Non Standard Evaluation in dplyr with Galaaz

Rodrigo Botafogo
Daniel Mossé - University of Pittsburgh

10/05/2019

Contents

1 Introduction

In this post we will see how to program with *dplyr* in Galaaz.

1.1 But first, what is Galaaz??

Galaaz is a system for tightly coupling Ruby and R. Ruby is a powerful language, with a large community, a very large set of libraries and great for web development. However, it lacks libraries for data science, statistics, scientific plotting and machine learning. On the other hand, R is considered one of the most powerful languages for solving all of the above problems. Maybe the strongest competitor to R is Python with libraries such as NumPy, Pandas, SciPy, SciKit-Learn and many more.

With Galaaz we do not intend to re-implement any of the scientific libraries in R. However, we allow for very tight coupling between the two languages to the point that the Ruby developer does not need to know that there is an R engine running. Also, from the point of view of the R user/developer Galaaz looks a lot like R, with just minor syntactic difference, so there is almost no learning courve for the R developer. And as we will see in this post, programming with *dplyr* is easier in Galaaz than in R.

R users are probably quite knowledgeable about *dplyr*, for the Ruby developer, *dplyr* and the *tidyverse* libraries are a set of libraries for data manipulation in R, developed by Hardley Wickham, chief scientis at RStudio and a prolific R coder and writer.

For the coupling of Ruby and R we use new technologies provided by Oracle: GraalVM, TruffleRuby and FastR:

```
 GraalVM is a universal virtual machine for running applications
 written in JavaScript, Python 3, Ruby, R, JVM-based languages like Java,
 Scala, Kotlin, and LLVM-based languages such as C and C++.

 GraalVM removes the isolation between programming languages and enables
 interoperability in a shared runtime. It can run either standalone or in
 the context of OpenJDK, Node.js, Oracle Database, or MySQL.

 GraalVM allows you to write polyglot applications with a seamless way to
 pass values from one language to another. With GraalVM there is no copying
 or marshaling necessary as it is with other polyglot systems. This lets
 you achieve high performance when language boundaries are crossed. Most
 of the time there is no additional cost for crossing a language boundary
 at all.

 Often developers have to make uncomfortable compromises that require them
 to rewrite their software in other languages. For example:

   * "That library is not available in my language. I need to rewrite it."
   * "That language would be the perfect fit for my problem, but we cannot
     run it in our environment."
   * "That problem is already solved in my language, but the language is
     too slow."

With GraalVM we aim to allow developers to freely choose the right language
for the task at hand without making compromises.
```

Interested readers should also check out the following sites:

- GraalVM Home
- TruffleRuby
- FastR
- Faster R with FastR
- How to make Beautiful Ruby Plots with Galaaz
- Ruby Plotting with Galaaz: An example of tightly coupling Ruby and R in GraalVM
- How to do reproducible research in Ruby with gKnit
- R for Data Science
- Advanced R

1.2 Programming with dplyr

This post will follow closely the work done in https://dplyr.tidyverse.org/articles/programming.html, by Hardley Wickham. In it, Hardley states:

> Most dplyr functions use non-standard evaluation (NSE). This is a catch-all term that means they don't follow the usual R rules of evaluation. Instead, they capture the expression that you typed and evaluate it in a custom way. This has two main benefits for dplyr code:
>
> Operations on data frames can be expressed succinctly because you don't need to repeat the name of the data frame. For example, you can write filter(df, x == 1, y == 2, z == 3) instead of df[df$x == 1 & df$y ==2 & df$z == 3,].
>
> dplyr can choose to compute results in a different way to base R. This is important for database backends because dplyr itself doesn't do any work, but instead generates the SQL that tells the database what to do.
>
> Unfortunately these benefits do not come for free. There are two main drawbacks:
>
> Most dplyr arguments are not referentially transparent. That means you can't replace a value with a seemingly equivalent object that you've defined elsewhere. In other words, this code:

```
df <- data.frame(x = 1:3, y = 3:1)
print(filter(df, x == 1))
#> # A tibble: 1 x 2
#>       x     y
#>   <int> <int>
#> 1     1     3
```

> Is not equivalent to this code:

```
my_var <- x
#> Error in eval(expr, envir, enclos): object 'x' not found
filter(df, my_var == 1)
#> Error: object 'my_var' not found
```

> This makes it hard to create functions with arguments that change how dplyr verbs are computed.

In this post we will see that programming with *dplyr* in Galaaz does not require knowledge of non-standard evaluation in R and can be accomplished by utilizing normal Ruby constructs.

2 Writing Expressions in Galaaz

Galaaz extends Ruby to work with expressions, similar to R's expressions build with 'quote' (base R) or 'quo' (tidyverse). Expressions in this context are like mathematical expressions or formulae. For instance, in mathematics, the expression $y = sin(x)$ describes a function but cannot be computed unless the value of x is bound to some value.

Let's take a look at some of those expressions in Ruby:

2.1 Expressions from operators

The code bellow creates an expression summing two symbols. Note that :a and :b are Ruby symbols and are not bound to any value at the time of expression definition:

```
exp1 = :a + :b
puts exp1
```

```
## a + b
```

We can build any complex mathematical expression such as:

```
exp2 = (:a + :b) * 2.0 + :c ** 2 / :z
puts exp2
```

```
## (a + b) * 2 + c^2L/z
```

The 'L' after two indicates that 2 is an integer.

It is also possible to use inequality operators in building expressions:

```
exp3 = (:a + :b) >= :z
puts exp3
```

```
## a + b >= z
```

Expressions' definition can also make use of normal Ruby variables without any problem:

```
x = 20
y = 30
exp_var = (:a + :b) * x <= :z - y
puts exp_var
```

```
## (a + b) * 20L <= z - 30L
```

Galaaz provides both symbolic representations for operators, such as (>, <, !=) as functional notation for those operators such as (.gt, .ge, etc.). So the same expression written above can also be written as

```
exp4 = (:a + :b).ge :z
puts exp4
```

```
## a + b >= z
```

Two type of expression, however, can only be created with the functional representation of the operators, those are expressions involving '==', and '='. In order to write an expression involving '==' we need to use the method '.eq' and for '=' we need the function '.assign'

```
exp5 = (:a + :b).eq :z
puts exp5
```

```
## a + b == z
```

```
exp6 = :y.assign :a + :b
puts exp6
```

```
## y <- a + b
```

In general we think that using the functional notation is preferable to using the symbolic notation as otherwise, we end up writing invalid expressions such as

```
exp_wrong = (:a + :b) == :z
puts exp_wrong
```

```
## Message:
##  Error in function (x, y, num.eq = TRUE, single.NA = TRUE, attrib.as.set = TRUE,  :
##   object 'a' not found (RError)
## Translated to internal error
```

and it might be difficult to understand what is going on here. The problem lies with the fact that when using '==' we are comparing expression (:a + :b) to expression :z with '=='. When the comparison is executed, the system tries to evaluate :a, :b and :z, and those symbols at this time are not bound to anything and we get a "object 'a' not found" message. If we only use functional notation, this type of error will not occur.

2.2 Expressions with R methods

It is often necessary to create an expression that uses a method or function. For instance, in mathematics, it's quite natural to write an expressin such as $y = sin(x)$. In this case, the 'sin' function is part of the expression and should not immediately be executed. When we want the function to be part of the expression, we call the function preceeding it by the letter E, such as 'E.sin(x)'

```
exp7 = :y.assign E.sin(:x)
puts exp7
```

```
## y <- sin(x)
```

Expressions can also be written using '.' notation:

```
exp8 = :y.assign :x.sin
puts exp8
```

```
## y <- sin(x)
```

When a function has multiple arguments, the first one can be used before the '.':

```
exp9 = :x.c(:y)
puts exp9
```

```
## c(x, y)
```

2.3 Evaluating an Expression

Expressions can be evaluated by calling function 'eval' with a binding. A binding can be provided with a list:

```
exp = (:a + :b) * 2.0 + :c ** 2 / :z
puts exp.eval(R.list(a: 10, b: 20, c: 30, z: 40))
```

```
## [1] 82.5
```

. . . with a data frame:

```
df = R.data__frame(
  a: R.c(1, 2, 3),
  b: R.c(10, 20, 30),
  c: R.c(100, 200, 300),
  z: R.c(1000, 2000, 3000))

puts exp.eval(df)
```

```
## [1] 32 64 96
```

3 Using Galaaz to call R functions

Galaaz tries to emulate as closely as possible the way R functions are called and migrating from R to Galaaz should be quite easy requiring only minor syntactic changes to an R script. In this post, we do not have enough space to write a complete manual on Galaaz (a short manual can be found at: https://www.rubydoc.info/gems/galaaz/0.4.9), so we will present only a few examples scripts using Galaaz.

Basically, to call an R function from Ruby with Galaaz, one only needs to preceed the function with 'R.'. For instance, to create a vector in R, the 'c' function is used. From Galaaz, a vector can be created by using 'R.c':

```
vec = R.c(1.0, 2, 3)
puts vec
```

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

A list is created in R with the 'list' function, so in Galaaz we do:

```
list = R.list(a: 1.0, b: 2, c: 3)
puts list
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## $c
## [1] 3
```

Note that we can use named arguments in our list. The same code in R would be:

```
lst = list(a = 1, b = 2L, c = 3L)
print(lst)
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## $c
## [1] 3
```

Now, let's say that 'x' is an angle of 45° and we acttually want to create the expression $y = sin(45°)$, which is $y = 0.850...$. In this case, we will use 'R.sin':

```
exp10 = :y.assign R.sin(45)
puts exp10
```

```
## y <- 0.850903524534118
```

4 Filtering using expressions

Now that we know how to write expression and call R functions let's do some data manipulation in Galaaz. Let's first start by creating the same data frame that we created previously in section "Programming with dplyr":

```
df = R.data__frame(x: (1..3), y: (3..1))
puts df
```

```
##   x y
## 1 1 3
## 2 2 2
## 3 3 1
```

The 'filter' function can be called on this data frame either by using 'R.filter(df, ...)' or by using dot notation. We prefer to use dot notation as shown bellow. The argument to 'filter' in Galaaz should be an expression. Note that if we gave to filter a Ruby expression such as 'x == 1', we would get an error, since there is no variable 'x' defined and if 'x' was a variable then 'x == 1' would either be 'true' or 'false'. Our goal is to filter our data frame returning all rows in which the 'x' value is equal to 1. To express this we want: ':x.eq 1', where :x will be interpreted by filter as the 'x' column.

```
puts df.filter(:x.eq 1)
```

```
##   x y
## 1 1 3
```

In R, and when coding with 'tidyverse', arguments to a function are usually not *referencially transparent*. That is, you can't replace a value with a seemingly equivalent object that you've defined elsewhere. In other words, this code

```
my_var <- x
filter(df, my_var == 1)
```

Generates the following error: "object 'x' not found.

However, in Galaaz, arguments are referencially transparent as can be seen by the code bellow. Note initally that 'my_var = :x' will not give the error "object 'x' not found" since ':x' is treated as an expression and assigned to my_var. Then when doing (my_var.eq 1), my_var is a variable that resolves to ':x' and it becomes equivalent to (:x.eq 1) which is what we want.

```
my_var = :x
puts df.filter(my_var.eq 1)
```

```
##   x y
## 1 1 3
```

As stated by Hardley

dplyr code is ambiguous. Depending on what variables are defined where, filter(df, x == y) could be equivalent to any of:

```
df[df$x == df$y, ]
df[df$x == y, ]
df[x == df$y, ]
df[x == y, ]
```

In galaaz this ambiguity does not exist, filter(df, x.eq y) is not a valid expression as expressions are build with symbols. In doing filter(df, :x.eq y) we are looking for elements of the 'x' column that are equal to a previously defined y variable. Finally in filter(df, :x.eq :y) we are looking for elements in which the 'x' column value is equal to the 'y' column value. This can be seen in the following two chunks of code:

```
y = 1
x = 2

# looking for values where the 'x' column is equal to the 'y' column
puts df.filter(:x.eq :y)
```

```
##   x y
## 1 2 2
```

```
# looking for values where the 'x' column is equal to the 'y' variable
# in this case, the number 1
puts df.filter(:x.eq y)
```

```
##   x y
## 1 1 3
```

5 Writing a function that applies to different data sets

Let's suppose that we want to write a function that receives as the first argument a data frame and as second argument an expression that adds a column to the data frame that is equal to the sum of elements in column 'a' plus 'x'.

Here is the intended behaviour using the 'mutate' function of 'dplyr':

```
mutate(df1, y = a + x)
mutate(df2, y = a + x)
mutate(df3, y = a + x)
mutate(df4, y = a + x)
```

The naive approach to writing an R function to solve this problem is:

```
mutate_y <- function(df) {
  mutate(df, y = a + x)
}
```

Unfortunately, in R, this function can fail silently if one of the variables isn't present in the data frame, but is present in the global environment. We will not go through here how to solve this problem in R.

In Galaaz the method mutate_y bellow will work fine and will never fail silently.

```
def mutate_y(df)
  df.mutate(:y.assign :a + :x)
end
```

Here we create a data frame that has only one column named ‘x’:

```
df1 = R.data__frame(x: (1..3))
puts df1
```

```
##   x
## 1 1
## 2 2
## 3 3
```

Note that method mutate_y will fail independetly from the fact that variable ‘a’ is defined and in the scope of the method. Variable ‘a’ has no relationship with the symbol ‘:a’ used in the definition of ‘mutate_y’ above:

```
a = 10
mutate_y(df1)
```

```
## Message:
##  Error in mutate_impl(.data, dots) :
##   Evaluation error: object 'a' not found.
## In addition: Warning message:
## In mutate_impl(.data, dots) :
##   mismatched protect/unprotect (unprotect with empty protect stack) (RError)
## Translated to internal error
```

6 Different expressions

Let’s move to the next problem as presented by Hardley where trying to write a function in R that will receive two argumens, the first a variable and the second an expression is not trivial. Bellow we create a data frame and we want to write a function that groups data by a variable and summarises it by an expression:

```
set.seed(123)

df <- data.frame(
  g1 = c(1, 1, 2, 2, 2),
  g2 = c(1, 2, 1, 2, 1),
  a = sample(5),
  b = sample(5)
)

as.data.frame(df)
```

```
##   g1 g2 a b
## 1  1  1 2 1
## 2  1  2 4 3
## 3  2  1 5 4
## 4  2  2 3 2
## 5  2  1 1 5
```

```
d2 <- df %>%
  group_by(g1) %>%
  summarise(a = mean(a))

as.data.frame(d2)
```

```
##   g1 a
## 1  1 3
## 2  2 3
```

```
d2 <- df %>%
  group_by(g2) %>%
  summarise(a = mean(a))

as.data.frame(d2)
```

```
##   g2        a
## 1  1 2.666667
## 2  2 3.500000
```

As shown by Hardley, one might expect this function to do the trick:

```
my_summarise <- function(df, group_var) {
  df %>%
    group_by(group_var) %>%
    summarise(a = mean(a))
}

# my_summarise(df, g1)
#> Error: Column `group_var` is unknown
```

In order to solve this problem, coding with dplyr requires the introduction of many new concepts and functions such as 'quo', 'quos', 'enquo', 'enquos', '!!' (bang bang), '!!!' (triple bang). Again, we'll leave to Hardley the explanation on how to use all those functions.

Now, let's try to implement the same function in galaaz. The next code block first prints the 'df' data frame define previously in R (to access an R variable from Galaaz, we use the tilda operator '~' applied to the R variable name as symbol, i.e., ':df'. We then create the 'my_summarize' method and call it passing the R data frame and the group by variable ':g1':

```
puts ~:df
print "\n"

def my_summarize(df, group_var)
  df.group_by(group_var).
    summarize(a: :a.mean)
end

puts my_summarize(:df, :g1).as__data__frame
```

```
##   g1 g2 a b
## 1  1  1 2 1
## 2  1  2 4 3
## 3  2  1 5 4
## 4  2  2 3 2
## 5  2  1 1 5
##
##   g1 a
## 1  1 3
## 2  2 3
```

It works!!! Well, let's make sure this was not just some coincidence

```
puts my_summarize(:df, :g2).as__data__frame
```

```
##   g2        a
## 1  1 2.666667
## 2  2 3.500000
```

Great, everything is fine! No magic, no new functions, no complexities, just normal, standard Ruby code. If you've ever done NSE in R, this certainly feels much safer and easy to implement.

7 Different input variables

In the previous section we've managed to get rid of all NSE formulation for a simple example, but does this remain true for more complex examples, or will the Galaaz way prove inpractical for more complex code?

In the next example Hardley proposes us to write a function that given an expression such as 'a' or 'a * b', calculates three summaries. What we want a function that does the same as these R statements:

```
summarise(df, mean = mean(a), sum = sum(a), n = n())
#> # A tibble: 1 x 3
#>    mean   sum     n
#>   <dbl> <int> <int>
#> 1     3    15     5

summarise(df, mean = mean(a * b), sum = sum(a * b), n = n())
#> # A tibble: 1 x 3
#>    mean   sum     n
#>   <dbl> <int> <int>
#> 1   9    45     5
```

Let's try it in galaaz:

```
def my_summarise2(df, expr)
  df.summarize(
    mean: E.mean(expr),
    sum: E.sum(expr),
    n: E.n
  )
end

puts my_summarise2((~:df), :a)
puts my_summarise2((~:df), :a * :b)
```

```
##   mean sum n
## 1    3  15 5
##   mean sum n
## 1    9  45 5
```

Once again, there is no need to use any special theory or functions. The only point to be careful about is the use of 'E' to build expressions from functions 'mean', 'sum' and 'n'.

8 Different input and output variable

Now the next challenge presented by Hardley is to vary the name of the output variables based on the received expression. So, if the input expression is 'a', we want our data frame columns to be named 'mean_a' and 'sum_a'. Now, if the input expression is 'b', columns should be named 'mean_b' and 'sum_b'.

```
mutate(df, mean_a = mean(a), sum_a = sum(a))
#> # A tibble: 5 x 6
#>      g1    g2     a     b mean_a sum_a
#>   <dbl> <dbl> <int> <int>  <dbl> <int>
#> 1     1     1     1     3      3    15
#> 2     1     2     4     2      3    15
#> 3     2     1     2     1      3    15
#> 4     2     2     5     4      3    15
#> # ... with 1 more row

mutate(df, mean_b = mean(b), sum_b = sum(b))
#> # A tibble: 5 x 6
#>      g1    g2     a     b mean_b sum_b
#>   <dbl> <dbl> <int> <int>  <dbl> <int>
#> 1     1     1     1     3      3    15
#> 2     1     2     4     2      3    15
#> 3     2     1     2     1      3    15
#> 4     2     2     5     4      3    15
#> # ... with 1 more row
```

In order to solve this problem in R, Hardley needs to introduce some more new functions and notations: 'quo_name' and the ':=' operator from package 'rlang'

Here is our Ruby code:

```ruby
def my_mutate(df, expr)
  mean_name = "mean_#{expr.to_s}"
  sum_name = "sum_#{expr.to_s}"

  df.mutate(mean_name => E.mean(expr),
            sum_name => E.sum(expr))
end

puts my_mutate((~:df), :a)
puts my_mutate((~:df), :b)
```

```
##   g1 g2 a b mean_a sum_a
## 1  1  1 2 1      3    15
## 2  1  2 4 3      3    15
## 3  2  1 5 4      3    15
## 4  2  2 3 2      3     15
## 5  2  1 1 5      3     15
##   g1 g2 a b mean_b sum_b
## 1  1  1 2 1      3     15
## 2  1  2 4 3      3     15
## 3  2  1 5 4      3     15
## 4  2  2 3 2      3     15
## 5  2  1 1 5      3     15
```

It really seems that "Non Standard Evaluation" is actually quite standard in Galaaz! But, you might have noticed a small change in the way the arguments to the mutate method were called. In a previous example we used df.summarise(mean: E.mean(:a), . . .) where the column name was followed by a ':' colom. In this example, we have df.mutate(mean_name => E.mean(expr), . . .) and variable mean_name is not followed by ':' but by '=>'. This is standard Ruby notation.

[explain. . . .]

9 Capturing multiple variables

Moving on with new complexities, Hardley proposes us to solve the problem in which the summarise function will receive any number of grouping variables.

This again is quite standard Ruby. In order to receive an undefined number of paramenters the paramenter is preceded by '*':

```ruby
def my_summarise3(df, *group_vars)
  df.group_by(*group_vars).
    summarise(a: E.mean(:a))
end

puts my_summarise3((~:df), :g1, :g2).as__data__frame
```

```
##   g1 g2 a
## 1  1  1 2
## 2  1  2 4
## 3  2  1 3
## 4  2  2 3
```

10 Why does R require NSE and Galaaz does not?

NSE introduces a number of new concepts, such as 'quoting', 'quasiquotation', 'unquoting' and 'unquote-splicing', while in Galaaz none of those concepts are needed. What gives?

R is an extremely flexible language and it has lazy evaluation of parameters. When in R a function is called as 'summarise(df, a = b)', the summarise function receives the litteral 'a = b' parameter and can work with this as if it were a string. In R, it is not clear what a and b are, they can be expressions or they can be variables, it is up to the function to decide what 'a = b' means.

In Ruby, there is no lazy evaluation of parameters and 'a' is always a variable and so is 'b'. Variables assume their value as soon as they are used, so 'x = a' is immediately evaluate and variable 'x' will receive the value of variable 'a' as soon as the Ruby statement is executed. Ruby also provides the notion of a symbol; ':a' is a symbol and does not evaluate to anything. Galaaz uses Ruby symbols to build expressions that are not bound to anything: ':a.eq :b' is clearly an expression and has no relationship whatsoever with the statment 'a = b'. By using symbols, variables and expressions all the possible ambiguities that are found in R are eliminated in Galaaz.

The main problem that remains, is that in R, functions are not clearly documented as what type of input they are expecting, they might be expecting regular variables or they might be expecting expressions and the R function will know how to deal with an input of the form 'a = b', now for the Ruby developer it might not be immediately clear if it should call the function passing the value 'true' if variable 'a' is equal to variable 'b' or if it should call the function passing the expression ':a.eq :b'.

11 Advanced dplyr features

In the blog: Programming with dplyr by using dplyr (https://www.r-bloggers.com/programming-with-dplyr-by-using-dplyr/) Iñaki Úcar shows surprise that some R users are trying to code in dplyr avoiding the use of NSE. For instance he says:

> Take the example of seplyr. It stands for standard evaluation dplyr, and enables us to program over dplyr without having "to bring in (or study) any deep-theory or heavy-weight tools such as rlang/tidyeval".

For me, there isn't really any surprise that users are trying to avoid dplyr deep-theory. R users frequently are not programmers and learning to code is already hard business, on top of that, having to learn how to 'quote' or 'enquo' or 'quos' or 'enquos' is not necessarily a 'piece of cake'. So much so, that 'tidyeval' has some more advanced functions that instead of using quoted expressions, uses strings as arguments.

In the following examples, we show the use of functions 'group_by_at', 'summarise_at' and 'rename_at' that receive strings as argument. The data frame used in 'starwars' that describes features of characters in the Starwars movies:

```
puts (~:starwars).head.as__data__frame
```

```
##             name height mass  hair_color  skin_color eye_color birth_year
## 1 Luke Skywalker    172   77       blond        fair      blue       19.0
## 2          C-3PO    167   75        <NA>        gold    yellow      112.0
## 3          R2-D2     96   32        <NA> white, blue       red       33.0
## 4    Darth Vader    202  136        none       white    yellow       41.9
## 5     Leia Organa    150   49       brown       light     brown       19.0
## 6      Owen Lars    178  120 brown, grey       light      blue       52.0
##   gender homeworld species
## 1   male  Tatooine   Human
## 2   <NA>  Tatooine   Droid
## 3   <NA>     Naboo   Droid
## 4   male  Tatooine   Human
## 5 female  Alderaan   Human
## 6   male  Tatooine   Human
##
## 1                                            Revenge of the Sith, Return of the Jedi, The
## 2                        Attack of the Clones, The Phantom Menace, Revenge of the Sith, Retu
## 3 Attack of the Clones, The Phantom Menace, Revenge of the Sith, Return of the Jedi, The
## 4                                                                     Revenge of the Sith, Retur
## 5                                                   Revenge of the Sith, Return of the Jedi, The
## 6                                                                                              Att
##                              vehicles                starships
## 1 Snowspeeder, Imperial Speeder Bike X-wing, Imperial shuttle
## 2
## 3
## 4                                          TIE Advanced x1
## 5              Imperial Speeder Bike
## 6
```

The grouped_mean function bellow will receive a grouping variable and calculate summaries for the value_variables given:

```
grouped_mean <- function(data, grouping_variables, value_variables) {
  data %>%
    group_by_at(grouping_variables) %>%
    mutate(count = n()) %>%
    summarise_at(c(value_variables, "count"), mean, na.rm = TRUE) %>%
    rename_at(value_variables, funs(paste0("mean_", .)))
    }

gm = starwars %>%
   grouped_mean("eye_color", c("mass", "birth_year"))

as.data.frame(gm)
```

```
##        eye_color mean_mass mean_birth_year count
## 1          black  76.28571        33.00000    10
## 2           blue  86.51667        67.06923    19
## 3      blue-gray  77.00000        57.00000     1
## 4          brown  66.09231       108.96429    21
## 5           dark       NaN             NaN     1
## 6           gold       NaN             NaN     1
## 7  green, yellow 159.00000             NaN     1
## 8          hazel  66.00000        34.50000     3
## 9         orange 282.33333       231.00000     8
## 10          pink       NaN             NaN     1
## 11           red  81.40000        33.66667     5
## 12     red, blue       NaN             NaN     1
## 13       unknown  31.50000             NaN     3
## 14         white  48.00000             NaN     1
## 15        yellow  81.11111        76.38000    11
```

The same code with Galaaz, becomes:

```
def grouped_mean(data, grouping_variables, value_variables)
  data.
    group_by_at(grouping_variables).
    mutate(count: E.n).
    summarise_at(E.c(value_variables, "count"), ~:mean, na__rm: true).
    rename_at(value_variables, E.funs(E.paste0("mean_", value_variables)))
end
```

```
puts grouped_mean((~:starwars), "eye_color", E.c("mass", "birth_year")).as__data__frame
```

```
##        eye_color mean_mass mean_birth_year count
## 1          black  76.28571        33.00000    10
## 2           blue  86.51667        67.06923    19
## 3      blue-gray  77.00000        57.00000     1
## 4          brown  66.09231       108.96429    21
## 5           dark       NaN             NaN     1
## 6           gold       NaN             NaN     1
## 7  green, yellow 159.00000             NaN     1
## 8          hazel  66.00000        34.50000     3
## 9         orange 282.33333       231.00000     8
## 10          pink       NaN             NaN     1
## 11           red  81.40000        33.66667     5
## 12     red, blue       NaN             NaN     1
## 13       unknown  31.50000             NaN     3
## 14         white  48.00000             NaN     1
## 15        yellow  81.11111        76.38000    11
```

12 Conclusion

Ruby and Galaaz provide a nice framework for developing code that uses R functions. Although R is a very powerful and flexible language, sometimes, too much flexibility makes life harder for the casual user. We believe however, that even for the advanced user, Ruby integrated with R throught Galaaz, makes a powerful environment for data analysis. In this blog post we showed how Galaaz consistent syntax eliminates the need for complex constructs such as quoting, enquoting, quasiquotation, etc. This simplification comes from the fact that expressions and variables are clearly separated objects, which is not the case in the R language.