LSINF2132 – Languages and translators

Implementing a domain specific language

Group 40
Arnaud GELLENS and Simon GUSTIN

May 2018

1 Introduction

For the course *Languages and translators*, we were asked to implement a Domain Specific Language using *Scala* and its *Scala-to-Javascript* compiler *Scala.js*. Namely, we had to translate a part of the *Javascript* library *D3.js* (which allows to manage plots on HTML web pages more easily than plain *Javascript*) and use it to create chord plots and migration plots. We then had to produce a Domain Specific Language that would allow the user to create those plots more simply (that is, with "*Scala*-like" abstractions).

2 Functionalities of the DSL

Our DSL allows to create easily two kinds of plots: chord plots and migration plots. We document here the usage of our DSL to do that. In appendix, we show two example codes to create a simple chord plot and migration plot.

2.1 Chord plot

To create a chord plot, several constructors are usable:

```
val plot = ChordPlot(
    List(
        List(1,2,3),
        List(4,5,6),
        List(7,8,9)
    )
)
val plot = ChordPlot(
    "Label A" -> (1,2,3),
    "Label B" -> (4,5,6),
    "Label C" -> (7,8,9)
)
val plot = ChordPlot("path-to/chord-plot-data.json")
```

We can see that we can thus give the data directly to the constructor (as a `List` of `List` or as tuples), or give it a *JSON* file. Several setters are provided as well, once again with several possible formattings:

```
plot.setDimension(460,480)
plot.dimension = (460,480)

plot.setTarget("#playground svg")
plot.target = ("#playground svg")

plot.setLabels(List("Label A", "Label B"))
plot.labels = List("Label A", "Label B")

plot.setColorPalette("#000000", "#FFDD89", "#957244", "#F26223")
plot.colorPalette = List("#000000", "#FFDD89", "#957244", "#F26223")
```

Those setters display a warning message in the developer console when the parameters are faulty and the system tries to cope with them (padding a non-square matrix with 0, ignoring extra labels, ...). When it is not possible, an exception is thrown (and displayed in the developer console) and the program stops. Note also that if no dimension is set, the plot automatically takes the size of its target. This is true for other attributes as well.

By default, the chord plot uses some listeners: when hovering sections of it with the cursor, the particular section and ribbons highlights: when hovering a ribbon, the corresponding data is displayed in a pop-up; when using the mouse wheel, the plot gets zoomed in and out and can be moved by dragging it with the mouse. Those behaviors are disableable using setters in the same way as before.

Then, it is possible to merge two sections of the plot either by calling the plot's `merge` method or simply by clicking two sections. The syntax is rather clear: `plot.merge("Label A" -> "Label B")` to merge the data labelled `"Label A"` into the one labelled `"Label B"`; `data.displayedMatrix = data.displayedMatrix.merge(("Label A", "Label B") -> "Label A and B")` merges `"Label A"` and `"Label B"` into a section now labelled `"Label A and B"`. Note that data is indexable using labels, but also indices in the data matrix.

Once again this behavior can be changed by setting the plot's `focusEvent` to `FocusEvent.none`, `FocusEvent.click` or `FocusEvent.hover`. The setters `dontFocusSections`, `focusSectionsOnClick` and `focusSectionsOnHover` have the same effect. Clicking out of the plot reverts the plot to its previous state (acting like some kind of *Ctrl+Z*).

Finally, the user can set some listeners by themself using the clear following syntax:

```
var i = 0
plot onClick {
    i += 1
    println(s"clicked $i times")
}
plot onClickDown {
    plot.colorPalette = List("#000000", "#AAAAAA", "#957244", "#F26223")
} onClickUp {
    plot.colorPalette = List("#000000", "#FFDD89", "#957244", "#F26223")
}
plot onHover {
    plot.merge("Label A" -> "Label B")
} onHoverOff {
    plot.revert()
}
```

Note that to display the graph, the user has to call the method `draw()` of the plot

2.2 Migration plot

As for the chord plot, several constructors are accessible, to create the data from the code or loading it from a JSON file:

```
val plot = MigrationPlot("path-to/map.geo.json",
    "FIN" -> (1,3,5),
    "GBR" -> (2,2,8),
    "FRA" -> (7,1,4)
)
val plot = MigrationPlot("path-to/map.geo.json", "path-to/data-migration-plot.json")
```

By default, straight arrows are displayed between regions of the plot to represent data (the thickness of them representing the relative size of a migration). Hovering them with the cursor shows a pop-up with the information of the migration (as for ribbons in the chord plots). Hovering a region of the map shows the data of this specific region.

As the class `MigrationPlot` extends the same trait as `ChordPlot`, most of the setters available in it are also available to the `MigrationPlot` (except for the specific attributes and behaviors of the chord plot, such as the color palette or the listeners that merge sections). Obviously, some additional setters are also provided: `colorArrow` and `colorRegion`. Note that it is also the case of most methods and most of the default behaviors we talked about.

2.3 Advanced features

Our DSL computes a certain number of things automatically and manages some data operations internally to avoid potential faults made by the user. Three of them are more important and are cited here.

First, for chord plots, the space between ticks (i.e. hatch marks around the circle) is automatically computed so that there are not too few nor too many of them around the circle. Where to put number labels onto those ticks (and which labels) is also computed automatically, the point being to be humanly readable (and correctly labelled).

Secondly, data stored in matrices are managed using the `RelationMatrix` (without labels) and `LabelizedRelationMatrix` (with labels). The constructors `FlowsMatrix` and `LabelizedFlowsMatrix` ensure that the diagonal is composed only of zeros. The merging of sections is also handled by those objects. Those classes handle the indexing of the elements, columns and rows using the following syntax:

```
matrix(0)(2) // Returns the element at index (0,2)
matrix(0,2) // Returns the element at index (0,2)
matrix(*)(1) // Returns the second column of the matrix
matrix(0)(*) // Returns the first row of the matrix
labelMatrix("A")("B") // Returns the element at index (indexof("A"), indexof("B")
labelMatrix("label A")(2) // Returns the element at index (indexof("Label A"), 2)
labelMatrix(*)("Label B") // Returns all the data that goes to "Label B"
labelMatrix("label A")(*) // Returns all the data that comes from "Label A"
```

Then, the trait extended by the plots manages the history of the plot: when the plot changes (two sections got merged, a label was updated, ...), its previous state is pushed in a history, and is thus retrievable at will. As we said, by default for the chord plot, clicking outside of the plot reverts it to its previous state.

Finally, our plots can be fed JSON files, which is very useful for large sets of data, with a rather straightforward format. The whole parsing and interpretation of those files is done by the DSL (in the classes `ChordPlot` and `MigrationPlot`) so that the user doesn't have to do it by themself.

3 Strong and weak points of the DSL

3.1 Strong points

First, the syntax of our DSL is easy to understand and to read, avoiding boiler plate code as much as possible.

Secondly, our DSL provides a lot of potential functionalities and behaviors, thanks to our use of setters. Tweaking the appearance of the plots is easy and fast. This is also the case of most of the implemented behaviors.

Thirdly, it is possible to use our DSL with different paradigms in mind: people who are used to OOP and Javascript will tend to use chain setters, while people who are used to functional programming would tend to set instance variables using the equal symbol.

Lastly, our DSL allows to create plots that are usable by default. That is, a plot returned by a `new` statement and displayed as is behaves well already without further setting.

3.2 Weak points

First, our encapsulation of *D3.js* removes some functionalities provided by it. A user that knows this library will not be able to do as much things in our DSL.

Then, obviously, deep customization of the plots is hardly achievable. The data used by *D3* being inaccessible from the outside, the user cannot customize the plots as much as they would want.

4 Further improvements

If we had had more time, we could have made a certain number of improvements to our DSL. First, we could have implemented the merging of regions in the migration maps: while the merging of the data is already implemented, this is not the case of the merging of regions on the displayed map.

Second, we would have added ways to set more things in the different plots, in order to approach the customizability of the underlying libraries. The same thing holds for listeners and behaviors.

Lastly, making more possible operations on the data matrices would have been interesting. We would obviously also had had to make it possible for the user to trigger those computations easily.

5 Conclusion

This document summarized what we made for this project. We can say we are rather proud of our final DSL, even though it doesn't feel as polished as we would have wanted. Note that the hardest part of the project seemed to be getting maps to display and Javascript to change the HTML webpage.

Appendix

A Example codes

For the following codes to work, the HTML page must have a `div` element with ID `playground`.

A.1 Chord plot

```
val plot = ChordPlot(
    "Label A" -> (1,2,3)
    "Label B" -> (4,5,6)
    "Label C" -> (3,2,1)
)

plot.setTarget("#playground svg")
    .setDimension(600, 600)
    .updateLabel("Label A" -> "Updated label")

plot.colorPalette = List("#000000", "#FFDD89", "#957244", "#F26223")

plot.draw()

plot onDoubleClick {
    println("double clicked")
}
```

A.2 Migration plot

```
val plot = MigrationPlot(
    "FRA" -> (0, 4, 5, 1, 3),
    "FIN" -> (1, 2, 9, 1, 0),
    "ITA" -> (3, 0, 1, 8, 0),
    "ESP" -> (5, 5, 2, 1, 2),
    "GBR" -> (3, 1, 0, 1, 12)
)

plot.setTarget("#playground svg")
plot.colorArrow = "orange"
plot.colorCountry = "#123456"

plot.draw()

plot onClickDown {
    plot.updateLabel("FRA" -> "France")
} onClickUp {
    plot.revert()
}
```