Lessons Learned
Directions for the Future

Ross Ihaka
University of Auckland

Primary R Developers

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

Plus a supporting cast of thousands.

Primary R Developers

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

Plus a supporting cast of thousands.

Outline

- How we got to where we are
- Where we need to be
- Future directions for research
- Some current meta-issues

Early R

- 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 and David Betz).
- This provided a platform for experimentation and extension.

R Version GC-13

```
> (define square (lambda (x) (* x x)))
square

> (define v 10)
v

> (square v)
100
```

Present R

- The S-like appearance of R was added incrementally.
- Similarity to S was driven by the desire to access already existing programming expertise and code.
- As R became more S-like the move towards making it *S compatible* became irresistible.
- This ultimately produced the present mature and widely-used system.
- Because R now has a large number of users who require a stable platform for getting work done, it is no longer suitable as a base for experimentation and development.

What R Provides

- An interactive, extensible, vectorised language.
- A large run-time environment which provides a good deal of statistical functionality.
- Good (static) graphics capabilities.
- Community support mechanisms.
- The freedom to inspect, modify and redistribute the source code.

R's Limitations

- R is not very good at handling large-scale problems.
- The following present particular difficulties.
  - Execution of large amounts of R code.
  - Scalar (element-by-element) computations.
  - Computations on large volumes of data.
- Some computational problems involve a mix of all of these.

An Example - Updating a Data Frame

- A problem encountered by a colleage required updates to corresponding elements of a collection of vectors.
- A natural way to do this was to store the variables in a data frame and to update the rows of the data frame.

```
df[i,] = new.row
```

- The computation was running very slowly.

Row-Wise Dataframe Updates

Hold the variables to be updated in a data frame.

```
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
    }
})
```

Run time: 100 seconds.

Multiple Vector Updates

Update the variables individually.

```
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]
    }
})
```

Run time: .2 seconds
(500 times faster than for the dataframe.)

What We (Will) Need To Deal With

- Multi-gigabyte data sets are now commonplace.
- Terabyte data sets are seen with increasing frequency.
- Petabye data sets are now beginning to appear.
- Statistical techniques are increasingly computationally intensive.
- To handle this we will need orders of magnitude increases in performance over what R (and other interpreters) can provide.

What Can We Do?

- Wait for faster machines.
- Introduce more vectorisation and take advantage of multicores.
- Make changes to R to eliminate bottlenecks.
  - Compilation.
  - Use non-copying semantics.
- Sweep the page clean and look at designs for new languages.
- Duncan Temple Lang, Brendan McArdle and I have begun examining what such new languages might look like.

Basic Language Speed I, Compilation

- R is an *interpreted* language.
- Using compilation into bytecode or machine code should speed up the language.
- A guess at how much the speed up will be is somewhere between a small multiple and an order of magnitude.
- Certain R language elements (`eval`, `get`, `assign`, `rm` and *scoping*) 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.

Basic Language Speed II, Scalar Types

- R is very slow at scalar computations.
- This produces limitations on the type of computations R is useful for.
- Example: Simulation of Markov chains and AR processes is inefficient because it cannot be vectorised.
- The limitation could perhaps be eliminated by introducing scalar data types.
- This would avoid the boxing and unboxing costs associated with using aggregate types (e.g. vectors) for scalar computations.

Basic Language Speed III, Avoiding Copying

- R uses *pass-by-value* semantics.
- This means that functions do not operate directly on their arguments, but rather on copies of the arguments.
- This can be very inefficient (e.g. model matrix copying in `lm` etc).
- This is one reason that the row-wise dataframe update process is so slow.
- Moving to *pass-by-reference* semantics should produce efficiency gains and make it possible to handle much larger problems.

Compiler Smarts I, Type Declarations

- 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.

Compiler Smarts II, Code Transformation

- Naive evaluation vector expression 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 (Scholz et al) provides an example of what can be done.

Building a New Language

- Given that we can determine suitable technologies, building a new language high-performance language is possible.
- Building such a language and a computational environment based on it will take time, but we have a model for how to go about the process.
- There are meta issues that need to be addressed.
  - How can development be supported?
  - How can the rights of contributors to the project be protected?