
FRR	3%
FRRRR	2%
FRRR	1%
M	0.5%

(a) Argument

EF	70.7%
FRER	5.4%
FER	5%
FRRER	4%
EFR	3%
FRRERR	1%
EFRR	1%

(b) Escaped

A	67.5%
	16.5%
AR	11.2%
AA	2%
F	2%
S	0.2%

(c) Non-Argument

Fig. 12. Promise lifecycle

Does context sensitive lookup force promises? Looking up a function name such as `f()` may force promises if that name is bound to one in the environment. If the promise yields a closure, that closure will be invoked, otherwise lookup continues. This allows “harmless” shadowing of function names as seen in Figure 13 where `addToGList.grob` from package `grid` defines a parameter `gList` that shadows a function of the same name defined by the same library.

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

Fig. 13. Shadowing

We observed 651.7 K funtion name lookups (out of a total of 831 B lookups) which caused a promise to be forced. Out of those, 86.3% forces yielded a closure, and 13.7% did not yield a closure. The latter are cases where a function name was shadowed. Figure 14 shows the 30 most commonly shadowed function names and the number of functions in which shadowing happens. Many of those names correspond to common R functions such as `c`, `names`, `print` and `max`.

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

Fig. 14. Context sensitive lookup

Do promises force each other? The parameter list of a function can include parameters with default values that refer to each other. This is a semantic quirk that can cause promises to force one another. Consider Figure 15 and function `sample.int` from the `base` package. Parameter `useHash` has a default value that depends on the other four arguments and `s` depends on `n`. Function `rmslash` has a recursive dependency between `center` and `Scatter`. The function expects at least one of those to be provided at each call site, if this is not the case an error will be reported.

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

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

Fig. 15. Argument lists

We found 4.8 K default value expression occuring in 3.9 K functions. At run-time, we observed 336.9 M occurences of default expressions forcing other parameters. This is 0.2% of the forced promises.

Does method dispatch force promises? In R, the two widely used object systems, S3 and S4, have an impact on promises. In order to find which method to invoke they force one or more arguments. Overall, only 1% of promises participate in method dispatch. 671.5 M promises are forced due to S3 dispatch and 536.2 M are forced due to S4 dispatch. As these numbers are small, we ignore dispatch for the remainder of the study.

How often does non-local return happen. In R, the `return` *function* pops the call stack until it arrives at the function with the same environment from which it was invoked. When a promise containing a call to `return` is forced by callee, stack unwinding abruptly ends execution of the callee and returns from the caller. Such cases are classified as non-local returns. Only 297.4 M arguments to 16 functions do non-local returns. One of the most common causes of non-local arguments is the `base::tryCatch` function in R. The function sequentially attaches handlers specified in its vararg list and executes `exp` by wrapping it in a call to the `return` function. The return function causes the control flow to exit `doTryCatch` and after that, `tryCatch`.

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

Takeways. Argument promises dominate the memory profile of R programs. They are mostly short lived, 80% are evaluated by their caller and over 99% do not survive a single GC cycle. The vast majority of promise expressions are a value or a variable. Only 17.3% contain expressions that need to be evaluated. Of those expression-carrying promises 80.7% are unused, 7.0% are evaluated in the caller and 12.3% are evaluated at a remote location or meta-programmed. Overall most promises lead a rather mundane life that one would hope a compiler could optimize out of existence.

7.2 Strictness

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

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

To answer this question we start with individual parameters. We only consider functions that are called more than once and have non-zero parameters. We observed 2.1 M distinct parameters. For a given parameter and a given function, we aggregate all calls and all uses of that parameter into three categories: parameters that are *Always* evaluated, parameters that are *Never* evaluated, and parameters that are *Sometimes* evaluated. Figure 16 summarizes this analysis. 87.6% of parameters

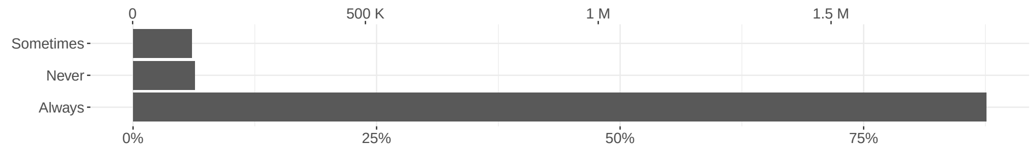

Fig. 16. Parameter strictness

are always evaluated, 6.0% parameters are evaluated in some calls and not others, 6.4% parameters are never used in our corpus.

For a function to be strict, all of its parameters should be always evaluated in the same order. Most functions, 93.6% to be precise, have a single observed order of evaluation. This means these functions are candidate for being strict. Functions with multiple orders of evaluation for their arguments are summarized in Figure 17. About 4.7% of the functions have two force order, and very few functions have more.

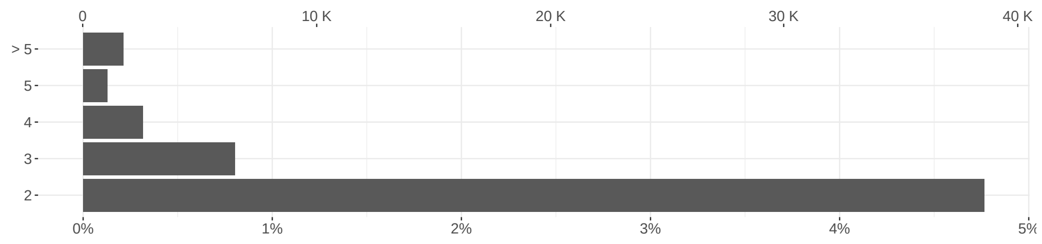

Fig. 17. Function force orders

Based on the above data, out of a total of 388.3 K functions, 83.7% are strict. Figure 18 gives a histogram of function strictness ratios per package. The majority of packages contain only strict functions. The packages that are less than 75% strict account for only 2.6 K packages (16.9% of all packages) and 81.7 K functions (21.0% of all functions).

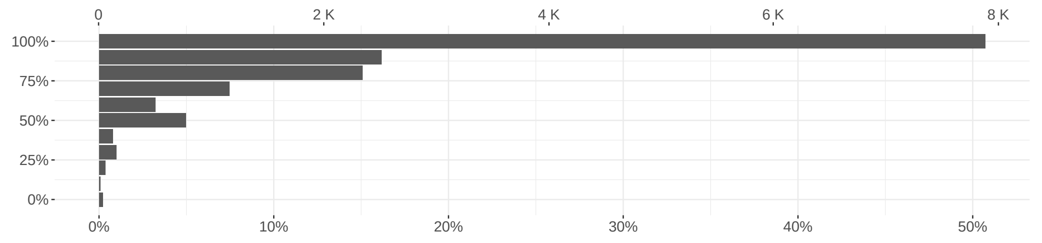

Fig. 18. Strictness per package (x-axis=packages; y-axis=strict function ratio)

One could consider relaxing strictness to allow multiple orders of evaluation if those order of evaluation could be shown to be semantically equivalent. Some features of R may help. First, all vectors have a copy-on-write semantics, thus many side-effects are hidden from view. Moreover, as Figure 19 shows only 25% of <u>Sometimes</u> promises perform any computation. In our experience most of those computations are side-effect free.
We performed an additional analysis to get an upper bound on the side effects performed during promise evaluation. A meager 16.5 M promises (out of 270.9 B) perform any side-effecting computation. There are several cases to consider, the simplest when the promise performs a side effect to its local environment. For example, consider the `stats::power` function:

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

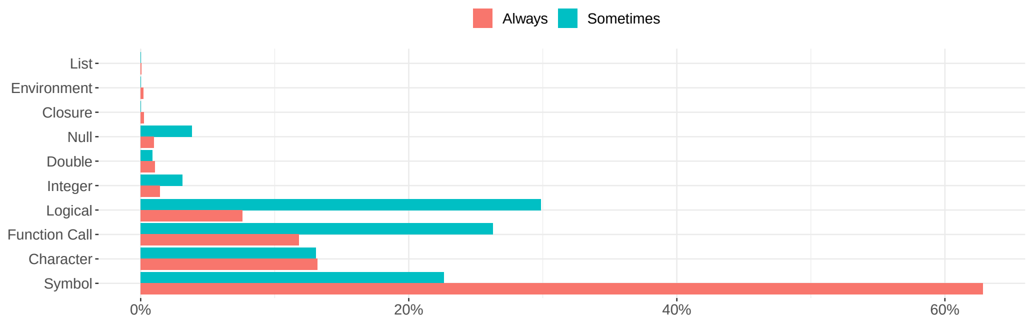

Fig. 19. Promise expressions

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

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

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

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

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

Qualitative analysis

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

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

Higher-order functions such as `apply` or `reduce` often enforce strictness following bug reports and confusion from users around unwanted lazy evaluation and variable capture interactions. This has been mitigated by the introduction of `forceAndCall` function which forces the arguments of a function prior to its execution. 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 1.3 B times. The `force` function is also widely used to enfore strictness. At runtime, 101.8 M calls 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. Improving the precision of dynamic analysis would require increasing code coverage. We also found numerous occasions where programmers require strictness to avoid unpredictable side-effects.

7.3 Meta-programming

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

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

For this purposes of this discussion we define meta-programming as the manipulation of code through calls to the `substitute` function which allows programmers to extract an abstract syntax tree from the body of a promise, modify it, and then evaluate it by calling `eval`. We observed 1.7 B calls (2% of all calls to functions) to `substitute`. Figure 20 shows the number of promises that were meta-programmed. The graph has four categories: promises that were created but never used, promises that were meta-programmed, promises that were both meta-programmed and accessed, and lastly, promises that were only accessed. The data shows that 0.5% of promises were used purely for meta-programming purposes, while 0.2% were both forced to obtain a value and used for meta-programming.

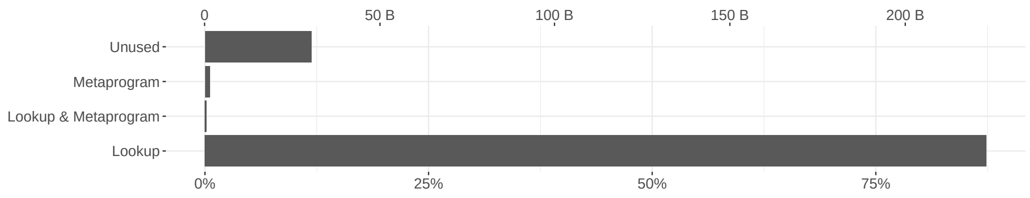

Fig. 20. Meta-programmed promises

The use of meta-programming is widespread and it is used many packages. There are 2 K (11.9%) packages that do meta-programming. One relevant feature of `substitute` is that it lets the programmer specify the environment in which to resolve names occuring in its argument. It is thus possible to access an environment up the call chain and invoke substitute on it. Over 99% of calls to `substitute` use the current evironment. Only 0.7% of calls use a custom replacement list or a new environment, and 209.9 K use a parent frame. These are almost entirely due to `deparse`, `eval` and `do.call` which allow specifying the environment in which to evaluate their arguments. To illustrate the discussion, the `envnames::get_env_names` function uses `substitute` in different contexts to extract user-defined names of environments. The first call to `substitute` is performed in the second frame from the top of call stack, the next call is performed in the first frame from the top of the call stack and the third call is performed in the same environment in which it is called.

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

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

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

Qualitative analysis

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

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

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

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

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

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

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

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

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

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

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

7.4 Revisiting the traditional benefits of laziness

The next research question asks whether the benefits of lazy evaluation that were advocated by Hudak [1989] are realized in the R ecosystem.

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

These benefits are that programmers do not have to worry about evaluation costs of potentially unused arguments and the ability to define and use unbounded data structures. We posit the following hypothesis, if programmers do not worry about the cost of evaluating terms, they will tend to write more expensive computations in non-strict parameters, whereas if they do worry, they will avoid costly arguments in non-strict positions. If true, one should observe a difference in running time for arguments known to be strict (promises passed to *Always* parameters) and those that are not (promises passed to *Sometimes* parameters). Figure 21 shows the probability density of promise 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. This data does not allow us to confirm that programmers are taking advantage of laziness to write potentially more expensive computations in non-strict argument positions.

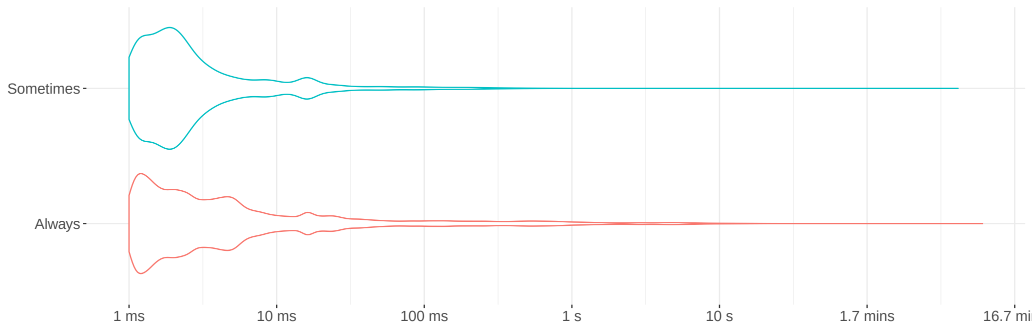

Fig. 21. Promise evaluation

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 programmers may have created some in their code. As we have shown, one can use promises together with environments to create unbounded data. While it is difficult to measure this directly. One thing that we can measure is *escaped promises*, the promises that outlive the function their are passed into. Of the 261 B promises we have observed, only 11.6 M escape. This is a rather small number. We need to establish the reason why the promises escape. Figure 22 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 symbols and closures as their return values. The next section performs a qualitative analysis to understand how those closures are used.

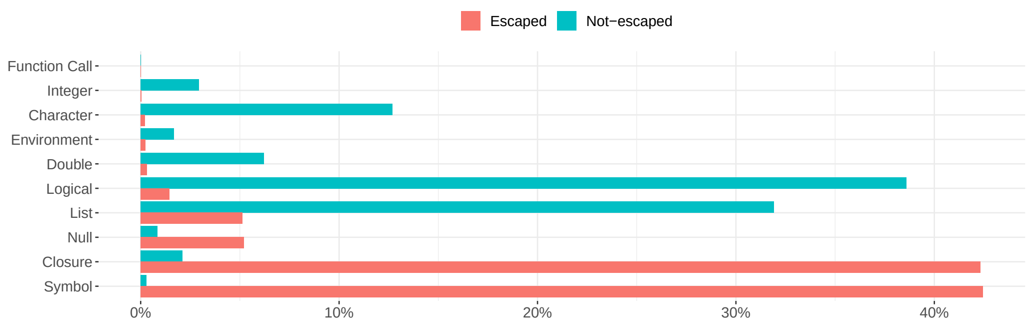

Fig. 22. Computation reuse

Qualitative analysis

It is not easy to asses from the quantitative results whether programmers benefit from laziness. Consider a call to a hypothetical function, `f(a+b,c)`, and imagine that depending on the value of `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 infrequently true and the programmer is aware of laziness, they would not worry about the performance implication of this code. But without laziness, the API of the function would have to change so that instead of passing the result of the computation one would pass the individual arguments and let the function perform the addition if needed. In our manual inspection, we have not found any code suggesting that programmers are concerned about the cost of evaluating expressions are described above. This is likely partly to the fact that many users are not performance sensitive.

One promising use of laziness is related to `delayedAssign`. We observed 82 M calls to this function. We inspected manually 36 packages that use it. We found several recurring patterns that aim to avoid unnecessary computation. Examples are: the `AzureML` package uses delayed assignment to avoid loading unneeded parts of its workspace; `crunch` uses it to delay fetching data from a server; and (slightly surprisingly) `callCC` uses it in conjunction with non-local return to implement the `call/cc` function. Overall 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. 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, (3) arguments stored in environments, (4) arguments passed into finalizers, (7) argument passed into delayed assignments, and (8) arguments passed into formulas. Figure 23 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. 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]. This package, which depends on lazy lists, is the only use of lazy data structures we are aware of in the R ecosystem.

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. 23. Escaping promises

8 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 laziness were modularity, referential transparency and the ability to work with infinite data structures [Hughes 1989]. These languages inspired the design of Haskell [Hudak et al. 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 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 quote, unquote and quasiquote user supplied expressions and evaluate them in 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 environments and existing laziness support in R.

Morandat et al. [2012] implemented a tool called TraceR for profiling R programs [Macnak et al. 2012]. The architecture of TraceR was similar to that of the pipeline presented here, but it did not target large scale data collection and has gone unmaintained for several years. Our current infrastructure is less invasive than the previous implementation and is being considered for inclusion in GNU R.

9 CONCLUSION

This paper offers 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 structures 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.

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 compilers and program analysis tools as they must deal with the possibility of any variable access causing side-effects. Lastly, the majority of users do not expect arguments to be evaluated in a lazy fashion, thus leading to hard to understand bugs.

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.

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.

Acknowledgments. We thank the reviewers for constructive comments that helped us improve the presentation. Early prototypes of our analysis were implemented by Konrad Siek and Jan Noha, we thank them for their contribution. We would also like to thank Ben Chung, Artem Pelenitsyn and Filip Křikava for their comments and encouragements. This work received funding from the Office of Naval Research (ONR) award 503353, the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement 695412), the NSF (awards 1518844, 1544542, and 1617892), and the Czech Ministry of Education, Youth and Sports (grant agreement CZ.02.1.010.00.015_0030000421).

REFERENCES

Lennart Augustsson. 1993. The Interactive Lazy ML System. *J. Funct. Program.* 3, 1 (1993). 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). https://doi.org/10.1145/366193.366201

Mark Bailey. 2007. The Grammar of Graphics. *Technometrics* 49, 1 (2007). 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.

Olivier Flückiger, Guido Chari, Jan Ječmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek. 2019. R Melts Brains: An IR for First-Class Environments and Lazy Effectful Arguments. In *Dynamic Language Symposium (DLS).*

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

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

John Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (1989). 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/

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

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.*

Ryan Macnak, Floreal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. 2012. TraceR: A framework for understanding R performance. In *The 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

Shawn T. O'Neil. 2015. Implementing Persistent O(1) Stacks and Queues in R. *The R Journal* 7 (2015). Issue 1. https://doi.org/10.32614/RJ-2015-009

Kent M. Pitman. 1980. Special Forms in LISP. In *LISP Conference.* https://doi.org/10.1145/800087.802804

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). http://www.gnu.org/s/parallel

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

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

David 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

David A. Turner. 2012. Some History of Functional Programming Languages. In *Trends in Functional Programming (TFP).* 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). https://doi.org/10.1023/A:100772063

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 Francois, Lionel Henry, and Kirill Muller. 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. https://doi.org/10.1006/inco.1994.1093