Under consideration for publication in J. Functional Programming

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.

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. Google Charts (Google, 2020) 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. 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, 2012). 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; 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.

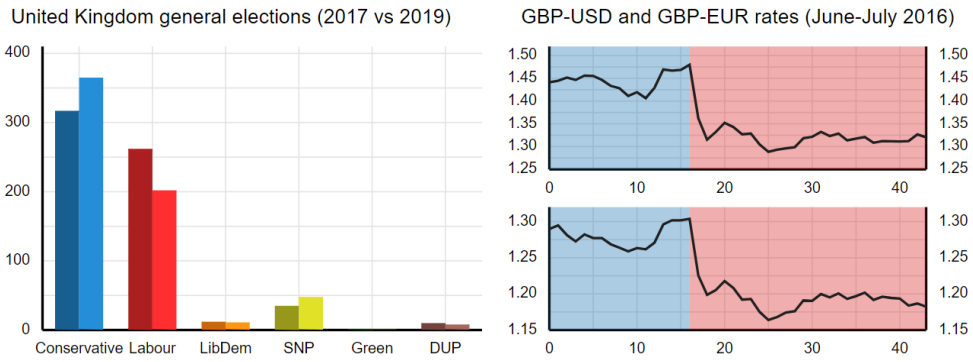

Fig. 1. Two charts about the UK politics: Comparison of election results from 2017 and 2019 (left) and GBP/USD exchange rate with highlighted areas before and after the 23 June 2016 Brexit vote.

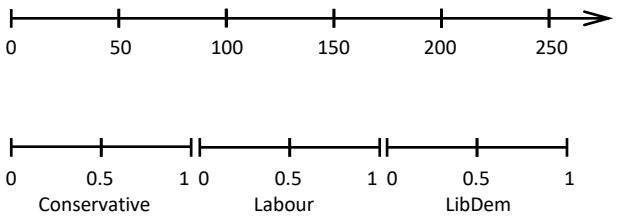

Fig. 2. On a continuous scale (above), an exact position is determined by a number. On a categorical scale (below), an exact position is determined by the category and a numerical ratio from 0 to 1.

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 an answer, 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:

- 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 all 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 67 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 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 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 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.

$$
\begin{array}{rcl}
v & = & \mathsf{cat}\ c, r \\
& | & \mathsf{cont}\ n
\end{array}
\qquad
\begin{array}{rcll}
s & = & \mathsf{line}\ \gamma, [v_{x1}, v_{y1}, \ldots, v_{xn}, v_{yn}] & | \ \mathsf{overlay}\ [s_1, \ldots, s_n] \\
& | & \mathsf{fill}\ \gamma, [v_{x1}, v_{y1}, \ldots, v_{xn}, v_{yn}] & | \ \mathsf{axis}_{l/r/t/b}\ s \\
& | & \mathsf{text}\ \gamma, v_x, v_y, t & | \ \mathsf{padding}\ n_t, n_r, n_b, n_l, s \\
& | & \mathsf{bubble}\ \gamma, v_x, v_y, w, h &
\end{array}
$$

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.

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. For continuous values, this involves 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.

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 (implemented as a string) and a ratio r between 0 and 1.

2.2 Basic primitives and combinators

Now that we know how Compost represents values, we can define the basic elements of its domain-specific langauge. A chart is represeted 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.

Figure 3 also defines three combinators. The most important is overlay, which overlays all shapes from a given list. When doing this, Compost automatically infers the scales of X and Y axes and calculates suitable projections using a method discussed in the next section. Finally, padding 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). We use the let construct of the host functional language to structure the code:

```
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) ]

axis_l (axis_b (overlay [ labour, conservative ]))
```

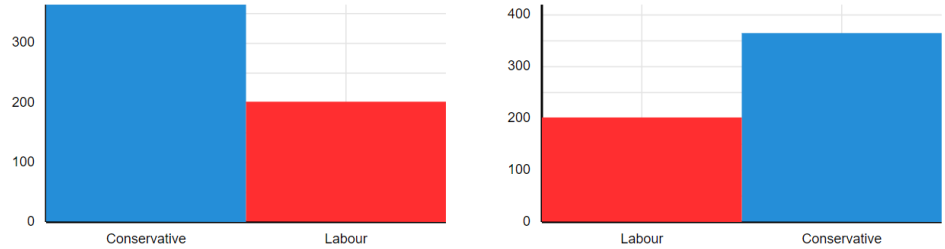

Fig. 4. Simple chart showing the UK election results; using automatically inferred scales (left) and using rounded Y scale and explicitly defined (reordered) X scale (right).

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. 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 easy. 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. A padding is specified in pixels rather than in terms of domain values. This is sometimes preferrable over, for example, drawing a bar using a range from 0.05 to 0.5. The chart is still missing a title, which we add in Section 4.

2.3 Inferring scales and projections

When composing shapes using the overlay primitive, the user does not need to specify how to position the child elements relatively to each other. The Compost library positions the elements automatically. This is done in two steps. First, Compost infers the *scales* for X and Y axes. A scale represents the range of values that needs to fit in the space available

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

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

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

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

for the chart. Second, Compost calculates a *projection*, a mapping from domain-specific values of the scale to the available screen space. 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.

Inferring scales 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:

$$\begin{array}{l} \mathsf{union}\ (\mathsf{continuous}\ n_l, n_h)\ (\mathsf{continuous}\ n'_l, n'_h) = \\ \quad \mathsf{continuous}\ \min(n_l, n'_l), \max(n_h, n'_h) \end{array}$$

$$\begin{array}{l} \mathsf{union}\ (\mathsf{categorical}\ [c_1, \ldots, c_p])\ (\mathsf{categorical}\ [c'_1, \ldots, c'_q]) = \\ \quad \mathsf{categorical}\ [c_1, \ldots, c_p]\ @\ [c'_i \mid \forall i \in 1 \ldots q, \nexists j . c_j = c'_i\,] \end{array}$$

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.

Once Compost computes scales for a given shape and its sub-shapes, it constructs a projection that maps domain-specific values to the available chart space. We discuss this in Section 6. 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 $\mathsf{cat}\ c, r$ 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 could make composing charts in Compost safer. First, a type representing a shape could specify whether the X and Y axes represent categorical or continous 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 $317\langle \mathsf{mp}\rangle$ would then be incompatible with an axis containing a value cont $1.32\langle \mathsf{gbp}/\mathsf{usd}\rangle$.

The design of the type system for Compost is straightforward, so we only present a brief sketch. 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:

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

Correspondingly, there are two kinds of 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:

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

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 axis 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 inferrence 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 utilize 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 two combinators for controlling the range of scales are roundScale and explicitScale. The oprations can be applied to either the X scale or the Y scale, which is indicated by the x/y sub-script. The roundScale primitve 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 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, the chart in Figure 4 (right) is constructed using the following code (reusing the labour and conservative variables defined earlier):

```
axis_l (axis_b (roundScale_y (explicitScale_x (categorical [Labour, Conservative]),
    overlay [ labour, conservative ] )))
```

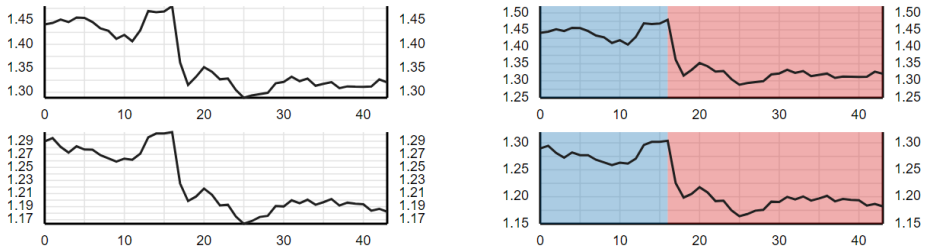

Fig. 7. Two charts showing currency exchange rates with a shared X scale and separate Y scales.

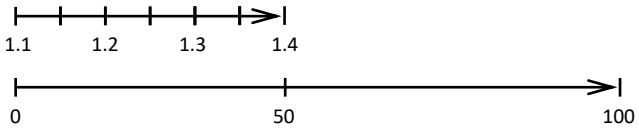

Fig. 8. A continuous scale with values from 0 to 6, nested in another scale.

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 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 defined in Figure 6 is $\mathsf{nest}_{x/y}$. The combinator takes two values, v_{min}, v_{max} and a shape s as arguments and it nests the scale of the shape s inside the region defined by v_{min}, v_{max}. When inferring scales of shapes, the scale of $\mathsf{nest}_{x/y}\ l, s$ will be a categorical or continuous scale inferred using 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 8. Here, a chart with a continous 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. We can, for example, nest a line chart inside a bar of a bar chart. In that case, the values for v_{min} and v_{max} would be cat ABC, 0 and cat ABC, 1, which define the start and the end of the region allocated to the ABC category on a categorical scale. A simpler use case for the combinator is showing multiple charts in a single view. For example, the motivating example in Figure 1 (right)

comapres 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 7 (left), using:

```
overlay [ nest_y (cont 0),(cont 50),(axis_l (axis_r (axis_b (line #202020 gbpusd))))
          nest_y (cont 50),(cont 100),(axis_l (axis_r (axis_b (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 nest_y. 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 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. An example of a more complex chart that follows a similar style, but does not have synthetic outer scale would be pairplot from the seaborn Python library (Waskom *et al.*, 2014).

For completeness, the following code snippet shows how to construct the full currency exchange rate chart shown in Figure 7 (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 a 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: Definining new abstractions

The Compost library does not introduce a rich collection of standard charts and chart features as most charting libraries. However, the functional domain-specific language design makes it easy to define high-level chart features based on the low-level primitives of the core language. To illustrate this, we give two examples.

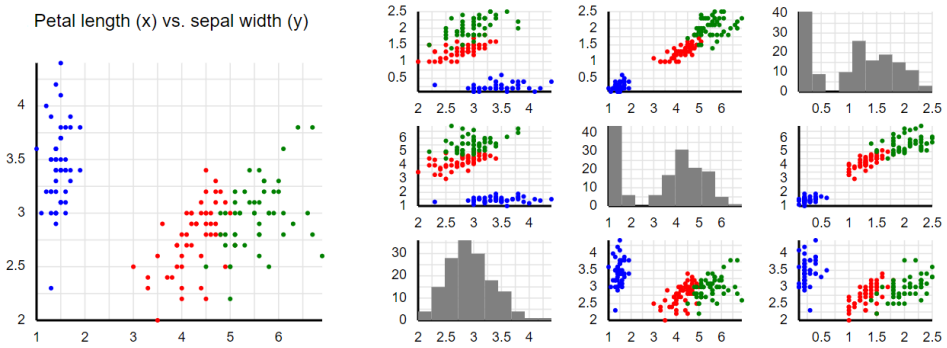

Fig. 9. Sample charts built using derived abstractions; a scatter plot visualizing the Iris dataset with a title (left) and a pairplot comparing two Iris features (right).

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) ]
```

The title combinator is a function defined using let in the host language. It takes 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 scale continuous $0, 15$ while the second has an outer Y scale continuous $15, 100$. These are defined using the nest_y primitive. Similarly, the outer X scale of both is continuous $0, 100$, defined using nest_x.

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 type of chart that can be easily composed using the Compost primitives is the pairplot chart 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 fature for X values and the other for Y values. At the diagonal (when the fatures are the same), it draws a histogram of the values of the feature. Additionally, 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 →
    nest_x (cat x,0),(cat x,1),(nest_y (cat y,0),(cat y,1),axis_l (axis_b
      ( 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 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 tripple 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

Data visualizations published on the web often also feature interactivity. Standard forms of interactivity include animations, hover labels or zooming. More interesting custom forms include the "You draw It" visualization 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 supported by high-level libraries. For example, Google Charts supports panning using drag & drop, zooming to a selected chart range and animations. Custom interactivity can be 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 $\lambda\, x\, y \to e$, 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 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.

$$\begin{array}{rcl} s & = & \mathsf{mouseUp}\ (\lambda x\ y \to e), s \quad \mid \quad \mathsf{mouseDown}\ (\lambda x\ y \to e), s \\ & \mid & \mathsf{mouseMove}\ (\lambda x\ y \to e), s \quad \mid \quad (\ldots) \end{array}$$

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

5.2 You draw it data visualizations

To illustrate building of interactive data visualizations using Compost, we look at one aspect of the "You draw It" visualizations. We want to create a bar chart where the user can use drag & drop to move individual bars. The resulting chart can be seen in Figure 11. The first step is to define types representing the state and events that can occur:

```
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, together with 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 captured by the Update event.

The next part of the visualization 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 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

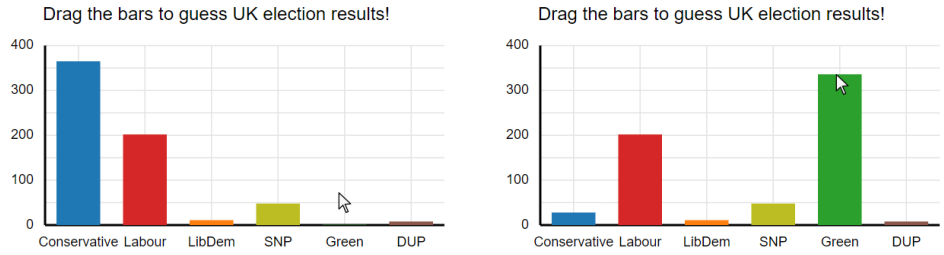

Fig. 11. Interactive "You draw it" data visualization. The user moves cursor to a bar (left), pushes a mouse button and drags the bar to the position that they think is the correct one (right).

the list s using the map function. If the user is not currently dragging, the event is ingored. The last case handles the Moving event, which simply replaces the first component of the state tuple, i.e. a flag indicating whether a mouse button is down.

Finally, the render 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 $\rightarrow$ 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 render trigger (_,state) =
  axis_l (axis_b (explicitScale_y (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 registed 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 render, together with an initial state is all that is needed to create an interactive data visualization. Compost calls render 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 projection

Compost is an open-source library, implmented 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 easy 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 implmentation 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 are 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 contiuous (displaying only continuous values). When inerring 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:

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

The operation does not need to transform the shape in most cases. The exception is the shape $\mathsf{nest}_{x/y}\ v_{min}, v_{max}, s$. 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 $\mathsf{nest}_{x/y}$ shape with an auxiliary shape $\mathsf{scaledNest}_{x/y}$:

$$s \quad = \quad \mathsf{scaledNest}_{x/y}\ v_{min},\ v_{max},\ s_{x/y},\ s \quad | \quad (\ldots)$$

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:

$$\begin{array}{l}
\mathsf{calculateScales}\ (\mathsf{nest}_x\ v_{min},\ v_{max},\ s) = \\
\quad \mathsf{let}\ (s_x, s_y), s' = \mathsf{calculateScales}\ s \\
\quad (\mathsf{union}\ (\mathsf{singleton}\ v_{min})\ (\mathsf{singleton}\ v_{max}), s_y),\ \mathsf{scaledNest}_x\ v_{min},\ v_{max},\ s_x,\ s'
\end{array}$$

$$\begin{array}{l}
\mathsf{calculateScales}\ (\mathsf{overlay}\ l) = \\
\quad \mathsf{let}\ \mathsf{scales},\ l' \ = \ \mathsf{unzip}\ (\mathsf{map}\ \mathsf{calculateScales}\ l) \\
\quad \mathsf{let}\ s_x, s_y \ = \ \mathsf{unzip}\ \mathsf{scales} \\
\quad (\mathsf{reduce}\ \mathsf{union}\ s_x,\ \mathsf{reduce}\ \mathsf{union}\ s_y), \mathsf{overlay}\ l'
\end{array}$$

When calculating the scales of the nest_x, 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 $\mathsf{scaledNest}_x$ shape stores the inferred X scale s_x of the sub-shape s. The second example is the overlay case which recursively infers 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 $0 \ldots \mathsf{width}$ and $0 \ldots \mathsf{height}$ where $\mathsf{width} \times \mathsf{height}$ 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:

$$\mathsf{project} \ : \ \mathsf{float} * \mathsf{float} \to \mathsf{Scale} \to \mathsf{Value} \to \mathsf{float}$$

The function is only defined if the value and scale are compatible. If both are continuous, the function performs a simple linear transformation. If both are categorical, the available pixel space is divided into a 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 corresponding to the area and a shape to be drawn. The result is a data structure representing a SVG document:

$$\mathsf{drawShape} \ : \ (\mathsf{float} * \mathsf{float}) * (\mathsf{float} * \mathsf{float}) \to \mathsf{Scale} * \mathsf{Scale} \to \mathsf{Shape} \to \mathsf{Svg}$$

For primitive shapes, the operation projects the coordinates using project and construct 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:

$$\begin{array}{l}
\mathsf{drawShape}\ a\ s\ (\mathsf{overlay}\ l) = \\
\quad \mathsf{concat}\ (\mathsf{map}\ (\mathsf{drawShape}\ a\ s)\ l)
\end{array}$$

$$\begin{array}{l}
\mathsf{drawShape}\ ((x_1, y_1), (x_2, y_2))\ (s_x, s_y)\ (\mathsf{scaledNest}_x\ v_{min},\ v_{max},\ ns_x,\ \mathsf{shape}) = \\
\quad \mathsf{let}\ x'_1 = \mathsf{project}\ (x_1, x_2)\ s_x\ v_{min} \\
\quad \mathsf{let}\ x'_2 = \mathsf{project}\ (x_1, x_2)\ s_x\ v_{max} \\
\quad \mathsf{drawShape}\ ((x'_1, y_1), (x'_2, y_2))\ (ns_x, s_y)\ \mathsf{shape}
\end{array}$$

When drawing overlay, the function draws all sub-shapes onto the same area using the same scales and then concatenates the returned SVG components the concat helper. The $\mathsf{scaledNest}_x$ 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 perspective on the problem of finding 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, 2012), 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 functional 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). D^3 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.

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. (2012). The grammar of graphics. *Pages 375–414 of: Handbook of computational statistics*. Springer.