JFP **31**, e13, 18 pages, 2021. © The Author(s), 2021. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution, and reproduction in any medium, provided the original work is properly cited.

doi:10.1017/S0956796821000046

FUNCTIONAL PEARLS

Composable data visualizations

TOMAS PETRICEK
School of Computing, University of Kent, Canterbury CT2 7NZ, 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 can draw bar charts and line charts, but extra features like multiple bars for each label, alignment of multiple charts, or custom color coding can only be used if the library 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 share 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, 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

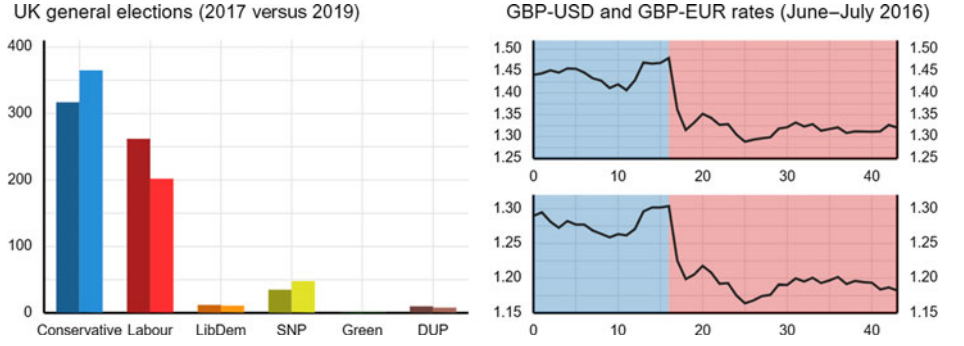


Fig. 1. Two charts about UK politics: comparison of election results from 2017 and 2019 (left) and GBP/USD exchange rate with highlighted areas before and after the 23 June 2016 Brexit vote.

https://doi.org/10.1017/S0956796821000046 Published online by Cambridge University Press

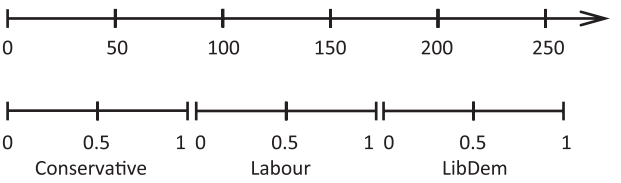


Fig. 2. On a continuous scale (above), an exact position is determined by a number. On a categorical scale (below), an exact position is determined by the category and a number between 0 and 1.

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.

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.
- Common chart types such as bar charts or line charts can be easily captured as high-level abstractions, but many interesting custom charts can be created as well.
- The approach can easily be integrated with the Elm architecture (Czaplicki, 2012) to create 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.

Compost is available as open source at http://compostjs.github.io. It is implemented in F# but is available as a plain JavaScript library thanks to the Fable F# to JavaScript compiler. The core logic consists of 800 lines of code and depends on the virtual-DOM library (http://npmjs.com/package/virtual-dom) for the implementation of the interactive features, making it easily portable to other functional programming languages.

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

https://doi.org/10.1017/S0956796821000046 Published online by Cambridge University Press

$$
\begin{array}{llll}
\textit{Category} & c & & \\
\textit{Ratio} & r & & \\
\textit{Number} & n & & \\
\textit{Text} & t & & \\
\textit{Color} & \gamma & & \\
\textit{Value} & v & = & \mathsf{cat}\ c, r \\
 & & \mid & \mathsf{cont}\ n
\end{array}
\qquad
\begin{array}{llll}
\textit{Shape} & s & = & \mathsf{line}\ \gamma, [\, v_{x1}, v_{y1}, \ldots, v_{xn}, v_{yn}\,] \\
 & & \mid & \mathsf{fill}\ \gamma, [\, v_{x1}, v_{y1}, \ldots, v_{xn}, v_{yn}\,] \\
 & & \mid & \mathsf{text}\ \gamma, v_x, v_y, t \\
 & & \mid & \mathsf{bubble}\ \gamma, v_x, v_y, n_w, n_h \\
 & & \mid & \mathsf{overlay}\ [\, s_1, \ldots, s_n\,] \\
 & & \mid & \mathsf{axis}_{l/r/t/b}\ s \\
 & & \mid & \mathsf{padding}\ n_t, n_r, n_b, n_l, s
\end{array}
$$

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.

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, and countries) or continuous (e.g. number of votes and exchange rates). The mapping from categorical and continuous values to positions on the chart is done automatically. We discuss this in Section 2.4.

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 2.5, continuous and categorical values can also be annotated with units of measure to make the values more descriptive.

2.2 Basic primitives and combinators

Compost is an embedded domain-specific language, implemented as a set of functions. In the subsequent code samples, we will use color to distinguish primitives of the Compost language, such as overlay or cat from primitives of the host language such as let or for.

A chart element is represented by a shape s, as 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

https://doi.org/10.1017/S0956796821000046 Published online by Cambridge University Press

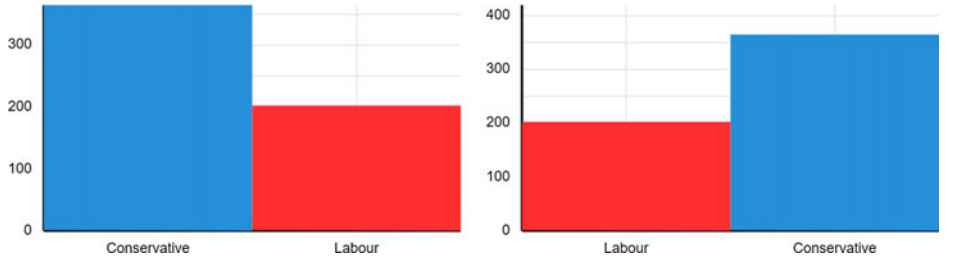


Fig. 4. Simple chart showing the UK election results; using automatically inferred scales (left) and using rounded Y scale and explicitly defined (reordered) X scale (right).

the 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) ]

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

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. 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, that is, 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) ] ] ] ))
```

https://doi.org/10.1017/S0956796821000046 Published online by Cambridge University Press

Aside from iterating over all available parties and splitting the bar, the example also adds padding around the bars, which is specified in pixels. A similar result could be 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 Choosing the level of language abstraction

Perhaps the most important aspect of the design of any domain-specific language is the level of abstraction it uses. The bar chart example discussed in the previous section illustrates the choice made in Compost. On the one hand, Compost gives us flexibility by letting us compose charts from shapes. On the other hand, Compost limits what we can do using two-dimensional space with positions determined by categorical or continuous values. In other words, the Compost design lies in the middle of a broader spectrum.

An example of a more general domain-specific language is the Pan language (Elliott, 2003) for producing images. Pan represents images as functions from a 2D point to a color. This makes it possible to create powerful combinators, for example, polar image transformation, but it makes it harder to express logic important for charting, for example, automatic alignment of shapes defined in terms of categorical values. Compost makes it easy to put two bars side by side in a bar chart, but harder to define generic combinators for aligning images.

An example of a less general domain-specific language is the Haskell Chart library (Docker, 2020). Haskell Chart provides a wide range of plots (such as lines, candles, areas, points, error bars, pies, etc.), but those can only be composed in limited ways by overlaying them or arranging them in a grid. The language is closer to the domain of the most common applications, but it places more restrictions on what can be expressed.

The design of the Compost domain-specific language aims to capture the key principles shared by most charts but avoid using a long list of different plot types. Different types of charts are all produced by composing shapes, but the ways in which shapes can be composed and transformed are limited to those that are needed for typical charts. We discuss the limitations of this approach in more detail in Section 7.

2.4 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 relatively 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 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 minimal and

https://doi.org/10.1017/S0956796821000046 Published online by Cambridge University Press

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

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

maximal ratios of the used categorical values as Compost will use 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:

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

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

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 $\mathsf{cat}\ c, r$ to the region corresponding to c according to the ratio r.

2.5 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 $\mathsf{cont}\ 317\langle \mathsf{mp} \rangle$ would then be incompatible with an axis containing a value $\mathsf{cont}\ 1.32\langle \mathsf{gbp/usd} \rangle$.

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:

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

https://doi.org/10.1017/S0956796821000046 Published online by Cambridge University Press

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

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

Correspondingly, there are two kinds of typing judgments; $v \vdash \sigma$ 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:

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

The rule for line ensures that all X and Y values have the same types, σ_x and σ_y, respectively, and infers $\mathsf{Shape}\ \sigma_x, \sigma_y$ 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 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 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 conservative and labour variables defined earlier):

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

Reading the code from the inside out, the snippet first overlays the two colored 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

https://doi.org/10.1017/S0956796821000046 Published online by Cambridge University Press

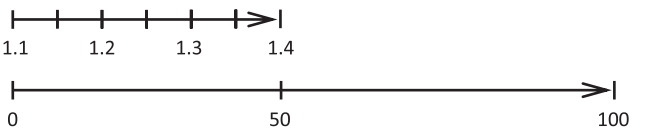


Fig. 7. A continuous scale with values from 0 to 6, nested in another scale.

shape, producing a usual labeled 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 $\mathsf{nest}_{x/y}$. As with other primitives like padding, the primitive takes a shape with some additional parameters and defines a new shape. Its behavior is similar to that of the SVG viewport (Dahlström *et al.*, 2011). The nest primitive takes two values, $v_{\min}$ and $v_{\max}$, 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 $\mathsf{nest}_{x/y}\ v_{\min}, v_{\max}, s$ 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. In other words, the operation defines a virtual coordinate system that exists only inside the newly created shape but is invisible to anything outside of the shape. 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, as shown in Figure 8 (left), using:

$$\begin{aligned}&\mathsf{overlay}\ [\ \mathsf{nest}_y\ (\mathsf{cont}\ 0), (\mathsf{cont}\ 50), (\mathsf{axis}_l\ (\mathsf{axis}_r\ (\mathsf{axis}_b\ (\mathsf{line}\ \#202020\ \mathsf{gbpusd}))))\\ &\qquad \mathsf{nest}_y\ (\mathsf{cont}\ 50), (\mathsf{cont}\ 100), (\mathsf{axis}_l\ (\mathsf{axis}_r\ (\mathsf{axis}_b\ (\mathsf{line}\ \#202020\ \mathsf{gbpeur}))))\ ]\end{aligned}$$

In this example, the x scale shows the days of the year. This scale is shared by both of the charts. Indeed, if data were 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

https://doi.org/10.1017/S0956796821000046 Published online by Cambridge University Press

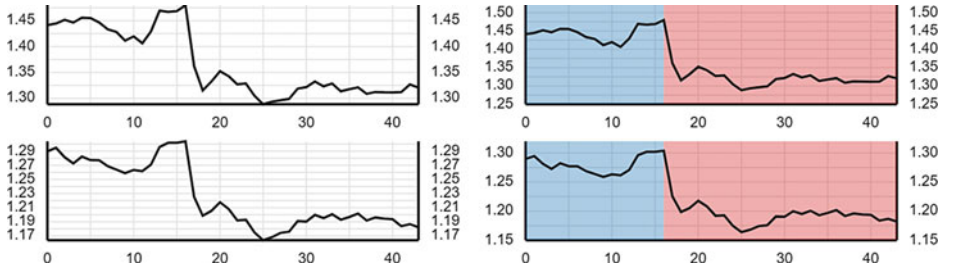


Fig. 8. Two charts showing currency exchange rates with a shared X scale and separate Y scales.

and 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 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 $\mathsf{nest}_{x/y}$.

https://doi.org/10.1017/S0956796821000046 Published online by Cambridge University Press

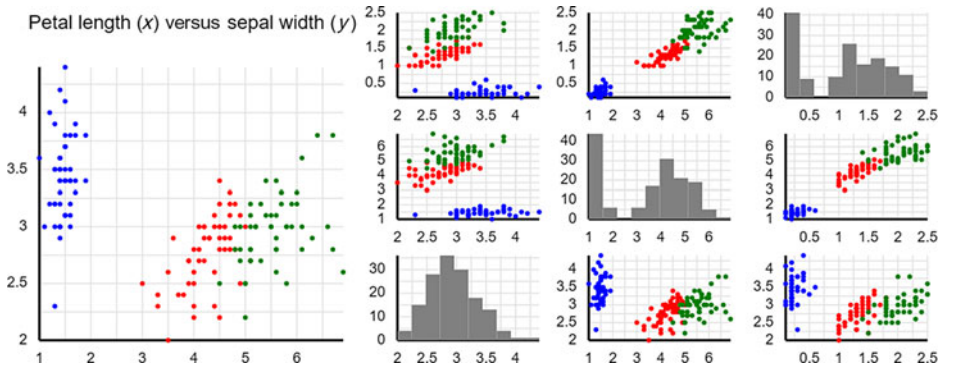


Fig. 9. Sample charts built using derived abstractions; a scatter plot visualizing the Iris dataset with a title (left) and a pairplot comparing two Iris features (right).

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). We assume that the text primitive centers the text, although the actual implementation also allows the user to specify horizontal and vertical alignment. 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, and 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 →
    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

https://doi.org/10.1017/S0956796821000046 Published online by Cambridge University Press

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

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

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 triple consisting of a lower and an upper range x_1, x_2 and the count y.

The example shows that 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.

Compost uses the Elm architecture (Czaplicki, 2016) to support interactive data visualizations. In this model, an interactive visualization is described using a pair of user-defined types and a pair of user-defined functions. The *state* type represents the current state of what is displayed (e.g. animation step or selection) and the *event* type represents actions that the user can perform (e.g. start an animation or draw a selection range). The two functions use the state and event types. The *view* function creates a chart based on the current state and the *update* function specifies how the state changes when an event occurs.

5.1 Domain-specific events

To support handling of mouse-based events, Compost adds three additional primitives to the definition of shape, as 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 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 .

https://doi.org/10.1017/S0956796821000046 Published online by Cambridge University Press

5.2 You Draw It data visualizations

To illustrate building 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:

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

The state is a pair of a 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, that is, a flag indicating whether a mouse button is down. The next two cases handle the Update event. The event carries two values, p and 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 an effectful function of type Event → unit that can be used to trigger events in handlers, registered using primitives such as mouseMove. The trigger function is provided by the Compost runtime. When it is invoked from an event handler, it takes the current state, transforms it using the *update* function, sets the new state as the current state, and invokes the *view* function to display the new state.

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) =
  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

https://doi.org/10.1017/S0956796821000046 Published online by Cambridge University Press

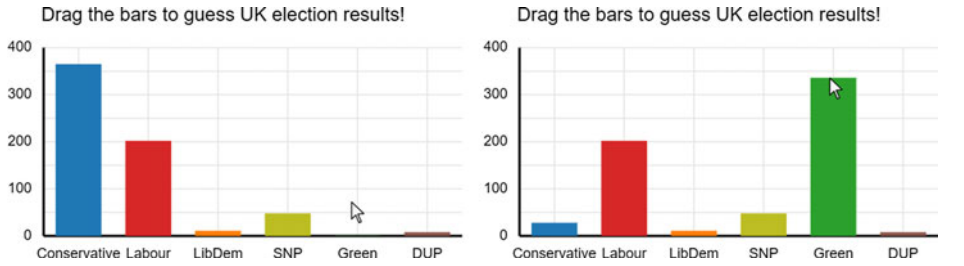


Fig. 11. Interactive "You Draw it" data visualization. The user moves cursor to a bar (left), pushes a mouse button, and drags the bar to the position that they think is the correct one (right).

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 primitives for specifying mouse event handlers can be nested or appear in multiple subshapes of the composed shape. This makes it possible to attach different event handlers to different parts of a chart and get event coordinates in local units. In case of nesting, the nested handler will capture events that occur in the space occupied by the shape it wraps, but it will ignore events occuring outside of this area.

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 2.4, 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

https://doi.org/10.1017/S0956796821000046 Published online by Cambridge University Press

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:

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

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 subshape, the operation calculates the scales of a subshape recursively and then adapts those somehow. To illustrate, we consider two interesting cases:

```
calculateScales (nest_x v_min, v_max, s) =
  let (s_x, s_y), s' = calculateScales s
  (union (singleton v_min) (singleton v_max), s_y), scaledNest_x v_min, v_max, s_x, s'

calculateScales (overlay l) =
  let scales, l' = unzip (map calculateScales l)
  let s_x, s_y = unzip scales
  (reduce union s_x, reduce union s_y), overlay l'
```

When calculating the scales of the nest_x, the function first calculates scales of the subshape 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 subshape s. The second example is the overlay case which recursively computes scales of all subshapes 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:

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

https://doi.org/10.1017/S0956796821000046 Published online by Cambridge University Press

The function is only defined if the value and scale are compatible. As discussed in Section 2.5, 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 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 mapped onto the area and a shape to be drawn. The result is a data structure representing an 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 constructs a corresponding SVG document. For shapes with subshapes, 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) (scaledNestx vmin, vmax, nsx, shape) =
  let x'1 = project (x1, x2) sx vmin
  let x'2 = project (x1, x2) sx vmax
  drawShape ((x'1, y1), (x'2, y2)) (nsx, sy) shape
```

When drawing overlay, the function draws all subshapes onto the same area using the same scales and then concatenates the returned SVG components using 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 $v_{\min}$ and $v_{\max}$. This defines the area corresponding to the nested scale ns_x, onto which the x coordinates in the subshape 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 Limitations and future work

As discussed in Section 2.3, the Compost library chooses a level of abstraction that makes it possible to express a wide range of charts but does not allow arbitrary image manipulation. The examples discussed so far provide a good review of what can be expressed using Compost. It is also worth considering what cannot currently be expressed. For many of the current limitations, we also consider what additional primitive would address the problem.

7.1 Radial charts and image transformations

Compost cannot currently produce pie charts and other radial charts. This could be supported by defining a primitive polar that renders a shape s specified as a parameter using a polar coordinate system instead of the default Cartesian system. Like the nest primitive, this would create a new shape that occupies a newly defined chart region. The polar primitive would make it possible to create pie charts, but also more elaborate Circos charts used to visualize genomic data (Krzywinski *et al.*, 2009).

https://doi.org/10.1017/S0956796821000046 Published online by Cambridge University Press

Radial charts provide a clear motivation for supporting polar geometries, but we do not currently expect the need for more general image transformations such as those supported by Pan (Elliott, 2003). Those are useful for producing visually appealing images but may not be necessary for data visualization. Arguably, we also do not expect the need for more general layout combinators such as above or besides (Yorgey, 2012). Those can be expressed elegantly using image transformations. In Compost, we can achieve similar effect using nest and explicitScale, as shown when defining title in Section 4.

7.2 Combinations and transformations of scales

Another area in which Compost could be extended is to allow more flexible handling of scales. Currently, categorical scales are mapped to bins of equal size and continuous scales are mapped using a linear transformation. The current design does not make it possible to use logarithmic scale or, for example, contracted axis where a subrange of values in the middle is omitted. Both of these could be supported if Compost allowed the user to specify a custom value transformation function.

Another interesting challenge is to allow overlaying of charts with multiple scales. This can currently be done using overlay together with nest. However, a more principled approach would be to allow the user to specify multiple, possibly named, scales for each shape. The calculateScales operation discussed in Section 6.1 would then need to return a list of scales rather than just a pair.

7.3 Controlling visual elements of a chart

There is also a number of occasions where the user might require more control over various visual elements of the chart such as fonts, text alignment, or visual aspects of the automatically generated axes. The current implementation of Compost already allows control over fonts, font sizes, and text alignment, but we omit the details for brevity.

Controlling the visual aspects of axes is a more interesting problem. In fact, the axis primitive described in this paper is not a primitive operation, but rather a derived one. It is implemented by calculating the scales of the shape specified as an argument and overlaying it with lines (for axes and grid), text elements (for labels), and adding a padding. The current implementation does not allow much customization, but the user can look at the implementation and easily create their own version, much like they can create their own version of the title operation described in Section 4.

8 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 spot 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 (Stolte *et al.*, 2002; Wickham, 2010; Satyanarayan *et al.*, 2015, 2016). In Grammar of Graphics, a chart is a mapping from

https://doi.org/10.1017/S0956796821000046 Published online by Cambridge University Press

data to 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 and size of bubbles) as scales, which would allow a more high-level specification of certain charts.

Acknowledgements

The Compost library is the result of my prolonged effort to create an elegant charting API for F#, which was supported, at various stages, by Don Syme at Microsoft Research and Howard Mansell at BlueMountain Capital. The idea of Compost first came together in discussion with Mathias Brandewinder and was (much much later) implemented thanks to the support of Google Digital News Initiative and The Alan Turing Institute. The final motivation for this paper was an invitation to talk at the Lambda Days conference in Kraków and the positive comments from the attendees. Finally, the anonymous referees provided valuable feedback that made this a better paper.

Supplementary materials

For supplementary material for this article, please visit http://doi.org/10.1017/S0956796821000046

Conflicts of Interest

None.

References

Aisch, G., Cox, A. & Quealy, K. (2015) *You Draw it: How Family Income Predicts Children's College Chances*. New York Times. Accessed May 24, 2020. Available at: https://www.nytimes.com/interactive/2015/05/28/upshot/you-draw-it-how-family-income-affects-childrens-college-chances.html.

Bostock, M., Ogievetsky, V. & Heer, J. (2011) D^3 data-driven documents. *IEEE Trans. Visualization Comput. Graphics* **17**(12), 2301–2309.

https://doi.org/10.1017/S0956796821000046 Published online by Cambridge University Press

Czaplicki, E. (2012) *Elm: Concurrent FRP for Functional GUIs*. Senior Thesis, Harvard University. Available at https://elm-lang.org/assets/papers/concurrent-frp.pdf.

Czaplicki, E. (2016) *A Farewell to FRP: Making Signals Unnecessary with The Elm Architecture*. Accessed May 24, 2020. Available at: https://elm-lang.org/news/farewell-to-frp.

DahlstrÃűm, E., Dengler, P., Grasso, A., Lilley, C., McCormack, C., Schepers, D. & Watt, J. (2011) *Scalable Vector Graphics (svg) 1.1*, 2nd ed. W3C Recommendation. Accessed May 24, 2020. Available at: http://www.w3.org/TR/2011/REC-SVG11-20110816/.

Docker, T. (2020) *Chart: A Library for Generating 2D Charts and Plots*. Haskell Hackage. Accessed December 9, 2020. Available at: https://hackage.haskell.org/package/Chart.

Elliott, C. (2003) Functional images. In *The Fun of Programming*, Chapter 7, Gibbons, J. & de Moor, O. (eds). Palgrave.

Google. (2020) *Google Charts: Interactive Charts for Browsers and Mobile Devices*. Google. Accessed May 24, 2020. Available at: https://developers.google.com/chart.

Kennedy, A. (2009) Types for units-of-measure: Theory and practice. In *Central European Functional Programming School*. Springer, pp. 268–305.

Krzywinski, M., Schein, J., Birol, I., Connors, J., Gascoyne, R., Horsman, D., Jones, S. J. & Marra, M. A. (2009) Circos: An information aesthetic for comparative genomics. *Genome Res.* **19**(9), 1639–1645.

Satyanarayan, A., Moritz, D., Wongsuphasawat, K. & Heer, J. (2016) Vega-lite: A grammar of interactive graphics. *IEEE Trans. Visualization Comput. Graphics* **23**(1), 341–350.

Satyanarayan, A., Russell, R., Hoffswell, J. & Heer, J. (2015) Reactive Vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE Trans. Visualization Comput. Graph.* **22**(1), 659–668.

Stolte, C., Tang, D. & Hanrahan, P. (2002) Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Trans. Visualization Comput. Graphics* **8**(1), 52–65.

Waskom, M., Botvinnik, O., Hobson, P., Warmenhoven, J., Cole, J. B., Halchenko, Y., Vanderplas, J., Hoyer, S., Villalba, S. & Quintero, E. (2014) *Seaborn: Statistical Data Visualization*. Accessed May 24, 2020. Available at: https://seaborn.pydata.org/.

Wickham, H. (2010) A layered grammar of graphics. *J. Comput. Graphical Stat.* **19**(1), 3–28.

Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer.

Wilkinson, L. (1999) *The Grammar of Graphics*. New York: Springer-Verlag.

Yorgey, B. A. (2012) Monoids: Theme and variations (functional pearl). In Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012, Voigtländer, J. (ed). ACM, pp. 105–116.

https://doi.org/10.1017/S0956796821000046 Published online by Cambridge University Press