Back to the Future: Lisp as a Base for a Statistical Computing System

Ross Ihaka[1] and Duncan Temple Lang[2]

[1] University of Auckland, New Zealand, *ihaka@stat.auckland.ac.nz*
[2] University of California, Davis, USA

Abstract. The application of cutting-edge statistical methodology is limited by the capabilities of the systems in which it is implemented. In particular, the limitations of R mean that applications developed there do not scale to the larger problems of interest in practice. We identify some of the limitations of the computational model of the R language that reduces its effectiveness for dealing with large data efficiently in the modern era.

We propose developing an R-like language on top of a Lisp-based engine for statistical computing that provides a paradigm for modern challenges and which leverages the work of a wider community. At its simplest, this provides a convenient, high-level language with support for compiling code to machine instructions for very significant improvements in computational performance. But we also propose to provide a framework which supports more computationally intensive approaches for dealing with large datasets and position ourselves for dealing with future directions in high-performance computing.

We discuss some of the trade-offs and describe our efforts to realizing this approach. More abstractly, we feel that it is important that our community explore more ambitious, experimental and risky research to explore computational innovation for modern data analyses.

Keywords: Lisp, optional typing, performance

1 Background

The growth in popularity of R over the last decade has been impressive and has had a significant impact on the practice and research of statistics. While the technical achievements have been significant, the fostering of a community which has continued the development of R and the myriad of packages that provide cutting-edge statistical methodology is perhaps the most significant achievement of the R project.

R is not unlike the S language that was developed at Bell Labs over the last 3 decades of the last century. At that time, S was revolutionary in concept and enabled a different approach to data analysis that continues today. A similar change in the way we do data analysis and statistical computing is needed again. This is no small part due to the changing nature of scientific computing (parallel and distributed computing, Web-based data access and computing,

massive data sets, computationally intensive methods). But also, we need to be undertaking bold research that involves experimenting with these new technologies and guiding statisticians to new computational paradigms rather than focusing mostly on making the existing, familiar facilities easier to use and implementing ideas available in numerous other programming languages.

It is important that the statistical community recognize the impact that R has had and not assume that it is sufficient for the long-term or that new developments will simply happen. Rather, they must encourage, support and participate in the development of new ideas and infrastructure.

In part, due to the success and popularity of R, it is no longer a research vehicle for more ambitious experiments. The focus of R development has changed gradually to be one of adding important usability features found in other languages, e.g. graphical user interfaces, support for Unicode and internationalisation, and improving portability and ease of use. People wanting to pursue more experimental research projects have been faced with the "nobody will use it" issue as there is a single, "official" R. Simply put, the phrase "the good is the enemy of the better" expresses well the sentiment that R has proven to be good enough for our needs and that an incremental, more localized mindset has developed and has made development of R somewhat conservative. This has inhibited significant changes in direction and has encouraged the more incremental, short term developments rather than a big picture research oriented view of statistical computing. Unfortunately, this has become dominant within the statistics community and journals, and we are now focused more on implementations of existing algorithms than novel new paradigms. To encourage and retain good minds in this field, we need to provide a more significant innovative and exciting research environment where concepts not code are the topics discussed and we are working on large problems, not just details of smaller issues.

2 Issues with R

Before commenting on any of R's deficiencies, we should note that R has been and continues to very effective and successful and there have been numerous significant developments within its history. However, modern data analysis and statistical and scientific computing are continuing to change at a dramatic rate and the essential computational model underlying R is tied to that of the early S systems from 20 to 30 years ago. We outline some of the issues below and note that they refer to efficiency of code execution and support for better programming practices with type specification.

Copying: R uses a pass-by-value semantic for function calls. This means that when a function modifies the contents of one of its arguments, it is a local copy of the value which is changed, not the original value. This has many desirable properties, including aiding reasoning about and debugging code,

and ensuring precious data is not corrupted. However, it is very expensive as many more computations need to be done to copy the data, and many computations require excessive memory due to the large number of copies needed to guarantee these semantics.

Whole-Object and Vectorized Computations versus Scalar Operations: There is a significant benefit to using vectorized functions that perform operations on the whole object rather than writing code that processes elements individually. However, many operations are hard to vectorise and operations that need to "unbox" individual values are extremely expensive. (See section 4 for timing results.)

Compiled Native Code: To obtain efficient code, it is quite common to move important code to C and access that from R. While this is not very difficult, it does pose a challenge to many users and requires knowledge of an additional programming language. Further, it make the resulting software less amenable to extensions by others, and involves significantly more work by the author to bridge the interface between the two languages, and especially debugging the two separate pieces of code.

Software and Type checking: Like many high-level programming languages, R does not require or support declarations and type specification for variables. This is very useful for rapid, interactive programming and prototyping. However, when developing larger systems or software for others to use, being able to annotate code with type information and have the system enforce it is an important productivity gain and produces more robust and reflective software.

These are issues with the language, not the implementation. They reflect sensible decisions that we need to reevaluate in the face of significant changes to computing and data analysis over the last and next decade.

3 Common Lisp

The R engine began life as a very simple Lisp interpreter. The similarities between S and Lisp made it easy to impose an S-like syntax on the interpreter and produce a result which looked very much like S. The fact that this approach has succeeded once raises the question of whether it might be possible to do even better by building a statistical language over a more robust, high-performance Lisp. There are both pluses and minus to taking this approach. On the minus side, "we" will no longer own the implementation details of all aspects of our computing environment. This reduces our independence to enact our own modifications. On the plus side, we gain the experience and effort of an entirely different and broader community in the implementation of an engine. This means that there is no necessity to add

features like namespaces, conditions/exceptions and an object system to the language. They are already present. One of the most important benefits of the approach is that we can use a version of Lisp that compiles to machine code and get significantly improved performance for general code via optional specification of the data types. This raises the possibility of greatly improving the performance of our statistical systems.

Common Lisp is a natural choice of Lisp for building large software systems. It is a formally standardized specification with many implementations – both open source and commercial. The implementations of interest to us provide various different types of small and very large number types (including rationals); language macros; lexical scoping/closures; dynamic scoping; optional type declarations; machine-code compilation; name spaces; a basic package mechanism; extensible I/O types via connections/streams; foreign function interface (FFI); thread support; reflection and programming on the language; additional user-level data structures (e.g. general hash tables, linked lists); unicode; reference semantics; destructive in-situ operations; error handling system (conditions and exceptions); profiling and debugging tools; interfaces for Emacs; IDEs for commercial versions of Lisp and an object/class system similar but richer than the S4 system in R. It is one of the few languages that is both high-level (interactive) and low-level & efficient (compiled to machine code) and offers features similar to those that have proven effective in statistics, with more idealized semantics for statistical computing. The syntax is quirky, but a thin layer on top of this that provides a more familiar form (see section 6) makes Lisp an extremely attractive candidate for moving forward in statistical computing practice and research.

Using Common Lisp provides significant advantages. It would free up the limited and valuable resources that the statistical computing community invests in maintaining, extending and innovating its own language and interpreter when a better one is already available to us. We do lose some control over aspects of the software environment, but the similarities of R and Lisp are such that this does not seem of any consequence. We can contribute changes to Open Source Lisp implementations (e.g. SBCL) or even fork development if we truly need such autonomy. But with the resources not tied to porting new features already existing in Lisp – both now and in the future – we can focus on innovations in statistical computing rather than computer science and information technology.

If we are willing to embark on building a new statistical computing environment, we need to consider all possible languages that might serve as a good base, and not just Lisp. We discuss other candidates in section 8.

4 Speed, compilation & timings

As mentioned previously, many R packages use compiled C/FORTRAN code in order to gain efficiency. As a result, R is a good prototyping environment but requires low-level programming for computationally intensive methods. And this has led people to disregard it for use in large-scale, high performance computing tasks. We want to reduce the gap between programming in the high-level language (R) and making things efficient in the system-level language (C), and also to allow methodology developed and implemented by statisticians to be used in real, industrial-strength applications. We believe that the optional type declaration and machine-code compiler provided by implementations of Lisp achieves this.

Let's consider a basic and overly simple example in which we implement the sum function directly within a high-level language. The following are obvious implementations of this in both R and Python[1], also a dynamic, interpreted language without type specification but with byte-code compilation.

R

```
Sum =
function(x) {
  ans = 0
  for(e in x)
    ans = ans + e
  ans
}
```

Python

```
def Sum(x):
    ans = 0.0
    for i in x:
        ans = ans + i
    return ans
```

We are ignoring issues such as missing values (NAs), and of course, both systems provide built-in, compiled versions of the sum function. However, we are interested in using this elementary example that focuses on scalar computations to compare the performance of our implementations written in the language with a similar implementation in Lisp.

We used a vector of length 100,000 and computed its sum 10,000 times to compare the relative performances of our two implementations above with the built-in ones and also a similar implementation Lisp. We also implemented and measured equivalent code in Java and C and explored different ways to compute the result in both Lisp and Python, i.e. using the general reduce function in both systems. The example is sufficiently small and we want to compare the naïve, obvious implementations, so we did not spend much time optimizing the code. The results are given in Table 1.

Python's built-in `sum` is much slower than R's built-in function because, while both are written in C, the Python code accepts generic, extensible Python sequences and must use generic dispatch (at the C-level or perhaps

[1] A potential point of confusion is that the compiler module within CMU Common Lisp is called Python and predates the programming language Python.

Implementation	*Time*	*Performance factor relative to slowest*
R interpreted	945.71	1
Python interpreted	385.19	2.50
Python reduce() function	122.10	7.75
Lisp no type declarations	65.99	14.33
Python built-in sum()	49.26	19.20
R built-in sum()	11.2	84.40
Lisp with type declarations*	2.49	379.80
Java	1.66	569.70
C	1.66	569.70

Table 1. Execution time (in seconds) of the summation of a vector of size 100, 000 repeated 10, 000 times. In all but the case of the call to R's built-in sum() function, there is no test for `NA`s. These measurements were taken on a Linux machine with a 2.4Ghz AMD 64 bit chip and 32 GB of RAM. *We also performed the experiments on an Intel Mac (2.33Ghz, 3Gb RAM) and the results were similar, but the actual values were quite different for some situations. The built-in R sum() took only 2.68 seconds and so is much more similar to Lisp which took 1.85 seconds on that machine. The Java code was 3 times slower than the C code.

to a Python function) to fetch the next element of the sequence and then similarly for adding the number to the total. While it is reasonable to point out that both the Python and R built-in functions are more general than the compiled lisp function in that they can handle arbitrary sequences and numeric and integer vectors respectively, this objection has one serious flaw. While the Lisp function has been limited to vectors of double-float elements, Lisp allows us to declare these limitations; R and Python do not. We can easily create a collection of specialized, fast sum functions for other data types in Lisp, but we cannot in R and Python. This optimization is not available to us in R and Python.

The timings show that the simple implementation entirely within Lisp is essentially as fast as R's C routine, taking into account that the latter tests for `NA`s. What is also informative is the factor of 35 between the Lisp code that has just two type declarations and the version that has none; optional type declarations are effective. But the important comparison is between the type-declared Lisp version and the equivalent version written entirely in both R and Python. Here we see that the Lisp version is 380 times faster than R and 150 times faster than Python.

Over the last several years, Luke Tierney has been making progress on byte-code compilation of R code. His results indicate an improvement of a factor between 2 and 5 (Tierney (2001)). Luke Tierney has also been experimenting with using multiple processors within the internal numerical computations done by R. This has the potential to speed up the code, but will yield, at best, a factor given by the number of available processors. Further,

this work would need to be done manually for all functions and would not directly apply to user-level code.

The timing results illustrate that the optimized Lisp code runs about 30% slower than optimized C code. Clearly, if the majority of the computations in the high-level language amount to calling primitives written efficiently in C, then this 30% slow-down will lead to an overall slow-down and the resulting system will be a potential step-backwards. However, we can of course implement such primitives ourselves in C and use them from within Lisp. But more importantly, we do not believe that these primitives form the majority of the operations, and further that copying objects is a large contributor to performance issues in R. While vectorized operations are fundamental, they are not relevant to the many common computations which cannot be readily vectorized. And the primary message from this section is that when we implement an algorithm that deals with individual elements of a vector in the high-level language, the gains in the Lisp approach are immense. Some Lisp implementations are not slow and the language is viable for high-performance computing. Furthermore, the gains in speed are available incrementally along a continuum ranging from an initial version that is subsequently annotated with increasing amount of information about the types of the data/variables. So the improvement in run-time will also be frequently accompanied by gains in development time as we don't have to switch to another language (e.g. C) to obtain the necessary performance improvements.

5 Actual examples

5.1 Reinforced random walk

Motivated by a research problem of a colleague, we simulated a simple discrete two dimensional reinforced random walk. This is a random walk in which the transition probabilities of moving North, South, East or West from the current position are a function of the number of times the walk has previously visited the current spot. In our simulation, the probability of moving East if this was the second or greater time we had visited the current location is $(1+\beta)/4$ and the probability of moving West is $(1-\beta)/4$; all other probabilities are $1/4$.

This is a potentially expensive simulation as we must keep a record of how often each location has been visited, and further we need to be able to quickly determine the number of times we have visited the a particular position. The choice of data structure and algorithm for computing this is important for the efficiency of this algorithm. We use a hash table with the location as a key (in Lisp the object can be used directly, but in R, we must create a string from the x, y pair). Furthermore, since this is a Markov process, it is not readily vectorized.

We implemented the algorithm in both R and Lisp using the same algorithm. With $\beta = .5$, we ran $100,000$ steps of the random walk on several

different machines. The execution times for 3 different machines are given below. (the times are in seconds).

Lisp	R	Machine characteristics
0.215	6.572	2.33Ghz/3GB Intel, Mac OS X
0.279	7.513	2.4Ghz/32GB AMD Opteron, Linux
0.488	8.304	1Ghz/2GB AMD Athlon, Linux

So we see a significant benefit from using Lisp, with a speedup of a factor ranging from 17 to 30.

The person interested in doing these simulations proposed looking at 50 different values of β and performing $10,000$ random walks, each of length $10,000$. The goal is to look at the distributions of both the drift and the standard deviation of the walk. On the Intel Mac laptop, the R version takes .75 seconds for $10,000$ iterations. 50 replications of this takes 39.912 seconds, and 100 takes 80.213 seconds. So this is close to linear and $10,000$ replications of $10,000$ iterations would take at least 133 minutes. And to do this for 50 values of beta would take at least $4\frac{1}{2}$ days! This assumes that the computation will complete and not run out of memory.

The Lisp version takes 212.9 seconds for $10,000$ iterations of $10,000$ steps for a given β. So for 50 values of β, the expected completion time is 3 hours in total.

5.2 Biham-Middleton-Levine traffic model

We also implemented Biham-Middleton-Levine traffic model in both R, with computationally intensive parts written in C code that are called from R and in pure, type declared Lisp code. The results again indicate that the Lisp code out-performed the combination of R and C code. While both R implementation could be further optimized, a reasonable amount was done using profiling in R and then recoding the bottlenecks in C.

6 Syntax

Lisp is a powerful computing language which provides a rich set of resources for programmers. Despite this, many programmers have difficulty with it because of its syntax. The S expression

```
sum(x)/length(x)
```

is represented in Lisp by the "s-expression"

```
(/ (sum x) (length x))
```

It is our intent to provide a thin layer of syntax over Lisp to provide a comfortable environment for carrying out data analysis. Although we intend

to change the appearance of Lisp, it is important that the layer which does this be as thin as possible. This would make it possible for users to work in Lisp, should they choose to do so. This would make the applications developed in the framework useful to the Lisp community as well as to statisticians.

There are a number of ways in which the syntax layer could be implemented. A standard LALR parser generator is available and this could be used to translate an S-like syntax into Lisp. As an alternative, we (together with Brendan McArdle of the University of Auckland) are examining the use of a PEG (parsing expression grammar) based parser. Such parsers provide the ability to extend the grammar at run-time which is useful for experimentation.

The syntax of the language is not yet finalised, but we would expect that a simple function definition such as the one below on the left would be translated to a Lisp form given on the right.

```
defun sum(x)
{
  local s = 0
  do i = 1, n {
    s = s + x[i]
  }
  s
}
```

```
(defun sum (x)
  (let ((s 0))
     (doloop (i 1 (length x))
       (setf s (+ s (elt x i))))
     s))
```

Here, `doloop` and `elt` are Lisp macros which implement a Fortran-style do-loop and 1-based element access for vectors.

Adding declarations to the original code would simply add corresponding declarations to the Lisp code. The annotated version of sum function above is given below on the left and the Lisp translation on the right.

```
defun sum(double[*] x)
{
  local double s = 0
  do i = 1, n {
    s = s + x[i]
  }
  s
}
```

```
(defun sum (x)
  (declare
    (type (simple-array double (*))
          x))
  (let ((s 0))
       (declare (type double s))
    (doloop (i 1 (length x))
      (setf s (+ s (elt x i))))
    s))
```

In fact, we will probably use macros to provide specialized versions for the different data types from a single “template”.

7 Other issues

Memory consumption & copying: As we have mentioned, the pass-by-value semantics of R impose a significant performance penalty. Moving to a pass-by-reference approach would avoid this but involve a very different style of programming. For common, interactive use this may not be desirable, but also would not be a significant issue. For more computationally intensive tasks and "production analyses", the approach may be very beneficial. So too would be a computational model that facilitated working on data sets record at a time or in blocks. This approach has been used very effectively in SAS, for example. We plan on making streaming data and out-of-memory computations a significant part of the fundamental framework. By combining pass-by-reference with a flexible, fast programming language and data delivery mechanism for streaming data, we expect that statisticians and others can use the same tools for interactive, exploratory data analysis and intensive, production-level data processing and mining tasks.

Parallel computing: Parallel computing using multiple cores executing code concurrently with shared memory is becoming increasingly important. Many statistical methods are "embarrassingly parallel" and will benefit greatly from such facilities. Thus, we want to be able use a high-level language to express parallel algorithms. Progress on this front has been slow in R for various reasons. By adopting another community's engine, i.e. SBCL or Allegro Lisp, we inherit much of the work that is already done to provide user-level parallel facilities which are close to completion for the different platforms. Additionally, some of the commercial vendors of Lisp platforms have rich thread support. Further, we expect that there will be advances in compiler technology in general, and implemented in Lisp systems, for identifying and automating aspects of parallelism that we are unlikely to achieve within the statistical community alone.

Backward compatibility?: The R community is already large and growing. There are over 1000 contributed R packages on CRAN (`www.r-project.org`), 150 from BioConductor (`www.bioconductor.org`) and 40 from Omegahat (`www.omegahat.org`). It is a not a trivial decision to embark on building a new system and losing access to this code. So backward-compatibility is an important early decision. We could attempt to re-implement R on a Lisp foundation and this would likely lead to improvements in performance. However, we feel that it is better to move to a new computational model. But, we might still implement a Lisp-based R interpreter that can run concurrently within the Lisp session and can interpret R code. Alternatively, we can develop a translator that converts R code to a Lisp equivalent. And an additional approach is to embed R within Lisp so that we can call R functions directly from within Lisp or our new language. rsbcl (Harmon (2007)) already

provides this interface and allows us access to arbitrary R functionality. We are exploring these different approaches to reusing R code.

Extensibility: The R interpreter is written in C and there is sharp divide between R-language code, interpreted code and the system itself. The fundamental internal data structures are compiled and fixed. With a system written using Lisp, however, we are working in a language that performs run-time compilation to machine code. There is no divide between the "interpreter" and the user-level language. This means that users can introduce new "core" data types within their code and they can be used in the same manner as the core data types provided by our "new" environment. This extensibility allows others outside of the language developers to perform new experiments on the system itself and to disseminate them to others without needing to alter the system. This gives us a great deal of flexibility to handle new tasks and explore alternative approaches to computing. This is also important if we are to foster research on the topic of statistical computing environments themselves, which is necessary if statistical computing is to continue to evolve.

8 Alternative systems

If we are prepared to build a new system, an obvious question is why choose Lisp as the underlying language/environment. Python is becoming increasingly widely used and supported. There is a great deal of advanced design in the upcoming Perl 6/Parrot environment. And each of Perl, Python and Java have extensive add-on modules that are of interest to the scientific and statistical communities.

Each of these systems is a worthy choice on which to build a new system. All are compiled languages in the sense of creating byte-code that is executed on a virtual machine. Java has just-in-time compilation (JIT) which gives it performance comparable to code compiled to machine-instructions. But this is what Lisp provides transparently. And Lisp provides optional type checking, whereas Java *requires* type specification and Python and Perl do no permit type specification (in the standard language). While Java is potentially very fast, its focus on secure code execution and hence array-bound checking introduces a significant overhead for scientific/numerical computing.

The timing results in section 4 indicate that a good Lisp implementation outperforms each of these other higher-level languages. While most of these are more popular than Lisp, we think it is important to engage in ambitious work with greater potential to improve statistical computing and its availability for, and impact on, scientific computing.

We could use a low-level language such as C++ and this would provide us with a potentially better foundation than we currently have in the C-based code underlying R. However, we would still be in the situation of owning our own interpreter and so be responsible for every detail, both now and

in the future. We could build on top of projects such as Root (Brun and Rademakers (1997)), which provides an interactive C++-like language that provides direct access to C++ libraries but we believe that there is a greater benefit to using a rich high-level language which is compiled to machine code rather than an interactive, interpreted C-based language.

Having chosen Lisp, we could elect to use one of the existing statistical systems based on Lisp, i.e. XLisp-Stat, Quail. The choice of Common Lisp will allow us to run the code on any of the standard Common Lisp implementations. For us, the main attraction is the presence of a good, high-performance machine-code compiler. If the code for these systems can be deployed on such a Common Lisp implementation and is not tied to a particular implementation of Lisp, then we will use it (license permitting). Otherwise, it is opportune to design the environment anew with fundamental support for more modern advanced data analysis, e.g. streaming data with out-of-memory algorithms.

9 Conclusion

The statistics community needs to engage in developing computing infrastructure for the modern and future challenges in computationally intensive, data rich analyses. The combination of run- and development-time speed and memory usage is important, and a language that supports optional/incremental type specification helps in both compilation and good programming, while enabling interactive use. We are pursuing Common Lisp, with its several high-performance implementations to develop a framework on which to implement a new statistical computing environment. And in this new development, we are seeking to build in at a fundamental level different, modern computing paradigms (e.g. streaming data and out-of-memory/record-at-a-time algorithms).

Starting the development of a new computing environment using Lisp is not a guaranteed success. Lisp is not a widely used language within the statistics community. And to a large extent, many people are content with their existing environments. This is a long-term project and we are also hoping to engage new and different additional communities and to benefit from their knowledge and activity.

By putting an R-like syntax on Lisp, we feel that the obvious benefits of Lisp can become accessible to a community in need of them, and allow software developed by statisticians to be used in real, high-performance applications.

References

BRUN, R. and RADEMAKERS, F. (1997): ROOT - An Object Oriented Data Analysis Framework, *Nucl. Inst. & Meth. in Phys. Res. A*, 389, 81–86. (Proceedings AIHENP '96 Workshop.)

HARMON, C. (2007): rsbcl - An Interface Between R and Steel Bank Common Lisp. Personal communication.

TIERNEY, L. (2001): Compiling R: A Preliminary Report, *DSC 2001 Proceedings of the 2nd International Workshop on Distributed Statistical Computing.*

http://www.springer.com/978-3-7908-2083-6

COMPSTAT 2008
Proceedings in Computational Statistics
Brito, P. (Ed.)
2008, XVII, 573 p. With CD-ROM., Softcover
ISBN: 978-3-7908-2083-6
A product of Physica-Verlag Heidelberg