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 value oriented "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, and return the same value when variables are replaced with their referred 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. Our usual example is: once you need to write a `for`-loop (or `lapply`) over a function you will really come to wish it made a standard or value oriented interface available.

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), the non-standard interface style is dominant creating a need for additional techniques and tooling.

The **wrapr** (Mount and Zumel, 2018) package provides tools for easily converting non-standard interfaces into value oriented interfaces. We will describe `wrapr::let()`, a function for re-adapting non-standard evaluation interfaces so one can script or program over them. With `let()` a package's preferred interface style becomes an inessential issue, as the conversion tools make either interface readily available.

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.

```
>     d <- data.frame(x = 1:3, y = 11:13, z = 21:23)
>     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')
>    frm <- as.formula(paste(OUTCOMEVAR, '~',
+                paste(INDEPENDENT_VARS, collapse = ' + ')))
>    print(frm)

z ~ x + y

>    lm(frm, data = d)

Call:
lm(formula = frm, data = d)

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

R formulas are already easy to construct either through the name capturing interface " ", or in a value oriented way via `paste()` and `as.formula()` so "`let()`" substitution was not required in the above example.[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.

wrapr Mount and Zumel (2018) the meta-programming solutions we will explain in this note.[2]

rlang Henry and Wickham (2017) a meta-programming system supplying "tidyeval" parsing and interpretation semantics, which are different than R semantics.[3]

nseval Meilstrup (2018) a meta-programming system attempting to correctly expose correct R quotation and evaluation handlers, without routing through mismatching S syntax and semantics.

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**. For more on the relative capabilities and priority of these packages please see our own note Mount (2018).

`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. **wrapr** is also a pure-R package in that it include no C/C++/FORTRAN code and uses only public R methods and interfaces.

[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.

[2] `let()` was originally publicly announced and shared December 3rd, 2016, http://www.win-vector.com/blog/2016/12/parametric-variable-names-and-dplyr/, https://github.com/WinVector/replyr/commit/f6a51f8b21af29aede4aad4f7bb9b682c6147992 it was made further available through CRAN December 9th, 2016 https://cran.r-project.org/src/contrib/Archive/replyr/.

[3] **rlang** was released to CRAN May 5th, 2017; well *after* `let()` (https://cran.r-project.org/src/contrib/Archive/rlang/).

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

The inspiration for `wrapr::let()` comes from the Lisp `let()` facility (Landin, 1965; Steele, 1984; Culpepper and Felleisen, 2010). `wrapr::let()` takes two arguments:

1. A mapping from the *stand-in* or *target* names to the *actual* names representing the final variable references. The mnemonic is: *substitution targets* are replaced by *actuals*. The mapping is specified by a named character vector with targets as keys, and actuals as values.
2. The expression or expression block to be transformed and executed.

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

```
>   library("dplyr")
>   data.frame(x = 1, y = 5) %>% mutate(x = x + 1)

  x y
1 2 5
```

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),
+       data.frame(x = 1, y = 5) %>% mutate(COLNAME = COLNAME + 1)
+   )

  x y
1 2 5
```

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),
+       data.frame(x = 1, y = 5) %>% mutate(COLNAME = COLNAME + 1),
+       eval = FALSE
+   )

data.frame(x = 1, y = 5) %>% 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 "`data.frame(x = 1,y = 5) %>% 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(data.frame(x = 1, y = 5), "y")

  x y
1 1 6
```

Notice the function correctly added one to "`y`" even though the target name was coincidentally "`x`". We will discuss how to manage potential name collisions and environments later in this note.

For this example a similar effect can be accomplished using the **rlang**, though the notation and semantics are different.

```
>   f2 <- function(d, z) {
+       x <- rlang::sym(z)
+       d %>% mutate(!!x := !!x + 1)
+   }
>   f2(data.frame(x = 1, y = 5), "y")
```

```
  x y
1 1 6
```

The amount of boiler plate code required by `wrapr::let()` is only proportional to the number of stand-ins targeted for replacement, and often independent of the size of the code being managed. In simple cases `wrapr::let()` requires little to no change to the code being parameterized. The boilerplate can also be made shorter by using the **wrapr**-supplied helper function `:=`[4]. Let's demonstrate this using the earlier `lm()` example.

```
>   d <- data.frame(x = 1:3, y = 11:13, z = 21:23)
>   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). A good history of FEXPR semantics (a core part of R and S function evaluation) can be found in Shutt (2010).

data.table (Dowle and Srinivasan, 2017) is, in our opinion, the most powerful package for in-memory data manipulation in R.[5] We mention **data.table** here as it uses a mixture of code capture and value oriented interfaces, giving as a useful new example. For example:

[4] `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** (Dowle and Srinivasan, 2017). **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.

[5] And it even has some important out of core extensions.

```
>   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 step uses values
+        dT[, NEW_NAME := NEW_NAME + 1]   # this step uses names
+        dT
+      })}
>   print(f(d, "x", "new_x")[])

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

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 functions

`wrapr::let()`'s design is based on a theory of *unbound variables*, that is variable names that may not have an associated value at the time or in the environment they are declared. The most common example of this is specifying the a column name that is to be interpreted with respect to a not yet available `data.frame`. This theory is different than concentrating on *bound variables* (which caries environments with variables). It is our principle that if one has a bound variable (i.e. a name and an environment that it is to be evaluated with respect to) then it makes sense to treat that as a value.[6].

When using `wrapr::let()` in a function the programmer must take explicit care in naming arguments and in specifying environments. Fortunately R already has good tools for these tasks. Let's make this concrete with a couple of examples.

Suppose we wish to create a function that returns the sum of two variables specified by name. We could attempt this as follows.

```
>   library("wrapr")
>   x <- 1
>   y <- 2
>   a <- 3
>   b <- 4
>   adder <- function(x, y) {
+       let(c(x = x, y = y),
+           x + y)
+       }
>   # works
>   adder("a", "b")

[1] 7

>   # fails
>   tryCatch(
+       adder("x", "y"),
+       error = function(e) { cat(format(e)) }
+   )

non-numeric argument to binary operator x + y
```

The above error is not due to any brittleness in `let()`, it is because we did not follow the following important conventions in using `let()` in a function.

1 Design your function and `let()` blocks so there are no coincidences between variables (especially function arguments) and `let()` stand-in substitution targets.

[6]In fact carrying environments around merely to delay de-references of values is a major source of reference leaks in R: http://www.win-vector.com/blog/2014/05/trimming-the-fat-from-glm-models-in-r/

2 Separate calculations into meaningful sub-units.

3 Be explicit in specifying which environment you wish values to be taken from for each step.

What went wrong is the function's execution environment the variable `x` contains the string `"x"`, which is not a suitable argument for standard numeric addition. We fix this by asking the expression to be evaluated in the environment the function was called from, where the values the user expected to be active are available.

In our case it is much better to write the function as follows.

```
>   adder <- function(x, y, envir = parent.frame()) {
+       let(c(X = x, Y = y),
+           X + Y,
+           envir = envir)
+       }
>   # works
>   adder("a", "b")

[1] 7

>   # works
>   adder("x", "y")

[1] 3
```

Above we used the mixed-case convention discussed earlier. It is *always* easy to avoid having substitution targets confused with other names in inner code, as both the stand-in target choices and the inner code are specified the exact same place: inside the `let()` block (and hence completely under the programmer's control).

Environment specification can get a bit more involved, takes a bit more some care. For instance suppose we have the following data.

```
>  x <- 1:3
>  y <- c(-1, -1, 3)
>  d <- data.frame(x = 1:3, y = c(1, 1, 3))
```

Further suppose that we wish to build a function that controls elements of a `lm()` formula and also what is returned. In our case we wish to parameterize an expression such as:

```
>  summary(lm(y ~ x, data = d))[["coefficients"]]

             Estimate Std. Error    t value  Pr(>|t|)
(Intercept) -0.3333333  1.2472191 -0.2672612 0.8337420
x            1.0000000  0.5773503  1.7320508 0.3333333
```

To do this we write our new function as follows.

```
> get_lm_prop <- function(x, y, df, property) {
+   let(c(X = x, Y = y),
+       summary(lm(Y ~ X, data = df))[[property]])
+ }
> get_lm_prop("x", "y", d, "coefficients")

             Estimate Std. Error    t value  Pr(>|t|)
(Intercept) -0.3333333  1.2472191 -0.2672612 0.8337420
x            1.0000000  0.5773503  1.7320508 0.3333333
```

Notice we did not take control of the environment, which allows values such as "`property`" to be taken from the function's environment (the function arguments in particular). Also, we are assuming (but not enforcing) that `x` and `y` are carrying column names of the `data.frame` `df`. If we wish to production harden the code we could re-write it thusly.

```
> get_lm_prop <- function(x, y, df, property) {
+   if(length(setdiff(c(x, y), colnames(df)))>0) {
+     stop("get_lm_prop referred to non-columns")
+   }
+   let(c(X = x, Y = y),
+       summary(lm(Y ~ X, data = df))[[property]])
+ }
```

Or if we wished for some values to come from the calling environment and some from the function environment we merely break up the steps allowing us to specify the distinctions.

```
> get_lm_prop <- function(x, y, df, property, envir = parent.frame()) {
+   let(c(X = x, Y = y),
+       form <- Y ~ X,
+       envir = envir
+   )
+   summary(lm(form, data = df))[[property]]
+ }
```

The above isn't a particularly useful refactoring (it is hard to imagine the formula construction benefits form being evaluated in the parent environment in this case). However, it gives us a chance to point out in this case the `lm()` portion of the task should not be in the environment controlled let-block we want the `lm()` to take the value for `df` from what was specified as function argument to `get_lm_prop` which means we want the `lm()` step evaluated in the function environment. For complicated sequences of operations it may be the case that we wish different portions evaluated with respect to different environments (some steps may want items from the function environment, and some may want to be isolated from the function environment). However, we point out this isn't a problem unique to `let()`, it is central issue in designing reliable functions in R. We feel the conventions we have just taught (unique target names, breaking up calculations into smaller units, and explicit control of environments) are sufficient tools for these challenges.

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** and **dplyr**.

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
1 1 2
2 2 2
3 3 4
```

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.

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

R. Culpepper and M. Felleisen. Fortifying macros. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 235–246, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863577. URL http://doi.acm.org/10.1145/1863543.1863577. [p3]

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]

P. J. Landin. Correspondence between algol 60 and church's lambda-notation: Part i. *Commun. ACM*, 8 (2):89–101, Feb. 1965. ISSN 0001-0782. doi: 10.1145/363744.363749. URL http://doi.acm.org/10.1145/363744.363749. [p3]

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]

P. Meilstrup. *nseval: A Clean API for Lazy and Non-Standard Evaluation*, 2018. URL https://CRAN.R-project.org/package=nseval. R package version 0.4. [p2]

J. Mount. Macro substitution in R, 2018. URL https://github.com/WinVector/wrapr/blob/master/extras/MacrosInR.md. [p2]

J. Mount and N. Zumel. *wrapr: Wrap R Functions for Debugging and Parametric Programming*, 2018. URL https://CRAN.R-project.org/package=wrapr. [p1, 2]

J. N. Shutt. *Fexprs as the basis of Lisp function application or $vau: the ultimate abstraction*. PhD thesis, Worcester Polytechnic Institute, August 2010. URL https://web.wpi.edu/Pubs/ETD/Available/etd-090110-124904/unrestricted/jshutt.pdf. [p4]

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. [p3]

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. *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