Proposals for Extending the R Language

Radford M. Neal, University of Toronto

Dept. of Statistical Sciences and Dept. of Computer Science

`http://www.cs.utoronto.ca/∼radford`

Some syntactic sugar (or maybe more than that...?)

- For any non-S4 object, `x`, make:

  `x@fred` equivalent to `attr(x,"fred")`

  `x@fred <- v` equivalent to `attr(x,"fred") <- v`

- For any matrix, `X`, make:

  `X$fred` equivalent to `x[,"fred"]`

- In any function call, make:

  `fun (=x, =abc, =pq)` equivalent to `fun (x=x, abc=abc, pq=pq)`

  Having to type the names again is not only tedious, it is also a potential source of errors. This change is also synergistic with some of my later proposals.

A New Sequence Operator that Operates Correctly

Problem: Using `i:j` to create an increasing sequence does not produce a zero-length sequence when `j` is less than `i`. This is very annoying, and leads to buggy code. Using **`seq_len`** is clumsy and not sufficiently general.

Another problem: `1:n-1` does not start at `1`.

Solution: A new operator, which produces only increasing sequences, including zero-length ones, and which has lower precedence than the arithmetic operators.

Question: What should be the name of the new operator?

It's not a trivial question. We need to make ":" obsolete, but retain it for compatibility, so we can't redefine it. But the new operator won't take over from ":" if its name is ugly and/or hard to type.

Examples using possibilities I've considered but don't like:

```
for (i in 1 %:% n-1) A[i, i %:% i+1] <- 0
for (i in 1 :> n-1) A[i, i :> i+1] <- 0
```

Proposal: Call the New Sequence Operator “..”

Examples of its use:

```
for (i in 1..n-1) A[i, i..i+1] <- 0

v[1..n] <- A[1..n,i]

if (any(v<i..j)) stop(...)
```

But... `i..j` is a valid symbol!

Yes. We’d need to disallow symbols with consecutive dots (except at the beginning of the symbol, so “`...`” and “`..1`” would still be legal).

Does anyone use symbols with “`..`” in the middle? I hope not.

It would be good (anyway) to encourage use of underscores rather than dots in symbols (except for S3 method names). I think the expression `i..max_pens` looks better than `i..max.pens`, even though the latter is unambiguous in this proposal.

Stopping Inadvertent Dimension Dropping

Problem: We want to create a sub-array of `A` with all its columns, but only those rows whose indexes are in `v`. We try to do that with `A[v,]`.

It usually works, but we get a vector rather than a matrix if either `v` has length one or `A` has only one column. So there's lots of buggy code. Adding `drop=FALSE` everywhere works, but is very tedious and unreadable.

Start of a solution:

First, define "`_`" to be a special object that selects all of a dimension without dropping it, even if the dimension is one. Writing "`_`" is also clearer than writing nothing.

Second, don't drop a dimension if the index is a 1D array, even if it is of length one. This might break some existing code, but probably very little.

Result: Now `A[array(v),_]` always produces a matrix.

Make the New Sequence Operator Produce a 1D Array

Many of the vectors used to index arrays are produced by a sequence operator. We can define the new sequence operator to produce a 1D array, so we don't have to use `array`.

Now, `A[1..n,_]` produces a matrix with one row when `n` is one, and a matrix with zero rows when `n` is zero.

Similarly, `A3[1..n,1,1..m]` delivers a 2D matrix even when `n` and/or `m` is zero or one. Note that adding `drop=FALSE` would not solve the problem here, since it would always produce a 3D array.

An Unfortunately Impossibility: Zero-Length Vectors Can't Contain Negative Elements

Problem: If `ix` is a vector of positive integers, `v[-ix]` gives a vector with all the elements of `v` except those in `ix`.

Well, almost. Unfortunately, it doesn't work when `ix` is of length zero!

Solution: Define a function `exclude(ix)` that returns `-ix`, except that it returns `_` when `ix` has length zero, and it returns a zero-length vector when `ix` is `_`.

Now `v[exclude(ix)]` works correctly.

It's maybe clearer too. Also, if we make it an error to pass `exclude` a vector with zero or negative elements, bugs will be found more easily.

Closed Functions

Problem: Sometimes a function inadvertently references a global variable when it meant to reference an argument (that isn't actually present, or has the wrong name). Sometimes such functions seem to work for a while.

Solution: Allow functions to be declared as "closed":

```
func <- function (x, y, a) : closed
{   c <- x*y
    c (exp(c), c^a)
}
```

Note that it's not necessary for "closed" to be a reserved word.

Symbol lookups inside a closed function don't look in the environment enclosing the function — except for lookups of functions (eg, `exp` above). Note that function lookup is already special (eg, `c` above).

Perhaps in some implementations there might be an efficiency gain from using closed functions.

Default Arguments in Closed Functions

I propose that expressions giving default values for arguments of a closed function *are* allowed to reference symbols in the enclosing environment. This would allow selective "importing" of the values of global variables.

My earlier proposal that an argument name in a function call can default to a symbol to the right of "=" could be applied to function definitions too. We could then import globals into a closed function as below:

```
gunk <- function (x, y, =xpow, =ypow) : closed
{   sum(x^xpow) + sum(y^ypow)
}
```

Of course, a call of `gunk` could override the default use of the global `xpow` and `ypow` if desired.

A variation: To give easy access to standard globals like `pi`, maybe lookup to outside a closed function should go directly to the base environment, rather than not going outside at all.

Beyond Call-by-Value

R's call-by-value semantics for function arguments is a serious limitation in some situations.

Example: Writing functions to update a large data structure.

Options: Store the data structure in a global variable, use `<<-`.
Pass the data structure as an argument, return an updated structure.

The first is inflexible, the second inefficient.

Even just accessing a large data structure passed as an argument is currently inefficient, since it flags the structure as needing to be copied when next updated. (Cleverer implementations might usually avoid this.)

Kludges such as putting the structure in an environment and then passing the environment aren't very appealing.

Allowing arguments to be passed by something other than call-by-value seems attractive. I propose the "call-by-name" mechanism of Algol 60.

Read-Only Call-by-Name

Read-only call-by-name should be very easy to implement in R — just keep on evaluating the "promise" rather than saving the value from the first evaluation. The effect is more-or-less as if the actual argument replaced all occurrences of the formal argument within the function.

Here's what a declaration might look like:

```
access_fun <- function (@struc, i)
{   struc$fred[i] + struc$sally[i]
}
```

Here are two calls:

```
access_fun (@a_struc, 13)
access_fun (@lis$st, i+j)
```

For call-by-name to be used, both the formal argument and the actual argument must be marked with "`@`". Otherwise call-by-value is used.

More Complex Call-by-Name Arguments

Complex call-by-name arguments are allowed, but may not be a good idea.

An example definition and two calls:

```
sum_corners <- function (@mat)
{   nr <- nrow(mat); nc <- ncol(mat)
    mat[1,1] + mat[1,nc] + mat[nr,1] + mat[nr,nc]
}
print (sum_corners(@lis[[f()]]$X))
print (sum_corners(@M[i..j,k..l]))
```

For the first call, `f()` will be evaluated six times (conceivably returning different values each time). Leaving the "`@`" out of this call might be better.

For the last call, the reference inside `sum_corners` to `mat[1,1]` would be treated like `M[i..j,k..l][1,1]`. This is inefficient, if implemented the obvious way. `M[i..j,k..l]` would be copied six times when evaluating `sum_corners(@M[i..j,k..l])`, though there would be no need to copy all of `M` (either immediately or when `M` is next modified).

Assigning to Call-by-Name Arguments

One could treat assignment to a call-by-name argument as turning it into a local variable, as for assignments to arguments in R at present.

But to be able to write update functions, one needs, as in Algol 60, to treat the assignment as if the the occurrence of the argument on the left side were replaced by the actual parameter.

An example definition and calls:

```
increment_top_left <- function (@mat)
{   mat[1,1] <- mat[1,1] + 1
}
increment_top_left(@M)
increment_top_left(@lis$X)
increment_top_left(@M+1)
```

The last call would give an error, just as `(M+1)[1,1] <- (M+1)[1,1] + 1` would give an error. A call without “`@`” would (pointlessly) increment element `[1,1]` of a local copy. (Or maybe it should signal an error?)

Default Call-by-Name Arguments for Closed Functions

Call-by-name argument passing would allow a closed function to “import” global variables for writing as well as reading.

Example:

```
update_fun <- function (i, v, =@big_data) : closed
{ big_data[i] <- v
  big_data[i+1] <- v^2
}
update_fun(12,3.1)
```

Here, `=@big_data` is an abbreviation for `@big_data=@big_data`.

The call-by-name argument with default allows references to `big_data` that act like references to a global variable (except that `<-` is used rather than `<<-`). Of course, there is still the flexibility to override the default `big_data` argument when calling `update_fun`.