FUNCTIONAL PEARLS
Composing data visualizations

TOMAS PETRICEK
University of Kent, UK
(*e-mail:* `t.petricek@kent.ac.uk`)

1 Introduction

Let's say we want to create the two charts in Figure 1. The chart on the left is a bar chart that shows two different values for each bar. The chart on the right consists of two line charts that share the X axis and highlight two parts of the timeline with two different colors.

There is a plenty of libraries that can draw bar charts and line charts, but adding those extra features will only be possible if the author already thought about your exact scenario. For example, Google Charts supports the left chart (it is called Dual-X Bar Chart) but there is no way for adding a background, or sharing an axis between charts. The alternative is to use a more low-level library such as D3. In D3 you construct the chart piece by piece, but then you have to tediously transform your values to coordinates in pixels yourself. For scientific plots, you could use an implementation of Grammar of Graphics such as ggplot2, where a chart is a mapping from data to geometric objects (such as points, bars, and lines) and their visual properties (X and Y coordinate, shape and color). However, the range of charts that can be created using this systematic approach is still somewhat limited.

What would an elegant functional approach to data visualization look like? A functional programmer would want a domain-specific language that has a small number of primitives; allow us to define high level abstractions such as a bar chart and its basic building blocks are expressed in terms of domain values such as the exchange rate, rather than pixels.

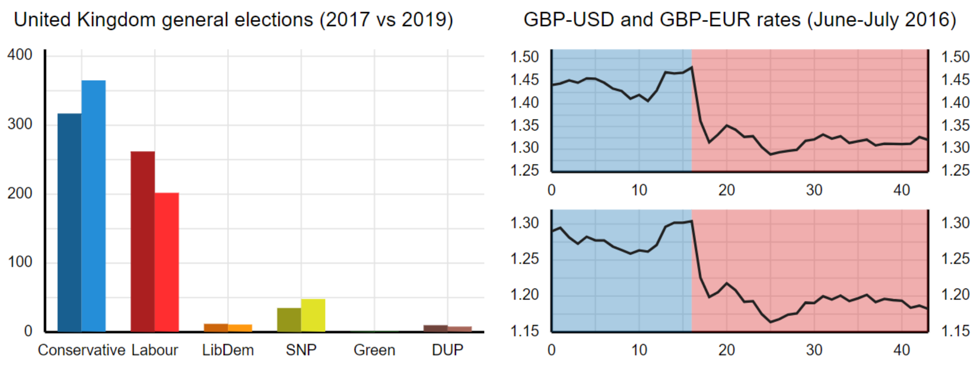

Fig. 1. Two charts about the UK politics: Comparison of election results from 2017 and 2019 (left) and GBP-USD exchange rate with highlighted areas before and after the 23 June 2016 Brexit vote.

As is often the case with domain-specific languages, finding the right primitives is more of an art than science. For this reason, I present my answer – a library named Compost – as a functional pearl. I hope to convince the reader that Compost is elegant and I illustrate this with a wide range of examples. Compost has a number of specific desirable properties:

- Charts are composed from a small number of primitive building blocks using a small number of combinators. In particular, concepts such as bar charts, line charts or charts with aligned axes are expressed in terms of more basic concepts.
- The primitives are specified in domain terms. When drawing a line, the value of an Y coordinate is an exchange rate of 1.36 USD/GBP, not 137 pixels from the bottom.
- Most common chart types can be easily captured as high level abstractions, but there is an elengant way of creating a majority of more interesting custom charts.
- The approach can easily be extended to creating web-based charts that involve animations or interaction with the user.

The presentation in this paper focuses on explaining the primitives and combinators of the domain-specific language. I outline the structure of an implementation, but omit the details. Filling those in requires careful thinking about geometry and projections, but there are no unexpected surprises. A complete F# implementation, including the examples used in this paper, is available at: http://github.com/compostjs.

2 Composing visualization primitives

I will introduce individual features of the Compost library gradually. The first important aspect of Compost is that properties of shapes are defined in terms of domain-specific values. I first explain what this means and then use domain-specific values to specify the core part of the UK election results bar chart.

2.1 Domain-specific values

In the election results chart in Figure 1 (left), the X values are categorical values representing the political parties such as Conservative or Labour. The Y values are numerical values representing the number of seats won such as 365 MPs. When creating data visualizations, those are the values that the user needs to specify. This is akin to most high-level charting libraries such as Google Charts, but in contrast with more flexible libraries like D3.

Our design focuses on two-dimensional charts with X and Y axes. Values mapped to those axes can be either categorical (such as different political parties or countries) or continuous (such as number of votes or exchange rates). The mapping from categorical and continuous values to exact positions on the chart is done automatically. For continuous values, this simply means applying a linear transformation. For categorical values, the mapping is more difficult.

For example, in the UK election results chart, the X axis is categorical. The library automatically divides the available space between the six categorical values (political parties). The value Green does not determine an exact position on the axis, but rather a range. To determine an exact position, we also need to attach a value between 0 and 1 to the categorical value. This identifies a relative position in the available range.

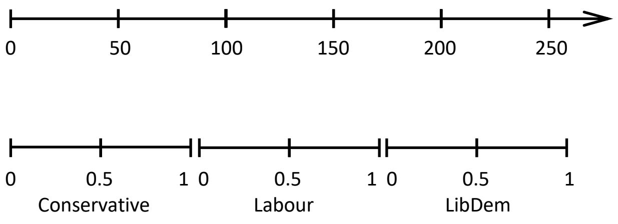

Fig. 2. On a continuous scale (above), an exact position is determined by a number. On a categorical scale (below), an exact position is determined by the category and a numerical ratio from 0 to 1.

Figure 2 illustrates the two kinds of values using the axes from the UK election results chart. More formally, we define a value v as either a continuous value $\mathsf{cont}\ n$ containing any number n or a categorical value $\mathsf{cat}\ c, r$, consisting of a categorical value c (implemented as a string) and a ratio r between 0 and 1:

$$v \;=\; \mathsf{cat}\ c, r \;\mid\; \mathsf{cont}\ n$$

2.2 *Visualization primitives*

$$\begin{array}{rcl} s & = & \mathsf{overlay}\ [s_1, \ldots, s_n] \\ & \mid & \mathsf{text}\ \gamma, v_x, v_y, t \\ & \mid & \mathsf{line}\ \gamma, v_{x1}, v_{y1}, \ldots, v_{xn}, v_{yn} \\ & \mid & \mathsf{fill}\ \gamma, v_{x1}, v_{y1}, \ldots, v_{xn}, v_{yn} \\ & \mid & \mathsf{axis}_{x/y} s \end{array}$$

xx

```
axis_x (axis_y (overlay [
  for party, clr17, clr19, mp17, mp19 in elections →
    padding 0, 10, 0, 10, overlay [
      fill clr, (cat party, 0), (cont 0), (cat party, 0), (cont mp17),
                (cat party, 0.5), (cont mp17), (cat party, 0.5), (cont 0)
      fill clr, (cat party, 0.5), (cont 0), (cat party, 0.5), (cont mp19),
                (cat party, 1), (cont mp19), (cat party, 1), (cont 0) ]
]))
```

2.3 *Inferring scales and projections*

3 Fancy scale features

4 Defininig abstractions

5 Interactive charts

6 Implementation structure

7 Domain-specific language

Domain-specific values

Language of composable charts

Now that we defined our representation of values, we can define a basic composable language of data visualizations. A shape s represents a chart. In the most basic form, it can be text label, line connecting a list of points and a filled polygon determined by a list of points. For line, text and polygon, we also include a parameter γ indicating the color to be used. In addition, a shape can be constructed by overlaying several other shapes:

$$\begin{array}{rcl} s & = & \mathsf{overlay}\ s_1, \ldots, s_n \\ & | & \mathsf{text}\ \gamma, v_x, v_y, t \\ & | & \mathsf{line}\ \gamma, v_{x1}, v_{y1}, \ldots, v_{xn}, v_{yn} \\ & | & \mathsf{fill}\ \gamma, v_{x1}, v_{y1}, \ldots, v_{xn}, v_{yn} \end{array}$$

As an example, consider bar chart with UK and CZ as two categorical values on the X axis and their population in millions (continuous value) on the Y axis. This can be constructed by creating two filled rectangles using fill and overlaying them using overlay:

```
overlay
  fill #0000ff, (cat UK, 0), (cont 00.00), (cat UK, 0), (cont 66.04),
                (cat UK, 1), (cont 66.04), (cat UK, 1), (cont 00.00),
  fill #ff0000, (cat CZ, 0), (cont 00.00), (cat CZ, 0), (cont 10.58),
                (cat CZ, 1), (cont 10.58), (cat CZ, 1), (cont 00.00)
```

The shape specification overlays two bars of different colours. To construct a rectangle, we define four points. For the UK, two of the points have Y value set to 0 (bottom of the bar) and two have it set to 66.04 (top of the bar indiciating the population value). The values on the X axis are categorical UK values – two are on the left as indicated by the ratio 0 and two are on the right as indicated by ratio 1. The ratio is always relative to the category, so the ratios for the second bar are also 0 and 1.

Calculating scales and projections

When processing shape specification such as the one given in the previous section, the Compost library infers the scales for the X and Y axes and computes a projection function that can be used for mapping domain-specific values to pixels. While doing this, it also checks for consistency – all values on one axis have to be either categorical or continuous, but not a mix of the two. The first step is to determine the scales for each of the axes. A scale can be continuous, defined by a minimal and maximal value, or a categorical, defined by a list of categorical values. Note that we do not need to track the ratios used as each category will be allocated equal amount of space, regardless of where in that space shapes need to appear. A scale l is thus defined as:

$$l \quad = \quad \mathsf{categorical}\ c_1, \ldots, c_k \ \mid\ \mathsf{continuous}\ n_{min}, \ldots, n_{max}$$

A scale is obtained by recursively walking over the shape and gradually constructing two scales, for X and Y axis, from the X and Y coordinates that appear in the shape. The

ZU064-05-FPR paper 25 April 2020 1:44

process is fairly straightforward and we won't discuss it in detail (for now). The first value encountered determines what kind of a scale will be constructed. If the type of a later value does not match the type of a scale, the process fails. Once we obtain scales for both X and Y axes, we calculate a projection function for each axis. The function takes a value v together with total space available (say, in pixels) and produces a position in pixels corresponding to v. We also omit the details here (for now), but briefly – for continuous values, we produce a simple linear transformation; for categorical values, we split the available space into k equally sized regions (where k is the number of categorical values in the scale) and then map a categorical value cat c, r to the region corresponding to c according to the ratio r.

Controlling and nesting scales

The process of computing scales can be controlled by additional primitives in the Compost domain-specific language for describing shapes that were not discussed earlier. The following definition lists additional primitives that are also a part of the definition of s. Note that the primitives can be applied either to the X scale or to the Y scale - we write x/y to indicate that a sub-script on those primitives can be set either to x or to y (both scales can be modified by nesting two primitives):

$$\begin{array}{rcl} s & = & (\ldots) \\ & | & \mathsf{axis}_{x/y} s \\ & | & \mathsf{roundScale}_{x/y} s \\ & | & \mathsf{explicitScale}_{x/y}\ l, s \\ & | & \mathsf{nest}_{x/y}\ v_{min}, v_{max}, s \end{array}$$

We first focus on the first three primitives, while the nest primitive will be discussed in the next section. The axis and roundScale constructs could be defined as derived constructs, but it is easier to consider them as primitives for now. The axis primitive simply takes the shape s and draws an axis around the contents of s, using the inferred scale of s to determine values displayed on the axis. This could be a derived construct, because axes can be rendered using lines and text labels. The roundScale operation takes the inferred X or Y scale of the shape s and, if it is a continuous scale, rounds its minimal and maximal values to "nice" numbers. For example, if a continuous scale has minimum 0 and maximum 66.04, the resulting scale would have maximum 70. For categorical scale, the operation does not have any effect. The explicitScale operation is similar, but it replaces the inferred scale with an explicitly provided scale (the type of the inferred scale has to match with the type of the explicitly given scale). For example, assuming populationBar is the example given earlier, we can write:

$$\begin{array}{l} \mathsf{axis}_x\ (\mathsf{axis}_y\ (\mathsf{roundScale}_y \\ \quad (\mathsf{explicitScale}_x\ (\mathsf{categorical}\ \mathsf{CZ,MN,UK})\ \mathsf{populationBar}))) \end{array}$$

This replaces the X axis with an explicitly given one that includes extra country code and also overrides the implicitly inferred order of the categorical values. We then ask Compost to automatically round the Y scale (showing population) so that we get a "nice" number as the maximum. Finally, we add axes around the shape, producing a usual labelled chart. As noted earlier, both axis and roundScale could be defined as derived – axis would have to infer the scales of the nested shape, calculate appropriate labels and insert suitable lines and labels; the roundScale operation would need to infer the scale too and then add an explicit explicitScale with a rounded minimal and maximal value of a continuous scale.

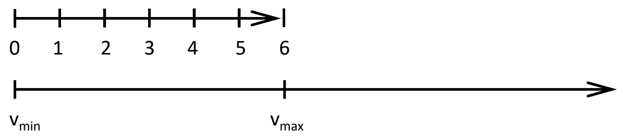

Fig. 3. A continuous scale with values from 0 to 6, nested in another scale.

Nesting of charts and scales

Finally, one more primitive that we added in the above definition and that we have not explained yet is $\mathsf{nest}_{x/y}$. The parameters of the primitive are two values, v_{min} and v_{max} together with a shape s. The primitive makes it possible to nest one scale inside another one. When inferring the scales, Compost infers the scale of the nested shape s. This will then take the area determined by the two points v_{min} and v_{max} in the scale determined by other shapes that are overlaid with the nest shape. The two points are also used to determine the scales of the overall shape, but the scales of s are nested and do not have any effect on the outer scales. In fact, the scale of s does not even have to be of the same type as the scale of the outer shape. Figure 3 illustrates this with an example. Here, we are nesting a continuous scale with values ranging from 0 to 6 into another scale. The values v_{max} and v_{min} do not even have to be continuous. We can, for example, nest a line chart inside a bar of a bar chart and then the two values could be cat One, 0 and cat One, 1 (which are two values that define a region of a categorical scale). One area where nesting of scales is particularly useful is when combining multiple charts in a single view. For example, let's say that we have two line charts showing prices of two different stocks over the same period of time (for simplicity, we can just use UNIX timestamp as a time). The prices of the two stocks are orders of magnitude different, so they cannot fit easily into the same chart. We want to show two line charts side-by-side (one above each other). They should each have their own Y axis (price), but the X axis (time) should be shared. Assuming we have googPrices and fbPrices as two line charts without any axes, covering the same date range, we can write:

```
axis_x (overlay
   (nest_y (cat top-chart, 0), (cat top-chart, 1), (axis_y fbPrices)),
   (nest_y (cat bottom-chart, 0), (cat bottom-chart, 1), (axis_y googPrices)) )
```

Reading the code from the inner-most part to the outer-most, we first add separate Y axes to both of the line charts. Given the difference in the prices, the axes will have quite different values. We then nest the (continuous) Y axes using the nest primitive and, at the same time, implicitly define a new outer Y axis. The outer Y axis is categorical with just two values, top-chart and bottom-chart. This means that the two nested charts will take equal amount

ZU064-05-FPR paper 25 April 2020 1:44

of space, one above the other (each of the charts takes the full space allocated for the category – the minimal value has ratio 0 while the maximal value has ratio 1). [TODO: We really need illustrations for those example charts, but that would be more work!] [TODO: Maybe use this chart as a motivation: https://wellcomeopenresearch.org/articles/4-63/v1]

8 Random ideas

- We could define a type system to catch categorical/continuous value mismatch in a single
- It would be worth thinking about ordinal values, which are categorical but can be sorted.
- There might be other kinds of scales - for example, color scale (can have meaning in scatter plot) or a scale for secondary markers (like sizes of bubbles in a bubble chart). We could really say that every value (including bar chart colors) should be coming from some scale...
- Implementing something like axis or roundScale as an actual derived primitive is a bit tricky, because it needs to invoke a part of the normal rendering workflow (to run the automatic inference of scales on the nested shape) – this might be just implementation issue, but it could be some more basic problem.