Profiling and debugging R code

Kylie Ariel Bemis

5 February 2020

Motivating example

How does R treat objects that are being modified?

Consider:

```r
x <- runif(10)
x[5] <- 10
x
```

```
##  [1]  0.8701411  0.7106866  0.2424019  0.5761244 10.0000000
##  [7]  0.9392003  0.2634997  0.9050442  0.8513078
```

Understanding memory and allocation in R

What happens to x?

```
x <- runif(10)
x[5] <- 10
x
```

```
##  [1]  0.6253091  0.6675531  0.2422563  0.1693882 10.0000000
##  [7]  0.1037115  0.2948248  0.9795762  0.2083172
```

There are two possibilities:

- x is modified in place
- R copies x and modifies the copy

Reference tracking in R

We can use `refs()` to understand whether R thinks an object needs to be copied or not.

```r
library(pryr)
```

```r
x <- runif(10)
address(x)
```

```
## [1] "0x7fada90a59b8"
```

```r
# [1] "0x7fcd419c21c8"
x[5] <- 10
x
```

```
##  [1]  0.6494538  0.6686644  0.1503944  0.6284494 10.0000000
##  [7]  0.9861735  0.1944980  0.4890946  0.1655930
```

```r
address(x)
```

```
## [1] "0x7fada89d1cb8"
```

What if we create another reference to `x`?

```
x <- runif(1)
c(address(x), refs(x))
# [1] "0x103100060" "1"

y <- x
c(address(y), refs(y))
# [1] "0x103100060" "2"
```

Now there are two references to the value of `x`.

```r
x <- runif(10)
y <- x
c(address(x), address(y))
```

```
## [1] "0x7fadab1235b8" "0x7fadab1235b8"
```

```r
x[5] <- 6L
c(address(x), address(y))
```

```
## [1] "0x7fadaa04a078" "0x7fadab1235b8"
```

In this case a copy of `x` is made, to avoid affecting `y`.

Another useful function is tracemem(). It prints a message every time the traced object is copied:

```r
x <- runif(10)
# Prints the current memory location of the object
tracemem(x)
# [1] "<0x7feeaaa1c6b8>"

x[5] <- 6L

y <- x
# Prints where it has moved from and to
x[5] <- 6L
# tracemem[0x7feeaaa1c6b8 -> 0x7feeaaa1c768]:
```

Figuring out when an object gets duplicated is complicated by the fact that most functions (any that evaluate their arguments, to be specific) will trigger an increment of the reference count.

```
x <- runif(10)
c(address(x), refs(x))
# [1] "0x103100060" "1"

identity(x)

c(address(y), refs(y))
# [1] "0x103100060" "7"
```

Primitive functions generally won't increase the reference count, which is why `[<-` can modify in-place without creating a copy.

Consider a simple use case where we modify each column of a data frame in a for loop.

```
x <- data.frame(matrix(runif(100 * 1e4), ncol = 100))
medians <- sapply(x, median)

tracemem(x)

for( i in seq_along(medians) )
    x[,i] <- x[,i] - medians[i]
```

Surprisingly, because the [<- method for data frames is a regular R function (i.e, non-primitive) a copy is made at each iteration.

The [<- method for lists is a primitive function, so it doesn't make a copy at every iteration.

```r
y <- as.list(x)

tracemem(y)

for(i in seq_along(medians))
    y[[i]] <- y[[i]] - medians[i]
```

While it's generally preferable to avoid creating unnecessary duplicates of objects, it can be surprisingly difficult to determine exactly when objects get copied.

Tips for avoiding duplication

- Use “primitive” replacement functions when possible
- Always pre-allocate a vector/ matrix before looping over it

```
+ Use `numeric()` or `matrix()` before the loop

+ Avoid using `c()` or `rbind()`/`cbind()` to append results
```

- Modify an vector/matrix as early as possible after creating it
- Use `tracemem()` to track when objects get copied and rewrite to avoid it

A note on reference counting

Currently, R does not use “true” reference tracking.

The `refs()` count of an object can only ever increase, and will never decrease. This leads to duplication in many places where it could be avoided if `refs()` were allowed to decrease.

This will change in R $>=$ 4.0. Future versions of R will use “true” reference counting that allows the `refs()` to decrease. Hopefully this means that many situations that result on duplication currently will no longer do so in the future.

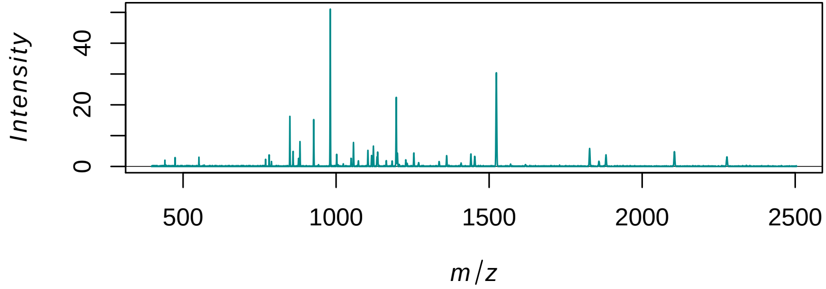
Most of the advice given here will still apply, but there may be fewer situations that trigger duplication in the near future.

Case study: finding local maxima

Suppose we want to find the local maxima of a vector, for use in a peak picking function.

```r
library(MSExample) # See "3-OOP" slides if you still need to ins
set.seed(2020) # set a seed for random number generator
s <- simSpectra(n=100, by=50, units="ppm") # simulate 100 mass s

plot(s[[1]])
```



Finding local maxima (version 1)

```r
locmax1 <- function(x, halfWindow = 2) {
    begin <- halfWindow + 1
    end <- length(x) - halfWindow
    out <- integer()
    if ( begin < 1L || end > length(x) )
        return(out)
    for ( i in begin:end ) {
        j1 <- i - halfWindow
        j2 <- i + halfWindow
        is_max <- TRUE
        for ( j in j1:j2 ) {
            if ( x[j] > x[i] )
                is_max <- FALSE
        }
        if ( is_max )
            out <- c(out, i)
    }
    out
}
```

Benchmarking `locmax1()`

The `microbenchmark` package is useful for accurately benchmarking functions that are intended to be called hundreds or thousands of times.

```
library(microbenchmark)

microbenchmark(locmax1 = locmax1(intensity(s[[1]])))
```

```
## Unit: milliseconds
##     expr      min       lq     mean   median       uq      max neval
##  locmax1 8.757202 9.533606 11.73233 10.46319 14.04444 25.55437   100
```

Not bad, but this function may get called thousands of times while processing thousands of spectra.

Can we do better?

Profiling `locmax1()`

We can use `Rprof()` to profile R code. `Rprof()` works by sampling at specific intervals and checking what function is currently being called.

To get enough data to profile performance, we either need to set a sufficiently small sampling interval, or use a function that runs for a sufficiently long time.

```r
Rprof(interval=0.005)
out <- sapply(s, function(si)
  locmax1(intensity(si)))
Rprof(NULL)
```

Summarizing code profiling

```r
summaryRprof()
```

```
## $by.self
##           self.time self.pct total.time total.pct
## "locmax1"     0.470    51.37      0.915    100.00
## "c"           0.445    48.63      0.445     48.63
##
## $by.total
##                        total.time total.pct self.time self.pct
## "locmax1"                   0.915    100.00     0.470    51.37
## "block_exec"                0.915    100.00     0.000     0.00
## "call_block"                0.915    100.00     0.000     0.00
## "eval"                      0.915    100.00     0.000     0.00
## "evaluate_call"             0.915    100.00     0.000     0.00
## "evaluate::evaluate"        0.915    100.00     0.000     0.00
## "evaluate"                  0.915    100.00     0.000     0.00
## "FUN"                       0.915    100.00     0.000     0.00
## "handle"                    0.915    100.00     0.000     0.00
## "in_dir"                    0.915    100.00     0.000     0.00
## "knitr::knit"               0.915    100.00     0.000     0.00
## "lapply"                    0.915    100.00     0.000     0.00
## "process_file"              0.915    100.00     0.000     0.00
## "process_group.block"       0.915    100.00     0.000     0.00
## "process_group"             0.915    100.00     0.000     0.00
## "rmarkdown::render"         0.915    100.00     0.000     0.00
## "sapply"                    0.915    100.00     0.000     0.00
## "timing_fn"                 0.915    100.00     0.000     0.00
## "withCallingHandlers"       0.915    100.00     0.000     0.00
## "withVisible"               0.915    100.00     0.000     0.00
## "c"                         0.445     48.63     0.445    48.63
##
## $sample.interval
## [1] 0.005
##
## $sampling.time
## [1] 0.915
```

Using `profvis`

The `profvis` package provides a nice way to visualize the output of `Rprof()`

```r
library(profvis)
profvis({
out <- sapply(s, function(si)
  locmax1(intensity(si)))
}, interval=0.005)
```

Finding local maxima (version 2)

```r
locmax2 <- function(x, halfWindow = 2) {
    begin <- halfWindow + 1
    end <- length(x) - halfWindow
    if ( begin < 1L || end > length(x) )
        return(integer())
    out <- logical(length(x))
    for ( i in begin:end ) {
        j1 <- i - halfWindow
        j2 <- i + halfWindow
        out[i] <- TRUE
        for ( j in j1:j2 ) {
            if ( x[j] > x[i] ) {
                out[i] <- FALSE
                break
            }
        }
    }
    which(out)
}
```

Finding local maxima (version 3)

```r
locmax3 <- function(x, halfWindow = 2) {
    begin <- halfWindow + 1
    end <- length(x) - halfWindow
    if ( begin < 1L || end > length(x) )
        return(integer())
    out <- vapply(begin:end, function(i) {
        j1 <- i - halfWindow
        j2 <- i + halfWindow
        if ( any(x[j1:j2] > x[i]) ) {
            FALSE
        } else {
            TRUE
        }
    }, logical(1))
    out <- c(rep(FALSE, halfWindow),
             out, rep(FALSE, halfWindow))
    which(out)
}
```

Verify functions

Before comparing our functions, we should make sure they're returning the same results.

```r
all.equal(locmax1(intensity(s[[1]])),
          locmax2(intensity(s[[1]])))
```

```
## [1] TRUE
```

```r
all.equal(locmax1(intensity(s[[1]])),
          locmax3(intensity(s[[1]])))
```

```
## [1] TRUE
```

Benchmark all versions

Which will be fastest?

```r
microbenchmark(locmax1 = locmax1(intensity(s[[1]])),
               locmax2 = locmax2(intensity(s[[1]])),
               locmax3 = locmax3(intensity(s[[1]])))
```

```r
microbenchmark(locmax1 = locmax1(intensity(s[[1]])),
               locmax2 = locmax2(intensity(s[[1]])),
               locmax3 = locmax3(intensity(s[[1]])))
```

```
## Unit: milliseconds
##     expr       min        lq      mean    median        uq        ma
##  locmax1  8.848769 10.088608 12.030649 10.958721 13.163781  32.24728
##  locmax2  3.772878  4.080353  4.707145  4.383595  4.964074   7.86117
##  locmax3 11.355587 13.345817 16.691295 14.367024 16.576744 101.27247
##  neval
##    100
##    100
##    100
```

Version 2 is fastest!

Despite the popular perception, `for` loops in R are not inherently slower than functions like `lapply()`, `sapply()` and `vapply()`.

Version 2 is able to update the output local maxima in place, while exiting the loop early as soon as we know the element is not a peak.

Performant code

It can be difficult to write high-performance code.

While R is sometimes criticized for being “slow”, profiling code can often reveal ways to improve performance dramatically. However, it’s not always obvious how to do so.

- Avoid premature optimization
  - Write the function correctly before trying to optimize it
  - Benchmark and profile your code before trying to optimize it
  - Our initial intuition of the “optimal” version is often incorrect!

Performant code (cont'd)

- Use functions that call efficient C code where possible
  - "Native" C code will always be faster than "interpreted" R code
  - Use functions that call C code rather than pure R implementations where possible
  - Recognized by functions with calls to `.C`, `.Call`, `.External`, `.Internal`, and `.Primitive` functions
- Use vectorized code wherever possible
  - Use functions that operate on whole vectors at a time
  - The `lapply()` family of functions are not "truly" vectorized – they provide a useful and convenient utility, but must still iterate over each element of the vector

Useful debugging tools

A major problem with programming is you almost never do something correctly the first time.

That's why debugging is important. R provides several tools for debugging.

- `traceback()`
  - Prints the stack at the time the error occurred
- `browser()`
  - Creates a pseudo-breakpoint that allows you to jump into code at a particular point
- `options(error=recover)`
  - Allows you to enter the `browser` anywhere on the stack after an error occurs
- `debug()` and `undebug()`
  - Enters `browser` at the beginning of a particular function call

Understanding the `traceback()`

Understanding *where* an error occured is generally the first step in debugging.

Examining the traceback can give us a hint of where the error occured.

```r
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) "a" + d
f(10)
```

```
## Error in "a" + d: non-numeric argument to binary operator
```

Either calling `traceback()` or clicking “Show Traceback” in RStudio will print the *call stack*.

Understanding the call stack

The call stack can be read from bottom to top:

```r
traceback()
# 4: i(c) at #1
# 3: h(b) at #1
# 2: g(a) at #1
# 1: f(10)
```

First `f(10)` was called, then `g(a)`, then `h(b)`, then `i(c)`.

This tells us that `i(c)` is where the actual error occured, but the cause of the error may have been earlier.

Sometimes knowing where the error occured is enough to reason what the cause of the bug, but often the traceback isn't enough on its own.

The debugger in R

Either manually adding `browser()` or using `debug()` on a function will put you in a debugging mode. RStudio provides a visual interface for stepping through the functions. These are also available as commands outside RStudio.

- `n` : Execute the next line
- `s` : Execute the next line and step into it if it's a function
- `f` : Finish execution of current loop or function
- `c` : Continue execution, exiting interactive debugging mode
- `Q` : Quit debugging mode without continuing execution

Using `options(error=recover)` also allows you to enter debugging mode anywhere in the stack after an error has occured.

Also note that if you need to debug warnings rather than errors, it is useful to convert them into errors using `options(warn = 2)`.

Use `options(warn = 0)` to return to the default behavior.

Debugging our call stack

Using `debug()` or `options(error=recover)` are the simplest ways to begin debugging.

We can debug `i()` specifically by using `debug()` or clicking “Rerun with Debug” in RStudio:

```r
debug(i)
f(10)
undebug(i)
```

Or we can jump into any function in the call stack after the error occured by using `options(error=recover)`

```r
options(error=recover)
f(10)
options(error=NULL)
```

Debugging S4 methods

S4 methods can be difficult to debug with `debug()`, because using `debug()` on the generic function will only debug the generic function, without letting you enter the actual method.

Use the `signature` parameter to define the signature for the S4 method you want to debug to debug the actual method instead.

```r
debug(plot, signature=c("MassSpectrum", "missing"))
```

Defensive programming

Debugging error-prone functions can be excessively difficult and is one of the most frustrating aspect of programming.

There are some “defensive programming” techniques you can make your life easier by avoiding common bugs and making your functions easier to debug when you do encounter bugs.

- “Fail fast” and “fail early”
  - It is better for a function to fail as soon as it receives bad input rather than propogate that bad input
  - Check whether the arguments passed to your function are valid values for those arguments – immediately `stop()` with an appropriate message if your input isn’t valid

Defensive programming (cont'd)

- Write short, simple functions that do one thing
    - Longer functions that do many things and handle multiple input types are more error-prone
    - Shorter, simpler functions are easier to jump into and `debug()`
- Avoid non-standard evaluation (NSE) when writing functions
    - Functions that use NSE like `base::subset()` or `dplyr::filter()` are convenient for top-level interactive data analysis, but can be extremely difficult to debug when used inside functions
    - Use simpler functions that use standard evaluation and ordinary subsetting operators like `[` and `[[`
- Avoid functions that may return different types of values
    - Use `lapply()` or `vapply()` rather than `sapply()`
    - Always specify `drop=FALSE` when subsetting with `[`
- Wrap error-prone code in `try()` or `tryCatch()`