R: Lessons Learned, Directions for the Future

Ross Ihaka[*]

Abstract
It has been nearly twenty years since Robert Gentleman and I began work on the R system. Although R has become a much more fully featured and well rounded system, its central core remains largely unchanged. With the benefit of hindsight it is possible look at how well this core design has held up, at ways in which it could be improved and whether or not this could lead to a successful new system.

Key Words: statistical computing

1. Introduction

I regularly find colleagues and students accusing me of badmouthing R because I can often be seen pointing out its shortcomings. Lest I leave you with the same impression, let me say at the outset that I have had a ball being part of the R development process and I take some pride in the results we have obtained. If I could only get to achieve one thing in life, being a part of creation and development of R would be enough.

R now provides a readily available system for carrying out quantitative and some non-quantitative calculations. It is comparatively easy to learn to get answers from R, although it can be more difficult to learn to get answers in an efficient way. There is now a large library of applications which can be used from R and a troll through the CRAN sites can turn up some real gems. I use R regularly and spend a good deal of my time teaching students how to use it.

However, despite my affection for R, I think it is important that we begin to look dispassionately at how it performs and to ask whether it is the system that we should be considering using into the indefinite future. My considered opinion is that it isn't and I hope to explain why this is the case and what we should be doing about it.

2. How R Developed

R did not always look like an alternative implementation of the S language. It started as a small Scheme-like interpreter (loosely based on work by Sam Kamin [4] and David Betz [2]). This provided a platform for experimentation and extension. The following dialog shows a very early version of the R interpreter at work.

```
> (define square (lambda (x) (* x x)))
square
> (define v 10)
v
> (square v)
100
```

The S-like appearance of R was added incrementally. We initially moved to an S-like syntax for our experiments because, although we were both familiar with Lisp syntax, we

[*]The University of Auckland, Auckland, New Zealand

didn't feel that it was really suitable for expressing statistical computations. This choice of syntax set us on a path towards compatibility with S. Once initiated, the move towards compatibility with S was irresistible. This was partly because we wanted to see just how far we could push it and partly because it have us access to code resources and developers.

The present success of R is largely due to the efforts of a dedicated group of developers joined in the effort and who have unselfishly contributed to creating what R has become. In particular the "R Core" group has been instrumental in R's success. Over the years the group has included the following members:

Douglas Bates
John Chambers
Peter Dalgaard
Seth Falcon
Robert Gentleman
Kurt Hornik
Stefano Iacus
Ross Ihaka
Friedrich Leisch
Thomas Lumley
Martin Maechler
Guido Masarotto
Duncan Murdoch
Paul Murrell
Martyn Plummer
Brian Ripley
Deepayan Sarkar
Heiner Schwarte
Duncan Temple Lang
Luke Tierney
Simon Urbanek

In addition, an even larger group has contributed bug-fixes and enhancements.

Work by R Core and other contributors ultimately produced the present mature and widely-used system. R has now reached a kind of comfortable maturity. It now has a large number of users who require a stable platform for getting their work done. For this reason it is less suitable as a base for experimentation and development.

The present work on R consists mainly of maintenance and the addition of small enhancements. A good deal of work is going into making R more efficient. Much of this work consists of reimplementing interpreted R code in compiled C (this is essentially hand compilation of critical code sections).

R provides An interactive, extensible, vectorised language with a large run-time environment which provides a good deal of statistical functionality and good graphics capabilities. It comes with the the freedom to inspect, modify and redistribute the source code. Extensive "third-party solutions" are available through the CRAN websites. User support is available through community mechanisms.

3. Why R is Not Enough

R does well at the tasks it was originally designed for. Indeed, there is an argument that the present R has far surpassed its original design goals (which were rather modest). This indicates that the original strategy of implementing a small language using a well trodden path and then extending it into something larger is a sound one.

Unfortunately, a number of factors place fairly severe limitations on how R performs, particularly for larger data sets. The following issues present particular problems:

- The R interpreter is not fast and execution of large amounts of R code can be unacceptably slow.

- R is set up to carry out *vectorised* computations and not scalar (element-by-element) computations. Not every computation can be expressed efficiently in a vectorised way and the use of a vector-based language to carry out scalar computations introduces unacceptable computational overheads.

- R was designed to hold its data “in core” and this places severe limitations of the size of problem which can be dealt with. These days, data sets on the petabyte and exabyte scale are not unknown and, even when the “heavy lifting” is offloaded to databases or special purpose tools like Hadoop [1], R is not a useful tool for handing them.

The following example serves to illustrate the kinds of problem that R has. Dataframes are a fundamental data structure which R uses to hold the traditional case/variable layouts common in statistics. The problem, brought to me by a colleague, concerns the process of updating the rows of a dataframe in a loop. A highly simplified version of this problem is presented below.

```
n = 60000
r = 10000
d = data.frame(w = numeric(n), x = numeric(n),
               y = numeric(n), z = numeric(n))
value = c(1, 2, 3, 4)
system.time({
    for(i in 1:r) {
        j = sample(n, 1)
        d[j,] = value
    }
})
```

This code fragment creates a dataframe with 60000 cases and 4 variables. A loop is run to update the values in 10000 randomly selected rows of the dataframe.

On my machine, this computation runs in roughly 100 seconds. This is a long time for a computation on such a small amount of data. On the scale that my colleague was working, the performance was unacceptably slow.

Knowing a little about the internal working of R, I was able to advise my colleague to simply take the variables out of the dataframe and update them one-by-one.

```
n = 60000
r = 10000
w = x = y = z = numeric(n)
value = c(1, 2, 3, 4)
system.time({
    for(i in 1:r) {
        j = sample(n, 1)
        w[j] = value[1]
        x[j] = value[2]
        y[j] = value[3]
        z[j] = value[4]
    }
})
```

This change reduces the computation time from 100 seconds to .2 seconds and my colleague found the 500-fold performance boost acceptable.

The problem in this case is caused by a combination of factors. R uses “call-by-value” semantics which means that arguments to functions are (at least conceptually) copied before any changes are made to them. The update

```
d[j,] = value
```

is carried out by a function call and so the dataframe `d` is copied. In fact it is copied multiple times for each update.

This kind of behaviour is completely unacceptable for a data structure as fundamental as data frames. The current approach to fixing this kind of problem is to move the update process into C code, where copying can be controlled. This does alleviate the symptom, but it does not fix the underlying problem which is likely to exhibit itself elsewhere.

A second problem is that R has no scalar data types and users spend a good deal of time looking for "cute" ways to vectorise problems which are much more naturally expressed in a scalar way. The following code carries out a matching technique, returning the values in the vector `x` closest to the values in `y`.

```
xymatch =
    function(x, y) {
        x[apply(outer(x, y, function(u, v) abs(u - v)),
              1, function(d) which(d == min(d))[1])]
```

This is certainly a "cute" one-line solution, but the use of the generalised outer product function `outer` expands the problem from an $m+n$ scale to an $m \times n$ one. While it works well for small problems, it is not a good way to solve large ones. It would be much more efficient to use loops and scalar computation.

4. Directions for Future Work

When we started work on R, my original hope was that we would develop a kind of toolkit which could be used to implement a variety of statistical languages, and these could be used to make judgements about what kind of language might be best. This ambition disappeared when we began to follow the path of "S compatibility."

Despite having been sidetracked by work on R for twenty or so years I've started looking again at the basics of how to implement a statistical language. What is immediately apparent is that we need orders of magnitude increases in performance over what R (and other interpreters) can provide. There are a number of ways in which we can look for such performance gains.

1. Wait for faster machines.
2. Introduce more vectorisation and take advantage of multicores.
3. Make changes to R to eliminate bottlenecks.
4. Sweep the page clean and look at designs for new languages.

At the moment I don't see the first three of these strategies as being able to deliver the performance we will need in future (or even those that are needed now), which leaves the last possibility.

Duncan Temple Lang, Brendan McArdle and I have begun examining what such new languages might look like. At the moment we are examining the basic constraints on computational speed and how we can achieve as close to the best possible performance as possible (see [3], for example). Some lessons are becoming clear.

1. Compilation (to byte-code or machine code) can provide speedups in the range from a small multiple to an order of magnitude. Some language constructs (e.g. `eval`, `get`, `assign`, `rm` and *scoping* rules like R's work against obtaining efficiency gains through compilation. Cleaning up (i.e. changing) language semantics should make it possible to get closer to the order of magnitude value.

2. R is very slow at scalar computations because it has no built-in scalar types. This means that boxing and unboxing overheads cannot be avoided. This could be overcome by introducing such types and using them to carry out scalar computation where such computations provide better performance.

3. Pass-by-value semantics can hurt computational performance because of unnecessary copying. Strategies for avoiding copying can be put in place, but these are invariably conservative and will not always alleviate the problem. Moving to *pass-by-reference* semantics should produce efficiency gains and make it possible to handle much larger problems.

4. Compiler performance can be boosted by the introduction of (optional) type declarations. Performance analysis often makes it possible to determine a few program locations which have a big impact on performance. By giving the compiler information about the types of variables in these locations it is often possible to eliminate the bottlenecks. In particular, it should be possible to eliminate method dispatch for common cases (like scalar arithmetic) in simple cases, making performance comparable with C and Fortran.

5. The use of compact "cute" computation specifications is attractive in an interactive language. Naive evaluation of vector expressions like `x+y+z` creates a vector intermediate `x+y` which is discarded immediately `x+y+z` is formed. Transforming this into an iteration over vector elements makes it possible to store intermediate values in machine registers avoiding the allocation of intermediate vectors. Type declarations make it possible to implement such optimisations. The SAC language (see Scholz [5]) provides an example of what can be done.

At present we are simply exploring these issues. If we are to develop a truly effective language for handing large-scale computations it is important that we understand, in depth, as many of the issues as we can before we begin development. What I have seen so far gives me hope that it will be possible to develop languages with much better performance than those in current use.

References

[1] Apache Foundation (2007). `http://hadoop.apache.org`.

[2] Betz, D. (2002). `http://www.xlisp.org`.

[3] Ihaka, R. and Temple Lang, D. (2008). *Back to the Future: Lisp as a Base for a Statistical Computing System*. Invited Paper, Compstat 2008. Porto, Portgal. .

[4] Kamin, S. (1990). *Programming Languages: An Interpreter-Based Approach*. Reading, Massachusetts: Addison-Wesley.

[5] Scholz, S. (2003). " Single Assignment C – Efficient Support for High-level Array Operations in a Functional Setting." *Journal of Functional Programming* **13**(6), pp.1005-1059.