Evaluating the design of a language for data analytics

Authors:
Floreal Morandat,
Brandon Hill,
Leo Osvald,
Jan Vitek
@ S3Lab,
Purdue University

Thanks:
NSF
Microsoft
Why?

... language for data analysis and graphics

... used in statistics, biology, finance ...

... based on S, but faster & open source

... books, conferences, user groups

... 4,338 packages

... 3 millions users
Scripting data

read data into variables
make plots
compute summaries
more intricate modeling
develop simple functions
to automate analysis

...
“the lazy functional language in wide use”

“seamless integration of objects and functions”

“interactive and intuitive”

“unencumbered by types”

“fully extensible”

“can be learned in a day”
How do we evaluate claims made about a language?

What can we learn from a language design?
EVALUATE

Our approach

• Extract core semantics by testing
  - *R has no official semantics*
  - *A single reference implementation*

• Observational study based on a large corpus
  - *Many open source programs come with “vignettes”*
  - *Dynamic analysis gives under-approximated behaviors*
  - *Static analysis gives over-approximation*

This talk

• An excursion in R’s wondrous world
The last group is the base library that is bundled with the R VM. Fig. 5 gives the size of these datasets.

<table>
<thead>
<tr>
<th>Name</th>
<th>Bioc.</th>
<th>Shoot.</th>
<th>Misc.</th>
<th>CRAN</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td># Package</td>
<td>630</td>
<td>11</td>
<td>7</td>
<td>1238</td>
<td>27</td>
</tr>
<tr>
<td># Vignettes</td>
<td>100</td>
<td>11</td>
<td>4</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>R LOC</td>
<td>1.4M</td>
<td>973</td>
<td>1.3K</td>
<td>2.3M</td>
<td>91K</td>
</tr>
<tr>
<td>C LOC</td>
<td>2M</td>
<td>0</td>
<td>0</td>
<td>2.9M</td>
<td>50K</td>
</tr>
</tbody>
</table>
Faces of R
Vectors

```r
x <- c(2, 7, 9, NA, 5)  
c(1, 2, 3) + x[1:3]  
x[is.na(x)] <- 0
```
Functions

\[ q \leftarrow \text{function}(x=5) x \times x \times x \]

\[ q() \]

\[ q(2) \]

\[ q(x=4) \]

\[ p \leftarrow \text{function}(x=5, \ldots, y=x+1) \]
Evaluating the Design of R

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.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 125 | 126 | 127 | 128 | 129 | 130 | 131 | 132 | 133 | 134 | 135 | 136 | 137 | 138 | 139 | 140 | 141 | 142 | 143 | 144 | 145 | 146 | 147 | 148 | 149 | 150 | 151 | 152 | 153 | 154 | 155 | 156 | 157 | 158 | 159 | 160 | 161 | 162 | 163 | 164 | 165 | 166 | 167 | 168 | 169 | 170 | 171 | 172 | 173 | 174 | 175 | 176 | 177 | 178 | 179 | 180 | 181 | 182 | 183 | 184 | 185 | 186 | 187 | 188 | 189 | 190 | 191 | 192 | 193 | 194 | 195 | 196 | 197 | 198 | 199 | 200 | 201 | 202 | 203 | 204 | 205 | 206 | 207 | 208 | 209 | 210 | 211 | 212 | 213 | 214 | 215 | 216 | 217 | 218 | 219 | 220 | 221 | 222 | 223 | 224 | 225 | 226 | 227 | 228 | 229 | 230 | 231 | 232 | 233 | 234 | 235 | 236 | 237 | 238 | 239 | 240 | 241 | 242 | 243 | 244 | 245 | 246 | 247 | 248 | 249 | 250 | 251 | 252 | 253 | 254 | 255+

---

f(1, 2)
f(x=1, y=2)
f(y=1, x=2)
f(2, x=1)
f(x=1, 2)
c(1, 2, 3, 4)
assert<-function(C,P)
    if (C) print(P)

assert(x == 2, report(x, "Bad"))

with(fd, carb*den)
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.

% of promises evaluated / vignette
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 functions 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 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.
Forcing promises

\[ C ::= [] \mid x \leftarrow \textcircled{C} \mid x[[C]] \mid x[[C]] \mid x[[e]] \leftarrow \textcircled{C} \mid x[[C]] \leftarrow \nu \mid \{C; e\} \mid \{\nu; C\} \]
assign <- function(x, val, pos=-1, env=as.environment(pos), immediate=TRUE)
  .Internal(assign(x, val, env))
Lexical scoping with context sensitive name resolution

\[
\begin{align*}
\texttt{c} & \leftarrow 42 \\
\texttt{c (1, 2, 3)} & \\
\texttt{d} & \leftarrow \texttt{c} \\
\texttt{d (1, 2, 3)} &
\end{align*}
\]
less than 0.05% context sensitive function name lookups

only symbols that rely on it are `c`
and `file`
Referential transparency

```
assert(x==4); assert(y[[1]]==5)
f(x, y)
assert(x==4); assert(y[[1]]==5)
f<-function(a, b) { x<-0; a<-1; b[[1]]<-0 }
```
Assignment

\[
\text{cpy}(H, \nu) = H', \nu'
\]

\[
\Gamma = \nu \star \Gamma' \quad H(\nu) = F
\]

\[
F' = F[x/\nu'] \quad H'' = H'[\nu/F']
\]

\[
x \leftarrow \nu \Gamma; H \rightarrow \nu; H''
\]
Assignment

cpy(H, ν’) = H’, ν’’
readn(ν, H’) = m
set(ν’’, m, ν’’, H’) = H’’

\[ x[[ν]] ← ν' \quad Γ; H \rightarrow ν'; H'' \]
Assignment

\[
\begin{align*}
\text{cpy}(H, \nu') & = H', \nu'' & \Gamma = \nu \ast \Gamma' & H'(\nu) = F & x \not\in F & \Gamma'(H', x) = \nu''' \\
\text{cpy}(H', \nu''') & = H'', \nu''''' & F' = F[x/\nu'''''] & H'''' = H''''[\nu/F'] \\
\text{readn}(\nu, H) & = m & \text{set}(\nu''''', m, \nu'', H''''') = H''''' \\
\mathbf{x}[\nu] & \leftarrow \nu' \Gamma; H \rightarrow \nu'; H''''''
\end{align*}
\]
Assignment

\[ \text{assign}(x, \nu', \Gamma', H') = H'' \]

\[ x \leftarrow \nu \Gamma; H \rightarrow \nu; H'' \]
45% of assignments are definitions

only 2 out of 217 million
assignments are definitions in a
parent frame

99.9% of side effects are local
Reflection

```r
with.default <- function(data, exp, ...)
  eval(substitute(exp), data, parent.frame))
```
x <- c(1,2,3,4)

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

who <- function(x) UseMethod("who")

who.man <- function(x) print("Ceasar!")

who.default <- function(x) print("??")

me <- 42; who(me)

class(me) <- 'man'; who(me)
Fig. 15. Object usage 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.

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.
**Objects**

```r
setClass("P", representation(x="numeric", y="numeric"))
setClass("C", representation(color="character"))
setClass("CP", contains=c("P", "C"))

r <- new("CP", x = 0, y = 0, color = "red")
r@color

setGeneric("add", function(a, b) standardGeneric("add"))
setMethod("add", signature("P","P"),
  function(a, b) new("P", 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))
```
### Object usage

<table>
<thead>
<tr>
<th></th>
<th>Bioc</th>
<th>Misc</th>
<th>CRAN</th>
<th>Base</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S3</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># classes</td>
<td>1535</td>
<td>0</td>
<td>3351</td>
<td>191</td>
<td>3860</td>
</tr>
<tr>
<td># methods</td>
<td>1008</td>
<td>0</td>
<td>1924</td>
<td>289</td>
<td>2438</td>
</tr>
<tr>
<td>Avg. redef.</td>
<td>6.23</td>
<td>0</td>
<td>7.26</td>
<td>4.25</td>
<td>9.75</td>
</tr>
<tr>
<td>Method calls</td>
<td>13M</td>
<td>58M</td>
<td>-</td>
<td>-</td>
<td>76M</td>
</tr>
<tr>
<td>Super calls</td>
<td>697K</td>
<td>1.2M</td>
<td>-</td>
<td>-</td>
<td>2M</td>
</tr>
<tr>
<td><strong>S4</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># classes</td>
<td>1915</td>
<td>2</td>
<td>1406</td>
<td>63</td>
<td>2893</td>
</tr>
<tr>
<td># singleton</td>
<td>608</td>
<td>2</td>
<td>370</td>
<td>28</td>
<td>884</td>
</tr>
<tr>
<td># leaves</td>
<td>819</td>
<td>0</td>
<td>621</td>
<td>16</td>
<td>1234</td>
</tr>
<tr>
<td>Hier. depth</td>
<td>9</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>Direct supers</td>
<td>1.09</td>
<td>0</td>
<td>1.13</td>
<td>0.83</td>
<td>1.07</td>
</tr>
<tr>
<td># methods</td>
<td>4136</td>
<td>22</td>
<td>2151</td>
<td>24</td>
<td>5557</td>
</tr>
<tr>
<td>Avg. redef.</td>
<td>3</td>
<td>1</td>
<td>3.9</td>
<td>2.96</td>
<td>3.26</td>
</tr>
<tr>
<td>Redef. depth</td>
<td>1.12</td>
<td>1</td>
<td>1.21</td>
<td>1.08</td>
<td>1.14</td>
</tr>
<tr>
<td># new</td>
<td>668K</td>
<td>64</td>
<td>-</td>
<td>-</td>
<td>668K</td>
</tr>
<tr>
<td>Method calls</td>
<td>15M</td>
<td>266</td>
<td>-</td>
<td>-</td>
<td>15M</td>
</tr>
<tr>
<td>Super calls</td>
<td>94K</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>94K</td>
</tr>
</tbody>
</table>

Fig. 15. Object usage in the corpus.

Fig. 15 summarizes the use of object-orientation in the corpus. In our corpus, 1055 S3 classes, or roughly one fourth of all classes, have no methods defined on them and 1107 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.

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.
To understand where the time is spent, we turn to more representative R programs. Fig. 7 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. Memory management time was further broken down into time spent in garbage collection (18.7%), allocating cons-pairs (3.6%), vectors (2.6%), and duplications (4%) for call-by-value semantics. The time spent in built-in functions represents the true computational work performed by R, this is on average 38% of execution time. There are some interesting outliers. The maximum spent in garbage collection is 70% and there is a program that spends 63% copying arguments. The lookup and match categories (4.3% and 1.8%) represent time spent looking up variables and matching parameters with arguments. Both of these would be absent in a more static language like C as they are resolved at compile time. Variable lookup will also be absent in Lisp or Scheme as, once bound, position of variables in a frame are known. Given the nature of R, many of the core numerical functions are written in C or Fortran. This can lead to the perception that execution time is dominated by native libraries. Looking at the amount of time spent in calls to foreign...
To understand where the time is spent, we turn to more representative R programs. Fig. 7 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. Memory management time was further broken down into time spent in garbage collection (18.7%), allocating cons-pairs (3.6%), vectors (2.6%), and duplications (4%) for call-by-value semantics. The time spent in built-in functions represents the true computational work performed by R, this is on average 38% of execution time. There are some interesting outliers. The maximum spent in garbage collection is 70% and there is a program that spends 63% copying arguments. The lookup and match categories (4.3% and 1.8%) represent time spent looking up variables and matching parameters with arguments. Both of these would be absent in a more static language like C as they are resolved at compile time. Variable lookup will also be absent in Lisp or Scheme as, once bound, position of variables in a frame are known. Given the nature of R, many of the core numerical functions are written in C or Fortran. This can lead to the perception that execution time is dominated by native libraries. Looking at the amount of time spent in calls to foreign
Not only is R slow, but it also consumes significant amounts of memory. Unlike C, C++ or Java, R is a dynamic language. 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.

An empty vector is 40 bytes long; 10^9 cells are 56 byte long, and take up 23 GB on average in the Shootout benchmarks. Yet, 36% of vectors allocated in R has to be boxed into a vector. As mentioned above, they are mostly used by the R VM. In R, call arguments are stack allocated, therefore, argument passing to functions is very simple, on average only 37% of arguments are copied.

Lists are created by pairlist. It is inter ws multiple functions). The graph clearly shows that R allocates orders of magnitude more data than the C++ and Java Shootout benchmarks. C++ and Java code mostly live on the stack, and they only allocate small amounts of data on the heap. R and Bioconductor none. The R User data is mostly used by the interpreter for arguments to functions. R is clearly slow and memory inefficient. Much more so than other dynamic languages. This is really simple, on average only 37% of arguments are copied.

Avoiding duplication reduces memory footprint. Even though the COW algorithm is under the covers, arguments are shared and only duplicated if there is actually a need. This is the reason why the COW algorithm is used in R.

Another reason for the large footprint is that all numeric data retain only a single numeric value. This is why, in many cases, the internal data required is more than the user data. This analysis is based on the R User data and R internal data collected. Fig. 8 shows the Shootout benchmarks. The costs involved in allocating and freeing these vectors, and the fact that even simple arithmetic requires a function call, further impacts run-time.
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

![Productivity Chart](chart.png)

<table>
<thead>
<tr>
<th>Productivity</th>
<th>Python</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shootout</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Size (LOC)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOC 10000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOC 20000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOC 30000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOC 40000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOC 50000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Fig. 11. Bioconductor R and C source code size. (LOC, no comments)
Morandat et al. indicated that 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 S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 A vg.

![Graph](image.png)

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

![Graph](image.png)

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