Evaluating the Design of the R Language

Objects and Functions For Data Analysis

Floréal Morandat Brandon Hill Leo Osvald Jan Vitek

Purdue University

Abstract. R is a dynamic language for statistical computing that combines lazy functional features and object-oriented programming. This rather unlikely linguistic cocktail would probably never have been prepared by computer scientists, yet the language has become surprisingly popular. With millions of lines of R code available in repositories, we have an opportunity to evaluate the fundamental choices underlying the R language design. Using a combination of static and dynamic program analysis we assess the success of different language features.

1 Introduction

Over the last decade, the R project has become a key tool for implementing sophisticated data analysis algorithms in fields ranging from computational biology [7] to political science [11]. At the heart of the R project is a *dynamic, lazy, functional, object-oriented* programming language with a rather unusual combination of features. This computer language, commonly referred to as the R language [15,16] (or simply R), was designed in 1993 by Ross Ihaka and Robert Gentleman [10] as a successor to S [1]. The main differences with its predecessor, which had been designed at Bell labs by John Chambers, were the open source nature of the R project, vastly better performance, and, at the language level, lexical scoping borrowed from Scheme and garbage collection [1]. Released in 1995 under a GNU license, it rapidly became the lingua franca for statistical data analysis. Today, there are over 4 000 packages available from repositories such as CRAN and Bioconductor.[1] The R-forge web site lists 1 242 projects. With its 55 user groups, Smith [18] estimates that there are about 2 000 package developers, and over 2 million end users. Recent interest in the financial sector has spurred major companies to support R; for instance, Oracle is now bundling it as part of its Big Data Appliance product.[2]

As programming languages go, R comes equipped with a rather unlikely mix of features. In a nutshell, R is a dynamic language in the spirit of Scheme or JavaScript, but where the basic data type is the vector. It is functional in that functions are first-class values and arguments are passed by deep copy. Moreover, R uses lazy evaluation by default for all arguments, thus it has a pure functional core. Yet R does not optimize recursion, and instead encourages vectorized operations. Functions are lexically scoped and their local variables can be updated, allowing for an imperative programming style. R targets statistical computing, thus missing value support permeates all operations. The

[1] `http://cran.r-project.org` and `http://www.bioconductor.org`.

[2] `http://www.oracle.com/us/corporate/press/512001`.

dynamic features of the language include forms of reflection over its environment, the ability to obtain source code for any unevaluated expression, and the `parse` and `eval` functions to dynamically treat text as code. Finally, the language supports objects. In fact, it has two distinct object systems: one based on single-dispatch generic functions, and the other on classes and multi-methods. Some surprising interactions between the functional and object parts of the language are that there is no aliasing, object structures are purely tree-shaped, and side effects are limited.

The R language can be viewed as a fascinating experiment in programming language design. Presented with a cornucopia of programming models, which one will users choose, and how? Clearly, any answer must be placed in the context of its problem domain: data analysis. How do these paradigms fit that problem domain? How do they strengthen each other and how do they interfere? Studying how these features are used in practice can yield insights for language designers and implementers. As luck would have it, the R community has several centralized code repositories where R packages are deposited together with test harnesses. Thus, not only do we have all the open source contributions made in the last 15 years, but we also have them in an executable format. This paper makes the following contributions:

- *Semantics of Core R:* Some of the challenges dealing with R come from the fact it is defined by a single implementation that exposes its inner workings through reflection. We make the first step towards addressing this issue. Combining a careful reading of the interpreter sources, the R manual [16], and extensive testing, we give the first formal account of the semantics of the core of the language. We believe that a precise definition of lazy evaluation in R was hitherto undocumented.
- *TraceR Framework:* We implemented TraceR, an open source framework for analysis of R programs. TraceR relies on instrumented interpreters and off-line analyzers along with static analysis tools.
- *Corpus Gathering:* We curated a large corpus of R programs composed of over 1 000 executable R packages from the Bioconductor and CRAN repositories, as well as hand picked end-user codes and small performance benchmark programs that we wrote ourselves.
- *Implementation Evaluation:* We evaluate the status of the R implementation. While its speed is not acceptable for use in production systems, many end users report being vastly more productive in R than in other languages. R is decidedly single-threaded, its semantics has no provisions for concurrency, and its implementation is hopelessly non-thread safe. Memory usage is also an issue; even small programs have been shown to use immoderate amounts of heap for data and meta-data. Improving speed and memory usage will require radical changes to the implementation, and a tightening of the language definition.
- *Language Evaluation:* We examine the usage and adoption of different language features. R permits many programming styles, access to implementation details, and little enforcement of data encapsulation. Given the large corpus at hand, we look at the usage impacts of these design decisions.

The code and data of our project are available in open source from:

`http://r.cs.purdue.edu/`

2 An R Primer

We introduce the main concepts of the R programming language. To understand the design of R, it is helpful to consider the end-user experience that the designers of R and S were looking for. Most sessions are interactive, the user loads data into the virtual machine and starts by plotting the data and making various simple summaries. If those do not suffice, there are some 4 338 statistical packages that can be applied to the data. Programming proper only begins if the modeling steps become overly repetitive, in which case they can be packaged into simple top-level functions. It is only if the existing packages do not precisely match the user's needs that a new package will be developed. The design of the language was thus driven by the desire to be intuitive, so users who only require simple plotting and summarization can get ahead quickly. For package developers, the goal was flexibility and extendibility. A tribute to the success of their approach is the speed at which new users can pick up R; in many statistics departments the language is introduced in a week.

The basic data type in R is the vector, an ordered collection of values of the same kind. These values can be either numerics, characters, or logicals. Other data types include lists (i.e., heterogeneous vectors) and functions. Matrices, data frames, and objects are built up from vectors. A command for creating a vector and binding it to `x` is:

```
x <- c(1, 2.1, 3, NA)
```

Missing values, `NA`, are crucial for statistical modeling and impact the implementation, as they must be represented and handled efficiently. Arithmetic operations on vectors are performed element by element, and shorter vectors are automatically extended to match longer ones by an implicit extension process. For instance,

```
v <- 1 + 3*x
```

binds the result of the expression `1+3*x` to `v`. There are three different vectors: `x` and two vectors of length one, namely the numeric constants `1` and `3`. To evaluate the expression, R will logically extend the constant vectors to match the length of `x`. The result will be a new vector equivalent to `c(4,7.3,10,NA)`. Indexing operators include:

```
v1 <- x[1:3];      v2 <- x[-1];      x[is.na(x)] <- 0
```

here `v1` is bound to a new vector composed of the first three elements of `x`, `v2` is bound to a new vector with everything but the first value of `x`, and finally, `x` is updated with 0 replacing any missing values.

In R, computation happens by evaluating functions. Even the assignment, `x<-1`, is a call to the built-in `assign("x",1)`. This design goes as far as making the `(` in a parenthesized expression a function call. Functions are first class values that can be created, stored and invoked. So,

```
pow <- function(b,e=2) if(e==1) b else b*pow(b,e-1)
```

creates a function which takes two arguments and binds it to `pow`. Function calls can specify parameters either by position or by name. Parameters that have default values, such as `e` above, need not be passed. Thus, there are three equivalent ways to call `pow`:

```
pow(3);      pow(3,2);      pow(e=2,3)
```

The calls all return the 1-element vector `9`. Named arguments are significantly used; functions such as `plot` accept over 20 arguments. The language also supports '`...`' in function definition and calls to represent a variable number of values. Explicit type declarations are not required for variables and functions. True to its dynamic roots, R checks type compatibility at runtime, performing conversions when possible to provide best-effort semantics and decrease errors.

R is lazy, thus evaluation of function arguments is delayed. For example, the `with` function can be invoked as follows:

```
with(formaldehyde, carb*optden)
```

Its semantics is similar to the JavaScript `with` statement. The second argument is evaluated in the context of the first which must be an environment (a list or a special kind of vector). This behavior relies on lazy evaluation, as it is possible that neither `carb` or `optden` are defined at the point of call. Arguments are boxed into *promises* which contain the expression passed as an argument and a reference to the current environment. Promises are evaluated transparently when their value is required. The astute reader will have noticed that the above example clashes with our claim that R is lexically scoped. As is often the case, R is lexically scoped up to the point it is not. R is above all a dynamic language with full reflective access to the running program's data and representation. In the above example, the implementation of `with` sidesteps lexical scoping by reflectively manipulating the environment. This is done by a combination of lazy evaluation, dynamic name lookup, and the ability turn code into text and back:

```
with.default <- function(env, expr, ...)
    eval(substitute(expr),env, enclose=parent.frame())
```

The function uses `substitute` to retrieve the unevaluated parse tree of its second argument, then evaluates it with `eval` in the environment constituted by composing the first argument with the lexically enclosing environment. The '`...`' is used to discard any additional arguments.

R associates attributes to all data structures, thus every vector, list, or function has a hidden map that associates symbols to values. For a vector, these include length, dimensions, and column names. Attributes are a key to R's extensibility. They are used as hooks for many purposes. As we will see in the next section, the object system relies on attributes to encode class membership. It is sometimes the case that all the interesting data for some value is squirreled away in attributes. Attributes can be updated by an assignment, e.g., to turn the vector `x` into a 2-by-2 matrix:

```
attr(x, "dim") <- c(2,2)
```

R has two different object systems. The simplest one uses a `class` attribute for implementing ad-hoc polymorphism. This attribute holds a series of strings denoting base class and parent classes in order. Any data structure can be labeled as the programmer wishes. The new object system allows for true class definitions and instance creation. It gives the programmer a similar multiple inheritance model as the early object system, but now allows for virtual classes and multi-method dispatch.

3 The Three Faces of R

We now turn to R's support for different paradigms.

3.1 Functional

R has a functional core reminiscent of Scheme and Haskell.

Functions as first-class objects. Functional languages manipulate functions as first-class objects. Functions can be created and bound to symbols. They can be passed as arguments, or even given alternate names such as `f<-factorize; f(v)`.

Scoping. Names are lexically scoped with no difference between variable and function declarations, as in Scheme [6]. All symbols introduced during the evaluation of a function are collected in a *frame*. Frames are linked to their lexically enclosing frame to compose environments. However, unlike many languages, bindings are performed dynamically [8]. Symbols can be added to an environment after it has been entered. This forces name resolution to be performed during function evaluation. The mechanism is subtle and has led to mistaken claims that R is not lexically scoped[3]. Consider,

```
function(){ f<-function()x;     x<-42;     f() }
```

where the function bound to `f` accesses a variable, `x`, that does not exist at the time of definition. But when `f` is called, lookup will succeed as `x` is in scope. Scoping has another somewhat surprising wrinkle, lookup is context sensitive. Looking up `c` and `c()` can yield different results from CommonLisp [19] or Scheme. By default, c is bound to the built-in function that creates new vectors. In these examples, the first code fragment binds 42 to `c`. When `c` is looked up in the function call, the binding is skipped because 42 is not a function, and lookup instead returns the default definition of `c` which is indeed a function. The second code fragment adds an assignment of `c` to `d`. When `d` is looked up, the only definition of `d` in the environment is the one that binds it to 42, so the call fails.[4] This is an example of best effort semantics; the lookup rules try to find a function when in a call context, which can yield surprising results.

```
c <- 42
c(2,3)
```

```
c <- 42
d <- c
d(2,3)
```

Lazy. R does not evaluate function arguments at calls. Instead, expressions are boxed into promises that capture the lexical environment. Promises are *forced* whenever their value is required by the computation. Languages like Haskell [9] delay evaluation as much as possible. This allows, e.g., for the elegant definition of infinite data structures. The R manual [16] does not state when promises are forced. As R is not purely functional, the order of evaluation is observable through side effects performed or observed by promises. Our investigation uncovered that promises are evaluated aggressively. They typically do not outlive the function that they are passed into.[5] Another surprising discovery is that name lookup will force promises in order to determine if a symbol is bound to a function.

[3] E. Blair, A critique of R (2004) `http://fluff.info/blog/arch/00000041.htm`

[4] Our core R semantics models this behavior with the [FORCEF] and [GETF] rules of Fig. 3.

[5] Unbounded data structures can be created by hiding promises in closures, e.g., `Cons <-function(x,y)list(function()x, function()y)`. An application is needed to get the value, e.g., `Car <-function(cons)cons[[1]]()`.

Consider the following three-argument function declaration:

```
function(y, f, z) { f(); return( y ) }
```

If it is not a function, evaluation forces `f`. In fact, each definition of `f` in the lexical environment is forced until a function is found. `y` is forced as well and `z` is unevaluated.

Referential transparency. A language is referentially transparent if any expression can be replaced by its result. This holds if evaluating the expression does not side effect other values in the environment. In R, all function arguments are passed by value, thus all updates performed by the function are visible only to that function. On the other hand, environments are mutable and R provides the super assignment operator (`<<-`) in addition to its local one (`<-`). Local assignment, `x<-1`, either introduces a new symbol or updates the value of an existing symbol in the current frame. This side effect remains local and is arguably easier to reason about than generalized side effects. The super assignment operator is a worse offender as it skips the current frame and operates on the rest of the environment. So, `x<<-1`, will ignore any definition of `x` in the current frame and update the first existing `x` anywhere in the environment or, if not found, define a new binding for `x` at the top-level. This form of side effect is harder to reason about as it is non-local and may be observed directly by other functions.

Parameters. R gives programmers much freedom in how functions are defined and called. Function declarations can specify default values and a variable number of parameters. Function calls can pass parameters by position or by name, and omit parameters with default values. Consider the `my` function declaration, it has three parameters, `a`, `k`, and `p`. The ellipsis specifies a variable number of parameters which can be accessed by the array notation or passed on to another function in bulk. This function can be called in different ways, for instance,

```
my <-
  function(a,k=min(d),...,p=TRUE){
    a <- as.matrix(a);
    d <- dim(a);
    l <- k+1;
    if (p) # some behavior.
}
```

```
my(x);    my(x, y);    my(x, y, z);    my(k=y, x, z, p=FALSE)
```

A valid call must have at least one, positional, parameter. Any argument with a default value may be omitted. Any argument may be passed by name, in which case the order in which it appears is irrelevant. Arguments occurring after an ellipsis must be passed by name. The default values of arguments can be expressions; they are boxed into promises and evaluated within the same environment as the body of the function. This means that they can use internal variables in the function's body. So `min(d)` above refers to `d`, which is only created during evaluation of the function. As `k` is always forced after `d` has been defined, the function will work as intended. But this shows that code can be sensitive to the order of evaluation, which can lead to subtle bugs.

3.2 Dynamic

Given R's interactive usage, dynamic features are natural. These features are intended to increase the expressiveness and flexibility of the language, but complicate the implementor's task.

Dynamic typing. R is dynamically typed. Values have types, but variables do not. This dynamic behavior extends to variable growth and casting. For instance:

```
v <- TRUE;    v[2] <- 1;    v[4] <- "1"
```

Vector `v` starts as a logical vector of length one, then `v` grows and is cast by the assignment to a numerical vector of length two, equivalent to `c(1, 1)`. The last assignment turns `v` into a string vector of length four, equivalent to `c("1", "1", NA, "1")`.

Dynamic evaluation. R allows code to be dynamically evaluated through the `eval` function. Unevaluated expressions can be created from text with the `quote` function, and variable substitution (without evaluation) can be done with `substitute` and partial substitution with `bquote`. Further, expressions can be reduced back to input strings with the `substitute` and `deparse` functions. The R manual [16] mentions these functions as useful for dynamic generation of chart labels, but they are used for much more.

Extending the language. One of the motivations to use lazy evaluation in R is to extend the language without needing macros. But promises are only evaluated once, so implementing constructs like a `while` loop, which must repeatedly evaluate its body and guard, takes some work. The `substitute` function can be used to get the source text of a promise, the expression that was passed into the function, and `eval` to execute it. Consider this implementation of a while loop in user code,

```
mywhile <- function(cond, body)
  repeat if(!eval.parent(substitute(cond))) break
             else eval.parent(substitute(body))
```

Not all language extensions require reflection, lazy evaluation can be sufficient. The implementation of `tryCatch` is roughly,

```
tryCatch <- function(expr, ...) {
  # set up handlers specified in ...
  expr
}
```

Explicit Environment Manipulation. Beyond these common dynamic features, R's reflective capabilities also expose its implementation to the user. Environments exist as a data type. Users can create new environments, and view or modify existing ones, including those used in the current call stack. Closures can have their parameters, body, or environment changed after they are defined. The call stack is also open for examination at any time. There are ways to access any closure or frame on the stack, or even return the entire stack as a list. With this rich set of building blocks, user code can implement whatever scoping rules they like.

3.3 Object Oriented

R's use of objects comes from S. Around 1990, a `class` attribute was added to S3 for ad-hoc polymorphism [4]. The value of this attribute was simply a list of strings used to dispatch methods. In 1998, S4 added proper classes and multi-methods [3].

S3. In the S3 object model there are neither class entities nor prototypes. Objects are normal R values tagged by an attribute named `class` whose value is a vector of strings. The strings represent the classes to which the object belongs. Like all attributes, an object's class can be updated at any time. As no structure is required on objects, two instances that have the same class tag may be completely different. The only semantics for the values in the `class` attribute is to specify an order of resolution for methods. Methods in S3 are functions which, in their body, call the dispatch function `UseMethod`. The dispatch function takes a string `name` as argument and will perform dispatch by looking for a function `name.cl` where `cl` is one of the values in the object's `class` attribute. Thus a call to `who(me)` will access the `me` object's class attribute and perform a lookup for each class name until a match is found. If none is found, the function `who.default` is called. This mechanism is complemented by `NextMethod` which calls the *super* method according to the Java terminology. The function follows the same algorithm as `UseMethod`, but starts from the successor of the name used to find the current function. Consider the following example which defines a generic method, `who`, with implementations for class the `man` as well as the `default` case, and creates an object of class `man`. Notice that the vector `me` dynamically acquires class `man` as a side effect. `UseMethod` may take multiple arguments, but dispatches only on the first one.

```
who <-          function(x) UseMethod("who")
who.man <-      function(x) print("Ceasar!")
who.default <- function(x) print("??")
me <- 42;           who(me)   # prints "??"
class(me) <- 'man';who(me)    # prints "Ceasar!"
```

S4. The S4 object model, reminiscent of CLOS [12], adds a class-based system to R. A class is defined by a call to `setClass` with an ordered list of parent classes and a representation. Multiple inheritance is supported, and repeated inheritance is allowed, but only affects the method's dispatch algorithm. A class' representation describes the fields, or slots, introduced by that class. Even though R is dynamically typed, slots must be assigned a class name. Slots can be redefined covariantly, i.e., a slot redefinition can be a subclass of the class tag used in the previous declaration. When a class inherits a slot with the same name from two different paths, the class tag coming from the first superclass is retained, and tags from other parents must be subclasses of that tag. Classes can be redeclared at any time. When a class is modified, existing instances of the class retain their earlier definition. Redeclaration can be prevented by the `sealClass` function. Objects are instantiated by calling `new`. A prototype object is created with the arguments to `new` and passed along to the `initialize` generic function which copies the prototype into the new object's slots. This prototype is only a template and does not contain any methods. Any values left unset become either NA, or a zero-length vector. Classes without a representation, or classes with `VIRTUAL` in their representation, are abstract classes and cannot be instantiated. Another mechanism for changing the behavior of existing classes is to define *class unions*. A class union introduces a new virtual class that is the parent of a list of existing classes. The main role of class union is to change the result of method dispatch for existing classes. The following example code fragment defines a colored point class, creates an instance of the class, and reads its `color` slot.

```
setClass("Point", representation(x="numeric", y="numeric"))
setClass("Color", representation(color="character"))
setClass("CP", contains=c("Point","Color"))
l <- new("Point", x = 1, y = 1)
r <- new("CP", x = 0, y = 0, color = "red")
r@color
```

Methods are introduced at any point outside of any class by a call to `setGeneric`. Individual method bodies for some classes are then defined by `setMethod`. R supports multi-methods [2], i.e., dispatch considers the classes of multiple arguments to determine which function to call. Multi-methods make it trivial to implement binary methods, they obviate the need for the visitor pattern or other forms of double dispatch, and reduce the number of explicit subclass tests in users' code. The following defines an `add` method that will operate differently on points and colored points:

```
setGeneric("add", function(a, b) standardGeneric("add"))
setMethod("add", signature("Point", "Point"),
  function(a, b) new("Point", x= a@x+b@x, y=a@y+b@y))
setMethod("add", signature("CP", "CP"),
  function(a, b) new("CP",x=a@x+b@x,y=a@y+b@y,color=a@color)
```

R does not prevent the declaration of ambiguous multi-methods. At each method call, R will attempt to find the best match between the classes of the parameters and the signatures of method bodies. Thus `add(r, l)` would be treated as the addition of two "Point" objects. The resolution algorithm differs from Clos's and if more than one method is applicable, R will pick one and emit a warning. One unfortunate side effect of combining generic functions and lazy evaluation is that method dispatch forces promises to assess the class of each argument. Thus when S4 objects are used, evaluation of arguments becomes strict.

4 A Semantics for Core R

This section gives a precise semantics to the core of the R language. To the best of our knowledge this is the first formal definition of key concepts of the language. The semantics was derived from test cases and inspection of the source code of version 2.12 of the R interpreter. Core R is a proper subset of the R language. Any expression in Core R behaves identically in the full language. Some features are not covered for brevity: logicals and complex numbers, partial keywords, variadic argument lists, dot-dot symbols, superfluous arguments, generalized array indexing and subsetting. Generalized assignment, `f(x)<-y`, requires small changes to be properly supported, essentially desugaring to function calls such as `` `f<-`(x,y) `` and additional assignments. Perhaps the most glaring simplification is that we left out reflective operation such as `eval` and

$$
\begin{array}{rcl}
\mathtt{e} & ::= & \mathtt{n} \mid \mathtt{s} \mid \mathtt{x} \mid \mathtt{x}[[\mathtt{e}]] \mid \{\mathtt{e};\mathtt{e}\} \\
& \mid & \mathtt{function}(\overline{\mathtt{f}})\ \mathtt{e} \\
& \mid & \mathtt{x}(\overline{\mathtt{a}}) \mid \mathtt{x} <\!\!- \mathtt{e} \mid \mathtt{x} <\!\!<\!\!- \mathtt{e} \\
& \mid & \mathtt{x}[[\mathtt{e}]] <\!\!- \mathtt{e} \mid \mathtt{x}[[\mathtt{e}]] <\!\!<\!\!- \mathtt{e} \\
& \mid & \mathtt{attr}(\mathtt{e},\mathtt{e}) \mid \mathtt{attr}(\mathtt{e},\mathtt{e}) <\!\!- \mathtt{e} \\
& \mid & u \mid \nu(\overline{\mathtt{a}}) \\
\mathtt{f} & ::= & \mathtt{x} \mid \mathtt{x} = \mathtt{e} \\
\mathtt{a} & ::= & \mathtt{e} \mid \mathtt{x} = \mathtt{e}
\end{array}
$$

Fig. 1. Syntax

`substitute`. As the object system is built on those, we will only hint at its definition. The syntax of Core R, shown in Fig. 1, consists of expressions, denoted by $\mathtt{e}$, ranging over numeric literals, string literals, symbols, array accesses, blocks, function declarations, function calls, variable assignments, variable super-assignments, array assignments, array super-assignments, and attribute extraction and assignment. Expressions also include values, u, and partially reduced function calls, $\nu(\overline{\mathtt{a}})$, which are not used in the surface syntax of the language but are needed during evaluation. The parameters of a function declaration, denoted by $\mathtt{f}$, can be either variables or variables with a default value, an expression $\mathtt{e}$. Symmetrical arguments of calls, denoted $\mathtt{a}$, are expressions which may be named by a symbol. We use the notation $\overline{\mathtt{a}}$ to denote the possibly empty sequence $\mathtt{a}_1 \ldots \mathtt{a}_n$. Programs compute over a heap, denoted H, and a stack, S, as shown in Fig. 2. For simplicity, the heap differentiates between three kinds of addresses: frames, ι, promises, δ, and data objects, ν. The notation $H[\iota/F]$ denotes the heap H extended with a mapping from ι to F. The metavariable $\nu_\perp$ denotes ν extended with the distinguished reference $\perp$ which is used for missing values. Metavariable α ranges over pairs of possibly missing addresses, $\nu_\perp\,\nu'_\perp$. The metavariable u ranges over both promises and data references. Data objects, κ^α, consist of a primitive value κ and attributes α. Primitive values can be either an array of numerics, $\mathtt{num}[\mathtt{n}_1 \ldots \mathtt{n}_n]$, an array of strings, $\mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s}_n]$, an array of references $\mathtt{gen}[\nu_1 \ldots \nu_n]$, or a function, $\lambda\overline{\mathtt{f}}.\mathtt{e}, \Gamma$, where Γ is the function's environment. A frame, F, is a mapping from a symbol to a promise or data reference. An environment, Γ, is a sequence of frame references. Finally, a stack, S, is a sequence of pairs, $\mathtt{e}\,\Gamma$, such that $\mathtt{e}$ is the current expression and Γ is the current environment.

$$
\begin{array}{l}
H ::= \emptyset \mid H[\iota/F] \\
\quad \mid H[\delta/\mathtt{e}\,\Gamma] \mid H[\delta/\nu] \\
\quad \mid H[\nu/\kappa^\alpha] \\
\alpha ::= \nu_\perp\,\nu_\perp \quad u ::= \delta \mid \nu \\
\kappa ::= \mathtt{num}[\overline{\mathtt{n}}] \mid \mathtt{str}[\overline{\mathtt{s}}] \\
\quad \mid \mathtt{gen}[\overline{\nu}] \mid \lambda\overline{\mathtt{f}}.\mathtt{e}, \Gamma \\
F ::= [] \mid F[\mathtt{x}/u] \\
\Gamma ::= [] \mid \iota * \Gamma \\
S ::= [] \mid \mathtt{e}\,\Gamma * S
\end{array}
$$

Fig. 2. Data

$$
\frac{\mathtt{e}\,\Gamma; H \;\rightarrow\; \mathtt{e}'; H'}{\mathbb{C}[\mathtt{e}]\,\Gamma * S; H \implies \mathbb{C}[\mathtt{e}']\,\Gamma * S; H'}\;[\textsc{Exp}]
\qquad
\frac{H(\delta) = \mathtt{e}\,\Gamma'}{\mathbb{C}[\delta]\,\Gamma * S; H \implies \mathtt{e}\,\Gamma' * \mathbb{C}[\delta]\,\Gamma * S; H}\;[\textsc{ForceP}]
$$

$$
\frac{\mathsf{getfun}(H, \Gamma, \mathtt{x}) = \delta}{\mathbb{C}[\mathtt{x}(\overline{\mathtt{a}})]\,\Gamma * S; H \implies \delta\,\Gamma * \mathbb{C}[\mathtt{x}(\overline{\mathtt{a}})]\,\Gamma * S; H}\;[\textsc{ForceF}]
\qquad
\frac{\mathsf{getfun}(H, \Gamma, \mathtt{x}) = \nu}{\mathbb{C}[\mathtt{x}(\overline{\mathtt{a}})]\,\Gamma * S; H \implies \mathbb{C}[\nu(\overline{\mathtt{a}})]\,\Gamma * S; H}\;[\textsc{GetF}]
$$

$$
\frac{H(\nu) = \lambda\overline{\mathtt{f}}.\mathtt{e}, \Gamma' \qquad \mathsf{args}(\overline{\mathtt{f}}, \overline{\mathtt{a}}, \Gamma, \Gamma', H) = F, \Gamma'', H'}{\mathbb{C}[\nu(\overline{\mathtt{a}})]\,\Gamma * S; H \implies \mathtt{e}\,\Gamma'' * \mathbb{C}[\nu(\overline{\mathtt{a}})]\,\Gamma * S; H'}\;[\textsc{InvF}]
$$

$$
\frac{H' = H[\delta/\nu]}{\mathbb{R}[\nu]\,\Gamma' * \mathbb{C}[\delta]\,\Gamma * S; H \implies \mathbb{C}[\delta]\,\Gamma * S; H'}\;[\textsc{RetP}]
\qquad
\frac{}{\mathbb{R}[\nu]\,\Gamma' * \mathbb{C}[\nu'(\overline{\mathtt{a}})]\,\Gamma * S; H \implies \mathbb{C}[\nu]\,\Gamma * S; H'}\;[\textsc{RetF}]
$$

Evaluation Contexts:

$$
\begin{array}{l}
\mathbb{C} ::= [] \mid \mathtt{x} \mathtt{<-}\, \mathbb{C} \mid \mathtt{x}[[\mathbb{C}]] \mid \mathtt{x}[[\mathtt{e}]] \mathtt{<-}\, \mathbb{C} \mid \mathtt{x}[[\mathbb{C}]] \mathtt{<-}\, \nu \mid \{\mathbb{C}; \mathtt{e}\} \mid \{\nu; \mathbb{C}\} \\
\quad \mid \mathtt{attr}(\mathbb{C}, \mathtt{e}) \mid \mathtt{attr}(\nu, \mathbb{C}) \mid \mathtt{attr}(\mathtt{e}, \mathtt{e}) \mathtt{<-}\, \mathbb{C} \mid \mathtt{attr}(\mathbb{C}, \mathtt{e}) \mathtt{<-}\, \nu \mid \mathtt{attr}(\nu, \mathbb{C}) \mathtt{<-}\, \nu \\
\mathbb{R} ::= [] \mid \{\nu; \mathbb{R}\}
\end{array}
$$

Fig. 3. Reduction relation $\implies$.

Reduction relation. The semantics of Core R is defined by a small step operational semantics with evaluation contexts [21]. The reduction relation $S;H \Longrightarrow S';H'$, shown in Fig. 3, takes a stack S and a heap H and performs one step of reduction. The rules rely on two evaluation contexts, $\mathbb{C}$, to return the next expression to evaluate and $\mathbb{R}$, to return the result of a sequence of expressions. There are seven reduction rules. Rule [EXP] deals with expressions, where $\mathbb{C}[\mathsf{e}]$ uniquely identifies the next expression e to evaluate. The expression is reduced in a single step, $\mathsf{e}\,\Gamma; H \;\rightarrow\; \mathsf{e}'; H'$, where e' is resulting expression. H' is the modified heap. If the expression is a promise, $\mathbb{C}[\delta]$, and δ has not been evaluated, rule [FORCEP] will push a new frame on the stack containing the body of the promise, $\mathsf{e}\;\delta * \Gamma'$. Rule [RETP] pops a fully evaluated promise frame and binds the result to a promise address. Context sensitive lookup is implemented by [FORCEF] and [GETF]. The former forces the evaluation of promises bound to the name of the function being looked up, the latter selects a reference, ν, to a function. The $\mathsf{getfun}()$ auxiliary function, defined in Fig. 4, looks up $\mathtt{x}$ in the environment, skipping over bindings to data objects. Function invocation is handled by [INVF], which retrieves the function bound to ν and invokes $\mathsf{args}()$ to process the arguments $\overline{\mathsf{a}}$ and the default values $\overline{\mathtt{f}}$ of the call. The output of $\mathsf{args}()$ is a mapping from parameters to values, F, an environment, Γ'', and a modified heap, H'. For each argument, a promise is allocated in the heap and the current environment is captured. The rule [RETF] simply pops the evaluated frame and replaces the call with its result.

$$\frac{\Gamma = \iota * \Gamma' \quad \iota(H,\mathtt{x}) = \nu \quad H(\nu) = \lambda\overline{\mathtt{f}}.\mathsf{e},\Gamma''}{\mathsf{getfun}(H,\Gamma,\mathtt{x}) = \nu}\;[\textsc{GetF1}] \qquad \frac{\Gamma = \iota * \Gamma' \quad \iota(H,\mathtt{x}) = \nu \quad H(\nu) \neq \lambda\overline{\mathtt{f}}.\mathsf{e},\Gamma''}{\mathsf{getfun}(H,\Gamma,\mathtt{x}) = \mathsf{getfun}(H,\Gamma',\mathtt{x})}\;[\textsc{GetF2}]$$

$$\frac{\Gamma = \iota * \Gamma' \quad \iota(H,\mathtt{x}) = \delta \quad H(\delta) = \nu \quad H(\nu) = \lambda\overline{\mathtt{f}}.\mathsf{e},\Gamma''}{\mathsf{getfun}(H,\Gamma,\mathtt{x}) = \nu}\;[\textsc{GetF3}] \qquad \frac{\Gamma = \iota * \Gamma' \quad \iota(H,\mathtt{x}) = \delta \quad H(\delta) = \mathsf{e}\,\Gamma''}{\mathsf{getfun}(H,\Gamma,\mathtt{x}) = \delta}\;[\textsc{GetF4}]$$

$$\frac{\Gamma = \iota * \Gamma' \quad \iota(H,\mathtt{x}) = \delta \quad H(\delta) = \nu \quad H(\nu) \neq \lambda\overline{\mathtt{f}}.\mathsf{e},\Gamma''}{\mathsf{getfun}(H,\Gamma,\mathtt{x}) = \mathsf{getfun}(H,\Gamma',\mathtt{x})}\;[\textsc{GetF5}]$$

$$\frac{\mathsf{split}(\overline{\mathsf{a}},P,N) = P',N'}{\mathsf{split}(\mathtt{x} = \mathsf{e}\,\overline{\mathsf{a}},P,N) = P',\mathtt{x} = \mathsf{e}N'}\;[\textsc{Split1}] \qquad \frac{\mathsf{split}(\overline{\mathsf{a}},P,N) = P',N'}{\mathsf{split}(\mathsf{e}\,\overline{\mathsf{a}},P,N) = \mathsf{e}P',N'}\;[\textsc{Split2}] \qquad \frac{}{\mathsf{split}([],P,N) = P,N}\;[\textsc{Split3}]$$

$$\frac{\mathsf{split}(\overline{\mathsf{a}},[],[]) = P,N \quad \iota\,\mathtt{fresh} \quad \Gamma'' = \iota * \Gamma' \quad \mathsf{args2}(\overline{\mathtt{f}},P,N,\Gamma,\Gamma'',H) = F,H' \quad H'' = H'[\iota/F]}{\mathsf{args}(\overline{\mathtt{f}},\overline{\mathsf{a}},\Gamma,\Gamma',H) = F,\Gamma'',H''}\;[\textsc{Args}]$$

$$\frac{(\mathtt{f}_0 \equiv \mathtt{x} \vee \mathtt{f}_0 \equiv \mathtt{x} = \mathsf{e}') \quad N \equiv N'\mathtt{x} = \mathsf{e}N'' \quad \mathsf{args2}(\overline{\mathtt{f}},P,N'N'',\Gamma,\Gamma',H) = F,H' \quad \delta\,\mathtt{fresh} \quad H'' = H'[\delta/\mathsf{e}\,\Gamma]}{\mathsf{args2}(\mathtt{f}_0\overline{\mathtt{f}},P,N,\Gamma,\Gamma',H) = F[\mathtt{x}/\delta],H''}\;[\textsc{Args1}]$$

$$\frac{(\mathtt{f}_0 \equiv \mathtt{x} \vee \mathtt{f}_0 \equiv \mathtt{x} = \mathsf{e}') \quad \mathtt{x} \notin N \quad \mathsf{args2}(\overline{\mathtt{f}},P,N,\Gamma,\Gamma',H) = F,H' \quad \delta\,\mathtt{fresh} \quad H'' = H'[\delta/\mathsf{e}\,\Gamma]}{\mathsf{args2}(\mathtt{f}_0\overline{\mathtt{f}},\mathsf{e}P,N,\Gamma,\Gamma',H) = F[\mathtt{x}/\delta],H''}\;[\textsc{Args2}]$$

$$\frac{\mathtt{x} \notin N \quad \mathsf{args2}(\overline{\mathtt{f}},[],N,\Gamma,\Gamma',H) = F,H'}{\mathsf{args2}(\mathtt{x}\,\overline{\mathtt{f}},[],N,\Gamma,\Gamma',H) = F[\mathtt{x}/\bot],H'}\;[\textsc{Args3}]$$

$$\frac{\mathtt{x} \notin N \quad \mathsf{args2}(\overline{\mathtt{f}},[],N,\Gamma,\Gamma',H) = F,H' \quad \delta\,\mathtt{fresh} \quad H'' = H'[\delta/\mathsf{e}\,\Gamma']}{\mathsf{args2}(\mathtt{x} = \mathsf{e}\,\overline{\mathtt{f}},[],N,\Gamma,\Gamma',H) = F[\mathtt{x}/\delta],H''}\;[\textsc{Args4}]$$

$$\frac{}{\mathsf{args2}([],[],[],\Gamma,\Gamma',H) = [],H}\;[\textsc{Args5}]$$

Fig. 4. Auxiliary definitions: Function lookup and argument processing.

The $\rightarrow$ relation has fourteen rules dealing with expressions, shown in Fig. 5, along with some auxiliary definitions given in Fig. 18 (where s and g denote functions that convert the type of their argument to a string and vector respectively). The first two rules deal with numeric and string literals. They simply allocate a vector of length one of the corresponding type with the specified value in it. By default, attributes for these values are empty. A function declaration, [FUN], allocates a closure in the heap and

$$\frac{\nu\,\texttt{fresh} \quad \alpha = \bot\,\bot \quad H' = H[\nu/\texttt{num}[\texttt{n}]^{\alpha}]}{\texttt{n}\,\Gamma; H \;\rightarrow\; \nu; H'}\;[\textsc{Num}] \qquad \frac{\nu\,\texttt{fresh} \quad \alpha = \bot\,\bot \quad H' = H[\nu/\texttt{str}[\texttt{s}]^{\alpha}]}{\texttt{s}\,\Gamma; H \;\rightarrow\; \nu; H'}\;[\textsc{Str}]$$

$$\frac{\nu\,\texttt{fresh} \quad \alpha = \bot\,\bot \quad H' = H[\nu/\lambda\overline{\texttt{f}}.\texttt{e}, \Gamma^{\alpha}]}{\texttt{function}(\overline{\texttt{f}})\;\texttt{e}\,\Gamma; H \;\rightarrow\; \nu; H'}\;[\textsc{Fun}]$$

$$\frac{\Gamma(H, \texttt{x}) = u}{\texttt{x}\,\Gamma; H \;\rightarrow\; u; H}\;[\textsc{Find}] \qquad \frac{H(\delta) = \nu}{\delta\,\Gamma; H \;\rightarrow\; \nu; H'}\;[\textsc{GetP}]$$

$$\frac{\mathsf{cpy}(H, \nu) = H', \nu' \quad \Gamma = \iota * \Gamma' \quad H(\iota) = F \quad F' = F[\texttt{x}/\nu'] \quad H'' = H'[\iota/F']}{\texttt{x} <\!\!- \nu\,\Gamma; H \;\rightarrow\; \nu; H''}\;[\textsc{Ass}]$$

$$\frac{\mathsf{cpy}(H, \nu) = H', \nu' \quad \Gamma = \iota * \Gamma' \quad \mathsf{assign}(\texttt{x}, \nu', \Gamma', H') = H''}{\texttt{x} <\!\!<\!\!- \nu\,\Gamma; H \;\rightarrow\; \nu; H''}\;[\textsc{DAss}]$$

$$\frac{\Gamma(H, \texttt{x}) = \nu' \quad \mathsf{readn}(\nu, H) = \texttt{m} \quad \mathsf{get}(\nu', \texttt{m}, H) = \nu'', H'}{\texttt{x}[[\nu]]\,\Gamma; H \;\rightarrow\; \nu''; H'}\;[\textsc{Get}]$$

$$\frac{\mathsf{cpy}(H, \nu') = H', \nu'' \quad \Gamma = \iota * \Gamma' \quad \iota(H', \texttt{x}) = \nu''' \quad \mathsf{readn}(\nu, H') = \texttt{m} \quad \mathsf{set}(\nu''', \texttt{m}, \nu'', H') = H''}{\texttt{x}[[\nu]] <\!\!- \nu'\,\Gamma; H \;\rightarrow\; \nu'; H''}\;[\textsc{SetL}]$$

$$\frac{\mathsf{cpy}(H, \nu') = H', \nu'' \quad \Gamma = \iota * \Gamma' \quad H'(\iota) = F \quad \texttt{x} \notin F \quad \Gamma'(H', \texttt{x}) = \nu''' \quad \mathsf{cpy}(H', \nu''') = H'', \nu'''' \quad F' = F[\texttt{x}/\nu''''] \quad H''' = H''[\iota/F'] \quad \mathsf{readn}(\nu, H) = \texttt{m} \quad \mathsf{set}(\nu'''', \texttt{m}, \nu'', H''') = H''''}{\texttt{x}[[\nu]] <\!\!- \nu'\,\Gamma; H \;\rightarrow\; \nu'; H''''}\;[\textsc{SetG}]$$

$$\frac{H(\nu) = \kappa^{\alpha} \quad \alpha = \nu_{\bot}\,\nu'_{\bot} \quad \mathsf{index}(\nu', \nu'_{\bot}, H) = \texttt{n} \quad \mathsf{get}(\nu_{\bot}, \texttt{n}, H) = \nu''}{\texttt{attr}(\nu, \nu')\,\Gamma; H \;\rightarrow\; \nu''; H}\;[\textsc{GetA}]$$

$$\frac{H(\nu) = \kappa^{\alpha} \quad \alpha = \nu_{\bot}\,\nu'_{\bot} \quad \mathsf{index}(\nu', \nu'_{\bot}, H) = \texttt{n} \quad \mathsf{set}(\nu, \texttt{n}, \nu'', H) = H'}{\texttt{attr}(\nu, \nu') <\!\!- \nu''\,\Gamma; H \;\rightarrow\; \nu''; H'}\;[\textsc{ReplA}]$$

$$\frac{\mathsf{cpy}(H, \nu'') = H', \nu''' \quad H'(\nu) = \kappa^{\nu_{\bot}\,\nu'_{\bot}} \quad \mathsf{index}(\nu', \nu'_{\bot}, H') = \bot \quad \mathsf{reads}(\nu', H') = \texttt{s} \quad H'(\nu_{\bot}) = \texttt{gen}[\overline{\nu}]^{\alpha} \quad H'(\nu'_{\bot}) = \texttt{str}[\overline{\texttt{s}}]^{\alpha'} \quad H'' = H'[\nu_{\bot}/\texttt{gen}[\overline{\nu}\nu''']^{\alpha}][\nu'_{\bot}/\texttt{str}[\overline{\texttt{s}}\texttt{s}]^{\alpha'}]}{\texttt{attr}(\nu, \nu') <\!\!- \nu''\,\Gamma; H \;\rightarrow\; \nu''; H''}\;[\textsc{SetA}]$$

$$\frac{\mathsf{cpy}(H, \nu'') = H', \nu^3 \quad H'(\nu) = \kappa^{\bot\,\bot} \quad \nu^4, \nu^5\,\texttt{fresh} \quad \mathsf{reads}(\nu', H') = \texttt{s} \quad H'' = H'[\nu^4/\texttt{gen}[\nu^3]^{\bot\,\bot}][\nu^5/\texttt{str}[\texttt{s}]^{\bot\,\bot}]}{\texttt{attr}(\nu, \nu') <\!\!- \nu''\,\Gamma; H \;\rightarrow\; \nu''; H''}\;[\textsc{SetB}]$$

Fig. 5. Reduction relation $\rightarrow$.

captures the current environment Γ. Variable lookup, [FIND], returns the value of the variable from the environment. The value of an already evaluated promise is returned by [GETP]. The assignment, [ASS], and super-assignment, [DASS], rules will either define or redefine the target symbol. The value being assigned and all of its attributes are copied recursively. The auxiliary function assign walks the stack and performs the assignment in the first environment that has a binding for the target symbol. If not found, the symbol is added at the top-level. The [GET] rule for array access, $\mathtt{x}[[\nu]]$, is straightforward, it accesses the array at the offset passed as argument. Note that the value returned must be packed in a newly allocated vector of length one of the right type. There are two rules for vector assignment $\mathtt{x}[[\nu]] <- \nu'$. Rule [SETL] applies when the vector is a local variable of the current frame. In that case, the value to be assigned is copied and the assignment is performed in place. Rule [SETG] is more complex. If the variable holding the vector does not occur in the current scope, a new variable will be added to the current scope, the vector is copied with its attributes into the new variable, and finally the assignment is performed.[6] Notice also that all assignment rules yield the right hand side value and not its copy. Finally, there are four rules dealing with attributes. Reading an attribute, $\mathtt{attr}(\nu, \nu')$, uses ν' as a key to find the corresponding value in the attribute vector ([GETA]).[7] The auxiliary function index() returns the index of a string in a vector of strings or $\perp$ if not found. The rules for updating attributes, $\mathtt{attr}(\nu, \nu') <- \nu''$, must consider the two cases. First, when an attribute already exists, the update is done directly ([REPLA]). Second, when an attribute is not present, then the value and name sequences must grow to accommodate the new attribute ([SETA]). Finally, if the attributes are empty, rule [SETB] will create them. It is noteworthy that attributes are modified in place; the objects that they decorate are not copied.

Observations. One of our discoveries while working out the semantics was how eager evaluation of promises turns out to be. The semantics captures this with $\mathbb{C}[]$; the only cases where promises are not evaluated is in the arguments of a function call and when promises occur in a nested function body, all other references to promises are evaluated. In particular, it was surprising and unnecessary to force assignments as this hampers building infinite structures. Many basic functions that are lazy in Haskell, for example, are strict in R, including data type constructors. As for sharing, the semantics cleary demonstrates that R prevents sharing by performing copies at assignments. The R implementation uses copy-on-write to reduce the number of copies. With super-assignment, environments can be used as shared mutable data structures. The way assignment into vectors preserves the pass-by-value semantics is rather unusual and, from personal experience, it is unclear if programmers understand the feature. Extending the semantics to supporting reflection and objects should be possible. Objects are encoded by vectors with attributes that hold their class, as a vector of strings, fields. Methods are functions that abide by a particular naming convention. Dispatch is done by reflecting over defined functions. It is noteworthy that objects are mutable within a function (since fields are attributes), but are copied when passed as an argument.

[6] Our semantics only allows extension of vector at the end. R allows vector to be extended at arbitrary offsets, with missing values added in unused positions.

[7] In R, attributes are represented by a normal vector (values) which, itself, has attributes (names). We simplify the structure for conciseness in the semantics.

5 Corpus Analysis

Given the mix of programming models available to the R user, it is important to understand what features users favor and how they are using those in practice. This section describes the tools we have developed to analyze R programs and the extensive corpus of R programs that we have curated.

5.1 The TraceR Framework

TraceR is a suite of tools for analyzing the performance and characteristics of R code. It consists of three data collection tools built on top of version 2.12.1 of R and several post-processing tools. *TrackeR* generates detailed execution traces, *ProfileR* is a low-overhead profiling tool for the internals of the R VM, and *ParseR* is static analyzer for R code.

TrackeR. To precisely capture user-code behavior, we built TrackeR, a heavily instrumented R VM which records almost every operation executed at runtime. TrackeR's design was informed by our previous work on JavaScript [17]. TrackeR exposes interactions between language features, such as evaluation of promises triggered by function lookups, and how these features are used. It also records promise creation and evaluation, scalar and vector usage, and internally triggered actions (e.g. duplications used for copy-on-write mechanisms). These internal effects are recorded through a mix of trace events and counters. Complex feature interactions such as lazy evaluation and multi-method dispath can result in eager argument evaluation. To capture the triggers for this behavior, prologues are emitted for function calls and associated with the triggering method. Properly tracking the uniqueness of short lived objects, like promises, is complicated by the recycling memory of addresses during garbage collection. R's memory allocations are too large and numerous to use memory maps to resolve this. Instead, a tagging system was used to track the liveness of traced objects. Since, at runtime, function objects are represented as closure with no name, we use R built-in debugging information to map closure addresses to source code. Moreover, control flow can jump between various parts of the call stack when executions are abandoned (e.g. with `tryCatch` or `break` function calls). Keeping the trace consistent requires effort since the implementation of the VM is riddled with calls to `longjmp`. Off-line analysis of traces can quickly exceed machine memory if they are analyzed in-core. Therefore, the tree is processed during its construction and most of it is discarded right away. Specialized trace filters use hooks to register information of interest (e.g. promises currently alive in the system).

ProfileR. While TrackeR reveals program evaluation flow and effects, its heavy instrumentation makes it unsuitable for understanding the runtime costs of language features. For this we built ProfileR, a dedicated counter based profiler which tracks the time costs of operations such as memory management, I/O and foreign calls. Unlike a sampling profiler, ProfileR is precise. It was implemented with care to minimize runtime overheads. The validity of its results was verified against sampling profilers such as oprofile and Apple Instruments. The results are consistent with those tools, and provide more accurate context information. The only notable differences are for very short functions called very frequently, which we avoided instrumenting. R also has a built-in sampling profiler but we found that it did not deliver the accuracy or level of detail we needed.

ParseR. Tracing only yields information on code triggered in a given execution. For a more comprehensive view, ParseR performs static analysis of R programs. It is built on a LL-parser generated with AntLR [14]. Our R grammar seems comprehensive as it parses correctly all R code we could find. Lexical filters can be easily written by using a mixture of tree grammars and visitors. Even though ParseR can easily find accurate grammatical patterns, the high dynamism of R forced us to rely on heuristics when looking for semantic information. ParserR was also used to synchronize the traces generated by TrackeR with actual source code of the programs.

5.2 A Corpus of R Code

We assembled a body of over 3.9 million lines of R code. This corpus is intended to be representative of real-world R usage, but also to help understand the performance impacts of different language features. We classified programs in 5 groups. The *Bioconductor* project open-source repository collects 515 Bioinformatics-related R packages.[8] The *Shootout* benchmarks are simple programs from the Computer Language Benchmark Game[9] implemented in many languages that can be used to get a performance baseline. Some R users donated their code; these programs are grouped under the *Miscellaneous* category. The fourth and largest group of programs was retrieved from the R package archive on CRAN.[10] The last group is the base library that is bundled with the R VM. Fig. 6 gives the size of these datasets.

Name	Bioc.	Shoot.	Misc.	CRAN	Base
# Package	515	11	7	1 238	27
# Vignettes	100	11	4	–	–
R LOC	1.4M	973	1.3K	2.3M	91K
C LOC	2M	0	0	2.9M	50K

Fig. 6. Purdue R Corpus.

A requirement of all packages in the Bioconductor repository is the inclusion of vignettes. Vignettes are scripts that demonstrate real-world usage of these libraries to potential users. Vignettes also double as simple tests for the programs. They typically come with sample data sets. Out of the 515 Bioconductor programs, we focused on the 100 packages with the longest running vignettes. Some CRAN packages do not have vignettes; this is unfortunate as it makes them harder to analyze. We retained 1 238 out of 3 495 available CRAN packages. It should be noted that while some of the data associated to vignettes are large, they are in general short running.

The Shootout benchmarks were not available in R, so we implemented them to the best of our abilities. They provide tasks that are purely algorithmic, deterministic, and computationally focused. Further, they are designed to easily scale in either memory or computation. For a fair comparison, the Shootout benchmarks stick to the original algorithm. Two out of the 14 Shootout benchmarks were not used because they required multi-threading and one because it relied on highly tuned low-level libraries. We restricted our implementations to standard R features. The only exception is the `knucleotide` problem, where environments served as a substitute for hash maps.

[8] `http://www.bioconductor.org`
[9] `http://shootout.alioth.debian.org/`
[10] `http://cran.r-project.org/`

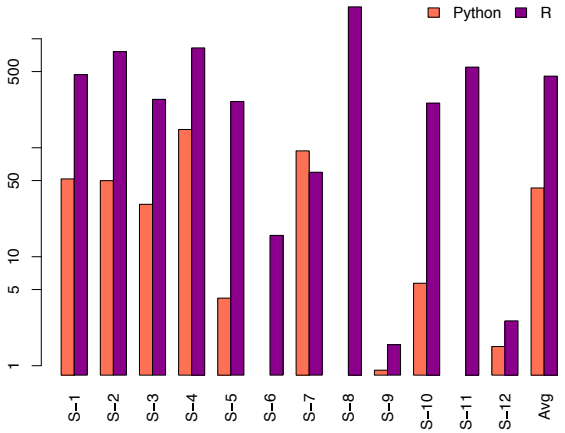
6 Evaluating the R Implementation

Using ProfileR and TraceR, we get an overview of performance bottlenecks in the current implementation in terms of execution time and memory footprint. To give a relative sense of performance, each diagnostic starts with a comparison between R, C and Python using the shootout benchmarks. Beyond this, we used Bioconductor vignettes to understand the memory and time impacts in R's typical usage.

All measurements were made on an 8 core Intel X5460 machine, running at 3.16GHz with the GNU/Linux 2.6.34.8-68 (x86_64) kernel. Version 2.12.1 of R compiled with GCC v4.4.5 was used as a baseline R, and as the base for our tools. The same compiler was used for compiling C programs, and finally Python v2.6.4 was used. During benchmark comparisons and profiling executions, processes were attached to a single core where other processes were denied. Any other machine usage was prohibited.

6.1 Time

We used the Shootout benchmarks to compare the performance of C, Python and R. Results appear in Fig. 7. On those benchmarks, R is on average 501 slower than C and 43 times slower Python. Benchmarks where R performs better, like `regex-dna` (only 1.6 slower than C), are usually cases where R delegates most of its work to C functions.[11]



	Name	Input
S-1	Binary trees	16
S-2	Fankuch redux	10
S-3	Fasta	2.5M
S-4	Fasta redux	2.5M
S-5	K-nucleotide	50K
S-6	Mandelbrot	4K
S-7	N-body	500K
S-8	Pidigits	500
S-9	Regex-dna	2.5K
S-10	Rev. complement	5M
S-11	Spectral norm	640
S-12	Spectral norm alt	11K

Fig. 7. Slowdown of Python and R, normalized to C for the Shootout benchmarks.

To understand where time is typically spent, we turn to more representative R programs. Fig. 8 shows the breakdown of execution times in the Bioconductor dataset obtained with ProfileR. Each bar represents a Bioconductor vignette. The key observation is that memory management accounts for an average of 29% of execution time.

[11] For C and Python implementations, we kept the fastest single-threaded implementations. When one was not available, we removed multi-threading from the fastest one. The `pidigits` problem required a rewrite of the C implementation to match the algorithm of the R implementation since the R standard library lacks big integers.

Memory management breaks down into time spent in garbage collection (18%), allocating cons-pairs (3.6%), vectors (2.6%), and duplications (4%) for call-by-value semantics. Built-in functions are where the true computational work happens, and on average 38% of the execution time. There are some interesting outliers. The maximum spent in garbage collection is 70% and one program spends 63% copying arguments. Lookup (4.3% and match 1.8%) represent time spent looking up variables and matching parameters with arguments. Both of these would be absent in Java as they are resolved at compile time. Variable lookup would also be absent in Lisp or Scheme as, once bound, the position of variables in a frame are known. Given the nature of R, many numerical functions are written in C or Fortran; one could thus expect execution time to be dominated by native libraries. The time spent in calls to foreign functions, on average 22%, shows that this is clearly not the case.

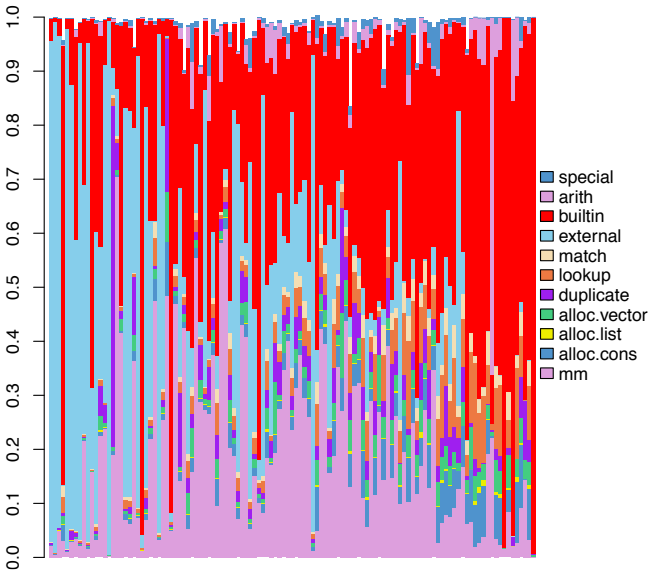


Fig. 8. Breakdown of Bioconductor vignette runtimes as % of total execution time.

6.2 Memory

Not only is R slow, but it also consumes significant amounts of memory. Unlike C, where data can be stack allocated, all user data in R must be heap allocated and garbage collected. Fig. 9 compares heap memory usage in C (calls to malloc) and data allocated by the R virtual machine. The R allocation is split between vectors (which are typically user data) and lists (which are mostly used by the interpreter for, e.g., arguments to functions). The graph clearly shows that R allocates orders of magnitude

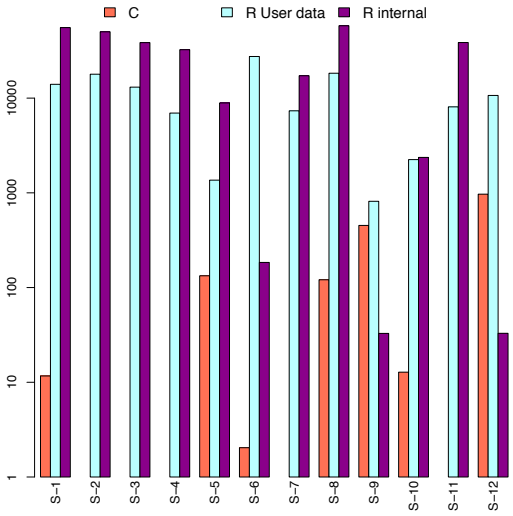


Fig. 9. Heap allocated memory (MB log scale). C vs. R.

more data than C. In many cases the internal data required is more than the user data. Call-by-value semantics is implemented by a copy-on-write mechanism. Thus, under the covers, function arguments are shared and duplicated when needed. Avoiding duplication reduces memory footprint; on average only 37% of arguments end up being copied. Lists are created by `pairlist` and mostly used by the R VM. In fact, the standard library only has three calls to `pairlist`, the whole CRAN code only eight, and Bioconductor none. The R VM uses them to represent code and to pass and process function call arguments. It is interesting to note that the time spent on allocating lists is greater than the time spent on vectors. Cons cells are large, using 56 bytes on 64-bit architectures, and take up 23 GB on average in the Shootout benchmarks.

Another reason for the large footprint, is that all numeric data has to be boxed into a vector; yet, 36% of vectors allocated by Bioconductor contain only a single number. An empty vector is 40 bytes long. This impacts runtime, since these vectors have to be dereferenced, allocated and garbage collected.

Observations. R is clearly slow and memory inefficient. Much more so than other dynamic languages. This is largely due to the combination of language features (call-by-value, extreme dynamism, lazy evaluation) and the lack of efficient built-in types. We believe that with some effort it should be possible to improve both time and space usage, but this would likely require a full rewrite of the implementation.

7 Evaluating the R Language Design

One of the key claims made repeatedly by R users is that they are more productive with R than with traditional languages. While we have no direct evidence, we will point out that, as shown by Fig. 10, R programs are about 40% smaller than C code. Python is even more compact on those shootout benchmarks, at least in part, because many

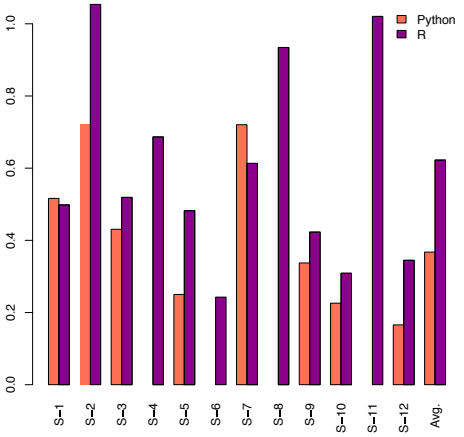


Fig. 10. Shootout Python and R code size, normalized to C.

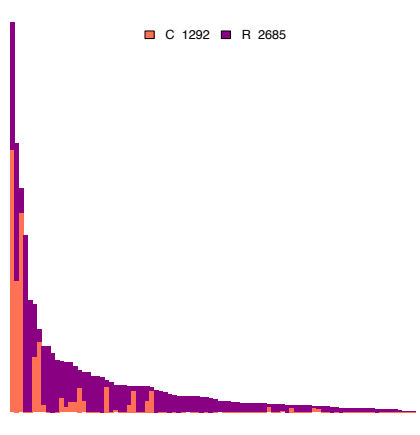


Fig. 11. Bioconductor R and C source code size. (LOC, no comments)

of the shootout problems are not easily expressed in R. We do not have any statistical analysis code written in Python and R, so a more meaningful comparison is difficult. Fig. 11 shows the breakdown between code written in R and code in Fortran or C in 100 Bioconductor packages. On average, there is over twice as much R code. This is significant as package developers are surely savvy enough to write native code, and understand the performance penalty of R, yet they would still rather write code in R.

7.1 Functional

Side effects. Assignments can either define or update variables. In Bioconductor, 45% of them are definitions, and only two out of 217 million assignments are definitions in a parent frame by super assignment. In spite of the availability of non-local side effects (i.e., `<<-`), 99.9% of side effects are local. Assignments done through functions such as `[]<-` need an existing data structure to operate on, thus they are always side effecting. Overall they account for 22% of all side effects and 12% of all assignments.

Scoping. R symbol lookup is context sensitive. This feature, which is neither Lisp nor Scheme scoping, is exercised in less than 0.05% of function name lookups. However, even though this number is low, the number of symbols actually checked is 3.6 on average. The only symbols for which this feature actually mattered in the Bioconductor vignettes are `c` and `file`, both popular variables names and built-in functions.

Parameters. The R function declaration syntax is expressive and this expressivity is widely used. In 99% of the calls, at most 3 arguments are passed, while the percentage of calls with up to 7 arguments is 99.74% (see Fig. 12). Functions that are close to this average are typically called with positional arguments. As the number of parameters increases, users are more likely to specify function parameters by name. Similarly, variadic parameters tend to be called with large numbers of arguments.

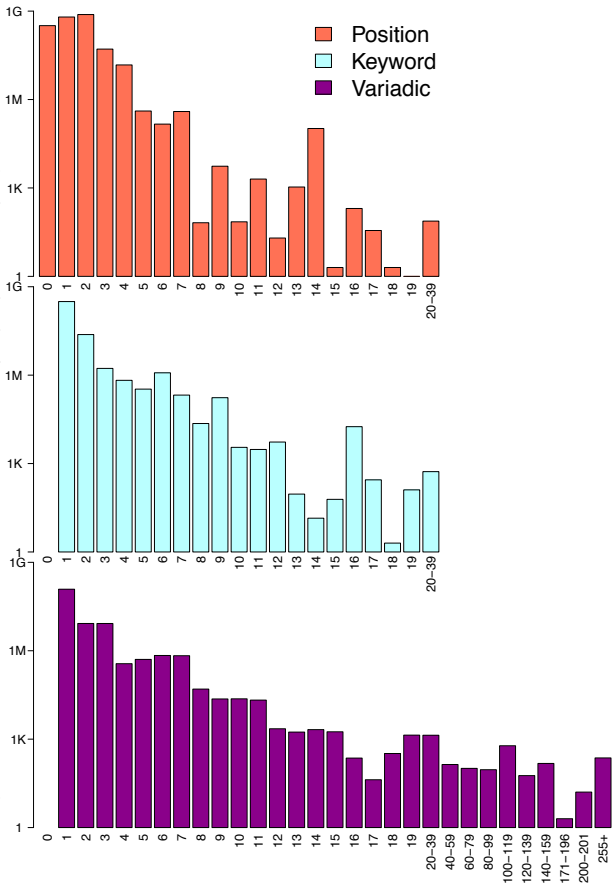


Fig. 12. Histogram of the number of function arguments in Bioconductor. (Log scale)

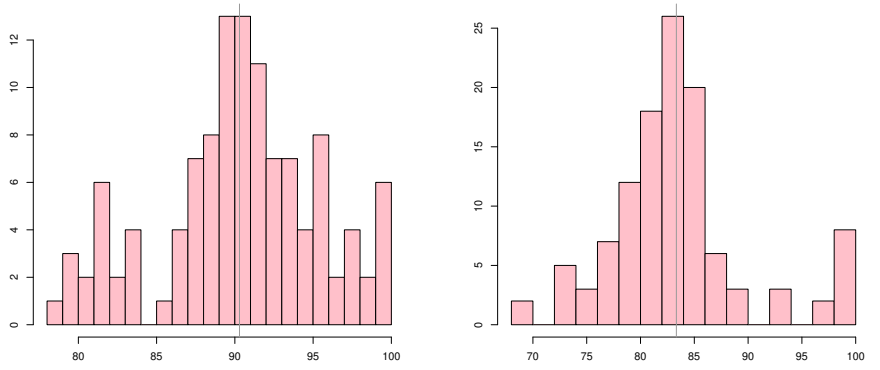
	Bioc		Shootout		Misc		CRAN	Base
	stat.	dyn.	stat.	dyn.	stat.	dyn.	stat.	stat.
Calls	1M	3.3M	657	2.6G	1.5K	10.0G	1.7G	71K
by keyword	197K	72M	67	10M	441	260M	294K	10K
# keywords	406K	93M	81	15M	910	274M	667K	18K
by position	1.0M	385M	628	143M	1K	935M	1.6G	67K
# positional	2.3M	6.5G	1K	5.2G	3K	18.7G	3.5G	125K

Fig. 13. Number of calls by category.

Fig. 13 gives the number of calls in our corpus and the total number of keyword and variadic arguments. Positional arguments are most common between 1 and 4 arguments, but are used all the way up to 25 arguments. Function calls with between 1 and 22 named arguments have been observed. Variadic parameters are used to pass from 1 to more than 255 arguments. Given the performance costs of parsing parameter lists in the current implementation, it appears that optimizing calling conventions for function of four parameters or less would greatly improve performance. Another interesting consequence of the prevalent use of named parameters is that they become part of the interface of the function, so alpha conversion of parameter names may affect the behavior of the program.

Laziness. Lazy evaluation is a distinctive feature of R that has the potential for reducing unnecessary work performed by a computation. Our corpus, however, does not bear this out. Fig. 14(a) shows the rate of promise evaluation across all of our data sets. The average rate is 90%. Fig. 14(b) shows that on average 80% of promises are evaluated in the first function they are passed into. In computationally intensive benchmarks the rate of promise evaluation easily reaches 99%. In our own coding, whenever we encountered higher rates of unevaluated promises, finding where this occurred and refactoring the code to avoid those promises led to performance improvements.

Promises have a cost even when not evaluated. Their cost in in memory is the same as a pairlist cell, i.e., 56 bytes on a 64-bit architecture. On average, a program allocates 18GB for them, thus increasing pressure on the garbage collector. The time cost of promises is roughly one allocation, a handful of writes to memory. Moreover, it is a data type which has to be dispatched and tested to know if the content was already evaluated.



(a) Promises evaluated (in %) (b) Promises evaluated on same level (in %)

Fig. 14. Promises evaluation in all data sets. The y-axis is the number of programs.

Finally, this extra indirection increases the chance of cache misses. An example of how unevaluated promises arise in R code is the `assign` function, which is heavily used in Bioconductor with 23 million calls and 46 million unevaluated promises.

```
function(x,val,pos=-1,env=as.environment(pos),immediate=TRUE)
  .Internal(assign(x,val,env))
```

This function definition is interesting because of its use of dependent parameters. The body of the function never refers to `pos`, it is only used to modify the default value of `env` if that parameter is not specified in the call. Less than 0.2% of calls to `assign` evaluate the promise associated with `pos`.

It is reasonable to ask if it would be valid to simply evaluate all promises eagerly. The answer is unfortunately no. Promises that are passed into a function which provides a language extension may need special processing. For instance in a loop, promises are intended to be evaluated multiple times in an environment set up with the right variables. Evaluating those eagerly will result in a wrong behavior. However, we have not seen any evidence of promises being used to extend the language outside of the base libraries. We infer this from calls to the `substitute` and `assimilate` functions. Another possible reason for not switching the evaluation strategy is that promises perform and observe side effects.

```
x <- y <- 0
fun <- function(a, b) if(runif(1)>.5) a+b else b+a
fun(x<-y+1, y<-x+2) # Result is always a+b, but can be either 4 or 5
```

This code snippet will yield different results depending on the order the two promises passed to `fun` are going to be evaluated. Taking into account the various oddities of R, such as lookups that force evaluation of all promises in scope, it is reasonable to wonder if relying on a particular evaluation order is a wise choice for programmers.

7.2 Dynamic

Eval. The `eval` function is widely used in R code with 8 500 static calls in CRAN and 5 800 calls in Bioconductor. The total number of dynamic calls in our benchmarks was 2.2 million. These are rather large numbers. We focus on the 15 call sites where each represents more than 1% of the total dynamic calls. Together these call sites account for 88% of `eval`. The `match.arg` function is the highest user with 54% of all calls to `eval`. In the 14 other call sites to `eval`, we see two uses cases. The most common is the evaluation of the source code of a promise retrieved by `substitute` in a new environment. This is done in the `with` function. The other use case is the invocation of a function whose name or arguments are determined dynamically. For this purpose, R provides `do.call`, and thus using `eval` is overkill.

Substitute. Promises provide a kind of limited automatic quoting of arguments as the `substitute` function can retrieve the textual representation of the source expression of any promise. A typical use case is to add a legend to a chart when no text is provided; this is done by retrieving the expression passed to the plot function and using it as a legend. However, this usage is limited to one level of nesting, and passing a promise to another function will destroy that information.

```
f <- function(x) substitute(x)
b <- function(x) (function(y) substitute(y))(x)
f(1 + 1) # 1 + 1
b(1 + 1) # x
```

The example above shows that `substitute` only retrieves the text of the last argument. In Bioconductor, `substitute` is called from 51 call sites 3.6 million times, but only 11 call sites are in user code (the rest comes from the standard library). They account for 2% of dynamic calls.

match.arg. The `match.arg` function takes arguments `arg` and `choices`; it matches `arg` against a table of candidate values as specified by `choices`. In practice, 75% of the calls to this function only pass the first argument, like the following code snippet:

```
magic <- function(type=c("mean","median","trimmed"))
            return (match.arg(type))
```

A call to `magic("t")` returns `trimmed`. Notice that the string `trimmed` only occurs in the default argument which is not used in this case as a value of `"t"` is provided. What happens is that `match.arg` reaches into its caller, reflectively finds the default value for `type` and uses it as the value of `choices`; this is done as follows,

```
match.arg <- function (arg, choices)
  if (missing(choices)) {
    formal.args <- formals(sys.function(sys.parent()))
    choices <- eval(formal.args[[deparse(substitute(arg))]])
  ...
```

The first line checks if the `choices` argument was passed to the function. The second line gathers the list of parameters of the caller. Finally, the last line extracts from this list the parameter that has the same name as `arg` and `eval`s it.

Environments. Explicit environment manipulation hinders compiler optimizations. In our benchmarks these functions are called often. But it turns out that they are most often used to short-circuit the by-value semantics of R. We discovered that 87% of the calls to `remove`, which deletes a local variable from the current frame, are used as part of an implementation of a hash map. R also allows programs to change the nesting of an environment with `parent.env`. But 69% of these changes are used by the R VM's lazy load mechanism, and 30% by the `proto` library which implements prototypes [20] and uses environments to avoid copies.

7.3 Objects

The S4 object model has been promoted as a replacement for S3 by parts of the R community [16,3]. However, our numbers do show this happening. Thirteen years after the introduction of S4, S3 classes still dominate. From the number of methods introduced and the number of times they are redefined, S3 classes seem to be used quite differently than S4 classes.

Fig. 15 summarizes the use of object-orientation in the corpus. In our corpus, 1 055 S3 classes, or roughly one fourth of all classes, have no methods defined on them and 1 107 classes, 30%, have only a `print` or `plot` method. Fig. 16 gives the number of redefinitions of S3 methods. Any number of definitions larger than one suggest some polymorphism. Unsurprisingly, `plot` and `print` dominate. While important, does the need for these two functions really justify an object system? Attributes already allow the programmer to tag values, and could easily be used to store closures for a handful of methods like `print` and `plot`. A prototype-based system would be simpler and probably more efficient than the S3 object system. Finally, only 30% of S3 classes are really object-oriented. This translates to one class for every two packages. This is quite low and makes rewriting them as S4 objects seem feasible. Doing so could simplify and improve both R code and the evaluator code.

		Bioc	Misc	CRAN	Base	Total
S3	# classes	1 535	0	3 351	191	3 860
	# methods	1 008	0	1 924	289	2 438
	Avg. redef.	6.23	0	7.26	4.25	9.75
	Method calls	13M	58M	-	-	76M
	Super calls	697K	1.2M	-	-	2M
S4	# classes	1 915	2	1 406	63	2 893
	# singleton	608	2	370	28	884
	# leaves	819	0	621	16	1 234
	Hier. depth	9	1	8	4	9
	Direct supers	1.09	0	1.13	0.83	1.07
	# methods	4 136	22	2 151	24	5 557
	Avg. redef.	3	1	3.9	2.96	3.26
	Redef. depth	1.12	1	1.21	1.08	1.14
	# new	668K	64	-	-	668K
	Method calls	15M	266	-	-	15M
	Super calls	94K	0	-	-	94K

Fig. 15. Object usage in the corpus.

S4 objects on the other hand, seem to be used in a more traditional way. The class hierarchies are not deep (maximum is 9), however they are not flat either. The number of parent classes is surprisingly low (see [5] for comparison), but reaches a maximum of 50 direct super-classes. In Fig. 15, singleton classes, i.e., classes which are both root and leaf, are ignored. At first glance, the number of method redefinitions seems to be a bit smaller than what we find in other object languages. This is partially explained by the absence of a root class, the use of class unions, and because multi-methods are declared outside of classes. The number of redefinitions, i.e., one method applied to a more specific class, is very low (only 1 in 25 classes). This pattern suggests that the S4 object model is mostly used to overcome an absence of structure declarations rather than to add objects in statistical computing. Even when biased by Bioconductor, which pushes for S4 adoption, the use of S4 classes remains low. Part of the reason may be the perception that S3 classes are less verbose and clumsy to write than S4; it may also come from the fact that the base libraries use S3 classes intensively and this is reflected in our data.

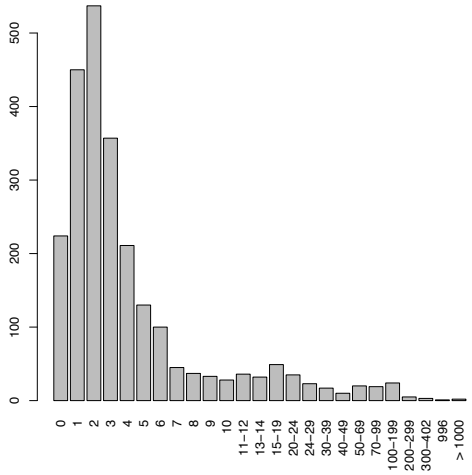


Fig. 16. S3 method redefinitions (on x axis).

7.4 Experience

Implementing the shootout problems highlighted some limitations of R. The pass-by-value semantics of function calls is often cumbersome. The R standard library does not provide data structures such as growable arrays or hash maps found in many other languages. Implementing them efficiently is difficult without references. To avoid copying, we were constrained to either use environments as a workaround, or to inline these operations and then make use of scoping and `<<-` as needed. Either choice makes the code unnecessarily verbose and readability suffers. Moreover, the former choice brings the question of whether the environments are used as intended, and the latter has a serious impact on code maintenance.

We found performance hard to predict. Without a solid understanding of the implementation, users are bound to be surprised by the impact of seemingly small changes to their code. In the binary tree program, adding an extra (and unneeded) `return` statement or a pair of parentheses `()` will impact performance in a noticeable way. Fig. 17 shows impact of adding these operations on performance for different input sizes.

Input size	12	13	14
Base time	6s	12s	31s
`()`	+7.4%	+5.3%	+6.3%
`return`	+4.6%	+5.4%	+5.4%

Fig. 17. Adding overheads.

8 Conclusions

This paper reports on our investigations into the design and implementation of the R language. Despite having millions of users and being, in many respects, a success story, R has received little attention from our community. With the exception of [13], which mistakenly characterized R as strict and imperative, ours is the first attempt to introduce R to a mainstream computer science audience.

Our first challenge was to understand the unconventional semantics of the language and the sometimes subtle interactions between its features. While some documentation exists, it is incomplete. The language is effectively defined by the successive releases of its implementation. Relying on an implementation as the authoritative specification of a language is unsatisfactory; the R interpreter is constrained by implementation decisions and presents a programming model that is at same time overconstrained and ambiguous. Implementation details are exposed and slowly bleed into the language. We have found it useful, for our own sake, to formalize the current implementation of R, focusing on features such as lazy evaluation, variable scoping and binding, and copy-semantics. Even though our semantics does not cover all of R, we did not oversimplify. We present a proper subset of R; we are confident that we will be able to extend it to larger portions of the language. As a language, R is like French; it has an elegant core, but every rule comes with a set of ad-hoc exceptions that directly contradict it.

A language definition is only part of the picture. The next question is how it is used in practice. Even the most elegant feature can be misused, and the ugliest language design can be used well when sufficient discipline is employed. To understand R in the wild, we implemented the TraceR framework and gathered over 3 million lines of code from various sources to form the largest open source R benchmark suite. Armed with these tools we started looking at how the exotic features of R are used by programs and what are the overheads and costs involved in supporting those features.

The R user community roughly breaks down into three groups. The largest groups are the end users. For them, R is mostly used interactively and R scripts tend to be short sequences of calls to prepackaged statistical and graphical routines. This group is mostly unaware of the semantics of R, they will, for instance, not know that arguments are passed by copy or that there is an object system (or two). The second, smaller and more savvy, group is made up of statisticians who have a reasonable grasp of the semantics but, for instance, will be reluctant to try S4 objects because they are "complex". This group is responsible for the majority of R library development. The third, and smallest, group contains the R core developers who understand both R and the internals of the implementation and are thus comfortable straddling the native code boundary.

One of the reasons for the success of R is that it caters to the needs of the first group, end users. Many of its features are geared towards speeding up interactive data analysis. The syntax is intended to be concise. Default arguments and partial keyword matches reduce coding effort. The lack of typing lowers the barrier to entry, as users can start working without understanding any of the rules of the language. The calling convention reduces the number of side effects and gives R a functional flavor. But, it is also clear that these very features hamper the development of larger code bases. For robust code, one would like to have less ambiguity and would probably be willing to pay for that by more verbose specifications, perhaps going as far as full-fledged type declarations. So, R is not the ideal language for developing robust packages. Improving R will require increasing encapsulation, providing more static guarantees, while decreasing the number and reach of reflective features. Furthermore, the language specification must be divorced from its implementation and implementation-specific features must be deprecated.

The balance between imperative and functional features is fascinating. We agree with the designers of R that a purely functional language whose main job is to manipulate massive numeric arrays is unlikely to be a success. It is simply too useful to be able to perform updates and have a guarantee that they are done in place rather than hope that a smart compiler will be able to optimize them. The current design is a compromise between the functional and the imperative; it allows local side effects, but enforces purity across function boundaries. It is unfortunate that this simple semantics is obscured by exceptions such as the super-assignment operator (`<<-`) which is used as a sneaky way to implement non-local side effects.

One of the most glaring shortcomings of R is its lack of concurrency support. Instead, there are only native libraries that provide behind-the-scenes parallel execution. Concurrency is not exposed to R programmers and always requires switching to native code. Adding concurrency would be best done after removing non-local side effects, and requires inventing a suitable concurrent programming model. One intriguing idea would be to push on lazy evaluation, which, as it stands, is too weak to be of much use outside of the base libraries, but could be strengthened to support parallel execution.

The object-oriented side of the language feels like an afterthought. The combination of mutable objects without references or cyclic structures is odd and cumbersome. The simplest object system provided by R is mostly used to provide printing methods for different data types. The more powerful object system is struggling to gain acceptance.

The current implementation of R is massively inefficient. We believe that this can, in part, be ascribed to the combination of dynamism, lazy evaluation, and copy semantics,

but it also points to major deficiencies in the implementation. Many features come at a cost even if unused. That is the case with promises and most of reflection. Promises could be replaced with special parameter declarations, making lazy evaluation the exception rather than the rule. Reflective features could be restricted to passive introspection which would allow for the dynamism needed for most uses. For the object system, it should be built-in rather than synthesized out of reflective calls. Copy semantics can be really costly and force users to use tricks to get around the copies. A limited form of references would be more efficient and lead to better code. This would allow structures like hash maps or trees to be implemented. Finally, since lazy evaluation is only used for language extensions, macro functions à la Lisp, which do not create a context and expand inline, would allow the removal of promises.

Acknowledgments. The authors benefited from encouragements, feedback and comments from John Chambers, Michael Haupt, Ryan Macnak, Justin Talbot, Luke Tierney, Gaël Thomas, Olga Vitek, Mario Wolczko, and the reviewers. This work was supported by NSF grant OCI-1047962.

References

1. R. A. Becker, J. M. Chambers, A. R. Wilks. *The New S Language*. Chapman & Hall, 1988.
2. D. G. Bobrow, K. M. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. In *Conference on Object-Oriented Programming, Languages and Applications (OOPSLA)*, 1986.
3. J. M. Chambers. *Software for Data Analysis: Programming with R*. Springer, 2008.
4. J. M. Chambers and T. J. Hastie. *Statistical Models in S*. Chapman & Hall, 1992.
5. R. Ducournau. Coloring, a Versatile Technique for Implementing Object-Oriented Languages. *Software: Practice and Experience*, 41(6):627–659, 2011.
6. R. Kent Dybvig. *The Scheme Programming Language*. MIT Press, 2009.
7. R. Gentleman, et. al., eds. *Bioinformatics and Computational Biology Solutions Using R and Bioconductor*. Statistics for Biology and Health. Springer, 2005.
8. R. Gentleman and R. Ihaka. Lexical scope and statistical computing. *Journal of Computational and Graphical Statistics*, 9:491–508, 2000.
9. P. Hudak, J. Hughes, S. Peyton Jones, P. Wadler. A history of Haskell: being lazy with class. In *Conference on History of programming languages (HOPL)*, 2007.
10. R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.
11. L. Keele. *Semiparametric Regression for the Social Sciences*. Wiley, 2008.
12. G. Kiczales, J. D. Rivieres, D. G. Bobrow. *The Art of the Metabobject Protocol: The Art of the Metaobject Protocol*. MIT Press, 1991.
13. Emily G. Mitchell. Functional programming through deep time: modeling the first complex ecosystems on earth. In *Conference on Functional Programming (ICFP)*, 2011.
14. T. Parr and K. Fisher. Ll(*): the foundation of the Antlr parser generator. *Conference on Programming Language Design and Implementation (PLDI)*, 2011.
15. R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2011.
16. R Development Core Team. *The R language definition*. R Foundation for Statistical Computing `http://cran.r-project.org/doc/manuals/R-lang.html`
17. G. Richards, S. Lesbrene, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Conference on Programming Language Design and Implementation (PLDI)*, 2010.
18. D. Smith. The R ecosystem. In *The R User Conference 2011*, August 2011.
19. G. L. Steele, Jr. *Common LISP: the language (2nd ed.)*, Digital Press, 1990.
20. D. Ungar and R. B. Smith. Self: The power of simplicity. In *Conference on Object-Oriented Programming, Languages and Applications (OOPSLA)*, 1987.
21. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1992.

$$\frac{H(\nu) = \mathtt{num}[\mathtt{n}]^{\alpha}}{\mathsf{reads}(\nu, H) = \mathtt{n}}\ [\textsc{reads}] \qquad \frac{H(\nu) = \mathtt{str}[\mathtt{s}]^{\alpha}}{\mathsf{readn}(\nu, H) = \mathtt{s}}\ [\textsc{readb}]$$

$$\frac{H(\nu) = \mathtt{num}[\mathtt{n}_1 \ldots \mathtt{n}_{\mathtt{m}} \ldots]^{\alpha} \quad \nu' \ \mathtt{fresh} \quad H' = H[\nu'/\mathtt{num}[\mathtt{n}_{\mathtt{m}}]^{\perp\,\perp}]}{\mathsf{get}(\nu, \mathtt{m}, H) = \nu', H'}\ [\textsc{GetN}]$$

$$\frac{H(\nu) = \mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s}_{\mathtt{m}} \ldots]^{\alpha} \quad \nu' \ \mathtt{fresh} \quad H' = H[\nu'/\mathtt{str}[\mathtt{s}_{\mathtt{m}}]^{\perp\,\perp}]}{\mathsf{get}(\nu, \mathtt{m}, H) = \nu', H'}\ [\textsc{GetS}]$$

$$\frac{H(\nu) = \mathtt{gen}[\nu_1 \ldots \nu_{\mathtt{m}} \ldots]^{\alpha}}{\mathsf{get}(\nu, \mathtt{m}, H) = \nu_{\mathtt{m}}, H'}\ [\textsc{GetG}]$$

$$\frac{\mathsf{readn}(\nu', H) = \mathtt{n} \quad H(\nu) = \mathtt{num}[\mathtt{n}_1 \ldots \mathtt{n}_{\mathtt{m}} \ldots]^{\alpha} \quad H' = H[\nu/\mathtt{num}[\mathtt{n}_1 \ldots \mathtt{n} \ldots]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}, \nu', H) = H'}\ [\textsc{SetN}]$$

$$\frac{\mathsf{reads}(\nu', H) = \mathtt{s} \quad H(\nu) = \mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s}_{\mathtt{m}} \ldots]^{\alpha} \quad H' = H[\nu/\mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s} \ldots]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}, \nu', H) = H'}\ [\textsc{SetS}]$$

$$\frac{H(\nu) = \mathtt{gen}[\nu_1 \ldots \nu_{\mathtt{m}} \ldots]^{\alpha} \quad H' = H[\nu/\mathtt{gen}[\nu_1 \ldots \nu' \ldots]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}, \nu', H) = H'}\ [\textsc{SetG}]$$

$$\frac{\mathsf{readn}(\nu', H) = \mathtt{n} \quad H(\nu) = \mathtt{num}[\mathtt{n}_1 \ldots \mathtt{n}_{\mathtt{m}}]^{\alpha} \quad H' = H[\nu/\mathtt{num}[\mathtt{n}_1 \ldots \mathtt{n}_{\mathtt{m}}\ \mathtt{n}]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}+1, \nu', H) = H'}\ [\textsc{SetNe}]$$

$$\frac{\mathsf{reads}(\nu', H) = \mathtt{s} \quad H(\nu) = \mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s}_{\mathtt{m}}]^{\alpha} \quad H' = H[\nu/\mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s}_{\mathtt{m}}\ \mathtt{s}]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}+1, \nu', H) = H'}\ [\textsc{SetSe}]$$

$$\frac{H(\nu) = \mathtt{gen}[\nu_1 \ldots \nu_{\mathtt{m}}]^{\alpha} \quad H' = H[\nu/\mathtt{gen}[\nu_1 \ldots \nu_{\mathtt{m}}\ \nu']^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}+1, \nu', H) = H'}\ [\textsc{SetGe}]$$

$$\frac{\mathsf{reads}(\nu', H) = \mathtt{s} \quad H(\nu) = \mathtt{num}[\mathtt{n}_1 \ldots \mathtt{n}_{\mathtt{m}} \ldots]^{\alpha} \quad H' = H[\nu/\mathtt{str}[s(\mathtt{n}_1) \ldots \mathtt{s} \ldots]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}, \nu', H) = H'}\ [\textsc{SetNs}]$$

$$\frac{H(\nu') = \mathtt{gen}[\nu_1 \ldots]^{\alpha'} \quad H(\nu) = \mathtt{num}[\mathtt{n}_1 \ldots \mathtt{n}_{\mathtt{m}} \ldots]^{\alpha} \quad H' = H[\nu/\mathtt{str}[g(\mathtt{n}_1) \ldots \nu' \ldots]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}, \nu', H) = H'}\ [\textsc{SetNg}]$$

$$\frac{\mathsf{readn}(\nu', H) = \mathtt{n} \quad H(\nu) = \mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s}_{\mathtt{m}} \ldots]^{\alpha} \quad H' = H[\nu/\mathtt{str}[\mathtt{s}_1 \ldots s(\mathtt{n}) \ldots]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}, \nu', H) = H'}\ [\textsc{SetSn}]$$

$$\frac{H(\nu') = \mathtt{gen}[\nu_1 \ldots]^{\alpha'} \quad H(\nu) = \mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s}_{\mathtt{m}} \ldots]^{\alpha} \quad H' = H[\nu/\mathtt{gen}[g(\mathtt{s}_1) \ldots \nu' \ldots]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}, \nu', H) = H'}\ [\textsc{SetSg}]$$

$$\frac{\mathsf{reads}(\nu', H) = \mathtt{s} \quad H(\nu) = \mathtt{num}[\mathtt{n}_1 \ldots \mathtt{n}_{\mathtt{m}}]^{\alpha} \quad H' = H[\nu/\mathtt{str}[s(\mathtt{n}_1) \ldots s(\mathtt{n}_{\mathtt{m}})\ \mathtt{s}]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}+1, \nu', H) = H'}\ [\textsc{SetNse}]$$

$$\frac{H(\nu') = \mathtt{gen}[\nu_1 \ldots]^{\alpha'} \quad H(\nu) = \mathtt{num}[\mathtt{n}_1 \ldots \mathtt{n}_{\mathtt{m}}]^{\alpha} \quad H' = H[\nu/\mathtt{str}[g(\mathtt{n}_1) \ldots g(\mathtt{n}_{\mathtt{m}})\ \nu']^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}+1, \nu', H) = H'}\ [\textsc{SetNge}]$$

$$\frac{\mathsf{readn}(\nu', H) = \mathtt{n} \quad H(\nu) = \mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s}_{\mathtt{m}}]^{\alpha} \quad H' = H[\nu/\mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s}_{\mathtt{m}}\ s(\mathtt{n})]^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}+1, \nu', H) = H'}\ [\textsc{SetSne}]$$

$$\frac{H(\nu') = \mathtt{gen}[\nu_1 \ldots]^{\alpha'} \quad H(\nu) = \mathtt{str}[\mathtt{s}_1 \ldots \mathtt{s}_{\mathtt{m}}]^{\alpha} \quad H' = H[\nu/\mathtt{gen}[g(\mathtt{s}_1) \ldots g(\mathtt{s}_{\mathtt{m}})\ \nu']^{\alpha}]}{\mathsf{set}(\nu, \mathtt{m}+1, \nu', H) = H'}\ [\textsc{SetSge}]$$

$$\frac{H(\iota) = F \quad F(\mathtt{x}) = \nu}{\iota(H, \mathtt{x}) = \nu}\ [\textsc{Look0}] \qquad \frac{\Gamma = \iota * \Gamma' \quad \iota(H, \mathtt{x}) = \nu}{\Gamma(H, \mathtt{x}) = \nu}\ [\textsc{Look1}] \qquad \frac{\Gamma = \iota * \Gamma' \quad H(\iota) = F \quad \mathtt{x} \notin dom(F) \quad \Gamma'(H, \mathtt{x}) = \nu}{\Gamma(H, \mathtt{x}) = \nu}\ [\textsc{Look2}]$$

$$\frac{\mathsf{cpy}(H, \nu_{\perp}) = H', \nu'_{\perp} \quad \mathsf{cpy}(H', \nu'_{\perp}) = H'', \nu'''_{\perp}}{\mathsf{cpy}(H, \nu_{\perp}, \nu'_{\perp}) = H'', \nu''_{\perp}, \nu'''_{\perp}}\ [\textsc{Copy0}] \qquad \frac{}{\mathsf{cpy}(H, \perp) = H, \perp}\ [\textsc{Copy1}]$$

$$\frac{H(\nu) = \kappa^{\alpha} \quad \alpha = \nu_{\perp}\nu'_{\perp} \quad \mathsf{cpy}(H, \nu_{\perp}, \nu'_{\perp}) = H'\,\nu''_{\perp}, \nu'''_{\perp} \quad \nu'' \ \mathtt{fresh} \quad H'' = H'[\nu''/\kappa^{\nu''_{\perp}\,\nu'''_{\perp}}]}{\mathsf{cpy}(H, \nu) = H'', \nu''}\ [\textsc{Copy2}]$$

$$\frac{\Gamma = \iota * \Gamma' \quad H(\iota) = F \quad \mathtt{x} \in dom(F) \quad F' = F[\mathtt{x}/\nu] \quad H' = H[\iota/F']}{\mathsf{assign}(\mathtt{x}, \nu, \Gamma, H) = H'}\ [\textsc{Super1}]$$

$$\frac{\Gamma = \iota * \Gamma' \quad H(\iota) = F \quad \mathtt{x} \notin dom(F) \quad \mathsf{assign}(\mathtt{x}, \nu, \Gamma', H) = H'}{\mathsf{assign}(\mathtt{x}, \nu, \Gamma, H) = H'}\ [\textsc{Super2}]$$

$$\frac{\Gamma = \iota \quad H(\iota) = F \quad F' = F[\mathtt{x}/\nu] \quad H' = H[\iota/F']}{\mathsf{assign}(\mathtt{x}, \nu, \Gamma, H) = H'}\ [\textsc{Super3}]$$

Fig. 18. Auxiliary definitions.