

FUNCTIONAL PEARLS

Composable 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 with parts of the timeline highlighted using two different colors.

Many 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. Google Charts (Google, 2020) supports the left chart (it is called Dual-X Bar Chart) but there is no way to add a background, or sharing an axis between charts. The alternative is to use a more low-level library. In D3 (Bostock *et al.*, 2011) you construct the chart piece by piece, but you have to tediously transform your values to coordinates in pixels yourself. For scientific plots, you could use ggplot2 (Wickham, 2016), based on the Grammar of Graphics (Wilkinson, 1999). A chart is a mapping from data to geometric objects (points, bars, lines) and their visual properties (x and y coordinate, shape, 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 that allow us to define high level abstractions such as a bar chart and that uses domain values such as the exchange rate, rather than pixels, in its basic building blocks.

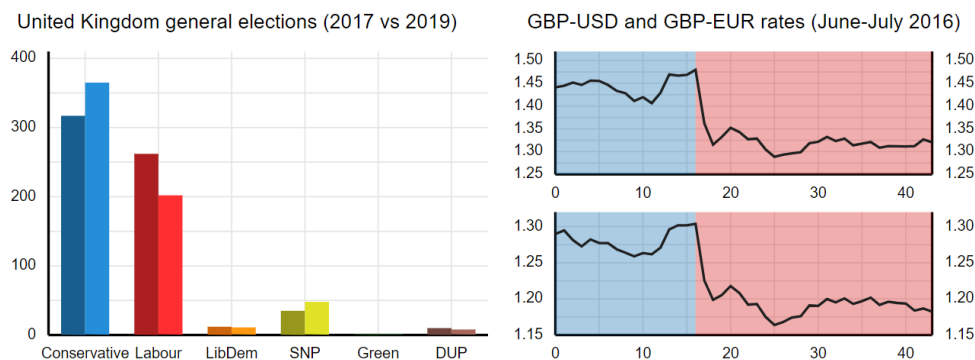


Fig. 1. Two charts about UK politics: Comparison of election results from 2017 and 2019 (left) and GBP/USD exchange rate with highlighted areas before and after the 23 June 2016 Brexit vote.

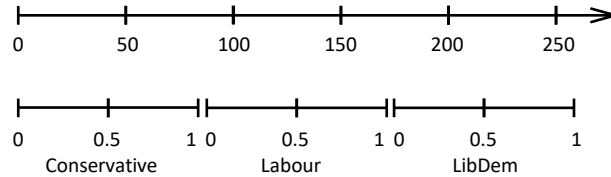


Fig. 2. On a continuous scale (above), an exact position is determined by a number. On a categorical scale (below), an exact position is determined by the category and a number between 0 and 1.

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

- Concepts such as bar charts, line charts or charts with aligned axes are all expressed in terms of more primitive building blocks using a small number of combinators.
- The primitives are specified in domain terms. When drawing a line, the value of a y coordinate is an exchange rate of 1.36 USD/GBP, not 67 pixels from the bottom.
- Most common chart types can be easily captured as high level abstractions, but there is a way of creating a majority of more interesting custom charts.
- The approach can easily be integrated with the Elm architecture (Czaplicki, 2012) 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. We outline the structure of an implementation, but omit the details; filling those in merely requires careful thinking about geometry and projections. An F# implementation, including all our examples, is available at: <http://github.com/compostjs>.

2 Basic charts: Overlaying chart primitives

We 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. In this section, we 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 axis shows categorical values representing the political parties such as **Conservative** or **Labour**. The y axis shows 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 (e.g. political parties, countries) or continuous (e.g. number of votes, exchange rates). The mapping from categorical and continuous values to positions on the chart is done automatically. We discuss this in Section 2.3.

v	=	<code>cat</code> c, r	s	=	<code>line</code> $\gamma, [v_{x1}, v_{y1}, \dots, v_{xn}, v_{yn}]$		<code>overlay</code> $[s_1, \dots, s_n]$	
						<code>fill</code> $\gamma, [v_{x1}, v_{y1}, \dots, v_{xn}, v_{yn}]$		<code>axis</code> _{$l/r/t/b$} s
						<code>text</code> γ, v_x, v_y, t		<code>padding</code> n_t, n_r, n_b, n_l, s
						<code>bubble</code> γ, v_x, v_y, w, h		

Fig. 3. Core primitives of the Compost domain-specific language. Values v are either categorical or continuous; a shape s is then defined as a simple recursive algebraic data type.

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.

Figure 2 illustrates the two kinds of values using the axes from the UK election results chart. In Figure 3, we define a value v as either a continuous value `cont` n containing any number n or a categorical value `cat` c, r consisting of a categorical value c and a number r between 0 and 1. As discussed in Section 2.4, continuous and categorical values can also be annotated with units of measure to make the values more descriptive.

2.2 Basic primitives and combinators

Now we can define the basic elements of the Compost domain-specific language. A chart element is represented by the shape s defined in Figure 3. A primitive shape can be a text label, a line connecting a list of points, a filled polygon defined by a list of points or a bubble at a given point with a given width and height. The position of points is specified by x and y coordinates, which can be either categorical or continuous values. For text, line, polygon and bubble, we also include a parameter γ that specifies the element color. The width and height of a bubble is given in pixels rather than in domain units.

Figure 3 also defines three combinators. The most important is `overlay`, which overlays given shapes. When doing this, Compost infers the range of values on the x and y axes and calculates suitable projections using a method discussed in the next section. The `padding` combinator adds padding around a specified shape and `axis` adds an axis showing the inferred scale on the left, right, top or bottom of a given shape. Using those primitives, we can construct the simple UK election results bar chart in Figure 4 (left):

```
let conservative, labour =
  fill #0000ff, [ (cat Conservative, 0), (cont 0), (cat Conservative, 0), (cont 365),
                (cat Conservative, 1), (cont 365), (cat Conservative, 1), (cont 0) ],
  fill #ff0000, [ (cat Labour, 0), (cont 0), (cat Labour, 0), (cont 202),
                (cat Labour, 1), (cont 202), (cat Labour, 1), (cont 0) ]
axisl (axisb (overlay [labour, conservative]))
```

We use the `let` construct of the host functional language to structure the code. The chart specification overlays two bars of different colors and then adds axes to the bottom and left of the chart. The two bars are filled rectangles defined using four corner points. The y coordinates are specified as continuous values, while the x coordinates are categorical.

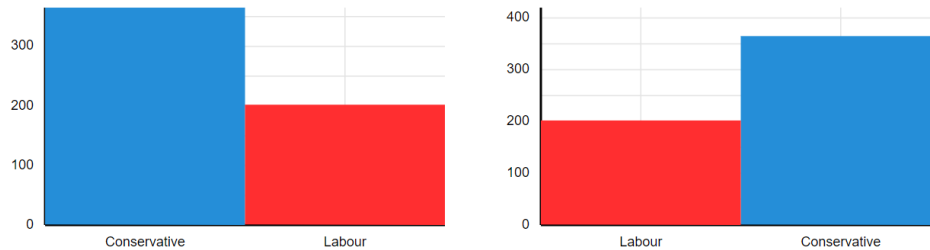
4 *Tomas Petricek*

Fig. 4. Simple chart showing the UK election results; using automatically inferred scales (left) and using rounded Y scale and explicitly defined (reordered) X scale (right).

For the Conservative party, two of the points have the y coordinate set to `cont 0` (bottom of the bar) and two have the y coordinate set to `cont 365` (top of the bar). The two x coordinates are the start and the end of the range allocated for the `Conservative` category, i.e. `cat Conservative, 0` on the left and `cat Conservative, 1` on the right.

Extending the snippet to generate a grouped bar chart that shows two results for each party as in Figure 1 is not much harder. Given a party p , we need to generate two rectangles, one with x coordinates `cat p, 0` and `cat p, 0.5` and the other with x coordinates `cat p, 0.5` and `cat p, 1`. In the following snippet, we use a `for` comprehension to generate the list. All remaining constructs are primitives of the Compost domain-specific language. Assuming elections is a list of election results containing a five-element tuple consisting of a party name, colors for 2017 and 2019 and results for 2017 and 2019 we create the chart using:

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

Aside from iterating over all available parties and splitting the bar, the example also adds padding around the bars. This is specified in pixels rather than in terms of domain values. A similar result could be achieved by drawing a bar using a range from `0.05` to `0.5`, but specifying padding precisely in pixels is sometimes preferable. The chart is still missing a title, which we add in Section 4.

2.3 Inferring scales and projections

We follow the terminology of Vega (Satyanarayan *et al.*, 2015) and use the term *scale* to refer to the mapping of values to positions on a screen; a *coordinate* is a value representing a position on a scale and the term *axis* is used to refer to the visual representation of a scale.

Scales are an important concept in Compost. When composing shapes using the `overlay` primitive, the user does not need to specify how to position the child elements relative to each other. The Compost library positions the elements automatically. This is done in two steps. During pre-processing, Compost infers the scales for x and y axes. A scale represents the range of values that needs to fit in the space available for the chart. When rendering a

$$l = \text{continuous } n_{min}, n_{max} \mid \text{categorical } [c_1, \dots, c_k]$$

Fig. 5. A scale l can be continuous, defined by a range, or categorical, defined by a list of values.

$$s = \begin{array}{l} \text{roundScale}_{x/y} s \quad \mid \quad \text{nest}_{x/y} v_{min}, v_{max}, s \\ \mid \quad \text{explicitScale}_{x/y} l, s \quad \mid \quad (\dots) \end{array}$$

Fig. 6. Additional combinators for controlling and nesting scales, extending earlier definition of s .

shape, Compost projects domain-specific values to the available screen space based on the inferred scale. A scale l is defined in Figure 5. A continuous scale is defined by a minimal and maximal value that need to be mapped to the available chart space. A categorical scale is defined by a list of individual categorical values. Note that we do not need a minimal and maximal ratios of the used categorical values as Compost will use an equal space for each category, regardless of where in this space a shape needs to appear.

Scale inference is done by a simple recursive function that walks over the given shape and constructs two scales for the x and y axis, using the x and y coordinates that appear in the shape. Most of the work is done by a simple helper function that takes two scales, l_1 and l_2 , and produces a new scale that represents the union of the two:

$$\begin{aligned} \text{union } (\text{continuous } n_l, n_h) (\text{continuous } n'_l, n'_h) &= \\ &\text{continuous } \min(n_l, n'_l), \max(n_h, n'_h) \\ \text{union } (\text{categorical } [c_1, \dots, c_p]) (\text{categorical } [c'_1, \dots, c'_q]) &= \\ &\text{categorical } [c_1, \dots, c_p] @ [c'_i \mid \forall i \in 1 \dots q, \nexists j. c_j = c'_i] \end{aligned}$$

When unioning two continuous scales, the minimum and maximum of the resulting scale is the smallest and largest of the two minimums and maximums, respectively. When unioning two categorical scales, we take all values of the first scale and append all values of the second scale that do not appear in the first one. Note that this means that the order of categorical values in a scale depends on the order in which they appear in the shape. (A possible improvement to Compost would be to support ordinal values, which are categorical values with a well-defined ordering.) It is also worth noting that a categorical scale cannot be combined with a continuous scale. In other words, mixing categorical and continuous values in a single scale results in an error.

The scales inferred during pre-processing are later used when rendering a shape. We discuss the implementation in Section 6. The key operation is projection which, given a coordinate, a scale and an area on the screen, produces a position on the screen. For a continuous scale, the projection is a linear transformation. For categorical scale with k values, we split the available chart space into k equally sized regions and then map a categorical value `cat` c, r to the region corresponding to c according to the ratio r .

2.4 Types and units of measure

We introduce the Compost domain-specific language as untyped, but there are some obvious ways in which types can make composing charts in Compost safer. First, a type representing a shape could specify whether the x and y axes represent categorical or continuous values. This would rule out mixing of different values on a single scale and guarantee that

the union operation, sketched in the previous section, is never called in a way leading to an undefined result. Second, the type of values mapped to an axis could be further annotated with units of measure (Kennedy, 2009). Using the F# notation where $n\langle u \rangle$ is a number n with unit u , an axis containing a value `cont 317⟨mp⟩` would then be incompatible with an axis containing a value `cont 1.32⟨gbp/usd⟩`.

We only outline the type system here. There are two kinds of types; σ is a type of values and τ is a type of shapes. Assuming u denotes a unit of measure, the types are defined as:

$$\sigma = \text{Cat } u \mid \text{Cont } u \qquad \tau = \text{Shape } \sigma_x, \sigma_y$$

Correspondingly, there are two kinds of typing judgements; $v \vdash \sigma$ indicates the type of a value, while $s \vdash \tau$ indicates the type of a shape. The typing rules for two of the basic chart primitives, `line` and `overlay` look as follows:

$$\frac{v_{xi} \vdash \sigma_x \quad v_{yi} \vdash \sigma_y}{\text{line } \gamma, [v_{x1}, v_{y1}, \dots, v_{xn}, v_{yn}] \vdash \text{Shape } \sigma_x, \sigma_y} \qquad \frac{s_i \vdash \text{Shape } \sigma_x, \sigma_y}{\text{overlay } [s_1, \dots, s_n] \vdash \text{Shape } \sigma_x, \sigma_y}$$

The rule for `line` ensures that all X and Y values have the same types, σ_x and σ_y , respectively and infers `Shape` σ_x, σ_y as the type of the shape. The rule for `overlay` ensures that all composed shapes have the same type, including the type of x and y scales.

3 Advanced charts: Controlling scale composition

Most charts have one x and one y scale that are determined by the values the chart shows, but there are interesting exceptions. The chart in Figure 1 (right) has two different y axes, one for GBP/USD and one for GBP/EUR. In the next two sections, we look at three combinators that control the scale inference process and what flexibility this enables.

3.1 Defining nice scale ranges

The automatic scale inference often results in scales where the maximum is a non-round number. This leads to charts that fully utilise the available space, but may not be easy to read. The first two primitives, shown in Figure 6 (left) allow the chart designer to adjust the automatically inferred range of scales. The operations can be applied to either the x scale or the y scale, which is indicated by the x/y subscript. The `roundScale` primitive 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 a “nice” number. For example, if a continuous scale has minimum `0` and maximum `365`, the resulting scale would have a maximum `400`. For categorical scale, the operation does not have any effect. The `explicitScale` operation 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, the chart in Figure 4 (right) is constructed using the following code (reusing the `labour` and `conservative` variables defined earlier):

```
axisl (axisb (roundScaley (explicitScalex (categorical [Labour, Conservative])),
  overlay [labour, conservative ] )))
```

Reading the code from the inside out, the snippet first overlays the two coloured bars defined earlier; it then replaces the X axis with an explicitly given one that changes the

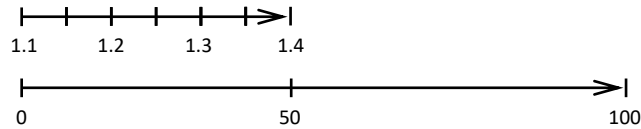


Fig. 7. A continuous scale with values from 0 to 6, nested in another scale.

order of the values. As a result, the bar for `Labour` will appear on the left, even though the value comes later in the list of overlaid chart elements.

The code next uses `roundScale` to automatically round the minimum and maximum of the continuous Y scale (showing the total number of seats). Finally, we add axes around the shape, producing a usual labelled chart. It is worth noting that `axis` and `roundScale` could be implemented as derived operations; `roundScale` would need to infer the scale of the nested shape and then insert `explicitScale` with a rounded number; `axis` would also need to infer the scales and then generates labels and lines in suitable locations.

3.2 Nested scales

The most interesting primitive for controlling scale composition is `nestx/y`. The behaviour is similar to that of the SVG viewport (Dahlström *et al.*, 2011). It takes two values, v_{min} , v_{max} and a shape s as arguments and nests the scale of the shape s inside the region defined by v_{min} , v_{max} . When inferring scales of shapes, the scale of `nestx/y l, s` will be a categorical or continuous scale constructed from the values v_{min} and v_{max} , regardless of the values that are used inside the shape s . The chart space between v_{max} and v_{min} will then be used to render the nested shape s using its inferred scale. An example of nesting is shown in Figure 7. Here, a chart with a continuous scale from 1.1 to 1.4 (e.g. GBP/EUR exchange rates) is nested in the left half of another chart, which has a continuous scale from 0 to 100.

The nesting of scales can be used in a variety of ways. For example, to nest a scatter plot showing individual data points inside a bar of a histogram, we would use `cat ABC, 0` and `cat ABC, 1` as the points that define the start and the end of the region. A simpler use case for the combinator is showing multiple charts in a single view. For example, the motivating example in Figure 1 (right) compares aligned line charts of exchange rates for two different currencies. Assuming `gbpusd` and `gbpeur` are lists containing days as x values and exchange rates as y values, we can construct a simple chart with two line charts, shown in Figure 8 (left), using:

```
overlay [ nesty (cont 0), (cont 50), (axisl (axisr (axisb (line #202020 gbpusd))))
          nesty (cont 50), (cont 100), (axisl (axisr (axisb (line #202020 gbpeur)))) ]
```

In this example, the x scale shows the days of the year. This scale is shared by both of the charts. Indeed, if data was only available for the second half of the month for one of the charts, we would want the line to start in the middle of the chart. However, the y scale needs to be separate for each of the charts. To achieve this, we use `nesty`. The scale of the inner shapes is continuous, from the minimal to the maximal exchange rate for a given period. The outer scale is determined by the explicitly defined points. For the upper chart, these

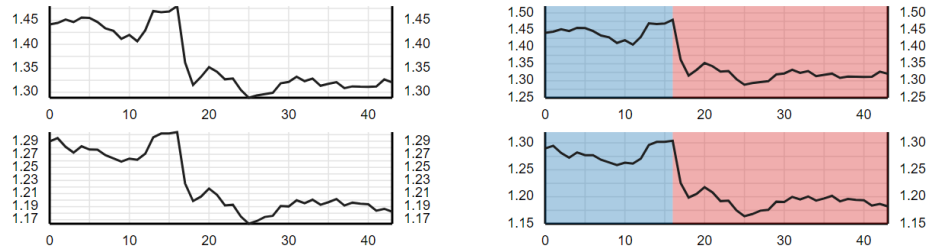


Fig. 8. Two charts showing currency exchange rates with a shared X scale and separate Y scales.

are `cont 0` and `cont 50`; for the lower chart, these are `cont 50` and `cont 100`. The continuous values define a scale that only contain two shapes – one in the upper half, one in the lower half – and so the three numbers could have equally been, for example, `0, 1, 2`. The outer scale used here is synthetic and it is not aligned with other chart elements. A chart that does not have synthetic outer scale is pairplot, discussed in the next section.

For completeness, the following code snippet shows how to construct the full currency exchange rate chart shown in Figure 8 (right), including the blue and red background:

```
let xrate (lo, hi) rates = overlay [
  fill #1F77B460, [ cont 0, cont lo, cont 16, cont lo, cont 16, cont hi, cont 0, cont hi ],
  fill #D6272860, [ cont 16, cont lo, cont 44, cont lo, cont 44, cont hi, cont 16, cont hi ],
  line #202020 rates ]

overlay [ nest_y (cont 0), (cont 50), (axis_l (axis_r (axis_b (xrate (1.25, 1.50) gbpusd))))
  nest_y (cont 50), (cont 100), (axis_l (axis_r (axis_b (xrate (1.15, 1.30) gbpeur)))) ]
```

Here, we use the `let` binding of the host language to define a function that takes the data rates together with the minimum and maximum. This is used for drawing two filled rectangles, covering the first 16 days of the view in blue and the rest in red. The shapes combined using `overlay` are rendered in the order in which they appear and so the line shape is last, so that it appears above the background.

4 Standard charts: Defining new abstractions

The functional domain-specific language design makes it easy to define high-level chart features and chart types, known from standard charting libraries, using the low-level primitives of the core language. To illustrate this, we give two examples.

First, one last remaining feature of the two charts in Figure 1 is a chart title. This can be added to any chart using the following derived combinator:

```
let title t s = overlay [
  nest_x (cont 0), (cont 100), (nest_y (cont 0), (cont 15),
  explicitScale_x (continuous 0, 100), (explicitScale_y (continuous 0, 100),
  text #000000, (cont 50), (cont 50), t )
  nest_x (cont 0), (cont 100), (nest_y (cont 15), (cont 100), s) ]
```

We use `let` in the host language to define `title` as a function taking a title `t` and a shape `s`. It overlays two shapes. To position the title above the chart, the first shape has an outer `y`

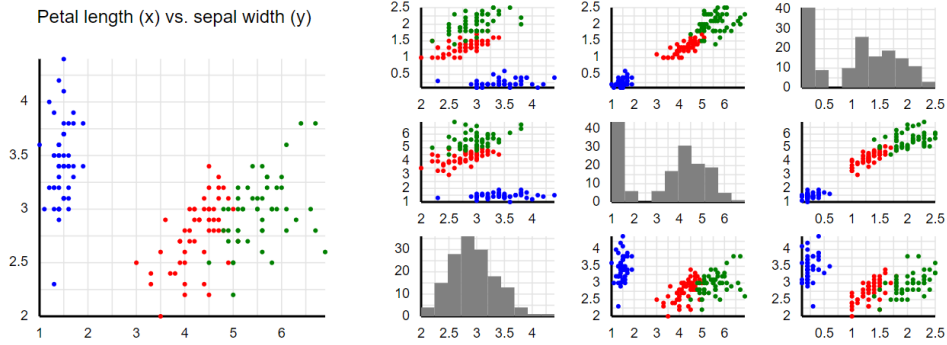


Fig. 9. Sample charts built using derived abstractions; a scatter plot visualizing the Iris dataset with a title (left) and a pairplot comparing two Iris features (right).

scale `continuous 0, 15` while the second has an outer `y` scale `continuous 15, 100`. Similarly, the outer `x` scale of both is `continuous 0, 100`. These are defined using `nestx/y`.

The second shape simply wraps the specified chart `s` to which we are attaching the title. The first positions the text title in the middle of the available space. To do so, we explicitly set the `x` and `y` scales inside the upper shape to continuous scales from `0` to `100` and then position the text label in the middle, at a point `(cont 50), (cont 50)`. Figure 9 (left) shows a sample scatter plot chart with a title created using the title combinator.

A more complex chart that can be composed using the Compost primitives is pairplot from the seaborn library (Waskom *et al.*, 2014). Pairplot visualizes pairwise relationships between features of a dataset. An example using three features (sepal width, petal width, petal length) from the Iris dataset is shown in Figure 9 (right). A pairplot draws a grid of charts, each visualizing the relationship between two numerical features. For distinct features, pairplot shows a scatter plot using one feature for `x` values and the other for `y` values. When the features are the same (the diagonal), it draws a histogram of the feature values. A categorical feature can be used to determine the color of dots in the scatter plots.

To generate a pairplot, we use `nest` to overlay and align a grid of plots. Each of those overlays a number of bubbles or filled shapes and adds left and bottom axis. As before, we use `let` to define a function and list comprehensions to generate individual chart elements. We assume that data is a list of rows, `attrs` is a list of available attributes and `get a r` obtains the attribute `a` of a row `r`. We also assume the dataset contains the `"color"` attribute.

```
let pairplot attrs data = overlay [
  for x in attrs → for y in attrs →
    nestx (cat x, 0), (cat x, 1), (nesty (cat y, 0), (cat y, 1), axisl (axisb
      ( if x ≠ y then overlay [ for r in data →
        bubble (get "color" r), (get x r), (get y r), 1, 1 ]
      else overlay [ for x1, x2, y in bins x data →
        fill #808080 [ x1, y, x2, y, x2, 0, x1, 0 ] ])))]
```

As before, `nest` is essential for composing individual charts. Here, the points that determine the locations of individual charts are categorical values defined by the attributes of the dataset. The choice between two possible nested charts is made using the host language `if` construct. Scatter plots are generated by overlaying bubbles with `x` and `y` coordinates

$$s = \begin{array}{l} \text{mouseUp } (\lambda x y \rightarrow e), s \quad | \quad \text{mouseDown } (\lambda x y \rightarrow e), s \\ | \quad \text{mouseMove } (\lambda x y \rightarrow e), s \quad | \quad (\dots) \end{array}$$

Fig. 10. Additional combinators for mouse-based interaction, extending earlier definition of s .

obtained using `get x r` and `get y r`. Histograms are composed from filled shapes. To obtain their locations, we use a helper function `bins x data`, which returns a list of bins specified by a triple consisting of a lower and an upper range x_1, x_2 and the count y .

The example shows that the Compost is simple yet expressive. With just a few lines of code, we are able to construct charts that, in other systems, require dedicated libraries. The essential aspect of the language making this possible is the automatic inference of scales and their mapping to the available space as well as the `nest` operation.

5 Interactive charts: Domain-specific event handling

Many data visualizations published on the web feature interactivity. Standard forms of interactivity include animations, hover labels or zooming. More interesting custom visualizations include “You Draw It” introduced by the New York Times (Aisch *et al.*, 2015). The chart shows only the first half of the data, such as a timeline, and the reader has to guess the second half before clicking a button and seeing the actual data. Standard forms of interactivity are often supported by high-level libraries; Google Charts supports panning using drag & drop, zooming to a selected chart range and animations. Custom interactivity is typically implemented using low-level libraries such as D3, but doing so requires directly handling JavaScript events and modifying the browser DOM.

5.1 Domain-specific events

Compost can be used in conjunction with a virtual-dom library to create interactive data visualizations based on the Elm architecture (Czaplicki, 2016). An interactive visualization is described using a pair of functions – the `view` function creates a chart based on the current state and the `update` function specifies how the state changes when an event occurs.

To support handling of mouse-based events, Compost adds three additional primitives to the definition of `shape`, shown in Figure 10. The three new primitives make it possible to handle three common mouse events using custom functions $\lambda x y \rightarrow e$, specified in the host language. The most interesting aspect is that the functions are given x and y coordinates of the event specified in the domain units of the chart. This means that if the user clicks on the bar representing the Conservative party in a bar chart, the values might be, for example, `cat Conservative, 0.75` for x and `cont 120.5` for y .

5.2 You Draw It data visualizations

To illustrate building of interactive data visualizations using Compost, we look at one aspect of “You Draw It”. We want to create a bar chart where the user can use drag & drop to move individual bars. Figure 11 shows the interactive chart before and after an interaction. The first step is to define types representing the state and events that can occur:

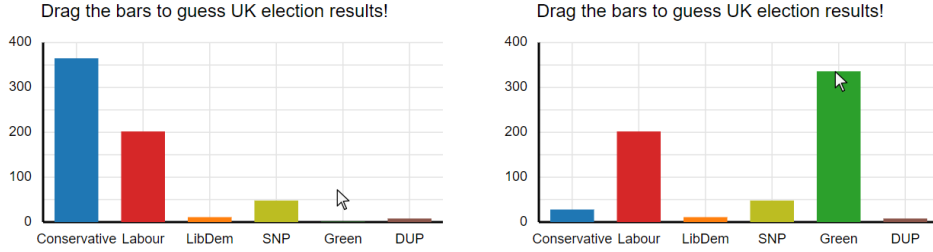


Fig. 11. Interactive “You Draw it” data visualization. The user moves cursor to a bar (left), pushes a mouse button and drags the bar to the position that they think is the correct one (right).

```
type State = bool * (string * int) list
type Event = Update of (string * int) | Moving of bool
```

The state is a pair of boolean, indicating whether the user is currently dragging, and a list of key value pairs, storing the number of seats for each political party. Two types of events can occur in the visualization. First, the user may start or stop dragging, which is indicated using `Moving(true)` and `Moving(false)`, respectively. Second, the user may change a value for a party, which is represented by the `Update` event.

The next part of the implementation is the update function which takes an old state together with an event and produces a new state:

```
let update (_,s) (Moving(m)) = m,s
    update (true,s) (Update(p,v)) = true, map (λ(k,o) → k, if k = p then v else o) s
    update (m,s) (Update(-,-)) = m,s
```

The first case handles the `Moving` event, which replaces the first component of the state tuple, i.e. a flag indicating whether a mouse button is down. The next two cases handle the `Update` event. The event carries two values, p, v , which represent the party (which bar the user is dragging) and the new value (new number of seats). If the user is currently dragging, we replace the value associated with the party p in the list s using the `map` function. If the user is not currently dragging, the event is ignored.

Finally, the view function takes the current state and builds the data visualization using the Compost domain-specific language. In addition, it also takes a parameter `trigger`, which is a function of type `Event → unit` that can be used to trigger events in handlers, registered using primitives such as `mouseMove`. To build the bar chart in Figure 11, we use the same approach as in Section 2.2. The only addition are the event handlers registered using `mouseMove`, `mouseUp` and `mouseDown`:

```
let view trigger (_,state) =
  axist (axisb (explicitScaley (continuous 0,400),
    (mouseMove (λ (cat p,-) (cont v) → trigger(Update(p,v))),
    (mouseUp (λ _ _ → trigger(Moving(true))),
    (mouseDown (λ _ _ → trigger(Moving(false))), overlay [
      for party,mps in state → padding 0,10,0,10, (fill (color party),
        [(cat party,0),(cont 0),(cat party,0),(cont mp),
        (cat party,0.5),(cont mps),(cat party,0.5),(cont 0)] ]))))))
```

When the user interacts with the visualization created using Compost, the library translates the coordinates associated with events from pixels to domain-specific values. In case of the above bar chart, when the user moves a mouse, the function registered using `mousemove` is given a categorical value `cat` p, r as the x coordinate and a continuous value `cont` v as the y coordinate. It then takes p , which is the name of the party corresponding to the bar and the value v corresponding to the number of seats and triggers the `Update(p, v)` event to update the state. The handlers for `mouseup` and `mousedown` do not use the coordinates. They simply switch the flag indicating whether the user is currently dragging or not.

The pair of functions, `update` and `view`, together with an initial state is all that is needed to create an interactive data visualization. Compost calls `view` each time the state changes and uses virtual-dom to update the chart displayed in the browser. Although creating an interactive visualization is more work than creating a static one, the domain-specific nature of Compost is invaluable. We can simply take the values p and v produced by a mouse event, use those to update the state and then, again, render an updated chart.

6 Implementation structure: Scale inference and rendering

Compost is an open-source library, implemented in the functional language F#. The full source code can be found at <http://github.com/compostjs>. As is often the case with functional domain-specific languages, the implementation is not difficult once we find the right collection of basic primitives and the right structure for the implementation. This largely applies to the Compost library and so we will not go into the implementation details. It is, however, worth giving an outline of the implementation structure.

As mentioned in Section 2.3, the rendering of shapes proceeds in two stages. First, the library infers the scales of a shape. When doing so, it also annotates some shapes with additional information that is needed later for rendering. Second, the library projects the shape onto an available space and produces the chart, represented as an SVG object.

6.1 Inferring the scales of a shape

In order to render a shape, we need to know the range of values that should appear on the x and y axes. This is done by inferring a scale for each of the axes from the individual x and y coordinates that specify shape locations. As discussed earlier, a scale can be either categorical (displaying only categorical values) or continuous (displaying only continuous values). When inferring scales, we use two helper operations; `union`, discussed earlier, combines two scales and `singleton` creates a scale from a single coordinate.

The operation that infers the scales of a shape is `calculateScales`. It takes a shape and produces a pair of x and y scales, together with a transformed shape:

$$\text{calculateScales} : \text{Shape} \rightarrow (\text{Scale} * \text{Scale}) * \text{Shape}$$

The operation does not need to transform the shape in most cases. The exception is the shape `nestx/y vmin, vmax, s`. In this case, the returned scale is based solely on the values of v_{min} and v_{max} . For rendering we need to keep the inferred scales of the nested shape s . To do so, the operation replaces the `nestx/y` shape with an auxiliary shape `scaledNestx/y`:

$$s = \text{scaledNest}_{x/y} \ v_{min}, v_{max}, s_{x/y}, s \mid (\dots)$$

There are two kinds of cases handled by `calculateScales`. For primitives, it constructs a pair of scales from individual coordinates using union and singleton. For shapes containing a sub-shape, the operation calculates the scales of a sub-shape recursively and then adapts those somehow. To illustrate, we consider two interesting cases:

```
calculateScales (nestx vmin, vmax, s) =
  let (sx, sy), s' = calculateScales s
      (union (singleton vmin) (singleton vmax), sy), scaledNestx vmin, vmax, sx, s'
```

```
calculateScales (overlay l) =
  let scales, l' = unzip (map calculateScales l)
      let sx, sy = unzip scales
          (reduce union sx, reduce union sy), overlay l'
```

When calculating the scales of the `nestx`, the function first calculate scales of the sub-shape s recursively. The resulting y scale s_y is returned as the result, while the x scale is obtained from the two coordinates v_{min} and v_{max} . This is also the case where the shape is transformed and the returned `scaledNestx` shape stores the inferred x scale s_x of the sub-shape s . The second example is the `overlay` case which recursively computes scales of all sub-shapes and combines those using the list folding function `reduce` with union as an argument.

6.2 Projecting coordinates and drawing

The key operation that needs to be performed when drawing a shape is projecting coordinates from domain-specific values to the screen coordinates. As we draw a shape, we keep the x and y scale and the space in pixels that it should be drawn on. Initially, the x and y scales are those inferred for the entire shape and the space in pixels is $0 \dots \text{width}$ and $0 \dots \text{height}$ where $\text{width} \times \text{height}$ is the size of the target SVG element.

The key calculation is done by the `project` function, which takes the space in pixels (as a pair of floating-point numbers representing the range), the current scale and a domain-specific value and produces a coordinate in pixels:

$$\text{project} : \text{float} * \text{float} \rightarrow \text{Scale} \rightarrow \text{Value} \rightarrow \text{float}$$

The function is only defined if the value and scale are compatible. As discussed in Section 2.4, this could be guaranteed using a simple type system. If both are continuous, the function performs a simple linear transformation. If both are categorical, the available pixel space is divided into an equally-sized bins, one for each categorical value on the scale, and the value is then projected into the appropriate bin.

The drawing of shapes is done by a function that takes the available area as a quadruple $(x_1, y_1), (x_2, y_2)$ together with the x and y scale mapped onto the area and a shape to be drawn. The result is a data structure representing an SVG document:

$$\text{drawShape} : (\text{float} * \text{float}) * (\text{float} * \text{float}) \rightarrow \text{Scale} * \text{Scale} \rightarrow \text{Shape} \rightarrow \text{Svg}$$

For primitive shapes, the operation projects the coordinates using `project` and constructs a corresponding SVG document. For shapes with sub-shapes it calls itself recursively, possibly with an adjusted scale or area. The two cases discussed earlier illustrate this:

```
drawShape a s (overlay l) =
  concat (map (drawShape a s) l)

drawShape ((x1,y1),(x2,y2)) (sx,sy) (scaledNest_x v_min, v_max, ns_x, shape) =
  let x'_1 = project (x1,x2) sx v_min
      x'_2 = project (x1,x2) sx v_max
      drawShape ((x'_1,y1),(x'_2,y2)) (ns_x,sy) shape
```

When drawing `overlay`, the function draws all sub-shapes onto the same area using the same scales and then concatenates the returned SVG components using the `concat` helper. The `scaledNest_x` case is more illuminating. Here, we first use `project` to find the range x'_1, x'_2 corresponding to the domain-values v_{min}, v_{max} . This defines the area corresponding to the nested scale ns_x , onto which the x coordinates in the sub-shape `shape` should be projected. To do this, we recursively call `drawShape`, but use x'_1 and x'_2 as the x coordinates of the target area and ns_x as the x scale. The y area and scales are propagated unchanged.

7 Conclusions

This paper presents a functional take on the problem of designing easy to use, but flexible abstractions for composing data visualizations. We hope to find a sweet between high-level, but inflexible approaches and low-level, but hard to use approaches.

Most work in this space is based on Grammar of Graphics (Wilkinson, 1999), designing more or less complex and powerful variants (Wickham, 2010; Satyanarayan *et al.*, 2016; Satyanarayan *et al.*, 2015; Stolte *et al.*, 2002). In Grammar of Graphics, a chart is a mapping from data to a chart elements and their visual attributes. In contrast, in Compost, the mapping is specified in the host programming language and a chart is merely a resulting data type describing the visual elements using domain-specific primitives.

Our approach is very flexible as it lets the user compose primitive visual elements in any way they want; it lets them define their own high-level abstractions and it also integrates well with reactive programming architectures to support interactive data visualizations.

In this paper, we focus on presenting the core ideas behind Compost. However, much remains to be explored, both in terms of finding the best set of primitives and in terms of their language integration. First, we only support categorical and continuous values, but there are also ordinal values (which cannot be compared, but can be sorted). Second, some of our primitives, namely `axis` and `roundScale` could be implemented as derived operations, but we treat those as built-in for simplicity. Third, we only treat x and y as scales, but we could similarly treat other visual features (colors of bars, size of bubbles) as scales, which would allow a more high-level specification of certain charts.

Even in the basic form, the Compost library makes it possible to create a wide range of standard, atypical and even interactive charts. Moreover, we hope that other functional programmers agree that it does so in a simple, logical and easy-to-understand way. In other words, we hope the presented functional library is, indeed, a functional pearl.

References

- Aisch, Gregor, Cox, Amanda, & Quealy, Kevin. 2015 (May). *You draw it: How family income predicts children's college chances*. New York Times. Retrieved May 24, 2020 from <https://www.nytimes.com/interactive/2015/05/28/upshot/you-draw-it-how-family-income-affects-childrens-college-chances.html>.
- Bostock, Michael, Ogievetsky, Vadim, & Heer, Jeffrey. (2011). D³ data-driven documents. *IEEE Transactions on visualization and computer graphics*, **17**(12), 2301–2309.
- Czaplicki, Evan. (2012). *Elm: Concurrent FRP for functional GUIs*. Senior thesis, Harvard University. Available at <https://elm-lang.org/assets/papers/concurrent-frp.pdf>.
- Czaplicki, Evan. 2016 (May). *A farewell to FRP: Making signals unnecessary with The Elm Architecture*. Retrieved May 24, 2020 from <https://elm-lang.org/news/farewell-to-frp>.
- Dahlström, Erik, Dengler, Patrick, Grasso, Anthony, Lilley, Chris, McCormack, Cameron, Schepers, Doug, & Watt, Jonathan. 2011 (Aug). *Scalable vector graphics (svg) 1.1 (second edition)*. W3C Recommendation. Retrieved May 24, 2020 from <http://www.w3.org/TR/2011/REC-SVG11-20110816/>.
- Google. 2020 (May). *Google charts: Interactive charts for browsers and mobile devices*. Google. Retrieved May 24, 2020 from <https://developers.google.com/chart>.
- Kennedy, Andrew. (2009). Types for units-of-measure: Theory and practice. *Pages 268–305 of: Central european functional programming school*. Springer.
- Satyanarayan, Arvind, Russell, Ryan, Hoffswell, Jane, & Heer, Jeffrey. (2015). Reactive Vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE transactions on visualization and computer graphics*, **22**(1), 659–668.
- Satyanarayan, Arvind, Moritz, Dominik, Wongsuphasawat, Kanit, & Heer, Jeffrey. (2016). Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics*, **23**(1), 341–350.
- Stolte, Chris, Tang, Diane, & Hanrahan, Pat. (2002). Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE transactions on visualization and computer graphics*, **8**(1), 52–65.
- Waskom, Michael, Botvinnik, Olga, Hobson, P, Warmenhoven, J, Cole, JB, Halchenko, Y, Vanderplas, J, Hoyer, S, Villalba, S, & Quintero, E. (2014). *Seaborn: statistical data visualization*. Retrieved May 24, 2020 from <https://seaborn.pydata.org/>.
- Wickham, Hadley. (2010). A layered grammar of graphics. *Journal of computational and graphical statistics*, **19**(1), 3–28.
- Wickham, Hadley. (2016). *ggplot2: Elegant graphics for data analysis*. Springer.
- Wilkinson, Leland. (1999). *The grammar of graphics*. Springer-Verlag New York.

