Galaaz Manual

How to tightly couple Ruby and R in GraalVM

Rodrigo Botafogo

2019

Contents

1 Introduction

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, Panda, SciPy, SciKit-Learn and a couple more.

2 System Compatibility

- Oracle Linux 7
- Ubuntu 18.04 LTS
- Ubuntu 16.04 LTS
- Fedora 28
- macOS 10.14 (Mojave)
- macOS 10.13 (High Sierra)

3 Dependencies

- TruffleRuby
- FastR

4 Installation

- Install GrallVM (http://www.graalvm.org/)
- Install Ruby (gu install Ruby)
- Install FastR (gu install R)
- Install rake if you want to run the specs and examples (gem install rake)

5 Usage

- Interactive shell: use 'gstudio' on the command line

  gstudio

```
vec = R.c(1, 2, 3, 4)
puts vec
```

```
## [1] 1 2 3 4
```

- Run all specs

  galaaz specs:all

- Run graphics slideshow (80+ graphics)

  galaaz sthda:all

- Run labs from Introduction to Statistical Learning with R

  galaaz islr:all

- See all available examples

  galaaz -T

  Shows a list with all available executalbe tasks. To execute a task, substitute the 'rake' word in the list with 'galaaz'. For instance, the following line shows up after 'galaaz -T'

  rake master_list:scatter_plot # scatter_plot from:. . . .

  execute

  galaaz master_list:scatter_plot

6 gKnitting a Document

This manual has been formatted usign gKnit. gKnit uses Knitr and R markdown to knit a document in Ruby or R and output it in any of the available formats for R markdown. gKnit runs atop of GraalVM, and Galaaz. In gKnit, Ruby variables are persisted between chunks, making it an ideal solution for literate programming. Also, since it is based on Galaaz, Ruby chunks can have access to R variables and Polyglot Programming with Ruby and R is quite natural.

gknit is described in more details here

7 Vector

Vectors can be thought of as contiguous cells containing data. Cells are accessed through indexing operations such as x[5]. Galaaz has six basic ('atomic') vector types: logical, integer, real, complex, string (or character) and raw. The modes and storage modes for the different vector types are listed in the following table.

typeof	mode	storage.mode
logical	logical	logical

typeof	mode	storage.mode
integer	numeric	integer
double	numeric	double
complex	complex	comples
character	character	character
raw	raw	raw

Single numbers, such as 4.2, and strings, such as “four point two” are still vectors, of length 1; there are no more basic types. Vectors with length zero are possible (and useful). String vectors have mode and storage mode “character”. A single element of a character vector is often referred to as a character string.

To create a vector the ‘c’ (concatenate) method from the ‘R’ module should be used:

```
vec = R.c(1, 2, 3)
puts vec
```

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

Lets take a look at the type, mode and storage.mode of our vector vec. In order to print this out, we are creating a data frame ‘df’ and printing it out. A data frame, for those not familiar with it, is basically a table. Here we create the data frame and add the column name by passing named parameters for each column, such as ‘typeof:’, ‘mode:’ and ’storage___mode¿. You should also note here that the double underscore is converted to a’.’. So, when printed ‘storage___mode’ will actually print as ‘storage.mode’.

Data frames will later be more carefully described. In R, the method used to create a data frame is ‘data.frame’, in Galaaz we use ‘data___frame’.

```
df = R.data__frame(typeof: vec.typeof, mode: vec.mode, storage__mode: vec.storage__mode)
puts df
```

```
##    typeof    mode storage.mode
## 1 integer numeric      integer
```

If you want to create a vector with floating point numbers, then we need at least one of the vector’s element to be a float, such as 1.0. R users should be careful, since in R a number like ‘1’ is converted to float and to have an integer the R developer will use ‘1L’. Galaaz follows normal Ruby rules and the number 1 is an integer and 1.0 is a float.

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

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

```
df = R.data__frame(typeof: vec.typeof, mode: vec.mode, storage__mode: vec.storage__mode)
outputs df.kable.kable_styling
```

typeof	mode	storage.mode
double	numeric	double

In this next example we try to create a vector with a variable ‘hello’ that has not yet being defined. This will raise an exception that is printed out. We get two return blocks, the first with a message explaining what went wrong and the second with the full backtrace of the error.

```
vec = R.c(1, hello, 5)
```

```
## Message:
##  undefined local variable or method `hello' for #<RC:0x2e0 @out_list=nil>:RC
```

```
## Message:
##  /home/rbotafogo/desenv/galaaz/lib/util/exec_ruby.rb:103:in `get_binding'
## /home/rbotafogo/desenv/galaaz/lib/util/exec_ruby.rb:102:in `eval'
## /home/rbotafogo/desenv/galaaz/lib/util/exec_ruby.rb:102:in `exec_ruby'
## /home/rbotafogo/desenv/galaaz/lib/gknit/knitr_engine.rb:650:in `block in initialize'
## /home/rbotafogo/desenv/galaaz/lib/R_interface/ruby_callback.rb:77:in `call'
## /home/rbotafogo/desenv/galaaz/lib/R_interface/ruby_callback.rb:77:in `callback'
## (eval):3:in `function(...) {\n          rb_method(...)'
## unknown.r:1:in `in_dir'
## unknown.r:1:in `block_exec:BLOCK0'
## /home/rbotafogo/lib/graalvm-ce-1.0.0-rc16/jre/languages/R/library/knitr/R/block.R:102:in
## /home/rbotafogo/lib/graalvm-ce-1.0.0-rc16/jre/languages/R/library/knitr/R/block.R:92:in `
## /home/rbotafogo/lib/graalvm-ce-1.0.0-rc16/jre/languages/R/library/knitr/R/block.R:6:in `p
## /home/rbotafogo/lib/graalvm-ce-1.0.0-rc16/jre/languages/R/library/knitr/R/block.R:3:in `<
## unknown.r:1:in `withCallingHandlers'
## unknown.r:1:in `process_file'
## unknown.r:1:in `<no source>:BLOCK1'
## /home/rbotafogo/lib/graalvm-ce-1.0.0-rc16/jre/languages/R/library/knitr/R/output.R:129:in
## unknown.r:1:in `<no source>:BLOCK1'
## /home/rbotafogo/lib/graalvm-ce-1.0.0-rc16/jre/languages/R/library/rmarkdown/R/render.R:16
## <REPL>:5:in `<repl wrapper>'
## <REPL>:1
```

Here is a vector with logical values

```
vec = R.c(true, true, false, false, true)
puts vec
```

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

7.1 Combining Vectors

The 'c' functions used to create vectors can also be used to combine two vectors:

```
vec1 = R.c(10.0, 20.0, 30.0)
vec2 = R.c(4.0, 5.0, 6.0)
vec = R.c(vec1, vec2)
puts vec
```

```
## [1] 10 20 30  4  5  6
```

In galaaz, methods can be chainned (somewhat like the pipe operator in R %>%, but more generic). In this next example, method 'c' is chainned after 'vec1'. This also looks like 'c' is a method of the vector, but in reallity, this is actually closer to the pipe operator. When Galaaz identifies that 'c' is not a method of 'vec' it actually tries to call 'R.c' with 'vec1' as the first argument concatenated with all the other available arguments. The code bellow is automatically converted to the code above.

```
vec = vec1.c(vec2)
puts vec
```

```
## [1] 10 20 30  4  5  6
```

7.2 Vector Arithmetic

Arithmetic operations on vectors are performed element by element:

```
puts vec1 + vec2
```

```
## [1] 14 25 36
```

```
puts vec1 * 5
```

```
## [1]  50 100 150
```

When vectors have different length, a recycling rule is applied to the shorter vector:

```
vec3 = R.c(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0)
puts vec4 = vec1 + vec3
```

```
## [1] 11 22 33 14 25 36 17 28 39
```

7.3 Vector Indexing

Vectors can be indexed by using the '[]' operator:

```
puts vec4[3]
```

```
## [1] 33
```

We can also index a vector with another vector. For example, in the code bellow, we take elements 1, 3, 5, and 7 from vec3:

```
puts vec4[R.c(1, 3, 5, 7)]
```

```
## [1] 11 33 25 17
```

Repeating an index and having indices out of order is valid code:

```
puts vec4[R.c(1, 3, 3, 1)]
```

```
## [1] 11 33 33 11
```

It is also possible to index a vector with a negative number or negative vector. In these cases the indexed values are not returned:

```
puts vec4[-3]
puts vec4[-R.c(1, 3, 5, 7)]
```

```
## [1] 11 22 14 25 36 17 28 39
## [1] 22 14 36 28 39
```

If an index is out of range, a missing value (NA) will be reported.

```
puts vec4[30]
```

```
## [1] NA
```

It is also possible to index a vector by range:

```
puts vec4[(2..5)]
```

```
## [1] 22 33 14 25
```

Elements in a vector can be named using the 'names' attribute of a vector:

```
full_name = R.c("Rodrigo", "A", "Botafogo")
full_name.names = R.c("First", "Middle", "Last")
puts full_name
```

```
##      First     Middle       Last
##  "Rodrigo"        "A" "Botafogo"
```

Or it can also be named by using the 'c' function with named paramenters:

```
full_name = R.c(First: "Rodrigo", Middle: "A", Last: "Botafogo")
puts full_name
```

```
##      First     Middle       Last
##  "Rodrigo"        "A" "Botafogo"
```

7.4 Extracting Native Ruby Types from a Vector

Vectors created with 'R.c' are of class R::Vector. You might have noticed that when indexing a vector, a new vector is returned, even if this vector has one single element. In order to use R::Vector with other ruby classes it might be necessary to extract the actual Ruby native type from the vector. In order to do this extraction the '>>' operator is used.

```
puts vec4
puts vec4 >> 0
puts vec4 >> 4
```

```
## [1] 11 22 33 14 25 36 17 28 39
## 11.0
## 25.0
```

Note that indexing with '>>' starts at 0 and not at 1, also, we cannot do negative indexing.

8 Accessing R variables

Galaaz allows Ruby to access variables created in R. For example, the 'mtcars' data set is available in R and can be accessed from Ruby by using the 'tilda' operator followed by the symbol for the variable, in this case ':mtcar'. In the code bellow method 'outputs' is used to output the 'mtcars' data set nicely formatted in HTML by use of the 'kable' and 'kable_styling' functions. Method 'outputs' is only available when used with 'gknit'.

```
outputs (~:mtcars).kable.kable_styling
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

9 Matrix

A matrix is a collection of elements organized as a two dimensional table. A matrix can be created by the 'matrix' function:

```
mat = R.matrix(R.c(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0),
               nrow: 3,
               ncol: 3)

puts mat
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

Note that matrices data is organized by column first. It is possible to organize the matrix memory by row first passing an extra argument to the 'matrix' function:

```
mat_row = R.matrix(R.c(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0),
                   nrow: 3,
                   ncol: 3,
                   byrow: true)

puts mat_row
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

9.1 Indexing a Matrix

A matrix can be indexed by [row, column]:

```
puts mat_row[1, 1]
puts mat_row[2, 3]
```

```
## [1] 1
## [1] 6
```

It is possible to index an entire row or column with the ':all' keyword

```
puts mat_row[1, :all]
puts mat_row[:all, 2]
```

```
## [1] 1 2 3
## [1] 2 5 8
```

Indexing with a vector is also possible for matrices. In the following example we want rows 1 and 3 and columns 2 and 3 building a 2 x 2 matrix.

```
puts mat_row[R.c(1, 3), R.c(2, 3)]
```

```
##      [,1] [,2]
## [1,]    2    3
## [2,]    8    9
```

Matrices can be combined with functions 'rbind':

```
puts mat_row.rbind(mat)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
## [4,]    1    4    7
## [5,]    2    5    8
## [6,]    3    6    9
```

and 'cbind':

```
puts mat_row.cbind(mat)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    2    3    1    4    7
## [2,]    4    5    6    2    5    8
## [3,]    7    8    9    3    6    9
```

10 List

A list is a data structure that can contain sublists of different types, while vector and matrix can only hold one type of element.

```
nums = R.c(1.0, 2.0, 3.0)
strs = R.c("a", "b", "c", "d")
bool = R.c(true, true, false)
lst = R.list(nums: nums, strs: strs, bool: bool)
puts lst
```

```
## $nums
## [1] 1 2 3
##
## $strs
## [1] "a" "b" "c" "d"
##
## $bool
## [1]  TRUE  TRUE FALSE
```

Note that 'lst' elements are named elements.

10.1 List Indexing

List indexing, also called slicing, is done using the '[]' operator and the '[[]]' operator. Let's first start with the '[]' operator. The list above has three sublist indexing with '[]' will return one of the sublists.

```
puts lst[1]
```

```
## $nums
## [1] 1 2 3
```

Note that when using '[]' a new list is returned. When using the double square bracket operator the value returned is the actual element of the list in the given position and not a slice of the original list

```
puts lst[[1]]
```

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

When elements are named, as dones with lst, indexing can be done by name:

```
puts lst[['bool']][[1]] >> 0
```

```
## true
```

In this example, first the 'bool' element of the list was extracted, not as a list, but as a vector, then the first element of the vector was extracted (note that vectors also accept the '[[]]' operator) and then the vector was indexed by its first element, extracting the native Ruby type.

11 Data Frame

A data frame is a table like structure in which each column has the same number of rows. Data frames are the basic structure for storing data for data analysis. We have already seen a data frame previously when we accessed variable '~:mtcars'. In order to create a data frame, function 'data___frame' is used:

```
df = R.data__frame(
  year: R.c(2010, 2011, 2012),
  income: R.c(1000.0, 1500.0, 2000.0))

puts df
```

```
##   year income
## 1 2010   1000
## 2 2011   1500
## 3 2012   2000
```

11.1 Data Frame Indexing

A data frame can be indexed the same way as a matrix, by using '[row, column]', where row and column can either be a numeric or the name of the row or column

```
puts (~:mtcars).head
puts (~:mtcars)[1, 2]
puts (~:mtcars)['Datsun 710', 'mpg']
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
## [1] 6
## [1] 22.8
```

Extracting a column from a data frame as a vector can be done by using the double square bracket operator:

```
puts (~:mtcars)[['mpg']]
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

A data frame column can also be accessed as if it were an instance variable of the data frame:

```
puts (~:mtcars).mpg
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

Slicing a data frame can be done by indexing it with a vector (we use 'head' to reduce the output):

```
puts (~:mtcars)[R.c('mpg', 'hp')].head
```

```
##                    mpg  hp
## Mazda RX4         21.0 110
## Mazda RX4 Wag     21.0 110
## Datsun 710        22.8  93
## Hornet 4 Drive    21.4 110
## Hornet Sportabout 18.7 175
## Valiant           18.1 105
```

A row slice can be obtained by indexing by row and using the ':all' keyword for the column:

```
puts (~:mtcars)[R.c('Datsun 710', 'Camaro Z28'), :all]
```

```
##             mpg cyl disp  hp drat   wt  qsec vs am gear carb
## Datsun 710 22.8   4  108  93 3.85 2.32 18.61  1  1    4    1
## Camaro Z28 13.3   8  350 245 3.73 3.84 15.41  0  0    3    4
```

Finally, a data frame can also be indexed with a logical vector. In this next example, the 'am' column of :mtcars is compared with 0 (with method 'eq'). When 'am' is equal to 0 the car is automatic. So, by doing '(~:mtcars).am.eq 0' a logical vector is created with 'true' whenever 'am' is 0 and 'false' otherwise.

```
# obtain a vector with 'true' for cars with automatic transmission
automatic = (~:mtcars).am.eq 0
puts automatic
```

```
##  [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [12]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE
## [23]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

Using this logical vector, the data frame is indexed, returning a new data frame in which all cars have automatic transmission.

```
# slice the data frame by using this vector
puts (~:mtcars)[automatic, :all]
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
```

```
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
```

12 Writing Expressions in Galaaz

Galaaz extends Ruby to work with complex expressions, similar to R's expressions build with 'quote' (base R) or 'quo' (tidyverse). Let's take a look at some of those expressions.

12.1 Expressions from operators

The code bellow creates an expression summing two symbols

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

```
## a + b
```

We can build any complex mathematical expression

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

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

It is also possible to use inequality operators in building expressions

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

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

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 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.

12.2 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 executed. Now, let's say that 'x' is an angle of 45° and we acttually want our expression to be $y = 0.850...$. 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)
```

12.3 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
```

13 Manipulating Data

One of the major benefits of Galaaz is to bring strong data manipulation to Ruby. The following examples were extracted from Hardley's "R for Data Science" (https://r4ds.had.co.nz/). This is a highly recommended book for those not already familiar with the 'tidyverse' style of programming in R. In the sections to follow, we will limit ourselves to convert the R code to Galaaz.

For these examples, we will investigate the nycflights13 data set available on the package by the same name. We use function 'R.install_and_loads' that checks if the library is available locally, and if not, installs it. This data frame contains all 336,776 flights that departed from New York City in 2013. The data comes from the US Bureau of Transportation Statistics.

```
R.install_and_loads('nycflights13')
R.library('dplyr')
```

```
flights = ~:flights
puts flights.head.as__data__frame
```

```
##   year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time
## 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
##   arr_delay carrier flight tailnum origin dest air_time distance hour
## 1        11      UA   1545  N14228    EWR  IAH      227     1400    5
## 2        20      UA   1714  N24211    LGA  IAH      227     1416    5
## 3        33      AA   1141  N619AA    JFK  MIA      160     1089    5
## 4       -18      B6    725  N804JB    JFK  BQN      183     1576    5
## 5       -25      DL    461  N668DN    LGA  ATL      116      762    6
## 6        12      UA   1696  N39463    EWR  ORD      150      719    5
##   minute           time_hour
## 1     15 2013-01-01 05:00:00
## 2     29 2013-01-01 05:00:00
## 3     40 2013-01-01 05:00:00
```

```
## 4      45 2013-01-01 05:00:00
## 5       0 2013-01-01 06:00:00
## 6      58 2013-01-01 05:00:00
```

13.1 Filtering rows with Filter

In this example we filter the flights data set by giving to the filter function two expressions: the first :month.eq 1

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

```
##   year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time
## 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
##   arr_delay carrier flight tailnum origin dest air_time distance hour
## 1        11      UA   1545  N14228    EWR  IAH      227     1400    5
## 2        20      UA   1714  N24211    LGA  IAH      227     1416    5
## 3        33      AA   1141  N619AA    JFK  MIA      160     1089    5
## 4       -18      B6    725  N804JB    JFK  BQN      183     1576    5
## 5       -25      DL    461  N668DN    LGA  ATL      116      762    6
## 6        12      UA   1696  N39463    EWR  ORD      150      719    5
##   minute           time_hour
## 1     15 2013-01-01 05:00:00
## 2     29 2013-01-01 05:00:00
## 3     40 2013-01-01 05:00:00
## 4     45 2013-01-01 05:00:00
## 5      0 2013-01-01 06:00:00
## 6     58 2013-01-01 05:00:00
```

13.2 Logical Operators

All flights that departed in November of December

```
puts flights.filter((:month.eq 11) | (:month.eq 12)).head.as__data__frame
```

```
##   year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time
## 1 2013    11   1        5           2359         6      352            345
## 2 2013    11   1       35           2250       105      123           2356
## 3 2013    11   1      455            500        -5      641            651
## 4 2013    11   1      539            545        -6      856            827
## 5 2013    11   1      542            545        -3      831            855
## 6 2013    11   1      549            600       -11      912            923
##   arr_delay carrier flight tailnum origin dest air_time distance hour
## 1         7      B6    745  N568JB    JFK  PSE      205     1617   23
## 2        87      B6   1816  N353JB    JFK  SYR       36      209   22
## 3       -10      US   1895  N192UW    EWR  CLT       88      529    5
## 4        29      UA   1714  N38727    LGA  IAH      229     1416    5
```

```
## 5        -24       AA    2243  N5CLAA     JFK  MIA       147      1089     5
## 6        -11       UA     303  N595UA     JFK  SFO       359      2586     6
##    minute           time_hour
## 1      59 2013-11-01 23:00:00
## 2      50 2013-11-01 22:00:00
## 3       0 2013-11-01 05:00:00
## 4      45 2013-11-01 05:00:00
## 5      45 2013-11-01 05:00:00
## 6       0 2013-11-01 06:00:00
```

The same as above, but using the 'in' operator. In R, it is possible to define many operators by doing %%. The %in% operator checks if a value is in a vector. In order to use those operators from Galaaz the '._' method is used, where the first argument is the operator's symbol, in this case ':in' and the second argument is the vector:

```
puts flights.filter(:month._ :in, R.c(11, 12)).head.as__data__frame
```

```
##    year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time
## 1 2013    11   1        5           2359         6      352            345
## 2 2013    11   1       35           2250       105      123           2356
## 3 2013    11   1      455            500        -5      641            651
## 4 2013    11   1      539            545        -6      856            827
## 5 2013    11   1      542            545        -3      831            855
## 6 2013    11   1      549            600       -11      912            923
##    arr_delay carrier flight tailnum origin dest air_time distance hour
## 1          7      B6    745  N568JB    JFK  PSE      205     1617   23
## 2         87      B6   1816  N353JB    JFK  SYR       36      209   22
## 3        -10      US   1895  N192UW    EWR  CLT       88      529    5
## 4         29      UA   1714  N38727    LGA  IAH      229     1416    5
## 5        -24      AA   2243  N5CLAA    JFK  MIA      147     1089    5
## 6        -11      UA    303  N595UA    JFK  SFO      359     2586    6
##    minute           time_hour
## 1      59 2013-11-01 23:00:00
## 2      50 2013-11-01 22:00:00
## 3       0 2013-11-01 05:00:00
## 4      45 2013-11-01 05:00:00
## 5      45 2013-11-01 05:00:00
## 6       0 2013-11-01 06:00:00
```

13.3 Filtering with NA (Not Available)

Let's first create a 'tibble' with a Not Available value (R::NA). Tibbles are a modern version of a data frame and operate very similarly to one. It differs in how it outputs the values and the result of some subsetting operations that are more consistent than what is obtained from data frame.

```
df = R.tibble(x: R.c(1, R::NA, 3))
puts df.as__data__frame
```

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

Now filtering by :x > 1 shows all lines that satisfy this condition, where the row with R:NA does not.

```
puts df.filter(:x > 1).as__data__frame
```

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

To match an NA use method 'is___na'

```
puts df.filter((:x.is__na) | (:x > 1)).as__data__frame
```

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

13.4 Arrange Rows with arrange

Arrange reorders the rows of a data frame by the given arguments.

```
puts flights.arrange(:year, :month, :day).head.as__data__frame
```

```
##   year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time
## 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
##   arr_delay carrier flight tailnum origin dest air_time distance hour
## 1        11      UA   1545  N14228    EWR  IAH      227     1400    5
## 2        20      UA   1714  N24211    LGA  IAH      227     1416    5
## 3        33      AA   1141  N619AA    JFK  MIA      160     1089    5
## 4       -18      B6    725  N804JB    JFK  BQN      183     1576    5
## 5       -25      DL    461  N668DN    LGA  ATL      116      762    6
## 6        12      UA   1696  N39463    EWR  ORD      150      719    5
##   minute           time_hour
## 1     15 2013-01-01 05:00:00
## 2     29 2013-01-01 05:00:00
## 3     40 2013-01-01 05:00:00
## 4     45 2013-01-01 05:00:00
## 5      0 2013-01-01 06:00:00
## 6     58 2013-01-01 05:00:00
```

To arrange in descending order, use function 'desc'

```
puts flights.arrange(:dep_delay.desc).head.as__data__frame
```

```
##   year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time
## 1 2013     1   9      641            900      1301     1242           1530
## 2 2013     6  15     1432           1935      1137     1607           2120
## 3 2013     1  10     1121           1635      1126     1239           1810
## 4 2013     9  20     1139           1845      1014     1457           2210
## 5 2013     7  22      845           1600      1005     1044           1815
## 6 2013     4  10     1100           1900       960     1342           2211
```

```
##   arr_delay carrier flight tailnum origin dest air_time distance hour
## 1      1272      HA     51  N384HA    JFK  HNL      640     4983    9
## 2      1127      MQ   3535  N504MQ    JFK  CMH       74      483   19
## 3      1109      MQ   3695  N517MQ    EWR  ORD      111      719   16
## 4      1007      AA    177  N338AA    JFK  SFO      354     2586   18
## 5       989      MQ   3075  N665MQ    JFK  CVG       96      589   16
## 6       931      DL   2391  N959DL    JFK  TPA      139     1005   19
##   minute           time_hour
## 1      0 2013-01-09 09:00:00
## 2     35 2013-06-15 19:00:00
## 3     35 2013-01-10 16:00:00
## 4     45 2013-09-20 18:00:00
## 5      0 2013-07-22 16:00:00
## 6      0 2013-04-10 19:00:00
```

13.5 Selecting columns

To select specific columns from a dataset we use function 'select':

```
puts flights.select(:year, :month, :day).head.as__data__frame
```

```
##   year month day
## 1 2013     1   1
## 2 2013     1   1
## 3 2013     1   1
## 4 2013     1   1
## 5 2013     1   1
## 6 2013     1   1
```

It is also possible to select column in a given range

```
puts flights.select(:year.up_to :day).head.as__data__frame
```

```
##   year month day
## 1 2013     1   1
## 2 2013     1   1
## 3 2013     1   1
## 4 2013     1   1
## 5 2013     1   1
## 6 2013     1   1
```

Select all columns that start with a given name sequence

```
puts flights.select(E.starts_with('arr')).head.as__data__frame
```

```
##   arr_time arr_delay
## 1      830        11
## 2      850        20
## 3      923        33
## 4     1004       -18
## 5      812       -25
## 6      740        12
```

Other functions that can be used:

- ends_with("xyz"): matches names that end with "xyz".
- contains("ijk"): matches names that contain "ijk".
- matches("(.)\1"): selects variables that match a regular expression. This one matches any variables that contain repeated characters.
- num_range("x", (1..3)): matches x1, x2 and x3

A helper function that comes in handy when we just want to rearrange column order is 'Everything':

```
puts flights.select(:year, :month, :day, E.everything).head.as__data__frame
```

```
##   year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time
## 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
##   arr_delay carrier flight tailnum origin dest air_time distance hour
## 1        11      UA   1545  N14228    EWR  IAH      227     1400    5
## 2        20      UA   1714  N24211    LGA  IAH      227     1416    5
## 3        33      AA   1141  N619AA    JFK  MIA      160     1089    5
## 4       -18      B6    725  N804JB    JFK  BQN      183     1576    5
## 5       -25      DL    461  N668DN    LGA  ATL      116      762    6
## 6        12      UA   1696  N39463    EWR  ORD      150      719    5
##   minute           time_hour
## 1     15 2013-01-01 05:00:00
## 2     29 2013-01-01 05:00:00
## 3     40 2013-01-01 05:00:00
## 4     45 2013-01-01 05:00:00
## 5      0 2013-01-01 06:00:00
## 6     58 2013-01-01 05:00:00
```

13.6 Add variables to a dataframe with 'mutate'

```
flights_sm = flights.
               select((:year.up_to :day),
                      E.ends_with('delay'),
                      :distance,
                      :air_time)

puts flights_sm.head.as__data__frame
```

```
##   year month day dep_delay arr_delay distance air_time
## 1 2013     1   1         2        11     1400      227
## 2 2013     1   1         4        20     1416      227
## 3 2013     1   1         2        33     1089      160
## 4 2013     1   1        -1       -18     1576      183
## 5 2013     1   1        -6       -25      762      116
## 6 2013     1   1        -4        12      719      150
```

```
flights_sm = flights_sm.
               mutate(gain: :dep_delay - :arr_delay,
                      speed: :distance / :air_time * 60)
puts flights_sm.head.as__data__frame
```

```
##   year month day dep_delay arr_delay distance air_time gain    speed
## 1 2013     1   1         2        11     1400      227   -9 370.0441
## 2 2013     1   1         4        20     1416      227  -16 374.2731
## 3 2013     1   1         2        33     1089      160  -31 408.3750
## 4 2013     1   1        -1       -18     1576      183   17 516.7213
## 5 2013     1   1        -6       -25      762      116   19 394.1379
## 6 2013     1   1        -4        12      719      150  -16 287.6000
```

13.7 Summarising data

Function 'summarise' calculates summaries for the data frame. When no 'group_by' is used a single value is obtained from the data frame:

```
puts flights.summarise(delay: E.mean(:dep_delay, na__rm: true)).as__data__frame
```

```
##      delay
## 1 12.63907
```

When a data frame is groupe with 'group_by' summaries apply to the given group:

```
by_day = flights.group_by(:year, :month, :day)
puts by_day.summarise(delay: :dep_delay.mean(na__rm: true)).head.as__data__frame
```

```
##   year month day     delay
## 1 2013     1   1 11.548926
## 2 2013     1   2 13.858824
## 3 2013     1   3 10.987832
## 4 2013     1   4  8.951595
## 5 2013     1   5  5.732218
## 6 2013     1   6  7.148014
```

Next we put many operations together by pipping them one after the other:

```
delays = flights.
           group_by(:dest).
           summarise(
             count: E.n,
             dist: :distance.mean(na__rm: true),
             delay: :arr_delay.mean(na__rm: true)).
           filter(:count > 20, :dest != "NHL")

puts delays.as__data__frame.head
```

```
##   dest count      dist     delay
## 1  ABQ   254 1826.0000  4.381890
## 2  ACK   265  199.0000  4.852273
## 3  ALB   439  143.0000 14.397129
## 4  ATL 17215  757.1082 11.300113
## 5  AUS  2439 1514.2530  6.019909
```

```
## 6   AVL   275  583.5818  8.003831
```

14 Using Data Table

```
R.library('data.table')
R.install_and_loads('curl')

input = "https://raw.githubusercontent.com/Rdatatable/data.table/master/vignettes/flights14
flights = R.fread(input)
puts flights
puts flights.dim
```

```
##         year month day dep_delay arr_delay carrier origin dest air_time
##      1: 2014     1   1        14        13      AA    JFK  LAX      359
##      2: 2014     1   1        -3        13      AA    JFK  LAX      363
##      3: 2014     1   1         2         9      AA    JFK  LAX      351
##      4: 2014     1   1        -8       -26      AA    LGA  PBI      157
##      5: 2014     1   1         2         1      AA    JFK  LAX      350
##     ---
## 253312: 2014    10  31         1       -30      UA    LGA  IAH      201
## 253313: 2014    10  31        -5       -14      UA    EWR  IAH      189
## 253314: 2014    10  31        -8        16      MQ    LGA  RDU       83
## 253315: 2014    10  31        -4        15      MQ    LGA  DTW       75
## 253316: 2014    10  31        -5         1      MQ    LGA  SDF      110
##         distance hour
##      1:     2475    9
##      2:     2475   11
##      3:     2475   19
##      4:     1035    7
##      5:     2475   13
##     ---
## 253312:     1416   14
## 253313:     1400    8
## 253314:      431   11
## 253315:      502   11
## 253316:      659    8
## [1] 253316     11
```

```
data_table = R.data__table(
  ID: R.c("b","b","b","a","a","c"),
  a: (1..6),
  b: (7..12),
  c: (13..18)
)

puts data_table
puts data_table.ID
```

```
##    ID a  b  c
## 1:  b 1  7 13
```

```
## 2:   b 2  8 14
## 3:   b 3  9 15
## 4:   a 4 10 16
## 5:   a 5 11 17
## 6:   c 6 12 18
## [1] "b" "b" "b" "a" "a" "c"
```

```
# subset rows in i
ans = flights[(:origin.eq "JFK") & (:month.eq 6)]
puts ans.head

# Get the first two rows from flights.

ans = flights[(1..2)]
puts ans

# Sort flights first by column origin in ascending order, and then by dest in descending or

# ans = flights[E.order(:origin, -(:dest))]
# puts ans.head
```

```
##    year month day dep_delay arr_delay carrier origin dest air_time
## 1: 2014     6   1        -9        -5      AA    JFK  LAX      324
## 2: 2014     6   1       -10       -13      AA    JFK  LAX      329
## 3: 2014     6   1        18        -1      AA    JFK  LAX      326
## 4: 2014     6   1        -6       -16      AA    JFK  LAX      320
## 5: 2014     6   1        -4       -45      AA    JFK  LAX      326
## 6: 2014     6   1        -6       -23      AA    JFK  LAX      329
##    distance hour
## 1:     2475    8
## 2:     2475   12
## 3:     2475    7
## 4:     2475   10
## 5:     2475   18
## 6:     2475   14
##    year month day dep_delay arr_delay carrier origin dest air_time
## 1: 2014     1   1        14        13      AA    JFK  LAX      359
## 2: 2014     1   1        -3        13      AA    JFK  LAX      363
##    distance hour
## 1:     2475    9
## 2:     2475   11
```

```
# Select column(s) in j
# select arr_delay column, but return it as a vector.

ans = flights[:all, :arr_delay]
puts ans.head

# Select arr_delay column, but return as a data.table instead.

ans = flights[:all, :arr_delay.list]
puts ans.head
```

```
ans = flights[:all, E.list(:arr_delay, :dep_delay)]
```

```
## [1]  13  13   9 -26   1   0
##    arr_delay
## 1:        13
## 2:        13
## 3:         9
## 4:       -26
## 5:         1
## 6:         0
```

15 Graphics in Galaaz

Creating graphics in Galaaz is quite easy, as it can use all the power of ggplot2. There are many resources in the web that teaches ggplot, so here we give a quick example of ggplot integration with Ruby. We continue to use the :mtcars dataset and we will plot a diverging bar plot, showing cars that have 'above' or 'below' gas consuption. Let's first prepare the data frame with the necessary data:

```
# copy the R variable :mtcars to the Ruby mtcars variable
mtcars = ~:mtcars

# create a new column 'car_name' to store the car names so that it can be
# used for plotting. The 'rownames' of the data frame cannot be used as
# data for plotting
mtcars.car_name = R.rownames(:mtcars)

# compute normalized mpg and add it to a new column called mpg_z
# Note that the mean value for mpg can be obtained by calling the 'mean'
# function on the vector 'mtcars.mpg'.  The same with the standard
# deviation 'sd'.  The vector is then rounded to two digits with 'round 2'
mtcars.mpg_z = ((mtcars.mpg - mtcars.mpg.mean)/mtcars.mpg.sd).round 2

# create a new column 'mpg_type'. Function 'ifelse' is a vectorized function
# that looks at every element of the mpg_z vector and if the value is below
# 0, returns 'below', otherwise returns 'above'
mtcars.mpg_type = (mtcars.mpg_z < 0).ifelse("below", "above")

# order the mtcar data set by the mpg_z vector from smaler to larger values
mtcars = mtcars[mtcars.mpg_z.order, :all]

# convert the car_name column to a factor to retain sorted order in plot
mtcars.car_name = mtcars.car_name.factor levels: mtcars.car_name

# let's look at the final data frame
puts mtcars.head
```

```
##                      mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
```

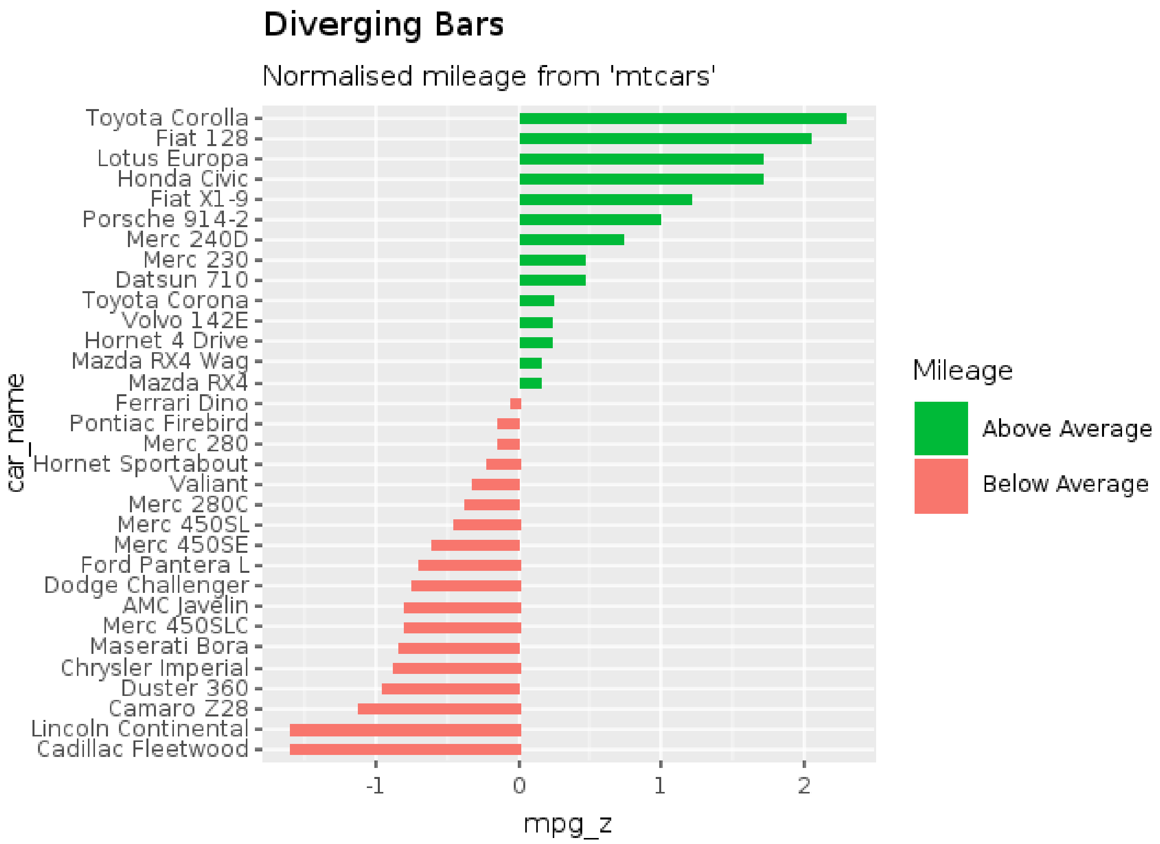
```
## Camaro Z28          13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
## Duster 360          14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
## Chrysler Imperial   14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
## Maserati Bora       15.0   8  301 335 3.54 3.570 14.60  0  1    5    8
##                                car_name mpg_z mpg_type
## Cadillac Fleetwood   Cadillac Fleetwood -1.61    below
## Lincoln Continental Lincoln Continental -1.61    below
## Camaro Z28                   Camaro Z28 -1.13    below
## Duster 360                   Duster 360 -0.96    below
## Chrysler Imperial     Chrysler Imperial -0.89    below
## Maserati Bora             Maserati Bora -0.84    below
```

Now, lets plot the diverging bar plot. When using gKnit, there is no need to call 'R.awt' to create a plotting device, since gKnit does take care of it. Galaaz provides integration with ggplot. The interested reader should check online for more information on ggplot, since it is outside the scope of this manual describing how ggplot works. We give here but a brief description on how this plot is generated.

ggplot implements the 'grammar of graphics'. In this approach, plots are build by adding layers to the plot. On the first layer we describe what we want on the 'x' and 'y' axis of the plot. In this case, we have 'car_name' on the 'x' axis and 'mpg_z' on the 'y' axis. Then the type of graph is specified by adding 'geom_bar' (for a bar graph). We specify that our bars should be filled using 'mpg_type', which is either 'above' or 'bellow' giving then two colours for filling. On the next layer we specify the labels for the graph, then we add the title and subtitle. Finally, in a bar chart usually bars go on the vertical direction, but in this graph we want the bars to be horizontally layed so we add 'coord_flip'.

```ruby
require 'ggplot'

puts mtcars.ggplot(E.aes(x: :car_name, y: :mpg_z, label: :mpg_z)) +
     R.geom_bar(E.aes(fill: :mpg_type), stat: 'identity', width: 0.5) +
     R.scale_fill_manual(name: 'Mileage',
                         labels: R.c('Above Average', 'Below Average'),
                         values: R.c('above': '#00ba38', 'below': '#f8766d')) +
     R.labs(subtitle: "Normalised mileage from 'mtcars'",
            title: "Diverging Bars") +
     R.coord_flip
```



16 Coding with Tidyverse

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. To see the problem, let's first define a 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
```

and now, let's look at this code:

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

It 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
```

16.1 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
```

16.2 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 defined 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'.

```
puts ~: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
```

We then create the 'my_summarize' method and call it passing the R data frame and the group by variable ':g1':

```
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 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.

16.3 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 "\n"
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'.

16.4 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 "\n"
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. . . .]

16.5 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 '*':

```
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
```

16.6 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'.

16.7 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, Retur
## 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
```

[TO BE CONTINUED…]

17 Contributing

- Fork it
- Create your feature branch (git checkout -b my-new-feature)
- Write Tests!
- Commit your changes (git commit -am 'Add some feature')
- Push to the branch (git push origin my-new-feature)
- Create new Pull Request