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 and highlight two parts of the timeline with two different colors.

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

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

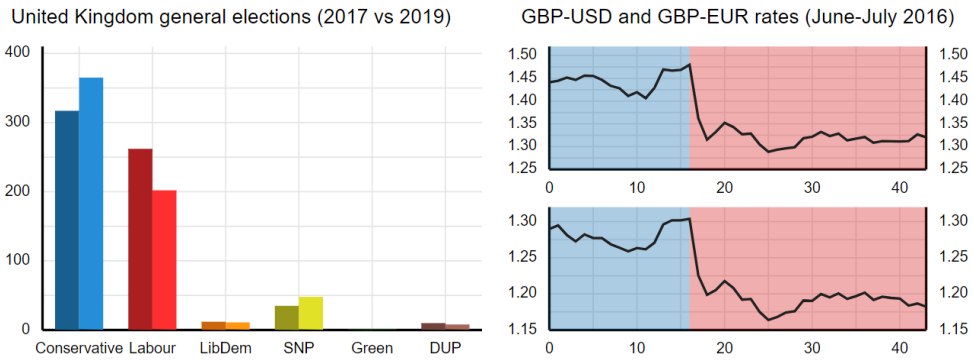

Fig. 1. Two charts about the UK politics: Comparison of election results from 2017 and 2019 (left) and GBP-USD exchange rate with highlighted areas before and after the 23 June 2016 Brexit vote.

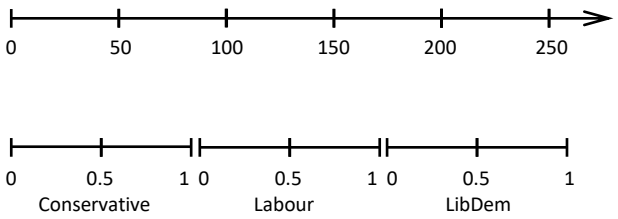

Fig. 2. On a continuous scale (above), an exact position is determined by a number. On a categorical scale (below), an exact position is determined by the category and a numerical ratio from 0 to 1.

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

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

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

2 Basic charts: Overlaying chart primitives

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

2.1 Domain-specific values

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

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

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 ]))
```

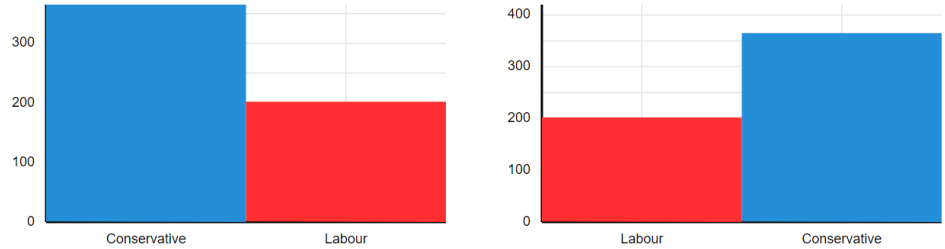

Fig. 4. Simple chart showing the UK election results; using automatically inferred scales (left) and using rounded Y scale and explicitly defined (reordered) X scale (right).

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:

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

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.

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.

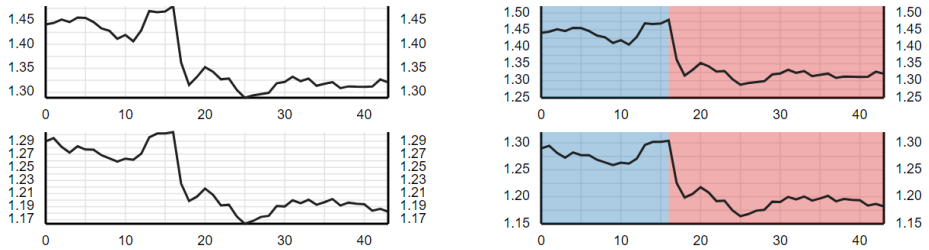

Fig. 7. Two charts showing currency exchange rates with a shared X scale and separate Y scales.

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):

$$\mathsf{axis}_l\ (\mathsf{axis}_b\ (\mathsf{roundScale}_y\ (\mathsf{explicitScale}_x\ (\mathsf{categorical}\ [\mathsf{Labour}, \mathsf{Conservative}]),\\ \mathsf{overlay}\ [\ \mathsf{labour}, \mathsf{conservative}\ ]\ )))$$

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 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

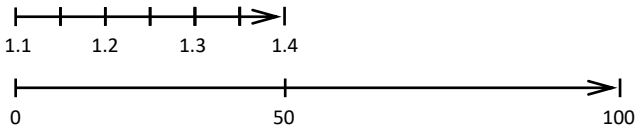

Fig. 8. A continuous scale with values from 0 to 6, nested in another scale.

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.

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)))) ]
```

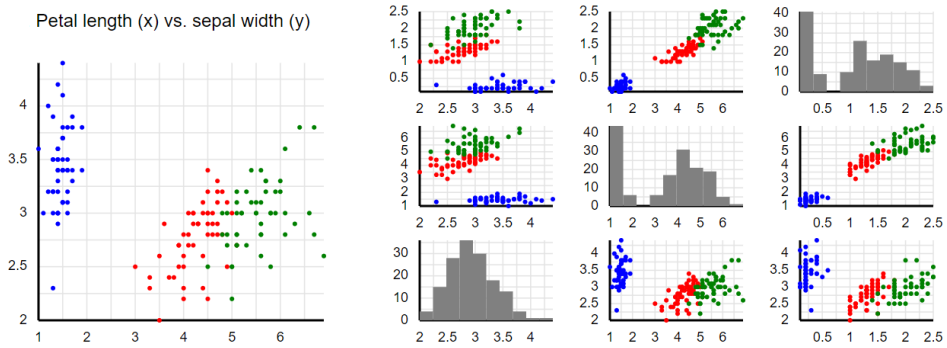

Fig. 9. Sample charts built using derived abstractions; a scatter plot visualizing the Iris dataset with a title (left) and a pairplot comparing two Iris features (right).

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.

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 $(\text{cont } 50), (\text{cont } 50)$. 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 Python library. 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

6 Implementation structure

Controlling and nesting scales

Nesting of charts and scales

[TODO: We really need illustrations for those example charts, but that would be more work!] [TODO: Maybe use this chart as a motivation: https://wellcomeopenresearch.org/articles/4-63/v1]

7 Random ideas

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