Non Standard Evaluation in dplyr with Galaaz

Rodrigo Botafogo Daniel Mossé - University of Pittsburgh

10/05/2019 (narrative updated for Galaaz 2.0, 2026)

Contents

1 Introduction

According to Steven Sagaert answer on Quora about "Is programming language R overrated?":

> R is a sophisticated language with an unusual (i.e. non-mainstream) set of features. It's an impure functional programming language with sophisticated metaprogramming and 3 different OO systems.
>
> Just like common lisp you can completely customise how things work via metaprogramming. The biggest example is the tidyverse: by creating it's own evaluation system (tidyeval) was able to create a custom syntax for dplyr.
>
> Mastering R (the language) and its ecosystem is not a matter of weeks or months but takes years. The rabbit hole goes pretty deep...

Although having a highly configurable language might give extreme power to the programmer, it can also be, as stated above, a question of years to master it. Programming with *dplyr* for instance, requires learning a set of complex concepts and rules that are not easily accessible for casual users or *unsofisticated* programmers as many users of R are. Being *unsofisticated* is NOT used here in a negative sense, as R was build for statitians and not programmers, that need to solve real problems, often in a short time spam and are not concerned about creating complex computer systems.

Unfortunatelly, if this *unsofisticated* programmer decides to move unto more sofisticated coding, the learning curve might become a serious impediment.

In this post we will see how to program with *dplyr* in Galaaz and how Ruby can simplify the learning curve of mastering *dplyr* coding.

2 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. It is also easy to learn. 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. We will not get here in the discussion on R versus Python, both are excellent languages with powerful features, benefits and drawbacks. Our interest is to bring to yet another excellent language, Ruby, the data science libraries that it lacks.

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 curve for the R developer. And as we will see in this post that 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, **Galaaz 2.0** uses **JRuby** (Ruby on the JVM) together with **GNU R**. A **bridge** sends expressions and data between Ruby and an R process so that Ruby can call **dplyr** and the rest of the tidyverse as if they were part of the same workflow. An **earlier** Galaaz line of work used Oracle's **GraalVM** with **TruffleRuby** and **FastR** in a single runtime; that approach is **no longer** the supported stack—see the project **manual** for setup, `bin/galaaz-jruby`, and **gKnit**.

3 Tidyverse and dplyr

In What is the tidyverse? the tidyverse is explained as follows:

> The tidyverse is a coherent system of packages for data manipulation, exploration and visualization that share a common design philosophy. These were mostly developed by Hadley Wickham himself, but they are now being expanded by several contributors. Tidyverse packages are intended to make statisticians and data scientists more productive by guiding them through workflows that facilitate communication, and result in reproducible work products. Fundamentally, the tidyverse is about the connections between the tools that make the workflow possible.

dplyr is one of the many packages that are part of the tidyverse. It is:

> a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges:
>
> 1. mutate() adds new variables that are functions of existing variables
> 2. select() picks variables based on their names.
> 3. filter() picks cases based on their values.
> 4. summarise() reduces multiple values down to a single summary.
> 5. arrange() changes the ordering of the rows.

Very often R is used interactively and users use *dplyr* to manipulate a single dataset without programming. When users want to replicate their work for multiple datasets, programming becomes necessary.

4 Programming with dplyr

In the vignette "Programming with dplyr", Hardley Wickham 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.

But then he goes on:

> 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.

As a result of this, programming with *dplyr* requires learning a set of new ideas and concepts. In this vignette Hardley goes on showing how to program ever more difficult problems with *dplyr*, showing the problems it faces and the new concepts needed to solve them.

In this blog, we will look at all the problems presented by Harley on the vignette and show how those same problems can be solved using Galaaz and the Ruby language.

This blog is organized as follows: first we show how to write expressions using Galaaz. Expressions are a fundamental concept in *dplyr* and are not part of basic Ruby. We extend the Ruby language create a manipulate expressions that will be used by *dplyr* functions.

Then we show very succintly how Ruby and R can be integrated and how R functions are transparently called from Ruby. Galaaz user manual (still in development) goes in much deeper detail about this integration.

Next in section "Data manipulation wiht *dplyr*" we go through all the problems on the *dplyr* vignette and look at how they are solved in Galaaz. We then discuss why programming with Galaaz and *dplyr* is easier than programming with *dplyr* in plain R.

The following section looks at another more advanced problem and shows that Galaaz can still handle it without any difficulty. We then provide further reading and concluding remarks.

5 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.

Expressions are fundamental in *dplyr* programming as they are the input to *dplyr* functions, for instance, as we will see shortly, if a data frame has a column named 'x' and we want to add another column, y, to this dataframe that has the values of 'x' times 2, then we would call a *dplyr* function with the expression 'y = x * 2'.

5.1 A note on notation

This blog was written in Rmarkdown and automatically converted to HTML or PDF (depending on where you are reading this blog) with gKnit (a tool provided by Galaaz). In Rmarkdown, it is possible to write text and code blocks that are executed to generate the final report. Code blocks appear inside a 'box' and the result of their execution appear either in another type of 'box' with a different background (HTML) or as normal text (PDF). Every output line from the code execution is preceeded by '##'.

5.2 Expressions from operators

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

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

```
## a + b
```

In Galaaz, we can build any complex mathematical expression such as:

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

```
## a + b * 2.0 + c ^ 2L / z
```

Expressions are printed with the same format as the equivalent R expressions. The 'L' after 2 indicates that 2 is an integer.

The R developer should note that in R, if she writes the number '2', the R interpreter will convert it to float. In order to get an interger she should write '2L'. Galaaz follows Ruby notation and '2' is an integer, while '2.0' is a float.

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 = 30.0
exp_var = (:a + :b) * x <= :z - y
puts exp_var
```

```
## a + b * 20L <= z - 30.0
```

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 types of expressions, however, can only be created with the functional representation of the operators. Those are expressions involving '==', and '='. This is the case since those symbols have special meaning in Ruby and should not be redefined.

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
```

Users should be careful when writing expressions not to inadvertently use '==' or '=' as this will generate an error, that might be a bit cryptic (in future releases of Galaza, we plan to improve the error message).

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

```
## Error: object 'a' not found
```

The problem lies with the fact that when using '==' we are comparing expression (:a + :b) to expression :z with '=='. When this comparison is executed, the system tries to evaluate :a, :b and :z, and those symbols, at this time, are not bound to anything giving the "object 'a' not found" message.

5.3 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 be immediately 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)
```

Function 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 '.'. For instance, the R concatenate function 'c', that concatenates two or more arguments can be part of an expression as:

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

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

Note that this gives an OO feeling to the code, as if we were saying 'x' concatenates 'y'. As a side note, '.' notation can be used as the R pipe operator '%>%', but is more general than the pipe.

5.4 Evaluating an Expression

Although we are mainly focusing on expressions to pass them to *dplyr* functions, expressions can be evaluated by calling function 'eval' with a binding.

A binding can be provided with a list or a data frame as shown below:

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

```
## [1] 72.5
```

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] 31 62 93
```

6 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 preced the function with 'R.'. For instance, to create a vector in R, the 'c' function is used. In 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
```

7 Data manipulation wiht *dplyr*

In this section we will give a brief tour *dplyr*'s usage in Galaaz and how to manipulate data in Ruby with it. This section will follow *dplyr*'s vignette that explores the nycflights13 data set. This dataset contains all 336776 flights that departed from New York City in 2013. The data comes from the US Bureau of Transportation Statistics.

Let's start by taking a look at this dataset:

```
R.library('nycflights13')
# check it's dimension
puts ~:flights.dim
# and the structure
~:flights.str
```

```
## ~(dim(flights))
## <environment: 0x61a6457991a8>
```

Now, let's use a first verb of *dplyr*: 'filter'. This verb, obviously, will filter the data by the given expression. In the next block, we filter by columns 'month' and 'day'. The first argument to the filter function is symbol ':flights'. A Ruby symbol, when given to an R function will convert to the R variable of the same name, in this case 'flights', that holds the nycflights13 data frame.

The second and third arguments are expressions that will be used by the filter function to filter by columns, looking for entries in which the month and day are equal to 1.

```
puts R.filter(:flights, (:month.eq 1), (:day.eq 1))
```

```
## # A tibble: 842 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
```

```
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # i 832 more rows
## # i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

7.1 Programming with *dplyr*: problems and how to solve them in Galaaz

In this section we look at the list of problems that Hardley describes in the "Programming with dplyr" vignette and show how those problems are solved and coded with Galaaz. Readers interested in how those problems are treated in *dplyr* should read the vignette and use it as a comparison with this blog.

7.2 Filtering using expressions

Now that we know how to write expressions and call R functions, let's do some data manipulation in Galaaz. Let's first start by creating a data frame. In R, the 'data.frame' function creates a data frame. In Ruby, writing 'data.frame' will not parse as a single object. To call R functions that have a '.' in them, we need to substitute the '.' with '___'. So, method 'data.frame' in R, is called in Galaaz as 'R.data___frame':

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

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

dplyr provides the 'filter' function, that filters data in a data brame. The 'filter' function can be called on this data frame either by using 'R.filter(df, ...)' or by using dot notation.

———-FIX———

We prefer to use dot notation as shown bellow. The argument to 'filter' 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
```

7.3 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)
```

```
## Error in strsplit(path, "/") : non-character argument
## Calls: mutate ... .rlang_purrr_map_mold -> vapply -> FUN -> path_trim_prefix -> strsplit
```

7.4 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 3 3
## 2  1  2 2 1
## 3  2  1 5 2
## 4  2  2 4 5
## 5  2  1 1 4
```

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

as.data.frame(d2)
```

```
##   g1        a
## 1  1 2.500000
## 2  2 3.333333
```

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

as.data.frame(d2)
```

```
##   g2 a
## 1  1 3
## 2  2 3
```

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 "
"

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

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

```
##   g1 g2 a b
## 1  1  1 3 3
## 2  1  2 2 1
## 3  2  1 5 2
## 4  2  2 4 5
## 5  2  1 1 4
##
## # A tibble: 2 x 2
##      g1     a
##   <dbl> <dbl>
## 1     1  2.5
## 2     2  3.33
```

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

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

```
## # A tibble: 2 x 2
##      g2     a
```

```
##   <dbl> <dbl>
## 1     1     3
## 2     2     3
```

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.

7.5 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'.

7.6 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 3 3      3    15
## 2  1  2 2 1      3    15
## 3  2  1 5 2      3    15
## 4  2  2 4 5      3    15
## 5  2  1 1 4      3    15
##   g1 g2 a b mean_b sum_b
## 1  1  1 3 3      3    15
```

```
## 2  1  2 2 1      3    15
## 3  2  1 5 2      3    15
## 4  2  2 4 5      3    15
## 5  2  1 1 4      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....]

7.7 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)
```

```
## # A tibble: 4 x 3
## # Groups:   g1 [2]
##      g1    g2     a
##   <dbl> <dbl> <dbl>
## 1     1     1     3
## 2     1     2     2
## 3     2     1     3
## 4     2     2     4
```

8 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'.

9 Advanced dplyr features

In the blog: 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
```

```
## # A tibble: 6 x 14
##   name      height  mass hair_color skin_color eye_color birth_year sex   gender
##   <chr>      <int> <dbl> <chr>      <chr>      <chr>          <dbl> <chr> <chr>
## 1 Luke Sky~    172    77 blond      fair       blue            19   male  mascu~
## 2 C-3PO        167    75 <NA>       gold       yellow         112   none  mascu~
## 3 R2-D2         96    32 <NA>       white, bl~ red             33   none  mascu~
## 4 Darth Va~    202   136 none       white      yellow          41.9 male  mascu~
## 5 Leia Org~    150    49 brown      light      brown           19   fema~ femin~
## 6 Owen Lars    178   120 brown, gr~ light      blue            52   male  mascu~
## # i 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

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

```r
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"))
```

```
## Warning: `funs()` was deprecated in dplyr 0.8.0.
## i Please use a list of either functions or lambdas:
##
## # Simple named list: list(mean = mean, median = median)
##
## # Auto named with `tibble::lst()`: tibble::lst(mean, median)
##
## # Using lambdas list(~ mean(., trim = .2), ~ median(., na.rm = TRUE))
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

```r
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"))
```

```
## # A tibble: 15 x 4
##    eye_color     mean_mass mean_birth_year count
##    <chr>             <dbl>           <dbl> <dbl>
##  1 black              76.3            33      10
##  2 blue               86.5            67.1    19
##  3 blue-gray          77              57       1
##  4 brown              66.1           109.     21
##  5 dark              NaN             NaN       1
##  6 gold              NaN             NaN       1
##  7 green, yellow     159             NaN       1
##  8 hazel              66              34.5     3
##  9 orange            282.            231       8
## 10 pink              NaN             NaN       1
## 11 red                81.4            33.7     5
## 12 red, blue         NaN             NaN       1
## 13 unknown            31.5           NaN       3
## 14 white              48             NaN       1
## 15 yellow             81.1            76.4    11
```

10 Further reading

- JRuby — Ruby on the JVM (Galaaz 2.0)
- How to make Beautiful Ruby Plots with Galaaz (plots; narrative partly pre-2.0)
- Ruby Plotting with Galaaz in GraalVM (older stack; ideas still useful)
- How to do reproducible research in Ruby with gKnit
- R for Data Science
- Advanced R
- Historical context: GraalVM, TruffleRuby, FastR

11 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.