High Quality Scientific Plotting with Ruby in GraalVM

Also: Allowing R to use classes, modules, blocks, etc.

Rodrigo Botafogo

19 October 2018

1 Introduction

Ruby is a dynamic, interpreted, reflective, object-oriented, general-purpose programming language. It was designed and developed in the mid-1990s by Yukihiro "Matz" Matsumoto in Japan. It reached high popularity with the development of Ruby on Rails (RoR) by David Heinemeier Hansson. RoR is a web application framework which was first release circa 2005 and makes extensive use of Ruby's metaprogramming features. With the advend of RoR, Ruby became extremely popular and it peeked in popularity around 2008 according to the Tiobe index (https://www.tiobe.com/tiobe-index/ruby/). From 2008 to 2015, it's popularity declined consistently and then started picking up again during the next 3 years. At the time of this writing (November 2018), Ruby is ranked 16th in the Tiobe index.

Python, considered a similar language to Ruby with similar features ranks 4th in the index. The first three positions are taken by Java, C and C++. One criticism often heard about Ruby, is that it is useful only for web applications while Python, with similar features has more diverse libraries, being useful for web applications with the Django framework, but also for scientific applications such as statistics, data analysis, big data, biology, etc. This criticism is by no way wrong. Although Ruby can do much more than just web applications: https://github.com/markets/awesome-ruby, for scientific computing, Ruby lags way behind Python and R, the two most prestigous languages in the field, with R being prefered by statisticians while Python is prefered by everyone else, because of it's gentle learning curve and more "natural" programming paradigm.

Until recently, there was no real perspective for Ruby to bridge this gap and have even the most basic scientific computing infrastructure. Comes GraalVM into the picture:

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

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

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

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

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

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

As stated above, GraalVM is a *universal* virtual machine that allows Ruby and R (and other languages) to run on the same environment. GraalVM allows polyglot applications to *seamlessly* interact with one another and pass values from one language to the other. Based on GraalVM, the Galaaz project was started. Galaaz indends to tightly couple Ruby and R and allow those languages to *seamlessly* interact in a way that the user will be unaware of such interaction.

Library wrapping is an usual way of bringing features from one library into another language. For instance, whenever Python needs to perform operations efficiently, C libraries are wrapped in Python. For the Python developer, the existence of such C library is of no concern. The problem with library wrapping is that for any new library of interest, there is the need to hand craft a new wrapper. With Galaaz, the same concept of wrapping was done, but instead of wrapping a single C or R library, Galaaz wraps the whole of the R language. Doing so, all thousands of R libraries are immediately available to Ruby developers and any new library developed in R will also become available without requiring a new wrapping effort.

In this article, the graphing ggplot2 library from R will be accessed by Ruby transparently, bringing to Ruby the power of high quality scientific plotting. It might seem, from the exposed above, that Galaaz mainly benefits Ruby developers and might be of no consequence to the R developer. This article will however show that migrating from R to Ruby with Galaaz is a matter of small syntactic changes. Furthermore, R lacks some fundamental constructs for code reuse and large system construction. Using Galaaz, the R developer can easily migrate to a powerful OO language, at virtually no cost and then, as needs requires, she can add them to her toolbox.

In this article we will explore the R ToothGrowth dataset. In doing so, we will create some plots. Furthermore we will create a "Corporate Template" for our plots ensuring that any plot of the same type will have a consistent visualisation.

2 gKnit

This document was written using rmarkdown and the corresponding HTML was generated by the gKnit application. gKnit is a wrapper around the powerful 'knitr' application which converts rmarkdown text to many different output formats such as HTML, Latex, docx, etc. The gKnit tool is still under active development and will soon be released.

In rmarkdown, text and code can be part of the same document, and code blocks are marked with a special markup. Interested readers can easily google 'knitr' and 'rmarkdown'. in gKnit, each Ruby block is evaluated independently and 'eval' in Ruby creates a new scope, so, in order for a variable defined in a block to be accessible in another block, it has to be a global variable, preceded by the '$' sign.

3 Exploring the Dataset

Let start by exploring our selected dataset. In this dataset the response is the length of odontoblasts (cells responsible for tooth growth) in 60 guinea pigs. Each animal received one of three dose levels of vitamin C (0.5, 1, and 2 mg/day) by one of two delivery methods, orange juice or ascorbic acid (a form of vitamin C and coded as VC).

In Galaaz, in order to have access to an R variable pointed by an R symbol we use the corresponding Ruby symbol preceeded by the tilda ('~') function.

```
# Read the R ToothGrowth variable and assign it to the
# Ruby tooth_growth variable
$tooth_growth = ~:ToothGrowth
# convert the dose to a factor
$tooth_growth.dose = $tooth_growth.dose.as__factor

# print the first few elements of the dataset
puts $tooth_growth.head
```

```
##    len supp dose
## 1  4.2   VC  0.5
## 2 11.5   VC  0.5
## 3  7.3   VC  0.5
## 4  5.8   VC  0.5
## 5  6.4   VC  0.5
## 6 10.0   VC  0.5
```

Great! We've managed to read the ToothGrowth dataset and take a look at its elements. Observe that we have three columns in this dataset: 'len', 'supp' and 'dose'. Accessing a column, for example the 'len' column, is done by doing '$tooth_growth.len'.

Let's explore some more details of this dataset. In particular, let's look at its dimensions, structure and summary statistics.

```
puts $tooth_growth.dim
# chdck why NULL
puts R.str(:ToothGrowth)
puts $tooth_growth.summary
```

```
## [1] 60  3
## NULL
##       len        supp     dose
##  Min.   : 4.20   OJ:30   0.5:20
##  1st Qu.:13.07   VC:30   1  :20
##  Median :19.25           2  :20
##  Mean   :18.81
##  3rd Qu.:25.27
##  Max.   :33.90
```

Let's now create our first plot with the given data by accessing ggplot2 from Ruby. For Rubyist that have never seen or used ggplot2, here is the description found on ggplot home page:

```
"ggplot2 is a system for declaratively creating graphics, based on _The Grammar of Graphics_.
You provide the data, tell ggplot2 how to map variables to aesthetics, what graphical
primitives to use, and it takes care of the details."
```

This description might be a bit cryptic and it is best to see it at work to understand it. Basically, in the *grammar of graphics* each component of the plot such as the grid, the axis, the data, title, subtitle, etc. is added to the plot in layers to form the final graphics.

In this plot bellow, the 'dose' is plotted on the 'x' axis and the tooth length on the 'y' axis. Note the specification in the the 'aes' method: 'E.aes(x: :dose, y: :len)', where ':dose' is the 'dose' column of the dataset and ':len' the 'len' column. The 'aes' method is the *aesthetics* for this plot. Then, to this layer, the 'geom_boxplot' is added and the whole plot is printed.

Note also that we have a call to 'R.png' before plotting and 'R.dev___off' after the print statement. 'R.png' opens a 'png' device for writing the plot. When 'R.dev___off' is called, the device is closed and a 'png' file is created. If no name is given to the 'png' function, a file named 'Rplot' is generated, where is the number of the plot. So, this first plot is called 'Rplot001.png'. We can then include the generated 'png' file in this document, by adding an rmarkdown directive.

```
require 'ggplot'

R.png

e = $tooth_growth.ggplot(E.aes(x: :dose, y: :len))
print e + R.geom_boxplot
```

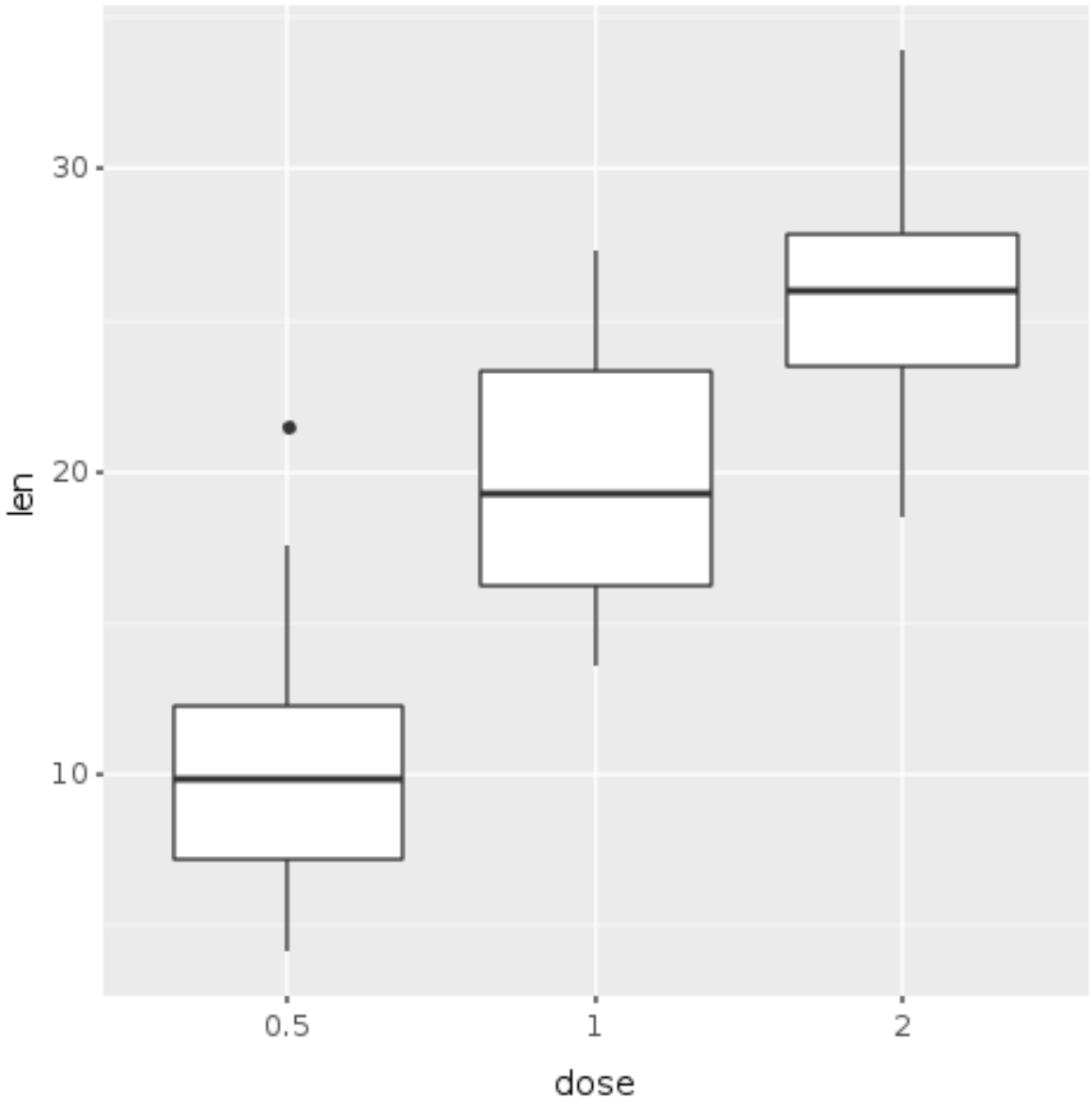

Figure 1: ToothGrowth

```
R.dev__off
```

We've just managed to generate our first plot in Ruby with only two lines of code. This plot, however, if far from being pleasing to the eye.

4 Conclusion

5 Installing Galaaz

5.1 Prerequisites

- GraalVM (>= rc8)

- TruffleRuby
- FastR

The following R packages will be automatically installed when necessary, but could be installed prior to using gKnit if desired:

- ggplot2
- gridExtra
- knitr

Installation of R packages requires a development environment and can be time consuming. In Linux, the gnu compiler and tools should be enough. I am not sure what is needed on the Mac.

5.2 Preparation

- gem install galaaz

5.3 Usage

- gknit