A Transformational Approach to Data Visualization

Karl Smeltzer Martin Erwig Ronald Metoyer
Oregon State University, USA
{smeltzek,erwig,metoyer}@eecs.oregonstate.edu

Abstract

Information visualization construction tools generally tend to fall in one of two disparate categories. Either they offer simple but inflexible visualization templates, or else they offer low-level graphical primitives which need to be assembled manually. Those that do offer flexible, domain-specific abstractions rarely focus on incrementally building and transforming visualizations, which could reduce limitations on the style of workflows supported. We present a Haskell-embedded DSL for data visualization that is designed to provide such abstractions and transformations. This DSL achieves additional expressiveness and flexibility through common functional programming idioms and the Haskell type class hierarchy.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

Keywords Embedded DSL, Information Visualization, Visualization Transformations

1. Introduction

Data collection and storage requirements are growing at an exponential rate [21]. As the data analysis process has evolved to keep up with this increasing demand, information visualization has emerged as an important component [17].

For further motivation, consider the following anecdote. In 2010, economists Reinhart and Rogoff published a now infamous paper [25] in which an Excel[1] error caused them to inaccurately report the relationships between debt and economic growth [16]. This incorrect work was then cited internationally in debates surrounding austerity measures and likely influenced economic policy. One of their primary errors was failing to select the correct number of data rows (of the 20 countries represented in the data only 15 were selected). Now consider a hypothetical situation in which these researchers had been using a visualization tool that encouraged quick, incremental visual data exploration. Figure 1 demonstrates a chart that such a tool could create quickly which might have immediately exposed that error. We have taken the liberty of numbering the bars to make the effect more obvious, but even without the numbers we can still observe that no countries in the A–E range are included or simply that there appear to be fewer than twenty bars by estimation. We claim that a powerful, incremental visualization tool could encourage scientists and data analysts to use these kinds of charts to perform a sanity check of their work, perhaps preventing a great deal of trouble in the process.

Creating effective visualizations is difficult, however, even for experienced data analysts. Despite the general prevalence of visualization construction tools and libraries, most are either too inflexible for custom problem domains or else require extensive low-level graphics expertise to construct even simple visualizations. For example, while Excel provides a diverse set of representations (bar charts, scatterplots, pie charts, etc.) there is limited ability to customize them, and they do not support incremental changes and transformations. Conversely, lower-level graphics libraries and languages such as Cairo[2] generally do offer increased flexibility, but typically lack visualization-specific abstractions completely.

Between these two extremes lies a continuum of domain-specific visualization software (see Section 7 for examples). Still, these options tend to treat visualizations as individual, finished products with little attention being paid to identifying abstractions which are reusable and support systematic analysis and transformation. Visualization and data analysis is an iterative process [30], which suggests that being able to generate new visualizations without having to start over could lead to a more suitable workflow. Because visualization transformations allow for incremental changes, they offer a way of achieving this.

As a possible solution, we present a Haskell-embedded domain-specific language (DSL) designed to provide a concise but expressive way to construct, compose, and transform data visualizations. Our implementation (built on Cairo) was used to generate all included figures. The design of this DSL is informed by a number of more specific goals. First, we provide multiple layers of abstractions in order to support multiple use cases. Sometimes a quick one-liner is all that is necessary to get an overview of data, while more customization and detail may be desirable for a figure designed to be shared. We also intentionally keep the number of core language constructs small, which is intended to simplify systematic analysis and ease the process of defining new, custom transformations. This can also allow the user to create custom abstraction layers to match particular requirements.

The embedded nature of the DSL also means that orthogonal problems such as data handling and rendering can be handled by the host language and treated separately. Additionally, the use of Haskell as a host language gives us access to its rich type class hierarchy which, in turn, can provide powerful operations using a style that is already familiar to many Haskell programmers. Finally, to further separate ourselves from low-level graphics programming tools, we provide an infinite, entirely scalable, unitless, and relative environment for sizing and layout. Large, complex visualizations

[1] `http://office.microsoft.com/en-us/excel/`

[2] `http://cairographics.org/`

GPCE'14, September 15-16, 2014, Västerås, Sweden
Copyright 2014 ACM 978-1-4503-3161-6/14/09...$15.00
http://dx.doi.org/10.1145/2658761.2658769

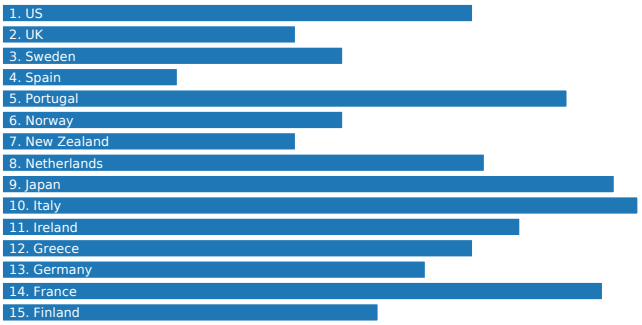


Figure 1. A chart showing some of the infamous Reinhart-Rogoff data with five missing values, as in the original work.

can be created without ever seeing pixel dimensions or dealing with raw geometric shapes.

We make the following specific contributions:

- A small set of core visualization construction components.
- Combinators and abstraction layers to hide low-level details.
- A set of transformations to systematically generate new visualizations from old ones.
- Haskell type class instantiations for functor and monad to support additional visualization operations and transformations.

The remainder of this paper is organized as follows. We begin in Section 2 by describing the core components that make up visualizations, how they are used, and how they can be transformed. In Section 3 we describe how visualizations can be merged and composed together. Sections 4 and 5 describe the use of Haskell's functor and monad type classes, respectively, in a visualization context. Section 6 offers evaluation based on a set of general-purpose visualization tasks. Finally, we outline related work in Section 7 and conclude in Section 8.

2. Visualization Components

Instead of a single, canonical way to build a given visualization, we provide a number of layered abstractions from which to choose the most appropriate for the situation. All of them, however, fundamentally produce a structure of type `Vis`. At the lowest level, the user can create an individual *mark*. A mark is a single graphical element consisting of a particular *primitive* shape and a set of *visual parameters*. Visual parameters are any components of a mark which can be bound to data, such as height, width, color, and orientation. All of these data types are then collected together in a `Vis`.

The following code snippet shows definitions for the data types corresponding to these ideas. While `Vis` is parameterized by a polymorphic type variable, in practice this is always instantiated with `Mark`. This will become useful in later, more complicated examples and, in theory, also presents an opportunity for later extensions to other types of visualizations. Note that the definitions reproduced here, and in many of the following examples, are incomplete. They are only intended to be illustrative.

```
data Vis a = Fill a | NextTo [Vis a] | Above [Vis a]

data Mark a = Mark Primitive [VisualParameter] a

data Primitive = R Rectangle | S Sector

data VisualParameter = Height Double | Width Double
```

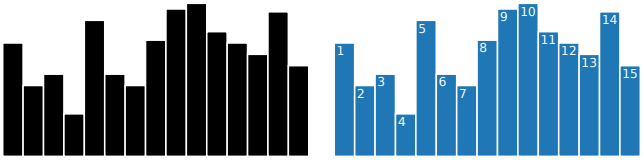


Figure 2. Two simple visualizations showing (left) the basic bar chart with only default values and (right) a basic bar chart with a few simple aesthetic customizations.

While it is possible to construct all visualizations by manually building structures out of these basic data types, it is rarely necessary to do so. For simple use cases, such as those covered by typical visualization templates, we offer a set of high-level functions which rely on simple defaults to construct basic charts. Examples of such combinators are shown in the following type signature.

```
barchart, piechart :: [Double] -> Vis (Mark Double)
```

Note that these functions and their peers all produce something of type `Vis (Mark Double)`. This type signature indicates that the visualization was constructed with a single floating point value for each mark. More complex visualizations may be parameterized differently. Also, these functions take only raw data lists as parameters, but we will demonstrate more flexible alternatives in later examples.

For intermediate cases where the defaults are insufficient, a number of mid-level abstractions are available. Marks and visualizations can be composed and transformed using any of a large set of combinators and smart constructors. The rest of this paper will demonstrate practical examples of these and how they can be used to visualize data.

2.1 Creating Simple Charts

Bar charts are one of the simplest and most common visualization types, making them a good way to demonstrate the practical use of this DSL. As mentioned in the previous section, a high-level function `barchart` is appropriate for the most straightforward use cases. It works by choosing a set of default options without prompting the user. In the following example, the `barchart` is used to construct the example chart already shown in Figure 1. For the sake of simplicity, we have extracted the data in lists of floating point values which are not reproduced here. Specifically, `rrData` contains data showing GDP growth by country for low debt/GDP ratios.

```
myBars = barchart rrData
```

The output of this simple visualization is shown on the left of Figure 2. In some cases this chart might already be sufficient to answer questions about the data, but the lack of color, missing labels, and dense marks could make it difficult to read. Fortunately, chart appearance can be customized via a number of aesthetic properties, thereby allowing for the creation of something that is not only more aesthetically pleasing but also more readable.

```
rrNumbers = map show [1..]

blueBars = myBars `color` blue `space` 0.1
                  `label` rrNumbers
```

Starting with the previously defined `myBars` visualization, three functions are applied to it. The `color` function simply assigns the given color to every mark in the visualization. Any RGB triple can be specified, as well as any of the X11 color names. In this case we color the bars blue.

Next we apply the `space` function, which intersperses whitespace between the bars. Since the environment is unitless, the amount of

space is specified with respect to the width of the bars. The use of the literal 0.1 means to use 10% of the bar width. The details of relative spacing are discussed later, in Section 3.

Finally, the `label` function is used to apply simple numeric labels to the bars showing their row number from the original data set. Any text strings can be used to create labels. The rendered result, including these customizations, can be seen on the right side of Figure 2. While probably adequate, this chart would be stronger if it included the country names as labels. Additionally, since names can be long, we want to rotate the chart so that the bars are horizontal to make them easier to read. We do that as follows.

```
rotRight :: Vis (Mark a) -> Vis (Mark a)

rrBars = rotRight $ blueBars `label` rrCountries
```

After adding new labels and performing a 90° rotation, the output produced is the same as seen originally in Figure 1. This chart emphasizes the error in the data set more directly, since both the numbering scheme and the country names indicate it.

Note that the change of the visualization could be expressed by applying a transformation to the existing visualization; it is not required to edit or rewrite the original visualization.

It is also worth noting that this process may still seem somewhat involved, but visualization customizations can be saved for later reuse. For example, we could write and keep the following function.

```
faveBars = rotRight . (`color` blue) . (`space` 0.1)
                                     . barchart
```

We could then use this just like the `barchart` function and it would automatically apply our customizations for color, spacing, and rotation.

2.2 Visualization Transformations

Bar charts are often an effective way to compare individual data points, but they are not ideal for every situation. However, knowing exactly which type of visualization to use for a given task requires an unrealistic amount of foresight. Fortunately, the use of visualization transformations allows switching between visualization types without losing customizations.

Our next example uses a data set of passenger information from the RMS Titanic, the ship which famously sank in 1912. We are specifically interested in examining the number of passengers in each of the travel classes, i.e., first class, second class, and third class. Additionally, we want to compare these with the number of crew members.

Imagine that we begin by creating a bar chart showing this information, as in the previous example.

```
classSize = map length $ group classDat
classLabels = map show classSize

classBars = barchart `color` forestgreen
                     `label` classLabels
```

After analyzing the output (which is not shown), we realize that a bar chart does not emphasize these data as ratios of all the passengers, making it difficult to see, for example, how big the crew was compared to the number of passengers. To fix this, we decide to create a pie chart[1] instead of a bar chart. We could start over and use the `piechart` function, which was briefly shown earlier, but that would require re-applying all of the previous customizations, too. Instead, an alternative solution is to generate a new pie chart by transforming our existing bar chart, which will then automatically inherit all of the customizations applied previously. To transform a bar chart into a pie chart we will need to make use of two transformation functions.

```
circular, reorient :: Vis (Mark a) -> Vis (Mark a)
```

To fully understand how these functions work, it is helpful to introduce some additional details. First, let us look more closely at the partial definition for the `Vis` data type that was reproduced in Section 2.

The `Fill` constructor is just a wrapper which serves as a base case for more complicated visualizations. The `NextTo` and `Above` hold lists of visualizations and compose them either next to or above one another. It is important to note that while `NextTo` places marks beside one another, that does not necessarily imply a rectangular coordinate system in which the list elements are aligned along a horizontal axis. An analogous point can also be made about `Above`. Some cases do behave that way, such as when creating bar charts, but `NextTo` and `Above` are actually more general. A pie chart, for instance, can also be constructed with `NextTo` by using pie wedges anchored around a single point rather than bars.

We call these pie wedges *sectors*, because they do not necessarily make actual wedge shapes. Notice that `Sector` is a constructor of type `Mark`. When `NextTo` is used with sectors, it assigns space by angle around a point, much like the theta angle measurement in a polar coordinate system. The sectors can still be understood to be "next to" one another in sequence, just with a nonlinear orientation. By convention we consider the zero angle to extend to the right from the center point, and an increase in the angle moves counterclockwise. Similarly, `Above` can be applied to sectors but rather than aligning them along a vertical access, it divides the space into sequential, concentric rings around the center point. Outer rings are understood to be "above" inner rings.

Given these definitions, similarities between bar charts and pie charts become more apparent. In fact, they are constructed in exactly the same way save two small differences: the use of sectors rather than rectangles and an angle-based orientation rather than a rectangular one.

We can now return to the aforementioned transformation functions and better understand how they work. The `circular` function will traverse a visualization and essentially convert every rectangle into a sector. It also inserts some bookkeeping information which is not relevant to the example. The actual parameters such as the embedded data, width, height, color, and label are not changed at all. Instead, the width of the original bar is used to determine the angle allocated to the sector, and the height of the original rectangle is used to determine the inner and outer radii of the sector.

It may intuitively seem as if applying `circular` is the only step necessary to convert between a bar chart and a pie chart. It does create a valid visualization, just not the one we want. By itself, as in the following code snippet, it actually transforms our bar chart into a rose chart, shown on the left in Figure 3.

```
rose = circular classBars
```

The reason for this is that, in addition to other parameters, our sectors have inherited an orientation. A mark's orientation determines which visual parameter the data is bound to. As we have seen, default bar charts have a vertical bar orientation, meaning that the data are bound to the bars' height parameter rather than width. When transformed into circular space, the radius of each new sectors will inherit the height of the corresponding bar and the angles will all be constant, as the bar widths were all constant.

If we had produced a horizontally-oriented bar chart in the first place then we would have immediately obtained a pie chart upon

[1] The use of pie charts is sometimes considered poor practice because it requires readers to perform area estimates, which can be inaccurate [9]. However, pie charts remain popular and widespread, and thus we support them.

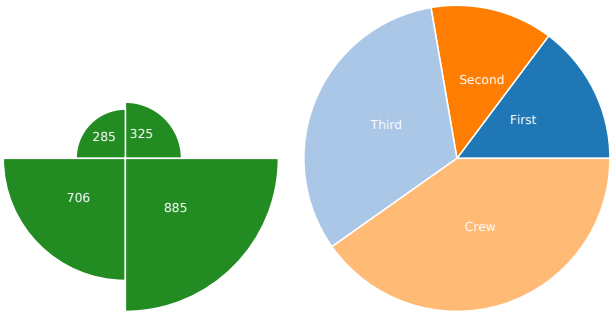


Figure 3. Two charts created by transforming an earlier bar chart. On the left is a rose chart in which the data is driving the radii of the wedges, and on the right is the same visualization recolored, relabeled, and reoriented so data drives the sector angles instead of the radii.

applying the `circular` function. Since that is not the case, the orientation of the sectors needs to be flipped. This is exactly the purpose of the `reorient` function. By itself, `reorient` will switch between vertical and horizontal bar charts or between rose charts and pie charts. Composed with the `circular` function, however, it will transform a vertical bar chart directly into a pie chart. Note that `reorient` could have been used in place of `rotRight` in the example from Section 2.

The complete code for the pie chart is shown below, and the output can be seen on the right in Figure 3. In order to produce a more readable visualization, the pie wedges are also recolored and relabeled.

```
classNames = ["First", "Second", "Third", "Crew"]

myPie = reorient . circular . autocolor $
          classBars `label` classNames
```

The `autocolor` function simply applies a list of infinitely repeated colors to the visualization, which is useful when you want to distinguish between marks more easily but are not concerned with the specific color choices. This can be configured rather easily, but the details of doing so are omitted for space.

This final pie chart provides insight into the proportion of people on board in each travel class to the number on the the entire ship. For instance, we can see that the crew made up approximately forty percent of passengers and that third class was larger than both first and second combined.

3. Chart Composition and Layout

Visualizations can also be composed together in a number of ways, allowing for the construction of more complex examples using simpler building blocks. Most fundamentally, visualizations can be composed spatially by dividing the canvas into parts, allowing multiple visualizations to be shown at once. There are a number of ways to accomplish this, but the simplest spatial composition functions are listed below.

```
above, nextTo :: Vis (Mark a) -> Vis (Mark a)
                              -> Vis (Mark a)
```

These functions divide the canvas in half either vertically or horizontally, respectively, and allocate each of the halves to one of the two visualizations, which is then scaled to fit. These can be nested arbitrarily deeply to produce hierarchical layouts. An attentive reader might wonder whether these functions simply wrap visualizations in `Above` and `NextTo` constructors. Aside from some bookkeeping related to scaling and framing, this is indeed the case.

It is often useful to compose charts in more meaningful ways than simply placing them near one another, however. Built-in functions are provided for a number of rich merging operations and, by virtue of keeping core data types simple, savvy users can create and save their own. Two of the built-in examples are listed below.

```
stacked, grouped :: Vis (Mark a) -> Vis (Mark a)
                                 -> Vis (Mark a)
```

As the name implies, `stacked` takes two visualizations and composes pairs of marks above one another, such as for stacked bar charts. Similarly, the `grouped` function zips pairs of marks from two visualizations together horizontally, useful for visualizations such as grouped bar charts.

To demonstrate the practical use of these operations, we return to our RMS Titanic data set. By now we have a high level understanding of the passenger breakdown by travel class, and we may want to examine the mortality rates, but broken down in a similar manner by travel class. The following example starts by building two distinct bar charts, one showing the survivors in each travel class and the other showing the deceased. We then compose them using both of the previous mentioned functions to construct a single, more complex visualization. To save space, the details of the original bar charts are omitted.

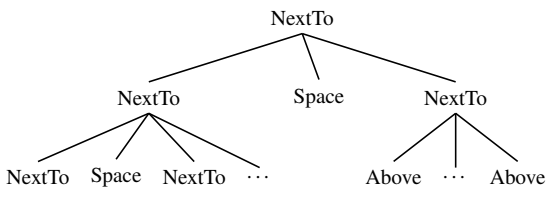
```
survived :: Vis (Mark Double)
deceased :: Vis (Mark Double)

groupedBars = grouped survived deceased
stackedBars = stacked survived deceased

composedVis = groupedBars `nextTo` stackedBars
                          `space` 0.3
```

The output of this code can be seen in Figure 4. This example demonstrates both kinds of composition mentioned. The first composes simple bar charts into more complex visualizations, specifically stacked and grouped charts. The second type is the spatial composing of these more complex visualizations into a single canvas using the `nextTo` function.

3.1 Whitespace

The output also provides further insight into the relative nature of the layout and scaling strategy in use. A previous example applied the `space` function to a single bar chart, which introduced space between the bars. The size was measured relative to the width of the bars. This example uses `space` again, but this time it is applied to the composition of two visualizations, and so it inserts whitespace between them relative in size to the entire visualization rather than between individual pairs of marks. The grouped bars have some space automatically inserted, but this is simply the default behavior for `grouped`.

Sizing determinations, especially those for whitespace, can be best understood by looking at visualizations as trees. Visualization trees are constructed according to user specification and are made up primarily of `NextTo` and `Above` nodes, each of which can have arbitrarily many children. In a well-defined tree, the leaf nodes are always `Fill` wrappers around individual `Marks`. A simplified tree partially representing the last example might look as follows.



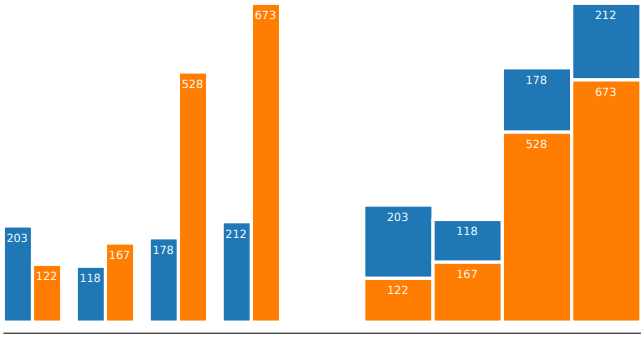


Figure 4. A single visualization showing two different representations of the same information. The blue bars show the number of survivors for the given travel class aboard the RMS Titanic and the orange show the number of deceased. This example demonstrates three types of composition: composing marks horizontally to construct bar charts, composing bar charts by stacking and grouping, and composing complete visualizations by dividing the canvas.

From this tree we can see how whitespace is captured internally. The `space` function always intersperses whitespace among the children of the node to which it was applied. Since we applied it to the root node (to the composition of the `groupedBars` and `stackedBars`), it interspersed whitespace between those two subtrees. The space sizing is also calculated relative to those sibling nodes, so in this case the specification of 0.3 produced whitespace equal in width to 30% of the width of the entire individual bar charts. Note that whitespace nodes were also automatically inserted in the left subtree, which represents the grouped bar chart, by the `grouped` function. These nodes create the spaces between each pair of bars.

Since the leaves in this tree are always `Fill` nodes, it may become evident that whitespace is actually represented internally using invisible marks. Not only has previous research [23] demonstrated that users often prefer to place whitespace as an actual visualization element rather than to describe it as negative space, but this also provides a great deal more flexibility in using whitespace to assist in layout tasks. Users can create their own whitespace marks and intersperse them throughout visualizations as desired, allowing them to use spaces to affect the layout of visualizations, or even to create unequal spaces to emphasize certain marks.

4. Visualization Functor

By virtue of being embedded in Haskell, we can make use of the type class hierarchy to provide additional mechanisms for building and transforming visualizations.

The foremost use of type classes is the visualization `Functor` instance. Implementing visualizations as functors provides a structure-preserving way of applying functions to individual parts of a visualization. In this case those individual parts are the graphical marks. Instantiating the visualization functor requires an implementation of the `fmap` function, which is reproduced below for the parts of `Vis` we have looked at.

```
instance Functor Vis where
  fmap f (Fill m)    = Fill (f m)
  fmap f (NextTo vs) = NextTo $ map (fmap f) vs
  fmap f (Above vs)  = NextTo $ map (fmap f) vs
```

This definition recursively applies `fmap` through each kind of visualization until it reaches the base case `Fill`, at which point the function being mapped is applied to the `Mark`. With this it is possible to apply any function of type `a -> b` (which typically means `Mark a -> Mark b` in practice) to any visualization. This can

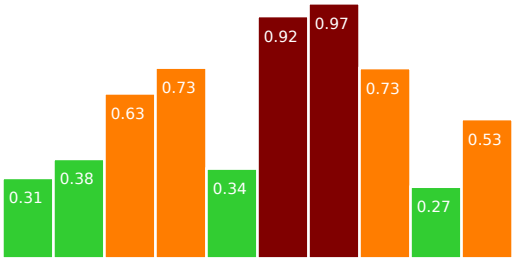


Figure 5. Conditional visualization formatting applied using `fmap` from the Haskell `Functor` type class. Bars with a height above 0.75 are colored red, bars with heights between 0.5 and 0.75 are colored orange, and the rest are colored green.

help simplify some routine visualization tasks, such as conditional formatting and scaling. The following example shows how it can be used to highlight particular values according to one or more thresholds by recoloring them. Suppose we have data in the range 0..1, where higher values indicate some increased risk, and we want to accentuate these high values to make them stand out from the rest of the data.

```
redHigh m = let h = mGetHeight m
                c | h > 0.75  = red
                  | h > 0.5   = orange
                  | otherwise = green

conditionalBars = fmap redHigh $ barchart myData
```

This example begins with the definition of a custom function `redHigh` which queries the height of a `Mark` and then applies a color depending on that value. We make all values above 0.75 red, all values from 0.5..0.75 orange, and the rest green. Next, `redHigh` is mapped across a newly created barchart. The result is shown in Figure 5. Remember that, by default, this barchart would have colored the bars black.

For the sake of comparison, accomplishing something similar in Excel typically requires the use of at least two spreadsheet equations to split the data into two columns, and then manually applying separate formatting to each column. If, at a later point, one wishes to change the threshold value, then that process needs to be repeated completely. While defining a custom formatting function as we have done is also nontrivial, one of the advantages of this approach is that such functions actually extend the language and can easily be saved for later use. The `redHigh` function is likely to be useful again later if similar data are regularly visualized. We could also continue to abstract the function by parameterizing the threshold values, which would increase reusability even further.

The visualization functor is useful for more than just coloring marks, though. We could, for example, use it to apply a mathematical operation to the visualization without modifying the original dataset. A common operation in data analysis is taking the logarithm of every value, which we demonstrate next. We assume a data set `expData` which we have reason to suspect might show an exponential trend and use the visualization functor to transform the initial visualization.

```
markLog :: Mark a -> Mark a
markLog m = mSetHeight m $ log $ mGetHeight m

expBars = barchart expData `label` myLabels
                           `color` blue
logBars = fmap markLog $ expBars `color` orange
```

The rendered output of both `expBars` and `logBars` can be seen on the left and right sides, respectively, of Figure 6. From this we

Figure 6. On the left is a simple barchart of data that appear to show an exponential trend. On the right is the same barchart after log-transforming each of the bars, showing what appears to be a linear trend.

can see that, indeed, the log transform appears to give the data a linear trend rather than an exponential one.

This may not seem like an interesting result since the data itself could always be log-transformed independently of the visualization creation process. Using a visualization functor has two advantages, in this case. First, it assumes no foresight and does not require a new visualization to be created, instead relying on the transformation of an existing one. A more traditional tool would require creating an additional data column and a complete new visualization. Second, the bars are only changed visually and the original data values are still accessible, allowing for further incremental exploration in case a log-transform turns out to be insufficient.

5. Visualization Monad

In addition to the `Functor` type class, we also make use of Haskell's `Monad` type class. This allows functions to be written that can take individual marks from a visualization as input, construct entirely new visualizations from them, and automatically stitch those parts back together. Monads are particularly under-explored in the context of visualization despite a number of compelling, intuitive use cases.

The relevant part of the `Monad` instance definition is reproduced below. It is abstracted out into two extra functions which each traverse the entire visualization. This improves clarity at the cost of performance, but could easily be changed for extremely complex visualizations.

```
instance Monad Vis where
  return         = Fill
  (Fill m) >>= f = f m
  v >>= f        = norm $ fmap f v

norm (Fill v)    = v
norm (NextTo vs) = prune $ NextTo (map norm vs)
norm (Above vs)  = prune $ Above (map norm vs)

prune (Fill m)    = Fill m
prune (NextTo vs) = NextTo $ map prune
                                (filter nEmpty vs)
prune (Above vs)  = Above $ map prune
                                (filter nEmpty vs)
```

The definition for the bind (`>>=`) function is similar to the definition for `fmap`. The function that is passed as a parameter is applied directly to a mark contained in a `Fill` node, and is mapped across the nested visualizations for other node types. The main difference from the `fmap` definition lies in the `norm` and `prune` functions. The `prune` function removes any visualization nodes which are empty, such as `NextTo`/`Above` nodes with no children. The `norm` function performs the flattening of visualizations from type `Vis (Vis a)` to type `Vis a`.

The visualization monad has a number of practical uses. We can, for example, use it to produce a more interesting version of the log-transform example in Section 4, which will show both the original and log-transformed bars at the same time. This is demonstrated in

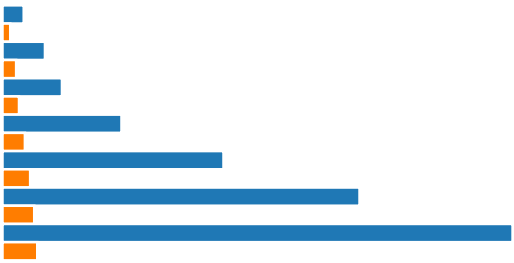


Figure 7. Another example of a log-transformed visualization, as in Figure 6, except this time using the visualization `Monad` instance to generate new bars in place beside their counterparts.

the following code, which assumes the definitions from that example are still available.

```
logBeside m = Fill ‘color‘ blue ‘nextTo‘
              (Fill (markLog m) ‘color‘ orange)

logExpBars = rotRight $ expBars >>= logBeside
```

The previous definition of `logMark` is conveniently reused here. Instead of just modifying each mark, however, this example generates completely new visualizations from them. This is made possible by the additional power monads offer over functors, and is also apparent when comparing the type signatures of `logBeside` and `markLog`, i.e., `Mark a -> Vis (Mark a)` versus `Mark a -> Mark a`. The `markLog` function is still applied to each mark, but now that result is composed beside the original mark. `Fill` constructors are used to wrap the marks, although more complex examples might use other `Vis` constructors. We also color each of the bars to match the colors from the previous example.

The rendered result is shown in Figure 7. It shows the log-transformed bars immediately beside the corresponding untransformed bars, as expected. This combination allows us to see both the exponential and linear trend at the same time in the same visualization. When the log-transformed bars are shown on the same scale as the full-length bars, they are quite small. This could be addressed by unzipping the visualization, and separating the bars into two distinct visualizations again so that each is scaled independently. The code for this is below, but the output is not shown since it is so similar to the output in Figure 6.

```
let (ex,ln) = vUnzip logExpbars
 in ex ‘nextTo‘ ln ‘space‘ 0.1
```

Achieving an effect like this in another tool would require the same data manipulation steps described previously in Section 4, but also another operation to somehow zip together and merge the two data columns. This is true even for tools such as ggplot2 [33], which includes coordinate transformations but does not allow separate transforms for different parts of the same visualization.

Another use case for the visualization monad is in implementing drill-down or roll-up operations, which increase and decrease the granularity of a visualization, respectively. Switching between an overview and a more detailed, granular view is a crucial component in visual data analysis [26]. While this kind of operation can theoretically be accomplished in most flexible visualization tools, it often requires a great deal of work and foresight on the part of the visualization creator. This difficulty is sometimes compounded by trying to drill-down or roll-up multiple data sets at once, which then requires complex, manual layout specification.

Leveraging the `Monad` instance allows us to achieve this affect by just defining a single function which generates the new visualizations of different granularity from the existing marks. As a lead-in

to the full example, the following code defines a bar chart with no drill-down or roll-up functionality.

```
barsFrom :: (a -> Double) -> [a] -> Vis (Mark a)

dat :: [[Double]]
dat = [dat2010, dat2011, dat2012, dat2013]

colorNegative :: Mark a -> Mark a

drilledBars = fmap colorNegative $ barsFrom (!! 2) dat
```

We have used a different function here for creating bar charts than in any of the previous examples, called `barsFrom`. The reason for this is that this example uses a more interesting data format than the previous examples. Unlike `barchart`, `barsFrom` takes data of a completely polymorphic type with the caveat that the user also supply a function for extracting floating point values from it.

A data set of profit and loss information for a business, called simply `dat`, is given. This data set contains information for each month for four years, structured as a nested list. For business reasons, we first want to compare the data associated with the month of March for each of the four years. To accomplish this, we pass `(!! 2)` to the `barsFrom` function, which is zero-indexed and will extract the value for the third month for each year. Finally, we make use of a function `colorNegative` (the definition of which is not shown) to color negative values and positive values differently. The corresponding bar chart can be seen on the left side of Figure 8.

After seeing this subset of data we decide that we also want to see context. That is, we want to see additional months to see how the March data fits in. To see this we effectively must perform a roll-up of the visualization. We can make use of monadic functions to accomplish this in the following way.

```
colorN :: Int -> Color -> Vis (Mark a) -> Vis (Mark a)

rollup = rotRight . colorN 2 lightorange .
           fmap colorNegative . barchart . mGetData

rolledBars = drilledBars >>= rollup
```

The first thing we have done is to define the custom `rollup` function. Written in point-free style, this function does five things. First, it extracts the original data set from the mark. In this particular case that means extracting the list of values for the entire corresponding year. That data is then passed to the familiar `barchart` function. We then map `colorNegative` to recolor this new visualization. Applying the `colorN` function will index into a visualization and recolor a particular element. We use this to highlight the original bars in our rolled-up view by coloring the March bars light orange. Finally we rotate the visualization to better make use of the vertical space.

With this function, we can transform the initial March-only chart that we created previously by using `>>=`. The output is shown on the right of Figure 8.

It is worth noting that Haskell's type system provides a practical safety guarantee here that the data extracted by `mGetData` are appropriate for creating the new visualization. The type of the data can be extracted from the type of the original visualization.

This way of extracting and manipulating data from marks is not intended to be an ideal data storage and lookup mechanism. Instead, it acts more as a lightweight alternative to querying a full database. It is worth noting, however, that this problem of data management is orthogonal to the visualization monad and what it offers. Just as we use `mGetData`, we could also query another data source based on a unique identifier of some sort.

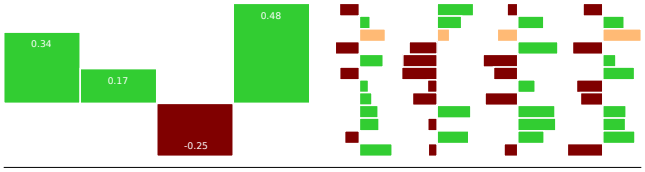


Figure 8. On the left is a simple chart showing some income data for a particular month across four years. On the right is a visualization of the data for every month across four years, obtained by transforming the visualization on the left using a custom monadic function. The original bars in the left visualization are highlighted on the right by coloring them distinctively.

5.1 Visualization Comprehensions

By using the GHC[1] extension for monad comprehensions, which provides syntactic sugar for Haskell's "do" notation, we can also allow users to create interesting, monadic visualizations using the same syntax used for list comprehensions. Visualization comprehensions are particularly useful in situations where we are interested in analyzing the Cartesian product of two data sets, or some filtered subset thereof, resembling a join operation in a database.

Here again, the strength of the visualization monad approach is that the actual data source need not be manipulated. If visualizations for each of the two data sets have already been generated, we can use a visualization comprehension with two generators to extract and join the marks from each with the existing data.

As an example, consider the situation where we want to analyze data related to employee performance in an office. There are two variables. The first is a (possibly negative) value for each employee in the office reflecting their performance relative to some standard measure. That is, a high value in the first data set represents an overachieving employee and a negative value represents an underachieving employee. There are 30 employees. The second data set contains values for five possible training courses. The actual data values show the estimated employee performance improvement associated with attending that particular training course. For bureaucratic reasons, the goal is to ensure as many employees as possible meet the standard by sending underachievers to training courses. Naturally, the optimal solution would seem to be sending all under-performing employees to the course with the highest estimated improvement value. However, the courses all have different costs and so we want to be able to select the most cost-efficient way of achieving our goal. To support this process, we want to see all possible, relevant combinations at once.

```
employees = barchart emplData `color` orange
courses = barchart courseData `color` blue
ecBars = employees `above` courses
```

We begin by charting the two data sets separately to get an overview. The output is not shown since it is similar to previous examples. This visualization is insufficient to easily meet our goal, however, since it shows two separate charts and we still need to manually estimate the effect of each course on each employee. Instead, we want to visualize the actual estimated end result of each possible combination. For this, we can use a visualization comprehension with two generators, a filter, and functions to combine bars together and give them an appropriate color.

[1] `http://haskell.org/ghc`

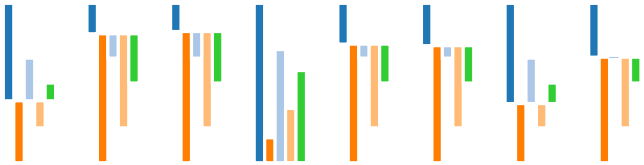


Figure 9. Estimated employee performance data created using a visualization comprehension to combine two existing bar charts. Labels identifying employees are omitted because of formatting limitations.

```
courseColor :: Mark a -> Mark a -> Mark a

addBars m1 m2 = let h = mGetHeight m1 + mGetHeight m2
                in mSetHeight m1 h

empCombo = clean [courseColor c (addBars e c)
                  | e <- employees
                  , c <- courses
                  , mGetHeight e < 0] `space` 0.6
```

We begin by defining the `courseColor` function. This is not shown because it is boilerplate code similar to the `redHigh` function from Section 4. Its purpose is to color each mark according to the training course it represents, i.e., it colors our final bars using one of five distinct colors corresponding to the data points in `courses`. Next we define `addBars` which combines two bars into a single one by adding their respective heights. This will allow us to merge one bar from each of the two existing visualizations together, giving the estimated final performance value for that employee and training course. Now we are ready to use a visualization comprehension to generate the new chart. The comprehension extracts all marks from `employees` and `courses`, filters out those employees who are already overachieving, and then combines all remaining pairs of bars. We also apply the `clean` function to the entire visualization to remove any leaves which are not `Fill` nodes, which may be introduced by filtering the marks. This could be done automatically before rendering, but we avoid doing so in case of unforeseen uses. Finally we add space between each group of bars to visually separate them. Note that we would ideally keep labels for the bars to identify which employee is being represented. Here we omit them only because making them legible would require too much space. The rendered output is shown on the right side of Figure 9.

The chart shows five bars—one per training course—for each of the underperforming employees. Each bar represents the estimated final performance value after having taken the course. Negative bars show cases where the improvement is not enough to cross the threshold. From this result we can select the least expensive course for each employee that will bring him or her above the standard measure, i.e., produces a positive value.

This kind of operation can be tedious in other visualization tools. They would typically require the user perform some kind of join between the two data sets, while filtering and sorting the values ahead of time. Our approach, by contrast, allows these operations to be performed on the visualization itself by transformations, leaving the data source alone. Perhaps more importantly, all the functions are reusable. Should another manager be interested in applying the same techniques, we could simply share the functions leaving only the simple task of plugging in the relevant employee data set.

6. Evaluation

Evaluating visualization tools is inherently difficult [24]. Common HCI evaluation techniques and user studies are challenging to apply because visualization tools often intentionally sacrifice accessibility or usability for gains in expressiveness or performance. Given such a sacrifice, a typical usability evaluation may offer only limited value. Additionally, a prototype tool such as presented here, almost certainly lacks some features that data analysts would find helpful or necessary, which precludes long-term case studies and deployments since users would not be able to use it for their full-time work.

Some popular visualization tools (e.g., [6]) have used Cognitive Dimensions [13] as an evaluation framework, although this is also problematic since it includes no objective measure of what is considered good or bad. Others have opted to measure success based on real-world user adoption rates [7, 31]. The most common option is to forego typical evaluation criteria completely and to instead demonstrate functionality equivalent with, or superior to, established tools [10, 12, 28, 32, 33].

Some work has proposed visualization-specific evaluation techniques and criteria, or proposed task-based taxonomies that could be used to extract functional requirements [1, 2, 8, 18, 27]. Many of these are still designed for interactive, graphical tools and are difficult to apply to a language-based solution, however.

6.1 Applicable Evaluation Schema

In this work, we turn to Brehmer and Munzner's typology of visualization tasks [8] to provide a set of requirements adequate for evaluating this work as a general-purpose visualization tool. That work is intended to classify "empirically observable events" and to (among other things) serve as an evaluation framework.

However, we make one important distinction. Our DSL need not necessarily serve as a complete, user-facing product in itself. As is the case with Tableau/Polaris [28], the DSL could hypothetically be used as the underlying representation for a tool which adds new types of interactivity and functionality. This means that the tasks in the "why?" and "what?" categories put forth by Brehmer and Munzner are actually orthogonal to our goals, as they are focused on visualization reading rather than visualization specification and implementation. Therefore, we focus exclusively on tasks in the "how?" category. These provide a set of functional requirements appropriate for evaluating the effectiveness of a visualization construction tool. Each is discussed here in turn.

Select This task involves selecting individual marks in a visualization, such as by clicking with the mouse. The actual interactions and input handling necessary to support this are orthogonal to our work, and can be achieved by embedding this work in a larger tool with an industrial-strength event-handling library. Of greater interest, then, is how the general concept of selection can be handled in our DSL. One possibility is illustrated by the following code snippet, where we assume that each mark contains a selection flag.

```
selectWhere f = fmap (\m -> if f m
                            then doSelect m
                            else m)
```

By defining a higher-order selection function in this manner, interactive events can be made to generate functions of type `Mark a -> Bool` automatically and then apply `selectWhere` accordingly. We could, for example, generate a function which selects marks in a certain geometric range selected by the users, or a function which selects marks according to a unique ID to support brushing across linked views. When combined with event-handling, this is sufficient to implement essentially any type of selection operation.

Navigate Navigation tasks are defined as methods that alter a user's viewpoint, which also includes showing data at different granularities. We have demonstrated this functionality directly in the roll-up example in Section 5. Additionally, we allow the spatial composition of arbitrarily many visualizations into a single canvas, which directly allows for multiple views of data to be shown at once.

Arrange Allowing visualization elements to be rearranged is one of the core strengths of our approach. It is possible to break any visualization down into its core components and then reconstruct them as desired. Furthermore, mechanisms are provided for indexing into visualizations (as shown by the highlighted bars in the roll-up example in Section 5), for sorting visualizations (via `vSort`, not demonstrated), for grouping and merging (as in Section 3), and for unzipping (via `vUnzip`, described in Section 5 but not shown). The swapping of axes is also mentioned specifically, which is available via the built-in `reorient` function.

Change This category discusses specific tasks including altering the size and color of marks, transforming scales and axes, and transforming between grouped and stacked bar charts. All of these are demonstrated here. Altering size and color can be achieved using the visualization functor as described in Section 4, transformation of scales is shown in Sections 4 and 5, and the transformation and composition of bar charts is shown in Section 3. We also support changing visualization types completely in Section 3.

Filter This group of tasks is concerned with filtering visualization elements based on user criteria. We support this in several ways. First, we can rely on the host language to provide rich filtering operations that operate on the data directly. More importantly, the visualization monad and visualization comprehensions allow filtering with arbitrary predicates. This is demonstrated in the example from Section 5.1 where negative values are filtered out.

Aggregate Aggregation (or changing the granularity) of visualization elements is another task that we support in two different ways. We can rely on Haskell to aggregate values in our data directly, such as by grouping, but the more idiomatic approach is to use the visualization monad again. We show an example of how drill-down and roll-up operations can be implemented in Section 5.

Annotate It is not immediately obvious how well we are able to support the annotation task. We certainly allow marks to be labeled, as demonstrated repeatedly. We also embed this label in the actual mark structure, as specifically suggested. We do not, however, currently support adding annotations in any empty spaces. While this case is not mentioned specifically, it may be intended to be implicit in the description. Because of that, we only claim to support annotation tasks partially. However, a GUI wrapper for our work could conceivably introduce additional annotation support without needing to modify the language itself.

Import This task type involves adding new visualization elements to existing ones. We demonstrate and discuss a number of ways this can be done in Section 3. We offer an array of techniques for combining visualizations, both spatially and in more interesting ways. This allows new data points to be added at any time without losing existing work. While scatterplots are not discussed here, adding new elements to a scatterplot could be achieved through the user of an overlay operation, which we support.

Derive This task is focused on creating new data elements from existing ones. This is one of the primary strengths of our approach and focus on transformation. Scaling is mentioned specifically, one form of which we have shown in Section 4 when log-transforming our visualization. We could also rely on support from the host language for this task, if desired.

Record Saving visualization elements is something we gain automatically via the embedded nature of our work. Visualizations (as well as marks and transformations) can be bound to Haskell identifiers, as shown in all examples. These persist indefinitely and can be used to reproduce and recall earlier work. Creating screen shots is also supported, witnessed by the figures shown throughout this work, which were all generated using the prototype. The specific graphical history mentioned could be implemented as a layer on top of our work by capturing each incremental visualization.

Encode Data encoding is not thoroughly discussed in Brehmer and Munzner due to space limitations. We adopt the technique of binding data directly to visual parameters, which is also standard in successful visualization tools such as Tableau/Polaris [28] and ggplot2 [33].

6.2 Evaluation Conclusions

According to this set of criteria, our work meets nearly all criteria for the implementation of a visualization tool. The only real failure is in supporting particular kinds of global annotations.

This is not meant to suggest that it is already a complete visualization tool ready for public distribution, but rather that the technical foundations are strong and, with further extensions, it could serve as a complete specification and transformation language. Reaching that goal would involve adding additional mark types, additional text handling for things like legends, support for guides and axes, and more.

7. Related Work

Much of the vocabulary used in this work as well as the idea of binding data to visual variables were first developed by Bertin [3, 4]. We make use of these ideas directly, with the exception of preferring the term *visual parameter* to visual variable.

Perhaps the most widely used visualization tools are D3 [7] and its now unmaintained predecessor Protovis [6]. Both are popular Javascript libraries for creating and manipulating data-driven graphics. Protovis, in particular, took a DSL-based approach based on the design described in earlier work by the same authors [14]. In addition to making some higher-level abstractions available than those provided by D3 and Protovis, our work is distinguished from this through the use of functional programming idioms and type classes, particularly the use of functors and monads.

The decision to approach this problem with a domain-specific language is partially motivated by Mackinlay [19], who introduced the idea that information visualizations are inherently sentences of a graphical language. This approach has been used before, however. Haskell DSLs have been developed for astrophysics visualization [11] and scientific visualization [5] for example. The Haskell community has also produced Diagrams [35], and its predecessor Chalkboard [22] designed for image and figure creation. Like our own, this work makes use of some functional programming idioms and part of the Haskell type class hierarchy. That work has also greatly influenced the scalable and unitless layout used here. Our work, however, focuses on incremental, data-driven visualizations rather than figures.

Some Haskell DSLs exist for information visualization, most notably the Chart package[1]. These are primarily focused on statistical plotting, however, and do not support transformation comprehensively.

Outside of the functional programming community, ggplot2 also used a DSL approach for visualization [33]. Serving as an implementation of a complete, object-oriented schema for statistical plotting [34], it influences the work on visualization transformations presented here. We extend what is available in ggplot2 by allowing users to change and create transformations, thereby allowing for custom and reusable abstractions.

Stencil [10] provides a Java-based DSL for creating information visualizations which shares the use of data-bound visual parameters with our work. The choice of Haskell as a host language in our work

[1] `https://github.com/timbod7/haskell-chart/wiki`

allows us to leverage the type system more than Stencil, especially when using functors and monads.

A number of tools offer a substantially higher-level abstraction than our work. Tableau [20], for instance, allows users to attach data to visual parameters via a drag-and-drop interface. Even higher-level tools, such as Excel, provide a set of templates for one-click solutions to simple use cases. By design, these tools offer a fixed number of representations, providing less flexibility than both domain-specific languages and lower-level programming libraries.

Finally, there are tools which are neither DSLs nor highly abstracted. These are too numerous to describe here, but relevant examples include Matlab[1], Improvise [32], Prefuse [15], and Mondrian [29].

8. Conclusion

We have presented a Haskell-embedded domain-specific language for creating and transforming data visualizations. Further, by leveraging Haskell's type hierarchy, we have demonstrated that functional programming idioms and type classes are useful in practical visualization scenarios. We have also shown that using visualization transformations to modify and generate visualizations can often supplant the creation of entirely new visualizations, offering the possibility of a more iterative and incremental workflow. Finally, an evaluation based on a set of typical visualization tasks has shown that we offer ample support in meeting the requirements of a complete visualization tool.

References

[1] R. Amar and J. Stasko. Knowledge precepts for design and evaluation of information visualizations. *IEEE Trans. Visualization and Computer Graphics*, 11(4):432–442, July 2005.

[2] R. Amar, J. Eagan, and J. Stasko. Low-level components of analytic activity in information visualization. In *IEEE Symp. Information Visualization*, pages 111–117, Oct 2005.

[3] J. Bertin. *Graphics and graphic information processing*. Morgan Kaufmann Publishers Inc., 1999. English translation.

[4] J. Bertin. *Semiology of Graphics: Diagrams, Networks, Maps*. University of Wisconsin Press, 2005. English translation.

[5] R. Borgo, D. Duke, M. Wallace, and C. Runciman. Multi-cultural visualization: how functional programming can enrich visualization (and vice versa). In *Proc. Vision, Modeling, and Visualization*, pages 245–252, 2006.

[6] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE Trans. Visualization and Computer Graphics*, 15(6):1121–1128, 2009.

[7] M. Bostock, V. Ogievetsky, and J. Heer. D3; data-driven documents. *IEEE Trans. Visualization and Computer Graphics*, 17(12):2301–2309, 2011.

[8] M. Brehmer and T. Munzner. A multi-level typology of abstract visualization tasks. *IEEE Trans. Visualization and Computer Graphics*, 19(12):2376–2385, Dec 2013.

[9] W. S. Cleveland and R. McGill. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American Statistical Association*, 79(387):531–554, 1984.

[10] J. A. Cottam. *Design and implementation of a stream-based visualization language*. PhD thesis, Indiana University, 2011.

[11] D. J. Duke, R. Borgo, M. Wallace, and C. Runciman. Huge data but small programs: Visualization design via multiple embedded dsls. In *Proc. Int. Symposium on Practical Aspects of Declarative Languages*, pages 31–45, 2009.

[12] J. Fekete. The Infovis Toolkit. In *IEEE Symp. Information Visualization*, pages 167–174, 2004.

[13] T. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.

[14] J. Heer and M. Bostock. Declarative language design for interactive visualization. *IEEE Trans. Visualization and Computer Graphics*, 16 (6):1149–1156, 2010.

[15] J. Heer, S. K. Card, and J. A. Landay. Prefuse: a toolkit for interactive information visualization. In *ACM Proc. SIGCHI Conf. Human Factors in Computing Systems*, pages 421–430, 2005.

[16] T. Herndon, M. Ash, and R. Pollin. Does high public debt consistently stifle economic growth? a critique of reinhart and rogoff. *Cambridge Journal of Economics*, 2013.

[17] D. Keim, F. Mansmann, J. Schneidewind, and H. Ziegler. Challenges in visual data analysis. In *IEEE Int. Conf. Information Visualization*, pages 9–16, July 2006.

[18] H. Kienle and H. Muller. Requirements of software visualization tools: A literature survey. In *IEEE Work. Visualizing Software for Understanding and Analysis*, pages 2–9, June 2007.

[19] J. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Trans. Graphics*, 5(2):110–141, 1986.

[20] J. Mackinlay, P. Hanrahan, and C. Stolte. Show me: Automatic presentation for visual analysis. *IEEE Trans. Visualization and Computer Graphics*, 13(6):1137–1144, 2007.

[21] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. Big data: The next frontier for innovation, competition, and productivity. *McKinsey Global Institute*, May 2011.

[22] K. Matlage and A. Gill. ChalkBoard: Mapping functions to polygons. In *Proc. Symp. Implementation and Application of Functional Languages*, volume 6041 of *LNCS*. Springer-Verlag, Sep 2009.

[23] R. Metoyer, B. Lee, N. Henry Riche, and M. Czerwinski. Understanding the verbal language and structure of end-user descriptions of data visualizations. In *ACM Proc. SIGCHI Conf. Human Factors in Computing Systems*, pages 1659–1662, 2012.

[24] C. Plaisant. The challenge of information visualization evaluation. In *ACM Proc. Working Conference on Advanced Visual Interfaces*, pages 109–116, 2004.

[25] K. Rogoff and C. Reinhart. Growth in a time of debt. *American Economic Review*, 100(2):573–578, 2010.

[26] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *IEEE Symp. Visual Languages*, pages 336–343, 1996.

[27] B. Shneiderman and C. Plaisant. Strategies for evaluating information visualization tools: multi-dimensional in-depth long-term case studies. In *ACM Proc. AVI Workshop on Beyond Time and Errors: Novel Evaluation Methods for Information Visualization*, pages 1–7, 2006.

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

[29] M. Theus. Interactive data visualization using Mondrian. *Journal of Statistical Software*, 7(11):1–9, 2003.

[30] J. van Wijk. The value of visualization. In *IEEE Visualization*, pages 79–86, Oct 2005.

[31] F. B. Viegas, M. Wattenberg, F. Van Ham, J. Kriss, and M. McKeon. Manyeyes: A site for visualization at internet scale. *IEEE Trans. Visualization and Computer Graphics*, 13(6):1121–1128, 2007.

[32] C. Weaver. Building highly-coordinated visualizations in improvise. In *IEEE Symp. Information Visualization*, pages 159–166, 2004.

[33] H. Wickham. *ggplot2: Elegant graphics for data analysis*. Springer-Verlag, 2nd edition, 2009.

[34] L. Wilkinson. *The Grammar of Graphics*. Springer-Verlag, 2nd edition, 2005.

[35] B. A. Yorgey. Monoids: theme and variations (functional pearl). In *ACM Proc. Haskell symposium*, pages 105–116, 2012.

[1] `www.mathworks.com.au/products/matlab/`