What We Eval in the Shadows

A Large-Scale Study of Eval in R Programs

AVIRAL GOEL, Northeastern University, USA
PIERRE DONAT-BOUILLUD, Czech Technical University in Prague, Czech Republic
FILIP KŘIKAVA, Czech Technical University in Prague, Czech Republic
CHRISTOPH M. KIRSCH, University of Salzburg, Austria and Czech Technical University in Prague, Czech Republic
JAN VITEK, Northeastern University, USA and Czech Technical University in Prague, Czech Republic

Most dynamic languages allow users to turn text into code using various functions, often named `eval`, with language-dependent semantics. The widespread use of these reflective functions hinders static analysis and prevents compilers from performing optimizations. This paper aims to provide a better sense of why programmers use `eval`. Understanding why `eval` is used in practice is key to finding ways to mitigate its negative impact. We have reasons to believe that reflective feature usage is language and application domain-specific; we focus on data science code written in R and compare our results to previous work that analyzed web programming in JavaScript. We analyze 49,296,059 calls to `eval` from 240,327 scripts extracted from 15,401 R packages. We find that `eval` is indeed in widespread use; R's `eval` is more pervasive and arguably dangerous than what was previously reported for JavaScript.

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

Additional Key Words and Phrases: eval, dynamic languages

ACM Reference Format:
Aviral Goel, Pierre Donat-Bouillud, Filip Křikava, Christoph M. Kirsch, and Jan Vitek. 2021. What We Eval in the Shadows: A Large-Scale Study of Eval in R Programs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 125 (October 2021), 23 pages. https://doi.org/10.1145/3485502

1 INTRODUCTION

Most dynamic languages provide their users with a facility to transform unstructured text into executable code and evaluate that code. We refer to this reflective facility as `eval` bowing to its origins in LISP, all the way back in 1956. `Eval` has been much maligned over the years. In computing lore, it is as close to a boogeyman as it gets. Yet, for McCarthy [1978], `eval` was simply the way to write down the definition of LISP; he was surprised that someone coded it up and offered it to end-users. Since then, reflective facilities have been used to parameterize programs over code patterns that can be provided after the program is written. The presence of such a feature in a language is a hallmark of dynamism; it is a form of delayed binding as the behavior of any particular call to `eval` will only be known when the program is run, and that particular call site is evaluated.

Authors' addresses: Aviral Goel, Northeastern University, USA; Pierre Donat-Bouillud, Czech Technical University in Prague, Czech Republic; Filip Křikava, Czech Technical University in Prague, Czech Republic; Christoph M. Kirsch, University of Salzburg, Austria and Czech Technical University in Prague, Czech Republic; Jan Vitek, Northeastern University, USA and Czech Technical University in Prague, Czech Republic.

2475-1421/2021/10-ART125
https://doi.org/10.1145/3485502

Trouble in Paradise. Reflective facilities hinder most attempts to reason about or apply meaning-preserving transformations to the code using them. In practice, `eval` causes static analysis techniques to lose so much precision as to become pointless. For compilers, anything but the most trivial, local optimizations are unsound after the use of `eval`. Furthermore, the addition of arbitrary code — code that could have been obtained from a network connection — as a program is running is a security vulnerability waiting to happen. To illustrate these challenges, consider the interaction of a static analysis tool with a dynamic language. A program analyzer computes an over-approximation of the set of possible behaviors exhibited by the program under study, a reflective facility must be represented by all behaviors that can be expressed in the target language. *i.e.*, any legal sequence of instructions can replace `eval`. As dynamic languages are permissive, the tool must assume that all functions in scope were redefined, *e.g.*, that `` `+` `` now opens a network connection. A single occurrence of `eval` causes the static analyzer to lose all information about the program state and meaning of identifiers. This loss of precision can sometimes be mitigated by analyzing the string argument [Christensen et al. 2003], but if the string comes from outside the program, not much can be done. A frustrated group of researchers argued giving up on soundness and, instead, under-approximating dynamic features [Livshits et al. 2015]. In their words, "a practical analysis, may pretend that `eval` does nothing unless it can precisely resolve its string argument at compile time." Alas, assuming that `eval` does not have side-effects or that side-effects will not affect results is unduly optimistic.

Is Past Prologue? Previous work investigated `eval` in web programming, specifically JavaScript web pages [Richards et al. 2010]. In 2011, 82% of the 10,000 most accessed sites used `eval` [Richards et al. 2011]. Yet, the strings passed to `eval`, and their behaviors, when executed, were far from random; it was shown that when one could observe several calls, the "shape" of future calls could be predicted with 97% accuracy [Meawad et al. 2012]. Overall, practical usage suggested that most reflective calls were relatively harmless. While this backed up the soundiness squad's approach, does it generalize to other application domains and to other languages?

The Here and Now. In this study, we investigate the usage of `eval` in programs written in the R language. R is a language designed by statisticians for applications in data science [R Core Team 2017]. What makes looking at R after JavaScript interesting is that, while both languages are dynamic, they are quite different. While one can program in an object-oriented style in R like in JavaScript, R is mostly a lazy functional language. JavaScript was designed to run untrusted code in a browser, while R is used for statistical computing on desktops. JavaScript is a general-purpose language used by a vast community of programmers, while R is used for scientific computing by data scientists and domain experts with, often, limited programming experience. One can distinguish between library implementers who have programming experience and a working knowledge of R, and end-users who are typically not expert programmers with a cursory knowledge of the language. Our goal is to highlight the differences in usage between JavaScript and R and explain them in terms of language features, application domain, and programmer experience. Hopefully, some of our observations generalize to other languages.

The What and How. One benefit of R is that every package in the CRAN repository comes with examples of typical usage. This gives us a codebase that we can analyze dynamically. To observe `eval`, we built a two-level monitoring infrastructure: we monitor programs by instrumentation and we also monitor the inner workings of the interpreter. Dynamic analysis is limited as it can only observe behaviors triggered by the particular inputs passed to a program. Luckily, R libraries come with many tests and use-cases. Our corpus is constructed to reflect the levels of sophistication of the R community. We distinguish between *CRAN packages* (15,401 curated packages that pass quality checks and have tests and sample data) and *Kaggle scripts* (7,931 end-user written programs) It is reasonable to expect `eval` usage to differ between these datasets, libraries are part of a lively and growing ecosystem, while end-user code is often thrown together, run once, and never revisited.

Why do we Eval? The results of our study suggest that `eval` is widely used for the implementation of the language, and in many libraries. End-user code makes less frequent and less sophisticated use of `eval`. In many ways, `eval` in R is as bad as it gets: it's varied, performs side-effects, and reaches many environments. By large, the motivations for `eval` relate to various forms of language extensions and meta-programming. `Eval` is used where other languages would provide macros. But, the expressive power of `eval` is higher as it can reach arbitrarily far back in the call stack.

Our data and code are open-source and publicly available at:

https://doi.org/10.5281/zenodo.5415230

2 BACKGROUND AND PREVIOUS WORK

This section provides a short introduction to R and the reflective features of the language; then looks at the semantics of `eval` in R and discusses design choices; lastly, this work is put in context.

2.1 R, Briefly

Morandat et al. [2012] give a programming language-centric overview of the R language. They characterized it as a lazy, vectorized, functional language with a rich complement of dynamic features expressive enough to layer several object systems on top of the core language. Most data types are sequences of primitive values. For instance, `c("Ha","bye")` evaluates to a vector of two strings. Constants such as `42` are vectors of length one. To enable equational reasoning, values accessible through multiple aliases are copied when written to. Furthermore, values can be tagged by attributes; these are key-value pairs. For instance, the attribute `dim-c(2,2)` can be attached to the value bound to `x` by `attr(x,"dim")←c(2,2)`. Adding this attribute turns `x` into a matrix. The `class` attribute gives a value a 'class' in the object-oriented sense. So, `class(x)← "human"` sets the class of `x` to `human`; classes are used for method dispatch. Every linguistic construct is desugared to a function call, even control flow statements, assignments, and bracketing. All functions can be shadowed and redefined, making R at the same time remarkably flexible and exceedingly challenging to compile as vividly detailed by Flückiger et al. [2019]. R uses a relaxed call-by-need convention for passing arguments to functions, studied in depth by Goel and Vitek [2019]. Each argument is a thunk composed of an expression, its environment, and a slot for the result; these are called *promises*. To get the value of an argument, the corresponding promise must be forced. Once forced, the promise's result is cached for future use.

2.2 On the Expressive Power of Eval

While a data-to-code facility is available in many languages, some design choices affect its expressive power. The key choices are the input format, the environment in which generated code evaluates, and the reflective operations available to that code. Table 1 summarizes a few designs.

The input to `eval` can be in any format convertible to code. JavaScript allows arbitrary strings; Julia and R are more restrictive as they require expressions (or abstract syntax trees). Finally, Java is the most restrictive as its classloader only accepts complete classes in bytecode format. These differences mostly affect users who seem more comfortable crafting strings.

The choice of the environment of `eval` is essential as it determines how much of a program `eval` can observe as well as the reach of potential side-effects performed by its execution. The most restrictive semantics is that of Java, where newly loaded code evaluates in the environment consisting of the classes visible from the current classloader. Julia limits `eval` to the symbols visible

Table 1. Design space of eval

Language	Input	Scope	Reflective operations
Julia [Bezanson et al. 2012]	expression	toplevel	data
Java [Liang and Bracha 1998]	bytecode	classloader	data
JavaScript [Richards et al. 2011]	text	current, toplevel	data
R [Ihaka and Gentleman 1996]	expression	programmatic	data, stack, environment

in the global environment, so does JavaScript's strict mode. Finally, R is the most flexible as any accessible environment can be selected and passed to `eval`. The choice of the environment is fully under the programmer's control.

The last degree of freedom is the expressive power of the code executed by `eval`. The main difference between languages lies in how much of the state of a program is accessible through reflective operations. Julia, Java and, JavaScript allow some form of introspection on the data that is visible in the environment in which `eval` executes. R is more flexible as it lets `eval` inspect the program's call stack as well as the code of any function. Thus, any environment and any binding in contains can be inspected and modified.

Given the above, the claim that R is amongst the languages with the most powerful `eval` seems plausible. The rationale for R's design seems to have been to expose as much of the language and its internals as possible in order to maximize expressivity. In R, `eval` is a key tool to extend the language and implement DSLs; it is also a replacement for macros. By contrast, the designers of Julia chose to limit `eval`. In Julia, only global variables can be side-effected, and environments cannot be readily manipulated. This is designed to shield optimized code from some of the most pernicious uses of the facility [Bezanson et al. 2018]. Furthermore, Julia provides a versioning mechanism to ensure that any method defined within an `eval` only becomes visible at well-defined points and thus that optimized code does not have to be invalidated [Belyakova et al. 2020].

2.3 Eval in R

The `eval` function in R takes three parameters: an expression to evaluate (`e`), an environment where to evaluate (`env`), and an enclosure (`encl`) that is used to look up objects not found in `env`.[1]

```
eval ← function(e, env = parent.frame(),
                encl = if(is.list(env)) parent.frame() else baseenv()) ...
```

The expression passed to `eval` can be thought of as an abstract syntax tree. Listings 1 and 2 show some of the ways of creating expressions: parsing from a string, manually, or via reflection. The call to `substitute(x)` extracts the unevaluated expression from the promise `x`. The call to `match.call()` returns an expression representing the current call which can then be further modified.

R is permissive in terms of what is considered an environment. Besides environments, it accepts lists, data frames (using element names or column names for variable resolution), or an integer `n` (in which case it would use the `n.` call frame). The default is the environment where the call to `eval` was made. Environments nest, each has a parent. A new environment created with `new.env` has the current environment as parent. Parent chains can be traversed with `parent.env`, until `emptyenv` is reached. The top-level environment is `.GlobalEnv`, its parents are the packages that have been

[1]R offers three other variants of `eval`: `evalq` automatically quotes passed expression—shorthand for `eval(quote(...))`, `eval.parent(e,n)` specifies the evaluating environment in terms of number of call frames (`n`) to go back—shorthand for `eval(e,parent.frame(n))`, and `local` evaluates `e` in a fresh environment—shorthand for `evalq(e,new.env())`.

```
parse(text="a+b")
quote(a+b)
call("+", quote(a), quote(b))

# reflect promise
f ← function(x) substitute(x)
f(a+b)
```

Listing 1. Examples of calls producing expression a+b

```
f ← function(x, y) {
  mc ← match.call() # reflect curr. call
  mc[[1]] ← as.name("g")
  mc[["x"]] ← 3
  mc
}
f(1,2) # returns an expr g(x=3,y=2)
```

Listing 2. Example of a call reflection

loaded. Environments are used as hash maps as they have reference semantics and a built-in string lookup. One can also directly read, modify or create new bindings, given any environment:

```
# reading
envir$v
get("v", envir=envir)
```

```
# writing
envir$v ← 2
assign("v", 2, envir=envir)
```

2.4 Previous Work

Richards et al. [2011] provided the first study at scale of the behavior of `eval` in JavaScript. A corpus of 10,000 popular websites was analyzed with an instrumented web browser to gather execution traces. Of those sites, 82% used `eval` for purposes such as on-demand code loading, deserialization of JSON data, or lightweight meta-programming to customize web pages. While many uses were legitimate, just as many were unnecessary and could be replaced with equivalent and safer code. The authors categorized inputs to `eval`. For inputs in which all named variables refer to the global scope, many patterns could be replaced by more disciplined code [Jensen et al. 2012; Meawad et al. 2012]. The work did not measure code coverage, so the numbers presented are a lower bound on possible behaviors. Furthermore, JavaScript usage in 2011 is likely different from today, *e.g.*, Node.js was not covered. More details about the dynamic analysis of JavaScript can be found in [Gong 2018].

Wang et al. [2015] analyzed the use of dynamic features in 18 Python programs to find if they affect file change-proneness. Files with dynamic features are significantly more likely to be the subject of changes than other files. Chen et al. looked at the correlation between code changes and dynamic features, including `eval`, in 17 Python programs [Chen et al. 2018]. They did not observe many uses of `eval`. Callaú et al. [2013] performed an empirical study of the usage of dynamic features in 1,000 Smalltalk projects. While `eval` itself is not present, Smalltalk has a rich reflective interface. The authors found that reflective methods are used in less than 2% of methods. The most common reflective method is `perform:`; it send a message that is specified by a string. These features are primarily used in the core libraries.

Bodden et al. [2011] looked at the usage of reflection in the Java DaCapo benchmark suite. They found that dynamic loading was triggered by the benchmark harness. The harness then executes methods via reflection. This caused static analysis tools to generate an incorrect call graph for the programs in DaCapo.

Arceri and Mastroeni [2021] studied `eval` in JavaScript from a software security point of view. The authors reported that 53% of the malware they studied used `eval` as a means to obfuscate attack code or mount attacks. They proposed an abstract interpretation-based approach to analyzing dynamic languages. One must construct a static approximation of the argument to `eval` and then analyze possible behaviors of the interpreter when evaluating the generated code.

Morandat et al. [2012] in their evaluation of the design of the R language, briefly looked at the use of `eval` in their corpus (1763 packages). They found 8500 `eval` call sites and recorded 2M `eval` calls. Because they included the base libraries, their results were dominated by a few functions with `match.arg`[2] being responsible for over half of the recorded `eval` calls. They discussed two use cases. The first one is the evaluation of an expression extracted from a promise using the `substitute` call in a new environment. The other is the invocation of a function with dynamically computed names and arguments.

In this work, we significantly expand that study. First, our corpus covers 15,401 packages (about the whole of CRAN at the time of writing) with 17,613 `eval` call sites, and 49.3M `eval` calls. Next, we track a lot more information such as the shapes of expressions passed to `eval`, their provenance, complexity, and side effects. This enables us to have an accurate and detailed picture of the use of `eval` in the R ecosystem. We use this to provide both quantitative and qualitative views on `eval`, including several use cases to demonstrate the popularity, diversity and powerfulness of `eval` in R.

3 METHODOLOGY

This section describes how the corpus of R program was selected and the analysis infrastructure.

3.1 Corpus

We distinguish three sources for calls to `eval`, those originating from *Base* libraries bundled with R, packages hosted on *CRAN*, and end-user scripts from *Kaggle*:

- *Base.* The 13 base libraries provide arithmetics, statistics, and operating system functionalities. Base has 200 functions with 342 calls to `eval`. These functions perform key tasks such as package loading, there is thus hardly any R program that does not invoke them. We consider base libraries as part of the language implementation and do not include them in our analysis.
- *CRAN.* Packages hosted on the Comprehensive R Archive Network (cran.r-project.org, aka CRAN) have three sources of runnable code: unit *tests* for individual functions, code *examples* embedded in the documentation, and use-cases called *vignettes.* All runnable snippets can be extracted into independent files. We download 15,401 packages and extract 240,327 scripts with 4,599,196 lines of code (1.5M in examples, 507.6K in vignettes and 2.6M in tests). CRAN packages contain 20 years of contribution from thousands of authors, often experienced R developers.
- *Kaggle.* Kaggle (kaggle.com) is an online platform that allows users to submit and compete to solve problems. We download 7,931 unique scripts and notebooks with their input files (2,339 duplicates were removed using SHA-1 hashes); 665K lines of R code making only 74 calls `eval`. The authors of these scripts have highly variable levels of expertise.

Discussion. When we started this project, our goal was to contrast the usage of `eval` by package developers and end-users, but the Kaggle dataset has so few calls to `eval` that there is little that can be said about those. The remainder of the paper focuses on CRAN with a few observations about the other sources when relevant.

3.2 Pipeline

We implemented an automated pipeline that acquires packages, extracts scripts, executes them, traces their behavior, and summarizes observations. Figure 1 shows the main steps along with their running time, data size, and the number of elements manipulated. The pipeline steps are:

[2] A base function that matches an argument against a set of candidate values, it uses `eval` to get the candidate values.

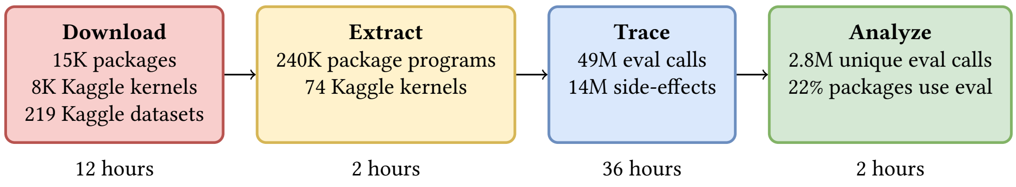


Fig. 1. Pipeline

(1) *Download.* Packages are downloaded from CRAN, for Kaggle a web crawler retrieves code, and a command-line tool gets data. Installation is complicated by native dependencies which sometimes have to be resolved manually.
(2) *Extract.* The *genthat* tool extracts all runnable code snippets and turns each of these into a self-standing program [Krikava and Vitek 2018], `knitr` extracts code from notebooks. Each extracted script is instrumented with calls to our dynamic analyzer in a way that avoids recording calls to `eval` originating from our execution harness.
(3) *Trace.* Scripts are run with a modified R interpreter to capture calls to `eval` in CRAN packages. Each script is run in its own process with the GNU R compiler turned off to avoid recording its execution.
(4) *Analyze.* Analysis outputs are merged, cleaned, and summarized. RMarkdown [Allaire et al. 2021] notebooks process the summarized data to generate all graphs (using `ggplot2` [Wickham 2016]) and numbers (exported as LaTeX macros) appearing in the paper.

The pipeline runs in parallel [Tange 2018] orchestrated by a Makefile. Servers have identical environments thanks to a docker image with all dependencies installed.

3.3 Dynamic Analysis

The tracer that performs dynamic analysis of R scripts is built on top of *R-dyntrace*, an extended R 4.0.2 virtual machine that exposes callbacks for various runtime events [Goel and Vitek 2019]. The tracer registers callbacks to all variants of `eval` and a few additional functions to locate the origin of `eval` arguments. It captures dynamic code loading, as well as variable definition and assignment, allowing us to record side effects that happen in environments while evaluating code in `eval`. A challenge was the lack of source code references for expressions that are not within a block surrounded by braces. This is unfortunately not easily fixed. We extend our dynamic analysis tool to attach synthetic source code references to all `eval` call sites, but the approach fails in some edge cases. For performance reasons, the tracer is an R package written in C++ (3.2K LOC) and R (1.3K LOC). While in theory, the implementation should be straightforward, it is not so in practice. Lazy evaluation requires delaying the processing of arguments until (and if) they are forced. Accounting for side-effects performed in an `eval` is complicated by the fact that R is implemented in a mixture of R and C, and that the language implementation can (and does) call `eval`. Lastly, large codebases exercise many edge cases of the highly underspecified R behavior.

Limitations. Even with the extension described above, there are 42.1K `eval` calls without source information (0.09% of all `eval` calls). This happens when `eval` is an argument to a higher-order function or when it is called from native code. Another limitation is that we do not record calls to the native `eval`. Neither of these limitations should invalidate our conclusions.

4 MEASURING EVAL

This section reports on the frequency of `eval` in CRAN. We use *site* to refer to an occurrence of a call site to the `eval` function in the source code and *call* to denote an observed invocation of the `eval` function.

There are 38,619 `eval` sites in 3,488 packages; 22.6% packages use `eval`. Over half of these packages have fewer than 3 sites, and with the exception VGAM which has 2,376 sites, all packages contain fewer than 800 sites. Fig 2 shows a histogram of sites per package. Sites appear in 15,532 functions; 2.5% of all functions use `eval`.

Our pipeline runs 240,327 scripts extracted from 15,401 packages. Any run that does not call `eval` is discarded, leaving us with 98,656 runs. In these runs, 49,296,059 `eval` calls were recorded originating from 17,613 sites. The runs exercised 52.9% of sites, a ratio similar to the package code coverage metric which is 51.7%. The reasons some sites are not exercised can be chalked down to incomplete tests and analysis failures (3% of the runs crashed or timed out). Fig. 3 plots exercised sites with respect to all sites for each program; as can be seen, coverage is unequal. Table 2 summarizes the frequency of calls, call counts on the left, and number of packages on the right. There are 1,540 packages with fewer than 100 calls, and 420 packages with more than 1,000 calls. ggplot2 calls `eval` 24,485,531 times and thus accounts for over half of our observations.

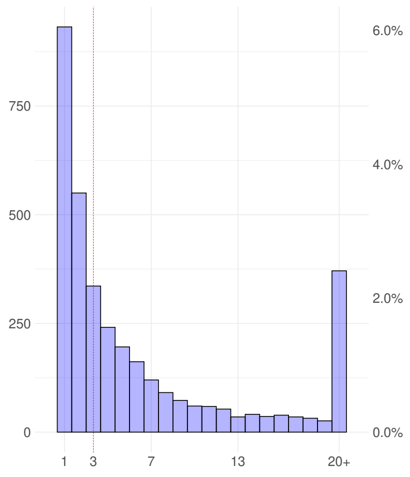


Fig. 2. `eval` call sites

Table 2. Call frequency

#calls	#pck	#calls	#pck
1 – 10	744	10K – 100K	117
11 – 100	796	100K – 1M	29
101 – 1K	522	1M – 10M	5
1K – 10K	268	10M – 100M	1

Table 3. Variants

	eval	evalq	eval .parent	local
Static sites	36,241	207	1,673	250
Exercised sites	16,412	8	1,064	129
Invocations	48.7M	1.1K	569.9K	39.5K

Table 3 summarizes the use of variants, the first row has sites (*static*), the second has sites encountered during analysis (*exercised*) and the last gives calls (*invocations*). Most sites and calls go to `eval` itself, `eval.parent` is rare, and both `evalq` and `local` are barely used at all.

Table 4. Normalized calls

#calls	#sites	#calls	#sites
0 – 50	16,274	501 – 1000	115
51 – 100	429	1001 – 1500	65
101 – 250	369	1501 – 2000	29
251 – 500	181	2001 – 3000	38

Table 4 shows the average count of calls from a given site and given run and how many sites fall in that range. For instance, 38 sites are called over 2,000 times per run. Larger call counts suggest the presence of loops or recursive functions, but given the data, this seems to be the exception;

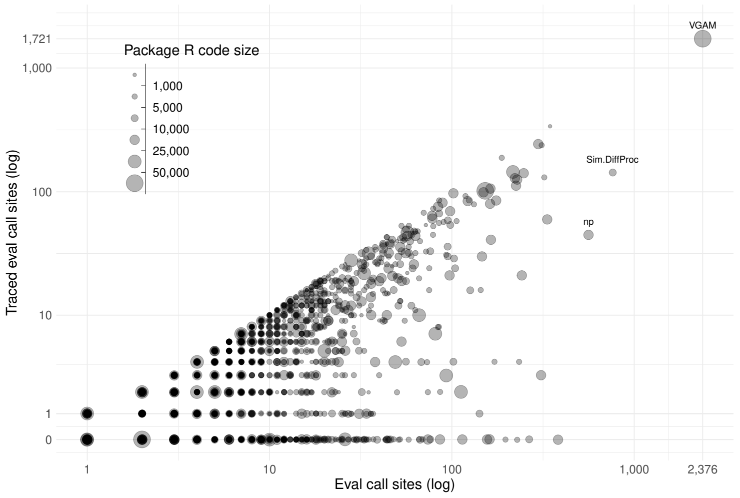


Fig. 3. `eval` call sites coverage of the 3,488 packages.

16,274 sites are called fewer than 50 times, most sites are invoked once and half as many are invoked twice.

`Eval` accepts any value as an argument, but if the value is not an expression, `eval` returns it unchanged. Expressions account for 90.2% of arguments. More specifically, 53.2% are symbols (variables such as `x`), 26.6% are language objects (expressions representing function calls), and 10.4% are expression objects (lists of expressions). Further inspection reveals that `_inherit` accounts for 93.1% of symbols and comes from one site in `ggplot2`.[3]

To estimate how much executable code is injected through `eval`, we measure the size of the arguments in terms of the number of nodes. For example, `expression(x+1)` has a size of 3. The large number of symbols skew the median to 1 but the average size is 4.5 nodes. The largest `eval` input we observed was an expression of 63,265 nodes, a significant chunk of code. Fig. 4 shows the distribution of sizes for arguments of fewer than 25 nodes. Sizes drop rapidly, there are few observations larger than 15 nodes. The long tail is omitted for legibility. As an alternative to counting nodes, we tried measuring string lengths of expressions after calling a function to convert values back to strings, but these measurements were dominated by massive data objects which could range in the MBs.

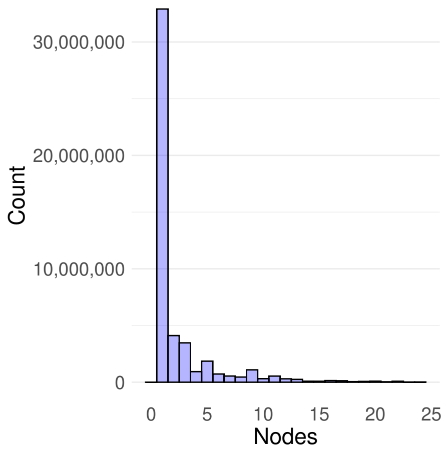
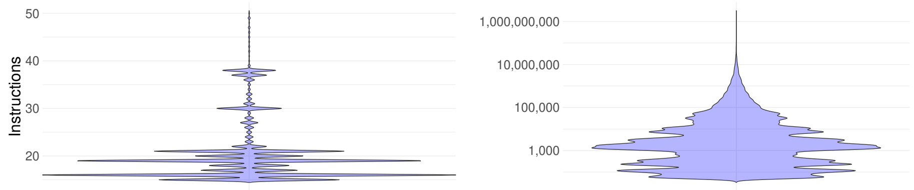


Fig. 4. Loaded code

To assess how much computational work is performed in evals, we count the instructions executed by the interpreter. Most calls do relatively little, with 89.8% of calls executing fewer than 50 instructions. The violin plot of Fig. 5(a) shows the distribution

[3]This field is used to model inheritance in `ggproto`, an object-oriented system used by `ggplot2`.

of short running evals; the data is dominated by evals that simply perform symbol lookup. Fig. 5(b) shows the distribution of work-intensive evals which go all the way to 2.1G instructions.



(a) Small (b) Large

Fig. 5. Instructions per call

Discussion. Eval is widely used in CRAN packages. Nearly every fourth package contains at least one eval call site. However, most packages use eval modestly. Packages that rely heavily on eval are mostly ones that provide statistical modeling and simulation functions. At runtime, over a third of our scripts triggered calls to eval. Note that all scripts call eval originating from the base package but these are disregarded here. Half of the calls come from a single site, for the rest, most packages have low call frequency. The data is consistent with the usage of eval for configuration and meta-programming, not in performance-sensitive contexts such as loops. While most arguments to eval are small, often just a variable name, and do little work, we have also observed large arguments and arguments that perform massive amounts of work.

5 A TAXONOMY OF EVAL

The previous section gave a quantitative view of eval usage; we now try to elucidate *what* it does.

5.1 The Expression in Eval

The expressions passed to eval vary widely. We categorize expressions using a minimization function which abstracts some of the incidental details of expressions and returns a "shape" that can be used to group similar evals. The function $min(e)$, for a given expression e, returns a normal form. Normal forms use V to stand in for values occurring in expression, X stands in for variables, and F stands for functions. The function performs constant folding of arithmetic and string expressions

Table 5. Minimized expressions

$min(e)$	#sites	%sites	#packages	#ops	%envir	example
X	4,701	27%	1,012	5	48%	y+1
F(F(X))	3,627	21%	831	81	79%	gbov(mean(x), a-1)
V	2,791	16%	786	2	71%	c(42,21,0)
F(X)	2,181	12%	768	16	68%	seq_len(iters)
$	1,708	10%	514	8	79%	DF**$**B
model.frame	1,261	7%	508	1.9K	46%	model.frame(formula = Z~U)
F()	1,184	7%	486	95	77%	rgamma(3, 2, n = 10L)
FUN	1,126	6%	129	10	96%	**function**(x)x+1
←	858	5%	274	60	78%	x[1, 2:3, 2:3] ← value
BLOCK	731	4%	150	5K	84%	{write(iris,tf);file.size(tf)}

over base operators; for instance, $min(1 + 1)$ simplifies to V, on the ground that addition of values likely returns a value. Value simplification takes $min(\mathsf{c}(1, 2, 3 + 2))$ to V, as a complex vector containing values is a value. Variable absorption has $min(\mathsf{x} + \mathsf{y})$ become X; this is motivated by the fact that addition is not an "interesting" operation and that `X` stands in for any number of variable lookups. Function absorption simplifies nested functions to keep an abstraction of the nesting $min(\mathsf{g}(\mathsf{f}(\mathsf{x}), \mathsf{h}(\mathsf{z}))) = \mathsf{F}(\mathsf{F}(\mathsf{X}))$. There are other simplifications; the full list can be found in our artifact. It is worth mentioning that these simplifications are heuristics; in R, `1+1` is not necessary `2`. The addition operator can be redefined, but it typically isn't, or at least not in a way that would invalidate arithmetic.

Table 5 gives the ten most frequent shapes with the number and ratio of sites that received arguments of that shape. Operations (ops) is the median count of instructions performed by the interpreter when evaluating an argument of the given shape. Envir is the ratio of sites that evaluate in a function environment. The last column has a sample expression that normalizes to the corresponding shape. We detail these shapes and discuss their implication for the behavior of `eval`.

$\boxed{min(e) = \mathsf{X}}$

Variable lookups are the most common shape, 50% are simple variable names, e.g. `x`. The shapes subsume `V` and built-in arithmetic operations, so a mixture of arithmetics and variables normalize to `X`. A lookup is one step of execution; if a promise is returned, an arbitrary number of additional steps may be needed. The median number of steps is 5, suggesting that most variable reads do little work. Variables are often evaluated in environments that have been constructed programmatically; 48% of expressions are evaluated in a function environment.

$\boxed{min(e) = \mathsf{F}(\mathsf{X})}$ $\boxed{min(e) = \mathsf{F}(\mathsf{F}(\mathsf{X}))}$

This shape corresponds to function calls whose arguments may include variable references and nested calls. Together they represent the most frequent shape and perform over 16 steps. They are frequently evaluated in function environments.

$\boxed{min(e) = \mathsf{V}}$

Values occur in 16% of sites, 74% of those are constants such as integers, the remainder evaluate to a value (1,436 of sites only ever see a value). `Eval` is needed when the expression denoting the values is constructed programmatically. Cases when a value is directly given to `eval`, and `eval` returns it unchanged, often occur when a computation has a default path and another, more interesting, path that requires evaluation. This shape usually runs in few interpreter steps.

$\boxed{min(e) = \$}$

This shape extends `X` to include lookup with the dollar operator, *e.g.*, `x$f`, and vector indexing, *e.g.*, `x[42]` and `x[[24]]`. This takes a few more steps of evaluation on average and is typically used in a function environment.

$\boxed{min(e) = \text{model.frame}}$

The `model.frame` function returns a data frame resulting from fitting the model described in a formula. This shape subsumes `F(F(X))`, `FUN`, and assignments. It is the single most popular function invoked from `eval`. Each call does a median of 2K instructions.

$\boxed{min(e) = \text{F()}}$

This shape represents function calls without variable lookups or assignment. A variable is allowed in the function position; looking up functions does not trigger computation unless the function's name is bound to a promise. This shape typically does not perform much work.

$\boxed{min(e) = \text{FUN}}$

This shape represents function definitions without any computation, *e.g.*, `function(x)x+1`. In addition to `FUN`, 10% sites have function definitions nested in other expressions.

$\boxed{min(e) = \leftarrow}$

This includes both direct assignment `<-` and parent environment assignment `<<-` as well as the `assign` function. It subsumes `$`. Assignments represent the most obvious source of side effects in `eval`. Most cases originate from trivial code generation where the assign expression is assembled using parse or substitute, often in a loop.

$\boxed{min(e) = \text{BLOCK}}$

These are multi-statement code blocks; as they can be large, we do not try to normalize their contents. They execute in a median 5K steps, usually in function environments. Essentially, the block denotes a fragment of a program to be evaluated in a particular way and a particular environment. The latter is what makes it different from a zero-argument closure. There are several use cases: unit testing frameworks (*e.g.*, `testthat`, `testit`), code benchmarking (*e.g.*, `rbenchmark`, `microbenchmark`), running code in parallel (*e.g.*, `foreach`, `doParallel`), or deferring code execution (*e.g.*, `withr`).

Discussion. The number of different shapes that any given `eval` site sees is an indication of the versatility of that site. 87.6% of sites only see a single expression shape. It would be encouraging if one could, given a small training set, predict the shape of evals to come. In our corpus, only a few sites are highly polymorphic with more than eight shapes. An example of those is the pipe operator of the `magrittr` package which is used to compose functions, *e.g.*, instead of `f(g(h(x),y),z)` one can write `h(x) %>% g(y)%>% f(z)`. As there are many different patterns of use, there are also many different shapes. In comparison with R, JavaScript's `eval` usage was more straightforward and more predictable, as reported by Meawad et al. [2012], with 98% of sites with only one shape. On the other hand, there are 7,217 sites that only receive one of the simple shapes (*i.e.*, `X`, `V`, `$` or $\leftarrow$), and 601 sites get multiple simple shapes. Among these sites, there are many cases where `eval` could be possibly replaced by simpler constructs such as `get` to lookup a name in an environment, `assign` to set a value to a name in an environment, or `do.call` to call a function reflectively.

5.2 The Environments of Eval

Contrary to JavaScript, in R, the environment of `eval` can be specified by its `env` argument. This gives users control over what is visible to the computation started by `eval` and the potential reach of its side-effects. We distinguish the following kinds of environments:

— *Function:* environments for the local variables of some function currently active on the call stack. Obtained by calling `parent.frame()` or `sys.frame()`.
— *Synthetic:* environments built from data structures such as lists, data.frames, or constructed explicitly with `new.env`, `list2env`, or `as.environment` or the empty environment.
— *Global:* environments in which scripts or interactive commands are evaluated.

— *Package:* environments of libraries.

Table 6 shows that most calls evaluate in the scope of a function; global is a distant second. This means that most reads and writes act on local variables. But of which function? Table 7 gives the offset of that function on the call stack: 0 is the function that called `eval`, 1 is that function's caller, and so on. In 81% of cases, `eval` uses its caller's environment – the variables of the function where the call to `eval` textually occurs are read and written. About 1.5% of sites evaluate their argument three frames up the call stack or above. Modular reasoning is thus impossible in R. Since the actions of `eval` happen at a distance, understanding any given code snippet requires knowing which functions may be called transitively from that snippet. In 91% of the sites, only one kind of environment is observed.

Table 6. Kinds per site

Kind	#sites	%sites
Function	12,842	72.9%
Synthetic	3,506	19.9%
Global	3,015	17.1%
Package	77	0.4%

Table 7. Function offsets

Offset	#sites	%sites
0	10.3K	81.5%
1	2K	15.5%
2	249	2%
≥ 3	193	1.5%

Table 8. Wrapper envs.

Parent	#sites	%sites
Function	2,497	71.2%
Package	1,362	38.8%
Global	378	10.8%
Empty	100	2.9%

Table 9. Multiplicities

#kinds	#sites	%sites
1	16K	91%
2	1.3K	7.6%
3	229	1.3%
4	11	0.06%

Each synthetic environment has a parent that is specified when calling `new.env`. When `envir` is a list or a data frame, `eval` uses its third argument (`enclos`). The parent is used to search for variables not found in its child (side-effects stay in the child). Table 8 shows parent kinds for synthetic environments. Most of them are functions.

Global evals split between intentional and accidental one. Direct references to the top-level, using `globalenv()` or `.GlobalEnv`, are rare; they occur in only 25 sites. Thus we suspect most uses of global are accidental. They arise from the fact that our corpus consists of many code snippets that are run at the top-level; global is thus the caller of `eval`. This is noteworthy because writes to global variables are visible to all functions and are not reclaimed by the garbage collector. So accidental uses may pollute that namespace. Table 9 gives the number of kinds seen at a given call site.

Discussion. The data presented here is not surprising. The main use of `eval` is to provide a customizable extension mechanism for the behavior of functions. They are a way to parameterize the function with any behavior that can be expressed in R. The behavior of `eval` within its enclosing function is to read and write variables, mostly read, and very rarely, delete variables. There are also some cases where new variables are injected in a function. Usually, this is in the direct caller, but it can sometimes take effect several frames up the call stack. There is something brittle about code relying on the position of a caller: a lot of refactoring may break code that does that.

One bit of information that we lack is how the environment was obtained. Our expectation is that the expression was extracted from a promise using `substitute`, perhaps modified, and then evaluated with the environment coming from the same promise. We believe the case where `eval` is provided with the results of programmatically selecting some call stack to be less frequent.

Synthetic environments are relatively frequent. Common uses-cases include the evaluation of an expression in a sandbox or using a data structure as environment. For example, one could evaluate a method of an object and use an environment to hold the object's fields.

We have already explained the popularity of global environments. As for package environments, they are used in only 77 sites. This is probably for the best as mutating the bindings of a loaded package is frowned upon, and R tries to discourage it.

5.3 The Origins of Eval

Where does the expression passed to `eval` come from? There are various means of creating an expression; often these means correlate with a particular use case. Our analysis records the values returned by some functions of interest as well as their arguments. We build a direct acyclic graph that tracks the origin of the values that are given to `eval`. Fig 6 shows an example of the origin graph for the expression `e`. The origin of `e` includes both `parse` and `quote`, the starting nodes. We pick one origin by traversing the graph from the terminal leaf, *i.e.*, from the `eval` node, following the left-hand side member of assignments until reaching a starting node. It represents the origin that is further modified to yield the expression passed to `eval`. In this example, that origin is `parse`.

```
> e ←parse(text = "a;b")
> e[[2]] ←quote(c) # e is a;c
> eval(e)
```

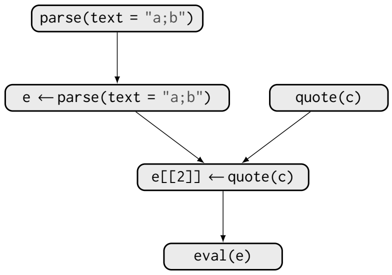


Fig. 6. Example of an origin graph

To have a high-level view of provenance, we classify functions into the following five categories:

— *Reflection:* use of `match.call` to reflectively capture the expression that invoked this function. Function arguments are promises, and `substitute` is used to retrieve their source.
— *String:* created from strings by invoking `parse`, `str2lang`, or `str2expression`.
— *Constructed:* invoked `quote` or `expression`, or by building with `call` or `as.call`.
— *Environment:* created with `as.name` or `as.symbol`, typically to read a non-local environment.
— *External:* call to a C function.

Table 10 summarizes origins. It shows frequencies by function and by category. For technical reasons, we misclassify a small number of sites. Manual inspection of numerous examples suggests that errors are rare. Some sites are counted multiple times as they are invoked with different origins.

Table 10. Origins

Provenance	**#sites**	**%sites**
parse	4,006	22.9%
substitute	3,916	22.4%
match.call	2,058	11.8%
expression	597	3.4%
quote	557	3.2%
as.name	482	2.8%
call	475	2.7%
as.call	450	2.6%
.External	251	1.4%
formals	112	0.6%

Origin	**#sites**	**%sites**
Reflection	7,857	44.9%
String	4,107	23.5%
Constructed	3,086	17.7%
Environment	1,200	6.9%
External	278	1.6%

Strings correlate with dynamic code loading as done by `source` and `sys.source`. Few calls (11 in total) consume the result of calling `parse` on a file. Most build strings programmatically.

Discussion. The main use cases for `eval` involve using `match.call`, or `substitute` to access and transform arguments of a function. Constructing expressions from strings is also a common use case. The constructed category shows that most evals come from existing code that was modified by the programmer before invoking `eval`. Both constructed and reflection categories roughly correspond to meta-programming. Some uses could be replaced by macros if the R designers could be convinced to overcome their distaste for those.

5.4 The Effects of `Eval`

`Eval` may perform side-effects. We care about effects that are visible after a call finishes, *i.e.*, variable definitions, updates, and removals. Knowing the environments in which these side-effects happen can help determine the impact of `eval` on our ability to reason about the code. Our analysis records information about every environment. From the recorded data, we discard side-effects coming from unit testing frameworks as they use `eval` to run all their tests and thus dominate the side-effect data.[4]

We record 14.4M side-effects from 2.9M calls and 3,091 sites. The challenge is to remove accidental side-effects caused by the R virtual machine implementation and not user code. For example, the `.Random.seed` variable is saved and restored from and to the global environment every time a statistical routine is called. Filtering out accidental side-effects leaves us with 7.9M writes from 2.5M calls (in 2,418 sites and 1,524 functions). In this set, 43 functions are responsible for 90% of side effects. Half of those come from just three functions: `plyr::allocate_column` (allocates space for a new data frame column), `withr::execute_handlers` (executes deferred expressions), and `foreach::doSEQ` (executes an expression on each element in a collection, possibly in parallel). Most of the sites (58.9%) update the environment specified by the `env` parameter.

Table 11. Target environments for side-effects

Environments	**#sites**	**%sites**	**#funs.**	**%funs.**
Local	945	39%	269	25%
Synthetic	394	16%	248	23%
Object	240	10%	136	12%
Function	153	6%	62	6%
Global	106	4%	47	4%
Package	5	0.2%	0	0%
multiple	575	24%	332	30%

Table 11 shows environment kinds where side-effects happen. Function environments distinguish between *local*, the caller environment (offset 0), and *function* (offset > 0). *Synthetic* represents constructed environments. *Object* is used to denote the environments attached to objects and classes in the S4 and R6 object systems. *Multiple* denotes cases where side-effects to more than one kind originate from one site. The table gives the number of call sites of `eval` (and the ratio) performing side-effects in a particular environment kind. The table also gives the number of functions in which these sites occur (and ratio).

[4]The corpus has `RUnit`, `testthat`, `tinytest` and `unitizer` testing frameworks.

Most sites, 76.2%, do all their side-effects consistently in one kind of environment. The same happens at the function level. Almost half of the sites (over a third of the functions) do side effects in either *Local* or *Object* environments.

Table 12 shows recorded effects. There are over 5M updates. In terms of calls, we primarily see assignments, followed by definitions. This is expected. A subsequent `eval` call will turn a definition into an update. The most dangerous side effect is variable removal, as it means that after a call to `eval` some binding in some environments will disappear. While rare, this happens, but the vast majority comes from a single site (`withr::execute_handlers`). This makes sense as it is used to defer evaluation of an expression to after the function exit and thus used for clean up. It is used almost exclusively by the `tidyselect` package for removing the reference to the current quosure environment while interpreting a data frame column selectors.[5]

Looking closely at the value types in variable updates, we observe that the majority of `eval` sites involve basic R vectors (53.8%) and lists (31%). From the perspective of a compiler, we would like to know how many sites change function bindings. In our corpus, this happens in only 62 sites; 75.8% of them do that in the local environment. We manually inspected a few of these sites; except for manual injection of parameters into a `model.frame` execution environment, we did not find a common use case.

Table 12. Types of `eval` side-effects

Side effect	**#events**	**%events**	**#calls**	**%calls**	**#sites**	**%sites**
update	5,155,985	66%	2,674,085	61%	1,372	43%
definition	2,458,727	31%	1,455,335	33%	1,700	54%
removal	243,260	3%	237,711	5%	104	3%

Out of the minimized expressions, it is `BLOCK`, $\leftarrow$ and `F(F(X))` that contribute to the vast majority of side effects. Concretely, 92% of `BLOCK`, 87% of $\leftarrow$, and 15% of `F(F(X))` expressions do side effects. In the case of `BLOCK`, most of them happen in the environment passed to the `eval` call (default is the parent environment). For $\leftarrow$ it is 15%. This form is dominated by the `plyr::allocate_column` function that side-effects in the environment containing the data frame to which it allocates a new column. Without this call, $\leftarrow$ is similarly predictable with 96% of side effects happening in the passed environment. For `F(F(X))`, it is less clear as 58% of side effects happen in a different environment to the one that was passed to `eval`.

Discussion. In JavaScript, assignments in `eval` can happen in either local scope or less often, when called through an alias, in the global scope. In R, given the support for first-class environments, it can happen anywhere, making `eval` more dangerous than it already is. However, the data suggest that first, side-effects from `eval` are not as widespread as in the case of JavaScript,[6] and that over half of them happen in a predictable environment. This gives a ray of hope for a hypothetical R compiler. Even though `eval` can do anything anywhere, the data suggests most effects are sane.

6 USAGE OF EVAL

The R language was intended to be extensible. The combination of lazy evaluation, `substitute`, and `eval` are some of the tools given to developers to this end. To understand *why* developers use eval, we manually inspected the top 117 sites, which contribute to 90% of the calls in the corpus. We

[5] *cf.* https://tidyselect.r-lib.org/

[6] [Richards et al. 2011] shows that in the *Interactive* scenario, `eval` in Javascript performs, store events can reach up to 40% of the events, and 7% to 8% of the `eval` do side effects in the global scope.

present examples from nine classes from this batch. This classification is not exhaustive since there are 38,619 `eval` call sites in our corpus, and many uses of eval do not fit in a specific category. We further categorize the nine classes into two groups: those that improve the interface for users and those that simplify the implementation for developers.

6.1 Better Interface

This group contains `eval` uses that enable the development of interfaces for quick interactive data exploration.

Non-standard Scoping. `Eval` allows functions to resolve symbols in different environments than the current one. For example, the `data.table` package redefines the subsetting operator, `[`[7], to enable compact queries on data frames. The example below shows the use of this operator on the `flights` data frame to compute the average departure delays of Delta flights in 2014 by the origin airport. The symbols `carrier`, `year`, `dep_delay`, and `origin` are not looked up in the current environment but in the `flights` data frame.

```
> flights[carrier=="DL" & year == 2014, mean(dep_delay), by=origin]
#    origin      V1
# 1:    LGA 11.35083
# 2:    EWR 14.96171
```

`data.table::[`

At its core, the `[` operator uses `substitute` to capture its operand expressions, transforms them, and evaluates them using `eval` in the given data frame. We omit the implementation as it spans over 1700 lines at the time of this writing.[8]

Non-standard scoping is an important use case of `eval` in R. It enables the design of functions that require less typing and syntactic noise at the cost of some ambiguity. Since R is used primarily interactively, this trade-off is justifiable.

Domain-Specific Language. A domain-specific language leverages R's grammar but redefines its semantics in a suitable way. The example below illustrates string interpolation. The `glue` function from the `glue` package extracts snippets of code enclosed between braces, evaluates them using `eval`, and splices their results to construct the result.

```
> greeting ← "Hello"
> glue("{greeting} World!")
# Hello World!
```

`glue::glue`

The `glue` package contains a variant of `glue` function that leverages non-standard scoping for interpolation in a custom environment. For example, the following snippet pipes the subsetted data frame into the `glue_data` function used to resolve the symbols for string interpolation.

```
> flights[carrier=="DL" & year == 2014, mean(dep_delay), by=origin] |>
   glue_data("{origin} has average delay of {round(V1)} min")
# LGA has average delay of 11 min
# EWR has average delay of 15 min
```

`glue::glue_data`

[7]Defined as a triplet `[i, j, k]`, where expression `j` is used to calculate a subset of rows from `i` grouped by `k`.
[8]Its implementation can be found at: https://github.com/Rdatatable/data.table/blob/master/R/data.table.R

Behavioral Extension. Sometimes it is convenient to allow users to extend the behavior of a library function. `Eval` allows clients to provide a simple text string that is executed in the context of the function. The example here allows users to provide formulas as strings, such as `"exp(-x^2/2)"`, the contract being that the text can refer to some variable `x` the called function will set up.[9]

```
function(n,formula) {
  p ← function(x) eval(parse(text=formula))
  ...
}
```

`AdapSamp::rARS`

Extracting Varargs. `Eval` is used to obtain the source code of the arguments passed to a vararg parameter (`...`). In the example below, `substitute(alist(...))` yields an expression representing a call to `alist` with the `...` expanded to the arguments supplied to the function. Passing this to `eval` returns a list of the unevaluated argument expressions. This pattern is needed to avoid premature evaluation of vararg arguments.

```
function(...) {
  for (e in eval(substitute(alist(...)))) {
```

`statnet.common::NVL`

Controlled Evaluation. `Eval` can be used to control when and if expressions are evaluated. For example, `assertthat` allows users to make assertions and, if they fail, produce legible error messages. The `assertthat::see_if` function accepts a vararg with a list of expressions. It extracts these expressions using the extracting varargs pattern and evaluates them sequentially in the caller's environment using `eval`. If an expression does not hold, an error is generated.

```
function(..., env=parent.frame(), msg=NULL) {
  asserts ← eval(substitute(alist(...)))
  for (assertion in asserts) {
    res ← tryCatch({
      eval(assertion, env)
    }, assertError=function(e) { structure(FALSE, msg=e$message) })
  ...
}
```

`assertthat::see_if`

Shadowing. First-class environments and `eval` enable R users to implement variations of core language functionality. For example, the `igraph` package provides the `do_call` function shown below. `do_call` calls a function (`f`) in the specified environment (`env`) with the arguments supplied to `...`, and `args`. This is a variant of the built-in `do.call` function with two differences: it resolves the function name differently, and it also allows arguments to be passed as varargs.

```
function(f,...,args=list(),env=parent.frame())
  eval(make_call(substitute(f),...,args), env)
```

`igraph::do_call`

[9]The function is used to generate a sequence of random numbers using the adaptive rejection sampling algorithm and the string formula is used to describe the target density.

6.2 Implementation Simplification

This group contains eval uses that simplify implementation for package developers.

Code Generation. Eval can be used to generate code that would either be tedious to write by hand or that is only known after deployment. The following code snippet from the Rcpp package uses eval to automatically generate R bindings to C++ methods.

```
function(methods, env) {
  for(what in cppMethods)
    methods[[what]] ←
      eval(substitute(function(...)CppObject$what(...)), env)
  ...
}
```

Rcpp::.makeCppMethods

Boilerplate Removal. A variant of code generation is to let users write compact code that is expended via eval and match.call. The example uses match.call to access the expression with which the function is called. The following three lines construct a call to stats::model.frame with argument text of parameters 'formula', 'data', and 'weights'. Then, the call expression is evaluated in the parent environment. The use of match.call avoids repeated use of substitute for extracting the argument expressions. It is extensible as new parameter names can be easily added. This example illustrates one of the most recurring uses of eval by packages that do statistical modeling. It is a typical example of the model.frame shape in Table 5.

```
function(formula, data, weights, ...) {
  Call ←match.call()
  indx ←match(c("formula", "data", "weights"), names(Call), nomatch=0)
  temp ←Call[c(1, indx)]
  temp[[1L]] ←quote(stats::model.frame)
  mf ←eval.parent(temp)
  ...
}
```

survival::survfit.formula

Wrapping functions in R is somewhat harder than in other languages because of default arguments, varargs, and missing arguments. The combination of match.call and eval can be used to forward the current call's arguments to another function without listing all the argument names. This pattern eases the maintenance of wrappers. For example, the base::write.csv function passes most of its arguments unchanged to the more general base::write.table function.

```
function (...) {
  Call ←match.call(expand.dots = TRUE)
  Call$sep ←","
  Call$dec ←"."
  Call$qmethod ←"double"
  Call[[1L]] ←as.name("write.table")
  eval.parent(Call)
}
```

base::write.csv

Obfuscation. A few packages use `eval` to bypass the restrictions imposed by the `R CMD CHECK` tool. This tool enforces some well-formedness rules on a submitted package before accepting it for inclusion in CRAN. Some static checks ensure that the package code does not use certain restricted functions. Package authors use `eval` to obfuscate their code to get around this limitation. The example below mutates the variable `s` in the package environment. However, it is locked by default, and unlocking it requires `unlockBinding` which is a restricted function. To work around this, the call to `unlockBinding` is done through `eval`.

```
function() {
  env ← parent.env(environment())
  eval(parse(text=paste0('unlockBinding("s", env)')))
  assign("s", s, envir=env)
  lockBinding("s", env)
  ...
}
```

`aibd::scalaEnsure`

7 DISCUSSION

In R, `eval` is used chiefly for meta-programming and accessing remote environments. Unlike JavaScript, which according to Richards et al. [2011] had a small set of well-defined patterns that could be rewritten without `eval`, our results suggest that `eval` is an integral part of programming in R. However, there are cases in which `eval` is not needed, and other, more stylized functions can be used. There are also alternatives to `eval` that provide a more principled way for meta-programming and dealing with lazy evaluation. In this section, we briefly discuss these issues.

Unnecessary use of eval. The cases where `eval` appears to be an overkill include simple expression shapes such as `X`, `V`, `$`, `←`, and `FUN`. Variable lookup in a remote environment can be performed by the built-in `get` function. A similar case can be made for assignment and the `assign` function. There is a dedicated `do.call` function to perform function calls. However, unnecessary uses of `eval` are not easy to find and often can only be uncovered by manual analysis. Dynamic analysis is limited due to coverage issues, and static analysis of R remains an open problem.

```
function (d="norm", dp, ...) {
  q.f ← eval(parse(text=paste0("q", d)))
  z ← NULL
  eval(parse(text=paste0("z←q.f(", dp, ", ...)")))
  ...
}
```

Listing 3. Unnecessary use of eval (`PerformanceAnalytics::chart.QQPlot`)

In the example in Listing 3, the function uses `eval` to lookup a target function name from a constructed string. Then it constructs a call with the arguments that are passed to the current function. `Eval` is overkill in this case: a semantically equivalent function body can use `get` to perform the lookup and then call the resulting function directly.

```
q.f ← get(paste0("q", d))
z ← q.f(dp, ...)
```

Eval alternatives. Alternative implementations of `eval` are also available from R packages. The `lazyeval` package provides `lazy` and `lazy_eval` as safer variants for the built-in `substitute` and `eval` functions, respectively. These variants differ from their counterparts in that they capture both the argument expressions and their environments, and they follow promises across function invocations. This helps to avoid scoping bugs, as shown in the example below.

```
filter ← function(df, cond) {
  conf ← substitute(cond)
  r ← eval(cond, df, parent.frame())
  df[r, ]
}

f ← function(...) {
  x ← 1
  filter(flights, ...)
}

x ← 2020
f(year > x) # incorrectly uses 1 for x
```

Use of R `eval`

```
filter ← function(df, cond) {
  cond ← lazy(cond)
  r ← lazy_eval(cond, df)
  df[r, ]
}

f ← function(...) {
  x ← 1
  filter(flights, ...)
}

x ← 2020
f(year > x) # correctly uses 2020 for x
```

Use of `lazyeval`

The `filter` function selects rows from `df` that match a given condition passed to `cond`. The implementation using the built-in `eval` suffers from a scoping bug since it always evaluates `cond` in the caller's environment irrespective of where it is passed from.[10] The `lazyeval` version avoids this problem since it correctly captures the environment associated with `cond`.

The tidyverse libraries[11] include their own implementation of `eval` called `eval_tidy` [Wickham et al. 2019] from the `rlang` package. This function which departs from the built-in `eval` in two ways. First, it supports the evaluation of quosures, custom objects used for metaprogramming that bundle expressions with an environment. Second, the `eval_tidy` function accepts a data mask argument, a set of bindings such as a data frame or list, which takes precedence over the environment. Consequently, effects such as assignments happen in the data mask, and expressions such as `return()` do not work since the data mask does not correspond to a frame on the call stack.

8 CONCLUSION

The `eval` function is used widely, and in varied ways in R. The function is an essential tool for language implementers. Our analysis observed that the base libraries heavily depend on `eval`, and any R program will end up calling it through the core of the language. Independently developed libraries often use `eval` in subtle ways, mostly to perform macro-programming tasks. Our review of code written by less experienced users suggests that `eval` is exceedingly rarely used. So, it is fair to conclude that most R programmers never have to write a call to `eval`, but the code they write would not run without it.

`Eval` is thus a challenge for automated program understanding. Program analysis and transformation tools or compilers must assume the worst when faced with code that calls this function. We have observed all sorts of side-effects with various degrees of visibility.

While our results are not encouraging in general, we have observed many cases where `eval` is used in disciplined and predictable ways. While in the general case `eval` is hell, there are many

[10]This is a simplified version of the built-in `subset` function which suffers from this exact problem.
[11]An opinionated collection of R packages designed for data science. It consists of some of the most popular packages.

cases where it is just a function. It does not appear that a general-purpose replacement for `eval` is possible; in future work, we hope to focus on special cases and propose tools that target particular subsets of call sites with some common properties.

ACKNOWLEDGMENTS

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

REFERENCES

JJ Allaire et al. 2021. *rmarkdown: Dynamic Documents for R.* https://github.com/rstudio/rmarkdown R package version 2.9.

Vincenzo Arceri and Isabella Mastroeni. 2021. Analyzing Dynamic Code: A Sound Abstract Interpreter for *Evil* Eval. *ACM Trans. Priv. Secur.* 24, 2 (2021). https://doi.org/10.1145/3426470

Julia Belyakova, Benjamin Chung, Jack Gelinas, Jameson Nash, Ross Tate, and Jan Vitek. 2020. World Age in Julia: Optimizing Method Dispatch in the Presence of Eval. *Proc. ACM Program. Lang.* 4, OOPSLA (2020). https://doi.org/10.1145/3428275

Jeff Bezanson, Jiahao Chen, Ben Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. 2018. Julia: Dynamism and Performance Reconciled by Design. *Proc. ACM Program. Lang.* 2, OOPSLA (2018). https://doi.org/10.1145/3276490

Jeff Bezanson, Stefan Karpinski, Viral Shah, and Alan Edelman. 2012. Julia: A Fast Dynamic Language for Technical Computing. *CoRR* abs/1209.5145 (2012).

Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/1985793.1985827

Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. 2013. How (and why) developers use the dynamic features of programming languages: the case of Smalltalk. *Empir. Softw. Eng.* 18, 6 (2013). https://doi.org/10.1007/s10664-012-9203-2

Zhifei Chen, Wanwangying Ma, Wei Lin, Lin Chen, Yanhui Li, and Baowen Xu. 2018. A study on the changes of dynamic feature code when fixing bugs: towards the benefits and costs of Python dynamic features. *Sci. China Inf. Sci.* 61, 1 (2018). https://doi.org/10.1007/s11432-017-9153-3

Aske Simon Christensen, Anders Møller, and Michael Schwartzbach. 2003. Precise Analysis of String Expressions. In *Static Analysis Symposium (SAS)*. https://doi.org/10.1007/3-540-44898-5_1

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

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

Liang Gong. 2018. *Dynamic Analysis for JavaScript Code.* Ph.D. Dissertation. University of California, Berkeley. http://www.escholarship.org/uc/item/7n30n4kd

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/

Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. 2012. Remedying the Eval That Men Do. In *International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/2338965.2336758

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

Sheng Liang and Gilad Bracha. 1998. Dynamic Class Loading in the Java Virtual Machine. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/286936.286945

Benjamin Livshits et al. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (2015). https://doi.org/10.1145/2644805

John McCarthy. 1978. History of LISP. In *History of programming languages (HOPL)*. https://doi.org/10.1145/960118.808387

Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. 2012. Eval begone!: semi-automated removal of eval from JavaScript programs. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/2384616.2384660

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

R Core Team. 2017. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. https://www.R-project.org/

Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The Eval that Men Do: A Large-scale Study of the Use of Eval in JavaScript Applications. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/978-3-642-22655-7_4

Gregor Richards, Sylvain Lesbrene, Brian Burg, and Jan Vitek. 2010. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Programming Language Design and Implementation Conference (PLDI)*. https://doi.org/10.1145/1809028.1806598

Ole Tange. 2018. *GNU Parallel*. Ole Tange. https://doi.org/10.5281/zenodo.1146014

Beibei Wang, Lin Chen, Wanwangying Ma, Zhifei Chen, and Baowen Xu. 2015. An empirical study on the impact of Python dynamic features on change-proneness. In *International Conference on Software Engineering and Knowledge Engineering*. https://doi.org/10.18293/SEKE2015-097

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

Hadley Wickham et al. 2019. Welcome to the tidyverse. *Journal of Open Source Software* 4, 43 (2019). https://doi.org/10.21105/joss.01686