Converting Between Value Oriented and Code Capturing Interfaces in R

by John Mount and Nina Zumel

Abstract

A number of popular R packages promote the use of non-standard or code-capturing function interfaces. The use of non-standard evaluation can produce concise and elegant code, especially in interactive situations. However, code produced in this style is difficult to parameterize or program over.

To address this issue, we describe R meta-programming tools from the **wrapr** package that easily convert between non-standard and standard interfaces. Our goal is to support popular programming paradigms in a more "R like" manner, supplying more good options for both interactive R users and package developers.

Introduction

Programming in R involves both interactive tasks where variable names and column names are known at the time of coding, and re-usable or parametric scripting, where variable names and column names are not known at the time of coding.

A typical interactive task might be printing a column from a data frame:

```
>   d <- data.frame(x = 1:3, y = 11:13, z = 21:23)
>   print(d$x)
```

```
[1] 1 2 3
```

A re-usable task may involve taking the name of the column to be printed from a variable:

```
>   colname <- "x"      # The value "x" may not be known until later.
>   print(d[[colname]])
```

```
[1] 1 2 3
```

For this paper we will call code of the form `d$x` the "code capturing," "name capturing," or "non-standard" (Wickham, 2014) interface style. And we will call code of the form `d[[colname]]` the "value oriented," "parametric," or "standard semantics" interface style.

Code capturing interfaces are more concise, and are convenient in interactive situations. Value oriented interfaces provide referential transparency (depend only on values), are easier to reason about, and are much easier to parameterize and program over. Value oriented interfaces are preferable when writing re-usable functions or packages.

For many situations the distinction between code capturing and value oriented interfaces is not an issue, as R typically supplies both interfaces. However, for some packages, for example **dplyr** (Wickham et al., 2017), one interface style is dominant.

The **wrapr** (Mount and Zumel, 2018) package provides tools for easily converting back and forth between the two interface styles. We will describe `wrapr::let()`, a function for re-adapting non-standard evaluation interfaces so one can script or program over them. With `let()` and its complementary function `qc()`, a package's preferred interface style becomes an inessential issue, as the conversion tools make either interface readily available. This note will emphasize how to use these tools in other scripts and packages.

Converting between conventions

An important code capturing interface in R is `"formula"`. When variables are known, creating and using a formula is straightforward, as in this linear regression example.

```
>     lm(z ~ x + y, data = d)
```

```
Call:
lm(formula = z ~ x + y, data = d)
```

```
Coefficients:
(Intercept)           x           y
         20           1          NA
```

When variable names are parameterized, creating formulas is less concise, but still relatively straightforward.

```
>    OUTCOMEVAR <- 'z'
>    INDEPENDENT_VARS <- c('x', 'y')
>    fmls <- paste(OUTCOMEVAR, '~',
+                  paste(INDEPENDENT_VARS, collapse = ' + '))
>    print(fmls)

[1] "z ~ x + y"

>    lm(as.formula(fmls), data = d)

Call:
lm(formula = as.formula(fmls), data = d)

Coefficients:
(Intercept)           x           y
         20           1          NA
```

In other words, R formulas are easy to construct either through the name capturing interface, or in a value oriented way via `paste()` and `as.formula()`.[1] Other R commands, such as `library()` and `help()`, directly implement both code capturing and value oriented interfaces.

For programming situations that are less straightforward, R supplies meta-programming facilities to assist with code capture, examination, and evaluation. These include `as.name()`, `bquote()`, `quote()`, `with()`, `substitute()`, `eval()`, and `do.call()`. Various R packages also supply additional meta-programming tools:

gtools Warnes et al. (2015) a package of tools to assist with R programming, including macro creation. A nice introduction to macros in R can be found here Lumley (2001).

lazyeval Wickham (2017) a non-standard evaluation package.

rlang Henry and Wickham (2017) a meta-programming system supplying "tidyeval" parsing and interpretation semantics, which are different than R semantics.

wrapr Mount and Zumel (2018) the meta-programming solutions we will explain in this note.

lazyeval and **rlang** are both meta-programming systems to allow value oriented programming over the highly popular **dplyr** and related packages. These packages all strongly depend on code capturing interfaces, and use either **rlang** or **tidyeval** to capture and reprocess arguments. Because this is an "inside the package" operation, users cannot activate this effect on packages not built using **rlang** or **tidyeval**.

`wrapr::let()` completes all substitutions before further code evaluation. Hence, users can apply `let()` without any prearrangement with package developers. The **wrapr** package tries to be "R-like": it adheres as much as possible to standard R tools, R semantics, and R conventions.

Using `wrapr::let()` to convert code capturing interfaces to value oriented interfaces

The inspiration for `wrapr::let()` comes from the Common Lisp `let()` facility (Steele, 1984). `let()` takes two arguments:

1. A mapping from the *stand-in* names in an expression to be executed to the *actual* names representing the final variable references. The mnemonic is: *stand-ins* are replaced by *actuals*. The mapping is specified by a named character vector.
2. The expression or expression block to be executed.

This is best shown with an example. Let us start with the following **dplyr** pipeline:

[1] Note that this `paste()` strategy obscures the content of the actual formula in `print.lm()` and `summary.lm()`. We will address this point again later.

```
>   library("dplyr")
>   d %>% mutate(x = x + 1)

  x  y  z
1 2 11 21
2 3 12 22
3 4 13 23
```

Suppose we want to replace the hard-coded column `x` with a column name coming from a variable. With `wrapr::let()` we write this as follows:

```
>   library("wrapr")
>   z <- "x"
>   let(c(colname = z),
+       d %>% mutate(colname = colname + 1)
+   )

  x  y  z
1 2 11 21
2 3 12 22
3 4 13 23
```

In the above code all uses of the stand-in "`colname`" are replaced with the actual ("`x`") stored in the variable "`z`" prior to execution. This is a bit clearer if we show the new code instead of executing it:

```
>   z <- "x"
>   let(c(colname = z),
+       d %>% mutate(colname = colname + 1),
+       eval = FALSE
+   )

d %>% mutate(x = x + 1)
```

`let()` is particularly useful when building functions by pasting existing ad-hoc code into a function block, requiring few edits and little clean-up. In the below example we show how the original ad-hoc pipeline "`d %>% mutate(x = x + 1)`" can be quickly abstracted to a re-usable parameterized function.

```
>   f <- function(d, z) {
+     let(c(x = z),
+       d %>% mutate(x = x + 1)
+     )}
>   f(d, "y")

  x  y  z
1 1 12 21
2 2 13 22
3 3 14 23
```

Implementing the same function using **rlang** requires additional modifications to the original pipeline. These modifications reduce code legibility, especially when the pipeline is non-trivial.

```
>   f2 <- function(d, z) {
+       x <- rlang::sym(z)
+       d %>% mutate( !!x := (!!x) + 1 )
+   }
>   f2(d, "y")

  x  y  z
1 1 12 21
2 2 13 22
3 3 14 23
```

While the amount of boiler plate code required by `wrapr::let()` is proportional to the number of stand-ins targeted for replacement, `wrapr::let()` often requires little to no change to the code being parameterized. The boilerplate can be made shorter by using the **wrapr**-supplied helper function `:=`[2]. Let's demonstrate this using the earlier `lm()` example.

```
>   symnames <- c("Y", "V1", "V2")
>   colnames <- c("z", "x", "y")
>   print(symnames := colnames)

  Y  V1  V2
"z" "x" "y"

>   let(symnames := colnames,
+       lm(Y ~ V1 + V2, data = d))

Call:
lm(formula = z ~ x + y, data = d)

Coefficients:
(Intercept)            x            y
         20            1           NA
```

Notice we have the additional benefit that the formula is completely legible in the above `print.lm()`. Note also that one cannot use **rlang** to directly perform the same substitution, as the **stats** package does not work with **rlang**:

```
>  Y <- rlang::sym("z")
>  V1 <- rlang::sym("x")
>  V2 <- rlang::sym("y")
>  lm(!!Y ~ (!!V1) + (!!V2), data = d)
Error in !Y : invalid argument type
```

Mixed case convention

To further improve code legibility, we recommend what we call the *mixed case convention*. In mixed case convention we use uppercase for stand-ins and lower case for variables containing the actuals.

```
>   colname <- "x"
>   let(c(COLNAME = colname),
+       d %>% mutate(COLNAME = COLNAME + 1)
+   )

  x  y  z
1 2 11 21
2 3 12 22
3 4 13 23
```

This convention allows the reader to distinguish the stand-ins from the actuals. This is particularly important when adapting code that is already using a mixture of code capturing and value oriented notation. Confusion between symbols and variables (and even values) is not just a problem for users, but part of the history of Lisp derived programming languages Harper (2012, 32.3 Notes, p. 268).

data.table (Dowle and Srinivasan, 2017) is a package that uses a mixture of code capture and value oriented interfaces. For example:

```
>   library("data.table")
>   f <- function(d, old_name, new_name) {
+    let(c(OLD_NAME = old_name,
+          NEW_NAME = new_name),
+       {
+         dT <- data.table::data.table(d)
+         setnames(dT, old_name, new_name) # this command uses values
+         dT[, NEW_NAME := NEW_NAME + 1]   # this command uses names
```

[2]`wrapr::`:=`` is an assignment operator implemented in the `wrapr` package. Due to package scoping rules the `wrapr` implementation will not interfere with other packages that use `:=` such as **dplyr** and **data.table**. **data.table**'s definition of `:=` can obscure **wrapr**'s implementation, so we suggest when using **data.table** to either load **wrapr** last or manually restore **wrapr**'s definition, as we show in a later example. Again, this will not interfere with `data.table` operation as **data.table** expressions are evaluated in the **data.table** package context.

```
+         withVisible(dT)
+       })}
>   print(f(d, "x", "new_x"))

$value
   new_x  y  z
1:     2 11 21
2:     3 12 22
3:     4 13 23

$visible
[1] TRUE
```

It was useful to have both the stand-in "NEW_NAME" and the variable "new_name" that holds the actual simultaneously available. Notice code sections that are longer than a single statement are placed in "{ }".

Using `wrapr::let()` in packages

Novel names such as NEW_NAME (from the **data.table** example in the section prior) superficially appear to be free or unbound symbols in the `let()` block. This can trigger warnings during package checks and inspections. This problem is not unique to **wrapr**; it also is seen with code capturing packages such as **ggplot2** (Wickham, 2009).

We suggest the following convention for working around this issue: assign a (not used) value to the free symbols prior to working with them. For example the "package hardened" version of the `data.table` example would be written as follows:

```
>   f <- function(d, old_name, new_name) {
+    OLD_NAME <- NULL # Indicate this is not an unbound symbol.
+    NEW_NAME <- NULL # Indicate this is not an unbound symbol.
+    let(c(OLD_NAME = old_name,
+          NEW_NAME = new_name),
+       {
+         dT <- data.table::data.table(d)
+         setnames(dT, old_name, new_name)
+         dT[, NEW_NAME := NEW_NAME + 1]
+         withVisible(dT)
+       })}
```

Alternatively, in some situations one can use what we call the *"x = x" convention*, where the stand-ins and the variables holding the actuals deliberately share names:

```
>   f <- function(d, x) {
+     let(c(x = x),
+       d %>% mutate(x = x + 1)
+     )}
>   f(d, "y")

  x  y  z
1 1 12 21
2 2 13 22
3 3 14 23
```

This convention has the advantages that there are no apparent unbound symbols to trigger package check warnings, and ad-hoc code can often be reused without any alteration. However, the resulting code is less explicit, and can be confusing to the novice. In addition, the "x = x" convention cannot easily be used in situations where we need access to both the stand-ins and the variables carrying the actuals, as in the **data.table** example above.

Using `wrapr::q*()` to convert value oriented interfaces to code capturing interfaces

`qc()` (quoting concatenate) performs a complementary task to `let()`: it converts value oriented interfaces into code capturing. It does this by capturing and quoting its arguments and returning them as a vector. With `qc()` we can use unquoted names with the first argument of `let()`.

```
>   # map := back to wrapr definition instead of data.table definition
>   `:=` <- wrapr::`:=`
>   print(qc(Y, V1, V2))

[1] "Y"  "V1" "V2"

>   let(qc(Y, V1, V2) := qc(z, x, y),
+       lm(Y ~ V1 + V2, data = d))

Call:
lm(formula = z ~ x + y, data = d)

Coefficients:
(Intercept)            x            y
         20            1           NA
```

wrapr also supplies additional quoting methods: `qae()` quotes assignment expressions, and `qe()` quotes expressions. Details and examples can be found in the package documentation.

Implementation discussion and details

Ideally we would like `wrapr::let(mapping,expression)` to behave as syntactic sugar for `eval(substitute(expression,mapping))`. In practice, we need different semantics, as we want to treat strings as names and we also intend to re-map left hand sides of function argument bindings. `substitute()` does not re-map left hand sides of function argument bindings, so it is not sufficient for our needs.

Usually one does not need to re-map left hand sides of function argument bindings, as function argument names are generally known at coding time (unlike argument values). However, some popular methods use named "`...`" arguments as if both the left and right hand sides were user controllable expressions. **dplyr** methods such as `mutate()`, `summarize()`, or `rename()` use such a convention. For example, even though the second argument of `mutate(d,y = x + z)` behaves as a column assignment, it is actually a function argument binding. For such code it is plausible a user may wish to replace the left hand side column name.

In our design we use `substitute()` to capture the unevaluated R language tree, but not to perform any substitution. We then walk the language tree recursively, replacing stand-ins with actuals.

The **gtools** `strmacro()` function is a text based macro generator that can also perform arbitrary left hand side substitutions. The **gtools** authors clearly saw the need for additional effects, as they implemented `strmacro()` *after* already implementing the `substitute()`-based `defmacro()`. `strmacro()` was the inspiration for the original string-based `let()` implementation, although `let()` now uses the more powerful and safer language based method described above.

Conclusion

We have demonstrated strategies for converting code capturing interfaces into standard value oriented interfaces, and vice-versa. The **wrapr** implementations are pure R, with low dependencies and without the use of external languages such as `C` or `C++`. The **wrapr** solutions are convenient, legible, and obey R semantics and conventions. We feel these tools address under-met needs, and will be of great value in writing maintainable R code for both R users and R package developers.

Bibliography

M. Dowle and A. Srinivasan. *data.table: Extension of 'data.frame'*, 2017. URL https://CRAN.R-project.org/package=data.table. R package version 1.10.4-3. [p4]

P. R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012. ISBN 1107029570, 9781107029576. doi: 10.1017/CBO9781316576892. [p4]

L. Henry and H. Wickham. *rlang: Functions for Base Types and Core R and 'Tidyverse' Features*, 2017. URL https://CRAN.R-project.org/package=rlang. R package version 0.1.6. [p2]

T. Lumley. Programmer's Niche: Macros in R. *R News*, 1(3):11–13, September 2001. URL https://www.r-project.org/doc/Rnews/Rnews_2001-3.pdf. [p2]

J. Mount and N. Zumel. *wrapr: Wrap R Functions for Debugging and Parametric Programming*, 2018. https://github.com/WinVector/wrapr, http://winvector.github.io/wrapr/. [p1, 2]

G. L. Steele. *COMMON LISP: the language*. 1984. ISBN 0-932376-41-X (paperback). With contributions by Scott E. Fahlman and Richard P. Gabriel and David A. Moon and Daniel L. Weinreb. [p2]

G. R. Warnes, B. Bolker, and T. Lumley. *gtools: Various R Programming Tools*, 2015. URL https://CRAN.R-project.org/package=gtools. R package version 3.5.0. [p2]

H. Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2009. ISBN 978-0-387-98140-6. URL http://ggplot2.org. [p5]

H. Wickham. *Advanced R*. Chapman & Hall/CRC The R Series. Taylor & Francis, 2014. ISBN 9781466586963. URL http://adv-r.had.co.nz. [p1]

H. Wickham. *lazyeval: Lazy (Non-Standard) Evaluation*, 2017. URL https://CRAN.R-project.org/package=lazyeval. R package version 0.2.1. [p2]

H. Wickham, R. Francois, L. Henry, and K. Müller. *dplyr: A Grammar of Data Manipulation*, 2017. URL https://CRAN.R-project.org/package=dplyr. R package version 0.7.4. [p1]

John Mount
Win-Vector LLC
552 Melrose Ave., San Francisco CA, 94127
USA
jmount@win-vector.com

Nina Zumel
Win-Vector LLC
552 Melrose Ave., San Francisco CA, 94127
USA
nzumel@win-vector.com