First-Class Environments in R

Aviral Goel
Northeastern University
USA

Jan Vitek
Northeastern University
USA
Czech Technical University in Prague
Czech Republic

Abstract

The R programming language is widely used for statistical computing. To enable interactive data exploration and rapid prototyping, R encourages a dynamic programming style. This programming style is supported by features such as first-class environments. Amongst widely used languages, R has the richest interface for programmatically manipulating environments. With the flexibility afforded by reflective operations on first-class environments, come significant challenges for reasoning and optimizing user-defined code. This paper documents the reflective interface used to operate over first-class environment. We explain the rationale behind its design and conduct a large-scale study of how the interface is used in popular libraries.

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

Keywords: first-class environments, dynamic languages

ACM Reference Format:
Aviral Goel and Jan Vitek. 2021. First-Class Environments in R. In *Proceedings of the 17th ACM SIGPLAN International Symposium on Dynamic Languages (DLS '21), October 19, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3486602.3486768

1 Introduction

The ability to name values is a building block of linguistic abstractions. Local variables, global variables, function parameters, all boil down to a mapping from names to values, commonly referred to as an *environment*. The semantics of environments have a profound impact on how languages can be implemented. At a first approximation, the more restricted the semantics, the easier it is to implement the language efficiently. Early languages did not support recursion; thus their compilers could generate code where each variable had its unique, pre-determined location in the computer's memory. When a variable's value was determined to be constant, it did not even need a memory location; the variable was effectively an alias for that value. As languages became more expressive, implementations had to store variables in data structures such as the stack for traditional imperative languages or heap-allocated records for functional languages. When the compiler could assume all variable accesses were known at compile-time, the variables could be represented by offsets from a known location. On the other hand, for languages that allowed symbolic lookups, the mapping between names and locations had to be retained. Each of these choices comes with performance implications.

The R programming language was created in 1993 [Ihaka and Gentleman 1996] as a direct descendent to S, whose origin dates to 1976 [Becker et al. 1988]. Both languages found inspiration in earlier work on Lisp [McCarthy 1959], CLOS [Steele 1982], and Scheme [Adams et al. 1998]. However, they chose to depart from those previous languages in small and large ways. R evolved to become a functional language, without type annotations, with delayed evaluation of function arguments, mutable state, and various mechanisms for supporting object-oriented programming.

At heart, R is a simple language. Its expressivity stems from the combination of delayed evaluation and the language's rich reflective interface that allows to extend the core semantics in various ways. One of the keys design choices was to make environments first-class and allow full programmatic access over environments.

In R, an environment is an unordered mutable map from symbols to values. Environments are omnipresent – they represent name spaces, or scopes, for variables within a function, but also the name spaces that are constructed when a package is loaded. As they are the only mutable data structure in the language, they are also used as hashmaps and objects. They are created implicitly when a function is called and explicitly with `new.env`. Operations are provided to read variables and update or delete them. It is even possible to change the association between a closure and its environment. R's interface allows, for example, to acquires the environment of the caller of the currently executing function, check if it contains a variable `x`, and rename it to `y`. Needless to say that this flexibility causes headaches to implementers, Flückiger et al. [2019] give an account of these challenges.

DLS '21, October 19, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9105-4/21/10.
https://doi.org/10.1145/3486602.3486768

This paper documents the interface that R exposes to environments. This interface evolved through the years. It is rich and not always consistent, and certainly not minimal. Through a dynamic analysis conducted on a corpus of popular R libraries and their clients, we report on the practical usage of environments. This allows us to explain the need for this interface and could, possibly, be a step towards a re-design or a warning for compiler writers.

2 Related Work

Imperative languages like C aim to have efficient implementations; variables are offsets from a stack pointer, and symbolic references are only allowed when debugging non-optimized code. Statically typed functional languages are more expressive with first-class functions. Their implementations allocate environments on the heap, but retain the variable-as-offset technique. Dynamic languages retain more information at run-time to support dynamic code generation through the `eval` function. Languages like Scheme or Python retain the mapping between symbols and locations. When languages allow to programmatically add and remove variables, achieving performance is even harder. Consider the design of Scheme and Python. While the Scheme Standard specifies that `eval` takes an environment-specifier which needs not be a first-class environment [Adams et al. 1998]. MIT/GNU Scheme supports first-class environments and, as a consequence, its `eval` takes an explicit environment. That implementation provides functions to create new environments, read and write bindings, examine parent environments, and obtain the current environment as a reified value. Unlike R, it does not provide access to caller environments. Python provides the `locals` function that returns the local bindings as a dictionary. Updates to this dictionary are not reflected in the function's scope. Caller's bindings can be accessed from their frame obtained by calling `inspect.stack`. Calling `locals` outside of a function returns a namespace which can be updated. The `globals` function returns a dictionary of the global namespace whose updates are also reflected in the namespace. Siskind and Pearlmutter [2007] proposed the `map-closure` construct to construct a new closure with a modified environment. By design, this hides the details of environments, and unlike R this does not allow addition and removal of variables.

Morandat et al. [2012] discussed the design of R, including its scoping and evaluation mechanism. While the paper presents some data on environments, it does not discuss explicit environment creation using `new.env`. In comparison to their work, we focuses on environments and provides a more detailed qualitative and quantitative account. Our study shows a significantly larger and richer use of explicit environments in the R ecosystem compared to theirs. Goel and Vitek [2019] studied the design and use of laziness in R. They provide a detailed account of the language's evaluation strategy with a small-step operational semantics and an empirical evaluation of laziness. Their semantics shows that promises are stored in environments and can outlive the frame that created them if they are returned as part of that environment. Turcotte et al. [2020] inferred type signatures for R functions by observing the type of argument and return values. Their type language includes a type for first-class environments.

3 The R Language

A presentation of the language requires introducing some key concepts. We will be brief; interested readers should consult [Wickham 2019].

Functions. Functions are first-class, anonymous, lexically-scoped values with optional parameters and default values. R is "functional" in the sense that values have a copy-on-write semantics, so side-effects to arguments inside a function are not reflected to the caller.

Promises. R is a mostly lazy language. Arguments to functions are packaged into promises which bundle an expression and its environment. When the value of a promise is needed, the expression is evaluated in the adjoined environment, and the result is cached in the promise.

Vectors. Most values in R are vectors, and most operations are vectorized. Vectors are homogeneous arrays of integer, double, character, logical, complex, or raw values.

Lists. Lists are heterogeneous vectors with optionally named elements. They can be indexed by position or name. Lists (and vectors) have a copy-on-write semantics.

Attributes. Values can be tagged with user-defined attributes which are a map from string to value. Consider the following code, which sets the attributes `dim` and `class` of some vector `x`.

```
x <- c(1,2,3,4)
attr(x, dim) <- c(2,2)
class(x) <- c("cat")
```

Setting the `dim` attribute causes the vector to be subsequently treated as a 2×2 matrix. Similarly, the class attribute is used for method dispatch in an object-oriented style.

Metaprogramming. Expressions can be evaluated explicitly in an environment using `eval`. Moreover, `substitute` extracts the unevaluated argument text from the promise bound to an argument in the supplied environment. The following example demonstrates how `substitute` constructs an AST by replacing parameters `x` and `y` with the argument expressions `a+b` and `c*d`, respectively.

```
f <- function(x, y) substitute((x) + y)
f(a + b, c * d)
# (a + b) + c * d
```

Formulas. A formula is a compact symbolic representation of models used by statistical functions. For example, the linear model `y~x-1` specifies a line through the origin. Each formula contains a reference to the environment in which it is defined.

4 Environments in R

The evolution of R has been gradual, rather a panoply of abstractions, the language designer opened up the internals of the interpreter for all to see, and packaged some commonly needed functionality with the core of the language. Thus, the task of describing the "interface" of environments as seen by developers and end-users involves some sleuthing. As we prepared this paper, we kept discovering new functions that accessed environments from native code. This section is not meant to be exhaustive, but it gives an overview of the most commonly used functions.

Environments are implemented either by an association list or a hashmap, as specified by a construction parameter. Each environment also has a parent, forming an acyclic chain terminated by the empty environment. That environment is returned by `emptyenv()`. Unlike other values, environments are mutable.

4.1 Environments as Packages

Packages are loaded by calls to the `library()` function and are represented by environments; their names are added to a global search path. A package can be retrieved by position, the *n*-th package is accessed by `as.environment(n)` or by name. Every package has a corresponding namespace that contains its private bindings and implementation specific metadata. A namespace can be obtained by calling `getNamespace(p)`. `baseenv()` returns the base package environment that is pre-loaded with R. The base package environment is also bound to the global variable, `.BaseNamespaceEnv`.

4.2 Environments as Lexical Scopes

In R, each function has a lexical scope. Furthermore, each nested function introduces a nested scope. When a function begins executing, its scope is initialized with parameters, and, as it runs, new variables are introduced implicitly each time an undefined symbol is assigned to. R allows obtaining the current scope with a call to `environment()`. From there, it is possible to read and update the environment through the reference returned by that function.

R provides access to the call stack. Frames start from 0 and increase by one for each nested call. Function `sys.nframe` returns the current frame number, and `sys.parent(n)` returns the number of the `n` th parent. The function `sys.frame(n)` returns environment at that position (counting backward if `n` is negative); `parent.frame(n)` optimises calls to `sys.frame(sys.parent(n))`. Lastly, `sys.frames()` returns a list of active environments.

```
f <- function() { print(environment()); g() }
g <- function() { print(parent.frame(1)); }
f()
# <env: 0x7f2> <env: 0x7f2>
```

In the above code snippet, `f` accesses its environment using the call to `environment()`, and its callee `g` accesses `f`'s environment by calling `parent.frame(-1)`.

The global environment, referred to by the variable `.GlobalEnv` (at offset 0), is returned by `globalenv()`. One can rebind the enclosing environment of a function with `environment(f)<-e`.

```
f <- function() print(environment())
environment(); environment(f); f()
# <env: Global> <env: Global> <env: 0x7ff>
e <- new.env()
print(e)
# <env: 0x7f1>
environment(f) <- e
environment(f)
# <env: 0x7f1>
```

The code snippet above modifies the enclosing scope of `f`, defined at the top-level. The environment `e`, created using `new.env`, is assigned the enclosing scope of `f` using `environment(f)<-e`.

4.3 Environments as Data Structures

Environments can be created by calls to `new.env`; it takes three optional arguments: a boolean to select how the environment is represented, the pre-allocation size, and an enclosing environment. `length` returns the number of bindings in an environment. `parent.env` yields the enclosing environment and `parent.env(e)<-p` sets `e`'s enclosing environment to `p`. The code snippet below illustrates these functions.

```
e <- new.env(parent=emptyenv())
length(e)
# 0
parent.env(e)
# <environment: R_EmptyEnv>
parent.env(e) <- globalenv()
parent.env(e)
# <environment: R_GlobalEnv>
```

`as.list` converts environments to lists. `list2env` copies a list to an environment. The variables of an environment can be retrieved as a vector using the `ls` and `objects` functions as shown below.

```
l <- list(x=1, y=2); e <- list2env(l)
length(e)
# 2
as.list(e)
# list(y = 2, x = 1)
ls(e)
# [1] "x" "y"
```

A variable's existence can be queried using `exists`. Its value can be retrieved using `$` and `[[` operators. Functions `get`, `get0`, `mget`, and `dynGet` are generalizations of these operators with options to perform lookup recursively in parent environments (`inherits=TRUE`) and validate the type of value bound to the variable being read (`mode="integer"`). `mget` is a vectorized version of `get`; it reads multiples variables supplied as a vector and returns a list of values. `dynGet` performs recursive lookups in caller frames, i.e., dynamic scopes, unlike the other functions which perform lookups in lexical scopes. Writes are performed using `assign` and `e$v<-x`, and `e[["v"]]<-x`. Bindings can be removed from an environment using the `rm` and `remove` function. Environments can be protected from addition or removal of bindings by calling `lockEnvironment`. This does not prevent updates of existing variables, those need to be explicitly locked using `lockBinding`.

5 Infrastructure and Corpus

The experimental results reported in this paper are produced by a dynamic analysis infrastructure running over a large corpus of R programs. The infrastructure has three tasks: assembling executable programs from R packages, generating execution traces using a modified R interpreter, and post-processing the traces to generate graphs and statistics. For reproducibility, the infrastructure lives in a Docker container based on Debian 10.9. Figure 1 shows the pipeline along with the duration of each stage and the size of returned results. We discuss the three tasks of this pipeline next.

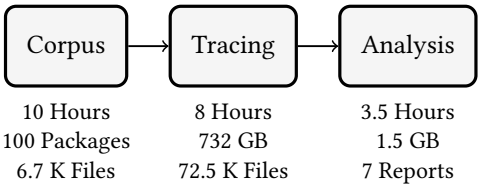

Figure 1. Pipeline

Corpus. Our corpus is assembled from R packages hosted on CRAN [Ligges 2017], the official R package repository. We mirror CRAN on our server and install its packages. We downloaded and installed 17,133 CRAN packages.[1] From these, we select the 100 CRAN packages with the highest number of clients. These 100 packages together have 11,786 clients (*ggplot2* has the highest number of clients, 2,320, and package *vctrs* has the fewest, 108). These packages contain 481K lines of R code and 1M lines of native code. During execution, these 100 packages call functions from 186 other packages, so our evaluation also includes them. These extra packages have 478K lines of R code and 1.1M lines of native code.

[1] Snapshot taken on 29 May 2021.

CRAN packages come equipped with runnable code in the form of tests, examples, and long-form examples called vignettes. Examples demonstrate the use of a package's functions, and vignettes illustrate a package's functionality with a larger example, typically using data supplied with the package. These programs are extracted as independently executable scripts for evaluation by the analysis pipeline. Overall, there are 6.9K scripts with 205.8K lines of code, Table 1 has details.

Table 1. Corpus

	Tests	Examples	Vignettes
Scripts	1.5K	5.0K	187
LOC	136.7K	55.2K	13.9K

Tracing. Execution traces are generated using envtracer, a dynamic analyzer built on top of R-dyntrace. R-dyntrace modifies GNU R 4.0.2 [Goel and Vitek 2019] to record events during program execution. envtracer collects execution information associated with environments from these events.

Analysis. The tracing step generates 681GB of data; analyzing data at this scale is thus the major challenge. We use a custom map-reduce analysis that first processes individual traces in parallel to generate smaller tables per program. This is expensive, but it substantially reduces data size. Then the tables are concatenated into a single table per analysis. Finally, summaries are computed from the concatenated tables. The report phase generates graphs and tables from these summaries.

6 How Frequent Are Environments?

The 286 corpus packages have 44K functions, of which 18K are exercised. From the un-exercised 26K functions, the majority belong to transitively included packages for which we do not have tests, and, some 8K functions are from our initial target packages but were unused.

Table 2. Package Size

Functions	Packages	Functions	Packages
1–25	169	251–300	4
26–50	40	301–400	6
51–100	17	401–500	2
101–150	14	501–600	3
151–200	11	601–700	0
201–250	15	701–800	3

Table 2 presents the distribution of exercised functions across these packages. We observe that 171 packages have 25 functions or less. There are few large packages; 8 with more than 500 functions.

We observed 42M calls to these functions. Figure 2 shows the distribution of calls: 53% of functions are called more than ten times, and 14% of functions are called only once.

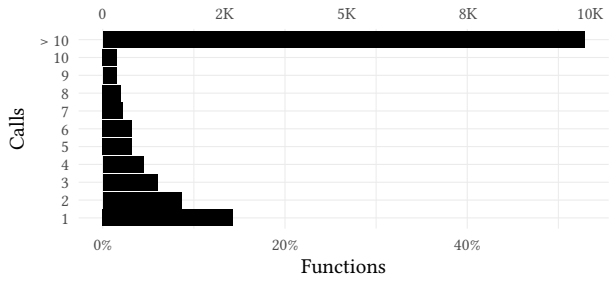

Figure 2. Call Distribution

These functions have a total of 67K parameter positions. Figure 3 shows the distribution of parameters: 3% functions have none, 22% have one parameter, and 5% have over 10. There are 4 functions with over 50 parameters, and the `ggplot2::theme` function has 95 parameters.

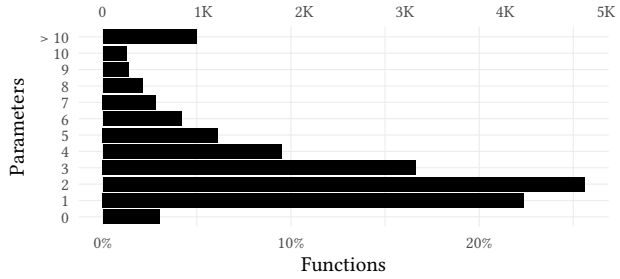

Figure 3. Parameter Distribution

We counted 1.2B environments, which makes them the second most widely allocated values. Table 3 shows the frequency of other values for comparison. Promises lead as there is one per parameter [Goel and Vitek 2019]. Vectors of logicals and characters are more frequent than integers, reals, and raw.

Table 3. Object Counts

Type	Count	Type	Count
Promise	2.8B	List	159.3M
Environment	1.2B	Closure	114.0M
Logical	1.0B	Real	113.4M
Character	929.9M	Symbol	73.5M
Language	483.9M	Raw	46.4M
Integer	453.5M	Other	15.2M

Table 4 gives the number of calls made to the various environment APIs. Each of these functions takes or returns an environment as an argument. Overall, they cover most of the non-traditional uses of environments.

Table 4. API Calls

Function	Calls	Function	Calls
`substitute`	15.8M	`sys.function`	3.2M
`environment`	13.0M	`list2env`	2.6M
`baseenv`	12.2M	`sys.call`	2.2M
`as.environment`	10.0M	`parent.env<-`	2.2M
`parent.frame`	6.9M	`parent.env`	2.1M
`getNamespace`	6.9M	`$`	2.1M
`sys.frame`	6.2M	`.subset2`	954.4K
`get0`	6.1M	`sys.nframe`	705.7K
`get`	4.9M	`environment<-`	697.0K
`sys.parent`	3.8M	`$<-`	659.0K
`[[`	3.6M	`exists`	505.1K
Function	**Calls**	**Function**	**Calls**
`assign`	392.4K	`[[<-`	21.1K
`lockBinding`	332.2K	`remove`	14.0K
`mget`	291.4K	`sys.parents`	7.2K
`emptyenv`	234.0K	`sys.frames`	3.6K
`as.list`	216.0K	`dynGet`	2.6K
`lockEnvironment`	206.2K	`ls`	2.5K
`globalenv`	179.1K	`unlockBinding`	1.3K
`~`	129.9K	`length`	375
`environmentName`	66.9K	`objects`	119
`rm`	21.1K	`pos.to.env`	7

7 Where Do Environments Come From?

Table 5 presents the distribution of environments by origin. The *Core* row is for environments created by R's implementation and its 16 core packages.[2] The *User* row is for environments that come from user-defined packages. We further differentiate between environments created in *native* code and in *R* code. Native environments are created using C APIs: `allocSExp`, `Rf_NewEnvironment`, and `R_NewHashedEnv`. R environments come from calls to `new.env`. Core is responsible for over 99% of environments, mostly from C code. Whereas the 0.03% of User environment are twice as likely to originate in R. We encountered 904K environments created during the initialization of the R session, we ignored those from the rest of the discussion.

Table 5. Environment Source

	Source	#	%
Core	*Native*	1.2B	99.62%
	R	3.1M	0.27%
User	*Native*	154.9K	0.01%
	R	240.4K	0.02%

[2]They are base, compiler, datasets, grDevices, graphics, grid, methods, parallel, profile, splines, stats, stats4, tcltk, tools, translations, and utils.

In the Core Native class, 99% of environments are needed to implement function environments. There 344K environment used for package namespaces, as well as 2.8M for the S4 object system implemented by the `methods` package. Some 165K were used for `eval` and 145K for `substitute`. In the Core R class, 94% of the environments come from the `base` package and 5% from `methods`.

8 How Are Environments Used?

We divide environments into three categories, presented in Table 6: **Calls** are environments used for evaluating function calls, **Explicits** are created for non-standard purposes, and **Packages** are needed for package loading.

Table 6. Environment Categories

Calls	1.2B	99.3%
Explicits	3.7M	0.3%
Packages	3.3M	0.2%

8.1 Calls

Of 1.2B calls, only 20M environments are passed to the functions of Table 4. The remaining environments are used only as "traditional" environments for creating, reading, and updating variables. To understand how those 20M environments are used, we summarize manipulation of these environments with four groups of operations:

- A: variable reads, writes and removes.
- V: `eval`.
- S: `substitute`.
- X: `parent.frame`, `sys.frame` or `sys.frames`.

Any environment may be used by a combination of the operations above. Table 7 has the frequency of the most common "sets" of operations, these are operations that happen to the same environment in no particular order or frequency. Overall, there are 63 sets but the top four explain 98% of non-trivial call environments.

Table 7. Operation mix

Event	#	Cum. %
S	9.8M	46%
X,A	8.3M	86%
X,V,A	2.2M	97%
X,S,V,A	312K	98%

S: Most uses of `substitute` originate from `base` package functions such as `::`. When users write `ns::sym`, this has the effect of reading variable `sym` publicly exported from namespace `ns`. Here, `substitute` is used to access the symbol and namespace names, the names are converted to strings, then `get` does the actual lookup.

```
`::` <- function(pkg, name) {
  pkg <- as.character(substitute(pkg))
  name <- as.character(substitute(name))
  getExportedValue(pkg, name)
}
```

X,A: These environments are obtained as values and then used for accessing their bindings. For instance, `registerS3method::assignWrapped` uses `parent.frame` to get the caller environment and then evaluates a promise in that environment accessing its variables.

```
assignWrapped <- function(x, method, home,
                          envir) {
  method <- method
  home <- home
  delayedAssign(x, get(method, envir = home),
                assign.env = envir)
}
home <- parent.frame()
assignWrapped(home = home, ...)
```

X,V,A: These environments are obtained for the purpose of evaluating code in them. The use of `glue::glue` by `waldo::path_attr` is an example where `glue` performs string interpolation by extracting the caller's environment and evaluating embedded code blocks.

```
path_attr <- function(path, i) {
  funs <- c("comment", "class", "dim")
  ifelse(i %in% funs,
         glue("{i}({path})"),
         glue("attr({path},␣'{i}')"))
}
```

X,S,V,A: These environments are used in a combination of `eval` and `substitute` use cases. This occurs in `match.arg` when the set of values against which the argument is to be matched are not provided, then `match.arg` uses `substitute` to get argument names and reflectively access their default values from the caller environment.

```
match.arg <- function(arg, choices,
                      several.ok = FALSE) {
  sysP <- sys.parent()
  formal.args <- formals(sys.function(sysP))
  argname <- as.character(substitute(arg))
  choices <- eval(formal.args[[argname]],
                  envir = sys.frame(sysP))
```

Apart from these, formula construction also stands out as a frequent operation. These formulas extract the environment of the call in which they are created and carry them around as attributes. We observed 66K formulas constructed in call environments. The most common example is the `stats::formula` function.

While one could hope an optimizing compiler would optimize most call environments, unfortunately, there are sufficient number of reflective accesses that it may be hard for the compiler to be able to determine that an environment can be elided.

8.2 Explicits

Explicit environments created using `new.env` mostly come from core, and 395K are created in user code.

8.2.1 Core Explicits. Nine packages are responsible for all explicits in Core. Table 8 shows these packages and the number of environments created. The `base` and `methods` packages alone account for 99% of environments. Table 9 shows the six functions that alone contribute to 98% of all explicit environments.

Table 8. Core Explicit Environment Packages

Package	#	%
`methods`	3.0M	89.2%
`base`	329.9K	99.0%
`grid`	15.7K	99.5%
`grDevices`	10.7K	99.8%
`stats`	2.7K	99.9%
`compiler`	2.4K	100%
`parallel`	610	100%
`tools`	217	100%
`utils`	7	100%

Table 9. Core Explicit Environment Functions

Function	#	Cum. %
`methods::new`	2.8M	84.3%
`base::eval`	165.5K	89.2%
`base::substitute`	145.8K	93.5%
`methods:.mlistAddToTable`	50.2K	95.0%
`methods:.resetInheritedMethods`	50.2K	96.5%
`methods::makeGeneric`	50.2K	98.0%

We now turn our attention to how these environments are used. Table 10 shows the top 5 of the full sets of operations performed on these environment. These sets include the following new operations:

- `L`: locking environments or bindings.
- `Z`: modifying parent environment.
- `!`: using environment as parent or lexical scope.

A: This is the most common case; environments that are only used to access variables, as in the `methods::new` function that is used for creating S4 objects.

A,V: These environments are used for evaluation by the `eval` and `evalq` functions.

Table 10. Core Explicit Environment Events

Event	#	Cum. %
`A`	3.0M	88.4%
`A,V`	165.6K	93.4%
`S`	145.8K	97.7%
`A,Z,!`	50.2K	99.2%
`A,L,!`	12.1K	99.6%

S: These environments are used with `substitute`.

A,Z,!: An example of this is the `methods::makeGeneric` function. It creates a new environment, assigns the field `".Generic"` to the name of the generic method, sets its parent as the lexical scope of the function, and finally, sets the new environment as the lexical scope of the function.

```
ev <- new.env()
parent.env(ev) <- environment(fdef)
environment(fdef) <- ev
packageSlot(f) <- package
assign(".Generic", f, envir = ev)
```

A,L,!: This happens in S4 objects. Their underlying data store consists of an environment with a reference to the object. This binding is locked to prevent modification.

```
selfEnv <- new.env(TRUE, objectParent)
for(field in fields) {
    fp <- prototypes[[field]]
    environment(fp) <- selfEnv
    assign(field, fp, envir = selfEnv)
}
selfEnv$.self <- .Object
lockBinding(".self", selfEnv)
lockBinding(".refClassDef", selfEnv)
```

Overall, 168K environments are passed to eval, and only 100 were used for formula construction. Overall, explicit environments are used for evaluation, substitution, and in the S4 object system. This category is integral to the language implementation.

8.2.2 User Explicits. User explicits come from 55 packages. Table 11 shows the distribution of the top 8 packages, which account for 96% of creations. The `vctrs` package allows for type-coercion and size-recycling of vectors. The `rlang` package provides utility functions for working with objects and a variant of `eval`. The `R6` package implements an object-oriented system. The `codetools` package implements code analysis. The `ggplot2` package is a popular plotting library. The `testthat` package is used for testing. The `dplyr` package implements a DSL for SQL-like queries on data frames. Lastly, `magrittr` implements the pipe operator for composing function. Table 12 shows the top ten functions creating environments; they contribute to 65% of all explicits.

Table 11. Explicits Packages

Package	#	Cum. %
vctrs	142.9K	36.2%
rlang	75.2K	55.2%
R6	74.1K	73.9%
codetools	39.2K	83.8%
ggplot2	24.3K	90.0%
testthat	9.1K	92.3%
dplyr	8.3K	94.4%
magrittr	6.1K	95.9%

Table 12. Explicits Functions

Function	#	Cum. %
R6::generator_funs::new	63.4K	16.0%
vctrs::vec_c	55.7K	30.1%
codetools::mkHash	31.3K	38.0%
ggplot2::ggproto	24.3K	44.2%
rlang::eval_tidy	18.1K	48.8%
vctrs::vec_slice	16.5K	52.9%
rlang::new_data_mask	13.8K	56.4%
vctrs::vec_cast_common	12.8K	59.7%
vctrs::vec_as_names	10.7K	62.4%
R6::create_super_env	10.6K	65.1%

Now, we turn our attention to how these environments are used. Table 13 shows the top 7 of the 82 operation mixes which characterize 96% of these environments. We observe a new operation, `@`, used to set class attributes.

Table 13. Explicits Operations

Events	#	Cum. %
A,V	154.0K	39.0%
A	102.8K	65.0%
A,!	43.8K	76.1%
A,@	38.9K	85.9%
A,L	30.4K	93.6%
A,L,@	8.3K	95.7%
A,@,!	3.2K	96.5%

A,V: These environments are created for custom evaluation strategies, *i.e.*, for evaluating expressions with custom bindings. For example, the `testthat` library uses them for running tests.

```
test_code <- function(code, env = test_env()) {
  test_env <- new.env(parent = env)
  eval(code, test_env)
```

A: These environments are used as hash tables and mutable state. For example, the `codetools::mkHash` function creates an environment to store intermediate static analysis information.

```
findGlobals <- function(fun, merge = TRUE) {
  funs <- mkHash()
  enter <- function(v) assign(v, TRUE, funs)
  collectUsage(fun, enterGlobal = enter)
  fnames <- ls(funs, all.names = TRUE)
```

A,!: These environments are used as parents of other environments or functions. For example, the `R6` package creates new environments and sets them as lexical scope of object methods.

```
assign_func_envs <- function(objs, env) {
  lapply(objs, function(x) {
    if (is.function(x)) environment(x) <- env
    x
  })
}

new <- function(...) {
  env <- new.env(parent=parent_env,hash=FALSE)
  methods <- assign_func_envs(methods, env)
```

A,@: These environments are used to create custom objects which can be used for dispatch. For example, the `ggproto` objects are explicit environments with the `"ggproto"` class attribute.

```
ggproto <- function(`_class` = NULL, ...) {
  e <- new.env(parent = emptyenv())
  e$super <- find_super
  class(e) <- c(`_class`, "ggproto", "gg")
```

A,L: This operation mix is seen in environments created by the R6 package which locks them during object instantiation to prevent any modification.

```
new <- function(...) {
    pub_env <- new.env(parent=emptyenv())
    lockEnvironment(pub_env)
```

A,L,@: These environments come from the `later` package which uses them as handles for event loop objects. These objects contain a unique loop identifier that is locked to prevent modification. The environments are given the class attribute `"event_loop"`.

```
create_loop <- function(...) {
loop <- new.env(parent = emptyenv())
class(loop) <- "event_loop"
loop$id <- id
lockBinding("id", loop)
```

A,@,!: These environments shows up in the `plyr` package, which creates environments with attribute `"idf"` for immutable data frames. It also assigns getter functions to these environments to access the columns of the data frame, and sets their lexical scope to be the environment itself.

```
idata.frame <- function(df) {
  self <- new.env()
  self$`_data` <- df
  self$`_getters` <- lapply(names(df), ...)
  names(self$`_getters`) <- names(df)
  for (name in names(df)) {
    f <- self$`_getters`[[name]]
    environment(f) <- self
  }
  structure(self, class = c("idf"))
}
```

Only 597 environments in this category were used for formula construction. Out of these, 389 were created in tests. The `survival` package stands out as it creates explicits for formulas. 162K of explicits were used for dynamic code evaluation. 50K of these environments have a class attribute. Table 14 presents the class attributes attached to environments.

Table 14. Environment Attributes

Package	Attributes	#	Cum. %
ggplot2	ggproto gg	24.3K	47.8%
rlang	rlang_ctxt_pronoun	12.1K	71.5%
R6	R6	9.2K	89.5%
rlang	r6lite	3.7K	96.8%
plyr	idf environment	1.2K	99.2%
later	event_loop	279	99.7%
R6	R6ClassGenerator	113	100%
shiny	session_proxy	12	100%
XML	XMLHashTree XMLAbstractDocument	10.0	100%
xts	replot_xts environment	2	100%

8.3 Packages

We observe 3.3M environments related to packages and namespaces. The package loading mechanism alone accounts for 2.9M of these. The remaining are used as package namespaces. 2.3M of these environments originate from `lazyLoadDBexec`, an internal function responsible for loading a package's code from a binary file. A few environments are created internally by the interpreter to store a package's native functions. The internal structure of these environments is unspecified.

9 Enclosing Scope Manipulation

A closure's enclosing scope can be accessed using `env(fun)` and modified using `env(fun)<-e`. Table 15 lists calls to these functions.

First, we look at the `environment` function called 97% of the time from Core. The `methods::registerS3methods` is responsible for 43.8% of calls to `environment`. This function registers

Table 15. Enclosing Scope API

		P#	F#	C#	C%
environment	*Core*	5	37	249.3K	97%
	User	15	33	6.7K	3%
environment<-	*Core*	2	4	114.4K	48.0%
	User	17	37	122.9K	52.0%

S3 methods for dispatch. It extracts the enclosing environment of the method and updates the method information in its `S3MethodsTable`.

```
defenv <- environment(genfun)
table <- new.env(hash = TRUE, parent = baseenv())
defenv[[".__S3MethodsTable__."]] <- table
```

Overall, the `methods` package is responsible for 52.2% of the calls.

The `compiler::cmpfun` function, used for byte code compilation is another client. It extracts the body, formal parameter list, and the enclosing scope of the function to be compiled.

```
cmpfun <- function (f, options = NULL) {
    cntxt <- make.toplevelContext(
        makeCenv(environment(f)), options)
    ncntxt <- make.functionContext(cntxt, formals(f),
                                   body(f))
```

Only, 3% of the calls to `environment` originate from User. The primary contributor to these calls is the `R.oo` package. This package implements objects with a `getStaticInstance.Class` function that calls the `environment` function.

```
setMethodS3("getStaticInstance", "Class",
  function(this, ...) {
    environment(static) <- environment(this)
```

We observe that `environment<-` is called almost equally by both Core and User packages. Table 16 gives the top five callers of `environment<-`, which account for 98.59% of calls.

Table 16. Top `environment<-` Callers

Function	Call %
`R6::assign_func_envs`	50.0%
`methods:.makeDefaultBinding`	32.6%
`stats::make.link`	11.8%
`methods::installClassMethod`	3.8%
`MASS::negative.binomial`	0.4%

On the Core side, `methods`, and `stats` are responsible for all calls to `environment<-`. Two functions in `methods`, `.makeDefaultBinding` and `installClassMethod`, are responsible for 75.47% of all Core calls.

```
.makeDefaultBinding <- function(...) {
  f <- function(value) ...
  environment(f) <- where
```

The stats::make.link function returns a list of functions related to a model. These functions are defined as its inner functions which don't use any of the parent scope bindings. Before returning, it modifies their definition environment to be the stats package namespace.

```
make.link <- function (link) {
 linkfun <- function(mu) ...
 linkinv <- function(eta) ...

 environment(linkfun) <- environment(linkinv) <-
 asNamespace("stats")
```

On the User side, R6 dominates calls to environment<-. R6::assign_func_envs is the most frequent caller, accounting for half of all calls. It uses environment<- to change the enclosing scope of object methods.

```
assign_func_envs <- function(objs, target_env) {
  lapply(objs, function(x) {
    if (is.function(x)) environment(x) <- target_env
    x
  })
}
```

The MASS::negative.binomial function uses environment<- to modify the parent scope of its inner functions, similar to the stats::make.link function described above.

10 Locking

Calling lockEnvironment prevents the introduction of new bindings while lockBinding prevents their mutation. Bindings can be unlocked with unlockEnvironment; the use of this function triggers a warning from the automated package checker. Table 17 presents the distribution of calls to these functions.

Table 17. Locking and Unlocking API

		P#	F#	C#	C%
lockEnvironment	*Core*	1	3	166.4K	80.1%
	User	2	3	41.3K	19.9%
lockBinding	*Core*	1	3	32.1K	9.4%
	User	6	7	309.0K	90.6%
unlockBinding	*Core*	1	1	688.0	53.8%
	User	5	5	590.0	46.2%

lockEnvironment is called 80.1% of the time by three base functions, sealNamespace, attachNamespace, and envhook. The first two initialize package and namespace environments. The third loads code from a database.

On the User side, only packages R6 and rlang lock environments. Most calls originate from R6. An example is the R6::clone method that locks the public and private method environments of object clones.

```
clone <- function(deep = FALSE) {
  lockEnvironment(new_1_binding)
  lockEnvironment(new_1_private)
```

The methods package is responsible for all calls to lockBinding and unlockBinding on the Core side. On the User side, the use of these functions is varied. Most package functions call them in a matched pair. For insance, gtools locks bindings to circumvent a a bug in R.[3] Some functions in gtools cause the bytecode interpreter to run out of stack space. The package has a function, unByteCodeAssign, that calls assignEdgewise to update the functions that trigger the bug to an equivalent non-bytecode version.

```
assignEdgewise <- function(name, env, value) {
  unlockBinding(name, env = env)
  assign(name, envir = env, value = value)
  lockBinding(name, env = env)
  invisible(value)
}

unByteCodeAssign <- function(fun) {
  FUN <- unByteCode(fun)
  retval <- assignEdgewise(name=name, env=environment
       (FUN), value=FUN)
}
```

Overall, R6 is the biggest user of locking. Bindings are rarely unlocked, and they are never used to inject new bindings in other packages. We only found one package, data.table, which unlocked bindings to update base functions cbind and rbind.

11 Conclusion

This paper looked at first-class environments in R. We introduced the main functions that operate on environments and reported on an observational study of 100 popular R packages. At the outset, our hope was that we could uncover some ways to simplify and rationalize the design of R's environments. We conclude with the rather disappointing observation that it seems that all of the generality of the environment interface is needed, or at least that it is used. While in the vast majority of cases environment access could be optimized and environment could be implemented in a straightforward manner, there are sufficient number of cases where environments escape and are used in a reflective manner that it is not clear such optimizations can be widely applied.

[3] https://bugs.r-project.org/bugzilla/show_bug.cgi?id=15215

Acknowledgments

This work has received funding from National Science Foundation awards 1759736, 1925644 and 1618732, 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

N. Adams et al. 1998. Revised[5] Report on the Algorithmic Language Scheme. *SIGPLAN Not.* 33, 9 (1998). https://doi.org/10.1145/290229.290234

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

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

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

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

John McCarthy. 1959. LISP: a programming system for symbolic manipulations. In *National meeting of the Association for Computing Machinery.* https://doi.org/10.1145/612201.612243

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

Jeffrey Mark Siskind and Barak A. Pearlmutter. 2007. First-Class Nonstandard Interpretations by Opening Closures. *POPL '07* (2007). https://doi.org/10.1145/1190216.1190230

Guy L. Steele. 1982. An Overview of Common Lisp. In *Symposium on LISP and Functional Programming (LFP).* https://doi.org/10.1145/800068.802140

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

Hadley Wickham. 2019. *Advanced R.* Chapman and Hall/CRC.