A Domain-Specific Language for Exploratory Data Visualization

Karl Smeltzer
Oregon State University
karl@lifetime.oregonstate.edu

Martin Erwig
Oregon State University
erwig@oregonstate.edu

Abstract

With an ever-growing amount of collected data, the importance of visualization as an analysis component is growing in concert. The creation of good visualizations often doesn't happen in one step but is rather an iterative and exploratory process. However, this process is currently not well supported in most of the available visualization tools and systems. Visualization authors are forced to commit prematurely to particular design aspects of their creations, and when exploring potential variant visualizations, they are forced to adopt ad hoc techniques such as copying code snippets or keeping a collection of separate files.

We propose variational visualizations as a model supporting open-ended exploration of the design space of information visualization. Together with that model, we present a prototype implementation in the form of a domain-specific language embedded in Purescript.

CCS Concepts • **Software and its engineering → Domain specific languages**; • **Human-centered computing** → *Visualization theory, concepts and paradigms*;

Keywords Data Visualization, Exploratory Programming

ACM Reference Format:
Karl Smeltzer and Martin Erwig. 2018. A Domain-Specific Language for Exploratory Data Visualization. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3278122.3278138

1 Introduction

Visualization has emerged as as a core component of modern data analysis. Data collection continues to grow exponentially [13], and with it the number of tools and systems for creating data-driven graphics and visualizations has increased accordingly. Many of these tools only support the creation of individual visualization artifacts with little attention paid to the exploratory nature of visual data analysis [10, 18].

The ggplot2[1] system is one exception. In part because it is embedded in the R language[2] and can interact with the corresponding ecosystem of statistical software packages, ggplot2 allows users to define their own workflow. By defining functions that generate visualizations, one can parameterize visualizations in order to easily apply them to different data or to explore different features and aesthetic options.

However, even this parameterization still fails to directly support one major component of exploratory analysis by forcing users to make decisions about their visualizations before they are ready to do so. Oftentimes, an analyst may not wish to fully commit to a particular arrangement of visualizations or particular aesthetic properties until after having seen some preliminary results.

For example, suppose we are analyzing a set of data for sales across three regions and each of the regions is comprised of a number of sub-regions. Our goal is to analyze which portions of our sales are coming from which regions in order to make some marketing decisions. However, we are not yet sure, first of all, whether we are interested in the extra details contained in the sub-region data. Moreover, there are a handful of questions regarding spacing and coloring of the visualization marks that we have yet to decide upon.

While an R programmer and ggplot2 user would find a way to parameterize the color and space information, there is no obvious way to support data that takes one of several different shapes. For instance, if we want to show subregion information for one particular region, but not for the rest, and adjust the colors accordingly, doing so requires committing to some details in the code generating the visualization.

Ideally, we could delay making those decisions about which regions to show detailed information for, how to color them, and how to lay them out, until we have seen some preliminary versions. Seeing such a chart could quickly tell us how cluttered the visualization would be with all the details shown or whether subregions with sequential colors are easier to visually parse than regions with divergent colors.

For example, given a function `piechart` which takes some data as input and generates a pie chart, we could generate a

GPCE '18, November 5–6, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6045-6/18/11…$15.00
https://doi.org/10.1145/3278122.3278138

[1] ggplot2.tidyverse.org
[2] www.r-project.org

version with no details shown and another with all the details shown by simply applying it. A version with all details omitted is shown in Figure 1a, and another with all details shown is depicted in Figure 1b. Ignoring color and considering only the level of detail for each of the regions, we might need to create as many as $2^3 = 8$ different charts. It is not difficult to see how this quickly becomes unreasonable.

To support such an exploratory workflow in which design decisions can be delayed, we propose *variational visualizations*, which are structures that encode arbitrarily many traditional visualizations in a systematic way. Variational visualizations empower visualization authors and don't force them to commit prematurely to particular design decisions. Variational visualizations are not limited to producing simply side-by-side comparisons of visualizations; due to their tight integration into a general visualization DSL, they facilitate versatile combinations of visualization variants such as overlays of alternative bar charts, as shown in Figure 1c.

The primary contribution of this work is a model of variational visualizations and its implementation as a domain-specific language (DSL) that allows the creation, manipulation, navigation, and rendering of variational visualizations.

In Section 2 we describe the basic design of the DSL for building programmable visualizations. In Section 3 we discuss a systematic way of representing variation that is the basis for Section 4 where we introduce variability into the DSL. In Section 5 we demonstrate how to add variability to visualizations and present several corresponding use cases. In addition to the creation of variability we also sometimes need to transform and aggregate variability. These two aspects are discussed in Sections 6 and 7, respectively. Finally, we discuss related work in Section 8 and present some conclusions in Section 9.

2 Programming Visualizations

Our DSL (https://github.com/karljs/vis) is based on the work presented in Smeltzer et al. [14], which introduced an embedded DSL in Haskell. In contrast, our DSL is embedded in Purescript[3], which is similar to Haskell, but has the additional feature of making it easy to produce browser-based graphics. Readers familiar with Haskell should have no problem understanding the definitions contained in this work.

The DSL groups functionality in to different layers. At the bottom-most layer, users can work directly with the core data types to customize every detail and make intricate extensions. Higher layers of the DSL provide increasingly abstracted combinator functions for cases when the details are less important than quickly building charts. At the top of this layering are functions for building simple default charts such as bar and pie charts.

```
barchart :: Array Number -> Vis
piechart :: Array Number -> Vis
```

[3] www.purescript.org

Functions like these generally take an array[4] of numbers as input and produce a visualization, a value of type `Vis`.

The `Vis` type is the core of all visualizations and serves two main purposes. First, it allows the composition of visualizations into larger visualizations. This could be spatial composition or something more sophisticated such as stacking chart elements. Second, it allows visualizations to be transformed between Cartesian and polar coordinate systems. The `Vis` type is defined as follows.

```
data Vis = Mark    MPs
         | NextTo  (NonEmptyList Vis)
         | Above   (NonEmptyList Vis)
         | Overlay (NonEmptyList Vis)
         | Stacked (NonEmptyList Vis)
         | Cartesian (Vis a)
         | Polar     (Vis a)


type MPs = { vps :: VPs
           , label :: Maybe Label
           , ... }
```

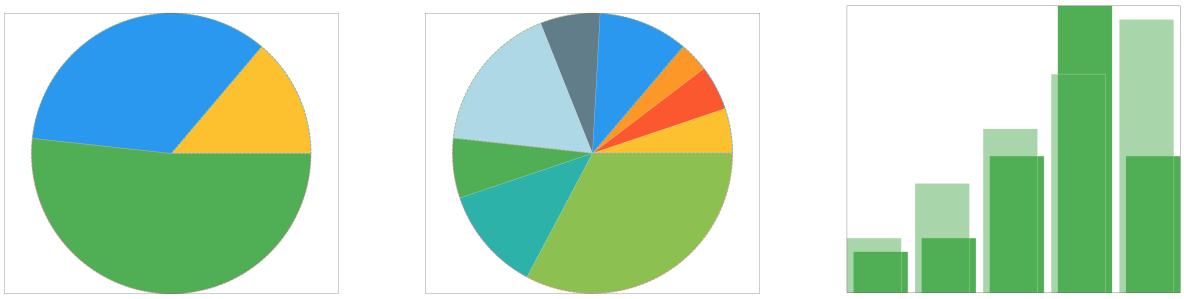
The general structure of a visualization is a tree of composition and transformation operators with `Mark` nodes as leaves. Every `Mark` node stores a set of *mark parameters*, represented as a record of type `MPs`, which include a set of *visual parameters*, represented as a record of type `VPs`. These visual parameters are based on Bertin's visual variables [2] and encode aspects of a visualization that can be bound to data such as width, height, and color.

```
type VPs = { height :: Number
           , width  :: Number
           , color  :: Color
           , ... }
```

Returning to the other `Vis` constructors, in a Cartesian coordinate system `NextTo` and `Above` spatially compose lists of charts horizontally or vertically, respectively. In contrast, in a polar coordinate system they respectively arrange charts in a circle or from the outside in. `Stacked` merges visualizations to produce charts such as stacked bar charts, and `Overlay` renders multiple charts into the same region of space. Finally, `Cartesian` and `Polar` convert the space a visualization will fill between coordinate systems. The constructors are best understood by looking at examples.

We will start by showing how to construct a simple bar chart. Ultimately, what we need is a `NextTo` composition of `Mark` elements. Each `Mark` should have its `height` parameter bound to a data value. We could just start constructing these values as desired, but there are a number of helper functions which can make it much easier. The first such helper functions are:

[4] The choice of arrays is strictly because Purescript has special syntax using square brackets `[]` for arrays, much like Haskell does for lists. We would prefer to use non-empty lists as we do elsewhere, but the syntactic overhead is significant.



(a) (b) (c)

Figure 1. Two pie charts and an overlay of two variants of a bar chart. The pie charts show the same data set: Chart (a) shows the totals for three regions but omits all details while chart (b) shows the details for each region. (c) The overlaid bar charts illustrate how different visualization variants can be integrated into single visualizations.

```
markHeight :: Array Number -> NonEmptyList Vis
markWidth  :: Array Number -> NonEmptyList Vis
```

These functions generate `Mark` values with data bound to the height and width respectively (as well as a constant value for the unbound dimension, a default color, and a default label). The implementation is uninteresting and is dominated by boilerplate for converting from arrays of values to non-empty lists. Again, this conversion is only used to improve the interactive interface. The list result type makes the definition of `barchart` very easy. Note that `(<<<)` is standard function composition.[5]

```
barchart :: Array Number -> Vis
barchart = NextTo <<< markHeight
```

The `piechart` function also constructs a composition of marks. However, we use the `Polar` constructor to ensure the chart is interpreted in a polar coordinate system.

```
piechart :: Array Number -> Vis
piechart = Polar <<< NextTo <<< markWidth
```

Using functions such as `barchart` and `piechart` does not prevent us from customizing the appearance of the rendered result. Visualizations can be transformed after they are defined by applying functions to them. Consider the following sequence, beginning with a simple, colored pie chart. Note that we use the > character to suggest an interaction with the REPL, that is, code that users of the DSL would write. This is to distinguish the code from the implementation of the DSL itself that users are not concerned with.

```
> let myPie = piechart myData `colorAll` green
```

We create a pie chart and then color it green. The result is shown in Figure 2a. The `colorAll` function works by traversing the tree-shaped visualization and changing the color anytime it reaches a `Mark` constructor. In Haskell this would be a candidate for an approach such as Scrap Your Boilerplate [11]. Unfortunately, Purescript does not have an analogous library yet, and so we have to define some generic traversal functions by hand, namely `updMP`, which applies an update on mark parameters to all marks in a visualization, and `updVP`, which uses `updMP` to apply an update on visual parameters to all marks in a visualization.

```
updMP :: (MPs -> MPs) -> Vis -> Vis
updMP f (Mark mps)    = Mark (f mps)
updMP f (Cartesian v) = Cartesian $ updMP f v
updMP f (Polar v)     = Polar     $ updMP f v
updMP f (... vs)      = ... $ map (updMP f) vs

updVP :: (VPs -> VPs) -> Vis -> Vis
updVP f = updMP (\mps->mps { vps = f mps.vps} )
```

Now we can implement the `colorAll` function relatively concisely in the following way.

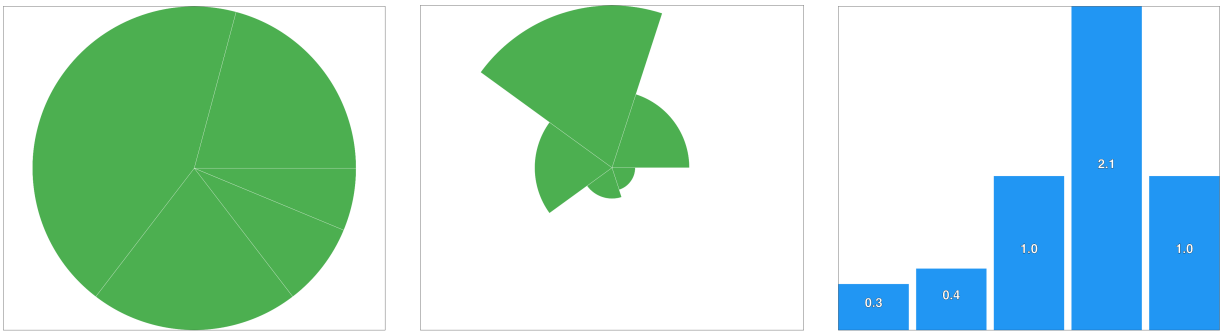
```
setColor :: Color -> Vis -> Vis
setColor c = updVP (\vps->vps { color = c })

colorAll :: Vis -> Color -> Vis
colorAll = flip setColor
```

Although coloring is conceptually straightforward, it is implemented here as a visualization transformation. By default, charts are colored black. In the preceding example we created a black pie chart, which was not shown, and we then transformed it into a green pie chart. This ability to transform visualizations is one of the main advantages of this DSL. This approach is not limited to simple visual tweaks either. To demonstrate a more interesting use, we can convert our pie chart into a bar chart without having to recreate any of its components.

Doing this requires two steps. We need to swap the orientation of the pie wedges, so that the data is bound to the radius (height) rather than the angle (width) because we want vertically oriented bars. This is because we want to reuse the `NextTo` constructor to create a horizontal composition of bars. We could also convert the `NextTo` constructor to an `Above` constructor, which is not demonstrated here. The second step is to ensure that the visualization is interpreted in a Cartesian coordinate system rather than a polar one.

[5] In Purescript `(<<<)` is used to avoid ambiguity, since the dot character `(.)` is used for record access.



(a) (b) (c)

Figure 2. A series of visualizations produced by transformation functions.

To swap the width and height values of our bars we can reuse the `updMP` function as follows.

```
swapWH :: VPs -> VPs
swapWH vps = vps { width = vps.height
                 , height = vps.width }
```

Until now we avoided introducing frames, which are structures that track contextual information about visualizations. When rendering, say, a bar in a bar chart, the size of the rectangle to draw will depend on the value of the data driving the visualization. However, a bar of height 3.0 will not always be rendered at the same height. Instead, it depends on the scale of the chart. If 3.0 is the highest data value being charted, then the bar will be tall, but if it is the lowest value then the bar may be quite short. Frames track the minimum and maximum values in a chart in order to know how to render a particular mark. The `MPs` type contains one frame for the width and one for the height which were hidden before.

```
type Frame = { frameMin :: Number
             , frameMax :: Number }

type MPs = { ...
           , frameH :: Frame
           , frameW :: Frame }
```

Since we are swapping the widths and heights, we also need to swap the frames. If we fail to also swap the frames when swapping the width and height, then the rendered output will be scaled incorrectly. We therefore arrive at the following definition for the function `reorient`.

```
reorient :: Vis -> Vis
reorient =
   updMP (\mps->mps { frameW = mps.frameH
                    , frameH = mps.frameW
                    , vps = swapWH mps.vps })
```

It is possible to delay computing these frames until the actual rendering, but making them accessible could be useful in that it allows the user to customize how charts are scaled. In most circumstances frames can be managed automatically and do not need to be exposed.

Referring back to the `myPie` value from the previous example, we can do the following.

```
> let myRose = reorient myPie
```

This produces a rose chart (also called a Coxcomb chart), as shown in Figure 2b. Now we have just one step remaining to produce our desired bar chart. This step is much simpler than the last because support for switching between coordinate systems is available directly via the `Cartesian` and `Polar` constructors. We just need to apply `Cartesian` here to produce a bar chart. When one of the coordinate system constructors (`Cartesian` or `Polar`) is applied directly to another, the outer one takes precedence. We will also apply some labels and insert whitespace to tweak the final output.

```
> let myBars = Cartesian myRose `space` 0.1
                                `label` myLabels
```

The purpose of the `space` function is to insert whitespace between the children of a visualization. For instance, when given a `NextTo` composition, it will create a whitespace element and intersperse it into the list of visualizations contained in the `NextTo`. The amount of space, or in this case the width of the whitespace element, is specified relative to the width of the children. Our bars have constant width 1.0 (inherited from the radius of the original pie chart) and so specifying the value 0.1 means the whitespace inserted will be 10% of the width of the bars.

```
space :: Vis -> Number -> Vis
space (NextTo vs) n =
       NextTo $ intersperse (hSpace n) vs
space (Above vs)  n =
       Above  $ intersperse (vSpace n) vs
space ...


hSpace v = spaceMark v 1.0 ...
vSpace v = spaceMark 1.0 v ...

spaceMark w h = Mark { vps: { height: h
                            , width: w, ...}, ... }
```

Finally, we use the `label` function to attach labels to each of the bars. It works by zipping together a list of visualizations with a list of labels. The `label` function itself deals with converting arrays (unsafely) to nonempty lists as explained earlier and is just a wrapper around `labelList`, which performs the real labeling and which works with lists.

```
labelList :: Vis -> NonEmptyList Label -> Vis
labelList (Mark mps) ls =
        Mark (mps { label = Just (head ls) })
labelList (NextTo vs) ls =
        NextTo $ zipWith labelAll vs ls
labelList (Above vs) ls =
        Above $ zipWith labelAll vs ls
labelList ...
```

The `labelAll` function referenced here just assigns a single label to all parts of a visualization which, in this case, will just be the bars.

3 Representing Variation

Before we can integrate variability into visualizations, we need a systematic way of structuring variation. To that end we make use of the choice calculus [7], which is a formal model for variation based on named *choices*. Each choice consists of a list of *alternatives*, and its name, called a *dimension*, synchronizes the selection of a specific alternative with the selection in other choices of the same name.

For example, the choice between the two numbers 3 and 4 can be represented as a variational integer in dimension A with two variants as $A\langle 3, 4\rangle$. Choices can appear as parts of expressions as in $2 + A\langle 3, 4\rangle$, and expressions can contain multiple choices in the same or different dimensions as in $A\langle 1, 2\rangle + A\langle 3, 4\rangle$ or $B\langle 1, 2\rangle + A\langle 3, 4\rangle$. The difference between these two expressions is that $A\langle 1, 2\rangle + A\langle 3, 4\rangle$ encodes only the two expressions $1 + 3$ and $2 + 4$ because the choices are synchronized through the name A, whereas $B\langle 1, 2\rangle + A\langle 3, 4\rangle$ represents four expressions $1+3$, $2+3$, $1+4$, and $2+4$ because selections in the two choices are independent of one another, since they occur in different dimensions. Choices can also be nested as in $A\langle B\langle 2, 3\rangle, 4\rangle$, and nested choice structures in particular are subject to a number of laws and transformation rules, but these are not of importance for the present paper.

Since we are building up to a Purescript DSL, we formalize the choice calculus as a parameterized data type. We restrict the type to binary choices for simplicity, which is not an essential constraint, since any n-ary choice can be represented by $n - 1$ (nested) binary choices.

```
type Dim = String

data V a = Chc Dim (V a) (V a)
         | One a
```

Each alternative comes with a corresponding *selector* which indicates that particular variant. Sets of selectors are called *decisions*, which we represent as mappings from dimensions to `L` or `R`.[6]

```
data Dir = L | R

type Selector = (Dim,Dir)

type Dec = Map Dim Dir
```

Selectors and decisions are used for eliminating variation and extracting plain values from choice expressions. This process is called *selection* and is defined as follows.

```
select :: Selector -> V a -> V a
select (L,d') (Chc d l r) | d==d' = select (L,d) l
select (R,d') (Chc d l r) | d==d' = select (R,d) r
select s      (Chc d l r) = Chc d (select s l)
                                  (select s r)
select _      (One x)     = One x
```

Repeated selection through a decision is supported by the `selectDec` function, which can completely eliminate variation more directly.

4 Integrating Variability into Visualizations

To work effectively with variational visualizations, we have to address two questions. First, how should we represent variability in visualizations? We discuss several different options together with their advantages and disadvantages in Section 4.1. Second, how can we a navigate variational visualization and select variants from it? We discuss this aspect in Section 4.2.

4.1 Representing Variational Visualizations

Choices can be integrated into visualizations in several different ways. Most straightforwardly, we can just apply the `V` type constructor to the existing `Vis` type.

```
type VVis = V Vis
```

This approach is general and allows us to define arbitrary variational visualizations by simply placing visualization alternatives in choices. For example, given visualizations `v1` and `v2`, we can write `Chc "A" (One v1) (One v2)` for a choice in dimension A.

Unfortunately, this method is somewhat crude. If we want to encode variant visualizations which are mostly similar, for example, differing only in a single bar or in color, we still need to copy the entire structure into each leaf of the tree of choices. Moreover, it could be valuable to be able to identify visually which portions of a visualization actually vary and which are constant. With this approach, identifying the constant parts in our visualization would, at best, require some kind of tree difference calculation. In the worst cases the proper alignment among visualizations might be completely ambiguous.

[6] We grant ourselves some syntactic liberty and use the Haskell tuple syntax `(a,b)` instead of Purescript's `Tuple a b`, both for values and types.

Alternatively, we could push the `V` type constructor down into the leaves of the tree. This would require us to modify the `Mark` constructor for the `Vis` type in the following way.

```
data VVis = Mark (V MPs)
          | ...
```

This change solves both of the problem mentioned for the previous approach. Everything is now shared, since only the leaves of a visualization may vary and we can always tell exactly which parts are constant and which vary based on whether or not there are any choices in a leaf. However, this benefit comes at an even larger cost because we are now unable to construct certain kinds of visualizations. For example, we can no longer construct a variational visualization where one variant is a vertical bar chart (`NextTo` composition of vertical bars) and the other is a horizontal bar chart (`Above` composition of horizontal bars). Since an important use case is the creation of comparable visualizations, this limitation is unacceptable, and rules out this approach.

Finally, a third approach is to let the DSL user decide where in the structure of their visualizations the variation should occur. This can be achieved by adding a new constructor to the original `Vis` type.

```
data VVis = Mark MarkParam
          | Chc Dim VVis VVis
          | ...
```

This hybrid approach removes the restriction on the kinds of variational visualizations we can construct by allowing the variation to occur anywhere. It puts the burden of this decision on the user, which has the benefit that a savvy user can design visualizations so that the constant parts and variational parts are clearly demarcated. However, it requires the user to understand the structure of visualizations more deeply than the other approaches before being able to use the DSL effectively. We believe this to be a reasonable tradeoff and, going forward, this third approach is the one we adopt.

4.2 Prototype Visualization Renderer

We have developed a prototype tool to render variational visualizations and navigate among their variants on top of the Purescript DSL. It is based on HTML5 canvas graphics and was used to generate all the figures in this paper.

The main design decision for the prototype is how to support a user in navigating among the different visualization variants. Much of the utility of variational visualizations is lost if the user is not able to make and modify selections. Our prototype interface uses a two-part interface consisting of checkboxes and radio buttons. Each dimension that occurs in a visualization produces a checkbox, which indicates whether a selection for this dimension has been made. If checked, two radio buttons are shown, which denote which alternative in the corresponding dimension is selected.

This checkbox/ratio button interface provides an interactive means to construct decisions with which to select variants from a variational visualization. If a selection is made in all dimensions, all variability is eliminated, and a single variant visualization is shown as a result. In general, however, selections are partial, and one or more dimension remain unselected. In this case, all possible variants that have not been eliminated through selections are displayed. We have adopted a small multiples approach to rendering multiple variants [17], which basically means to composing the variant visualizations into a grid.

An example is shown in Figure 3, which shows two dimensions corresponding to some variational data associated with months of the year. The small format makes it somewhat difficult to see, but the two dimensions are *February* and *June*. The selection in the *June* dimension is {*June.r*}. Since *February* is unselected, no decision for that dimension is included, and therefore both *February* visualization variants are displayed.

The figure also shows how color is used to map the user control to parts of the visualization. The *June* dimension has been assigned a green color (automatically), and the corresponding region in the visualization is surrounded by a green, dotted outline. This means that if we change the selection in that dimension, we should expect that part of the visualization to change. This variation region indicator appears in both of the shown variants.

Using colors and small multiples in this way faces a number of issues, particularly with respect to scalability. However, this interface is only intended to be a prototype and not to be a perfect solution to this problem.

In the following sections we explore the different ways a programmer can construct and manipulate variational visualizations, and generate corresponding rendered output using the prototype. There are three major categories of tasks a user of our DSL can engage in, namely generating variation, transforming variation, and consuming variation. The following sections will focus on these three aspects in order.

5 Generating Variation

Creating variation in a visualization is the most basic step in working with variational visualizations. The produced visualizations can then be either explored using the browser prototype briefly discussed in Section 4.2 or subjected to further transformations.

One frequent use case for generating variational visualizations is the manual introduction of variation. For example, to look at and compare several alternative visualizations, we can place them in choices and decide on the one we want by simply loading them in the variational visualization browser. We may also encounter situations in which we want to vary part of a bigger visualization. We can achieve this with our representations easily by creating a choice of the parts and

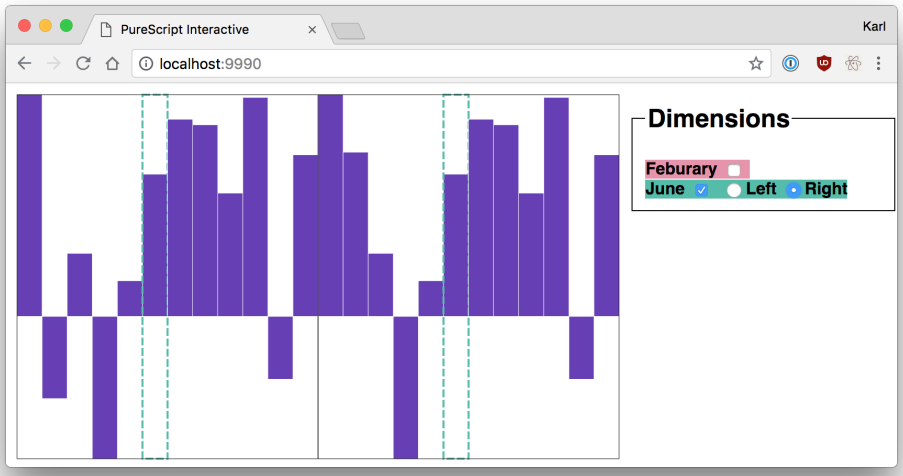


Figure 3. The prototype user interface showing a variational visualization to which the decision {*June.r*} has been applied.

then combining it with the remaining parts of the visualization using the other `VVis` constructors such as `NextTo`, `Above`, etc.

In addition to constructing variation manually, there are at least two common scenarios in which variation is added programmatically to a visualization.

5.1 Visualization Provenance

One simple and useful way of systematically adding variation to an existing visualization is offered by the function `vary`, which is defined as follows.

```
vary :: (Dim,VVis -> VVis) -> VVis -> VVis
vary (d,f) v = Chc d v (f v)
```

This function is reminiscent of function application, with good reason. Its purpose is to apply a visualization transformation function, `f`, but it does so in a way that preserves the original visualization as a variant along with the newly transformed result. Both visualizations are enclosed in a choice with dimension `d`.

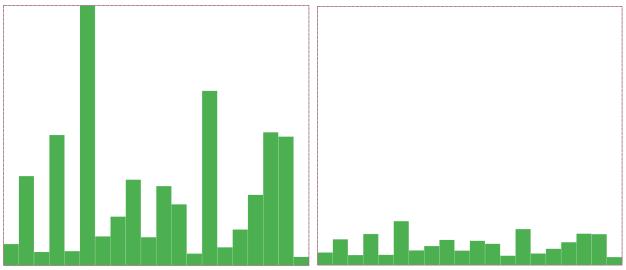
Since we often vary visualizations by changing visual parameters, we also provide as a special instance of `vary` the function `varyVP`.

```
varyVP :: (Dim,VPs -> VPs) -> VVis -> VVis
varyVP (d,f) = vary (d,updVP f)
```

To illustrate the use of `vary`, suppose we have created a simple bar chart `myBars`.

```
> let myBars = barchart myData
```

After viewing it, we decide that it may be valuable to see the same data after applying a square-root transformation. We could of course transform the data and produce a new chart, but that causes us to lose any customizations we may have applied. Another possibility is to define a visualization



(a) (b)

Figure 4. Generating variation. By using the `vary` function we can transform a visualization and keep the original as an alternative. By chaining applications, it is possible to track the provenance of a visualization completely, ensuring every variant is always reachable.

transformation that actually changes the heights of the bars in the existing chart. We achieve this by first defining a higher-order function that manipulates the heights of bars.

```
onHeight :: (Number -> Number) -> VPs -> VPs
onHeight f vps = vps { height = f vps.height })
```

We can then use `varyVP` to apply `onHeight` across the visualization.

```
> let vBars = varyVP ("Sqrt",onHeight sqrt) myBars
```

The two variants are shown in Figure 4. Note that the square-root transformed visualization is *not* automatically rescaled to fill the frame. If the user prefers rescaling, this can be easily achieved by applying the function:

```
fillFrame :: VVis -> VVis
```

By chaining applications of `vary` (and `varyVP`) it is possible to build increasingly large choice structures that track all the

variants of a visualization that are ever produced, serving as a visualization provenance technique. The programmer can then use selection to navigate through the visualization in order to revisit older states.

5.2 Branching Visualizations

To truly support visualization history and provenance, our DSL needs to be able to do more than just create a linear sequence of changes. In particular, we also need to support branching transformations, that is, we want to support a tree of history information rather than a list. To support this, the DSL must provide a way to indicate which previous variant should serve as the starting point of a new transformation. When working with the DSL in a REPL environment, it is likely that visualizations will be created (such as with `vary`) that are not bound to a particular name, meaning the user does not have a reference to them. Fortunately, we can make use of the existing choice calculus machinery to identify visualizations.

The `branch` function serves this purpose and is analogous to `vary`, but with an additional parameter in the form of a decision. This decision is used to indicate where in the choice tree the transformation should be applied and the new variant should be inserted.

```
branch :: (Dim,VVis -> VVis) -> Dec -> VVis -> VVis
```

However, the implementation of this function is not quite as simple as `vary`. Since the variational type constructor is integrated recursively in the visualizations type, it is not obvious when traversing the visualization whether we have made all the relevant selections or whether some still remain nested further down.

For this reason, we need to check explicitly. The helper function `lacksDims` performs this task, searching recursively for any of the dimensions selected in a particular decision. We will show an example after the complete definition.

```
lacksDims :: Dec -> VVis -> Boolean
lacksDims dec (Chc d l r) = case lookupDim d dec of
  Nothing -> lacksDims dec l && lacksDims dec r
  _       -> false
lacksDims _ (Mark _) = true
lacksDims dec ...
```

The cases which are not shown just employ boilerplate code to recursively check their child visualizations. With this completed, we can return to the definition of `branch`.

```
branch :: (Dim,VVis -> VVis) -> Dec -> VVis -> VVis
branch (newDim,f) dec (Chc d l r) =
  case lookupDim d dec of
    Just L  -> Chc d (branch (newDim,f) dec l) r
    Just R  -> Chc d l (branch (newDim,f) dec r)
    Nothing -> if lacksDims dec (Chc dim l r)
               then vary (newDim,f) (Chc dim l r)
               else Chc d (branch (newDim,f) dec l)
                          (branch (newDim,f) dec r)
branch (newDim,f) dec ... = if lacksDims dec ...
```

Again, the omitted cases are repetitive boilerplate, and essentially the same as the `Nothing` branch of the above case statement, checking whether the traversal is finished and dispatching accordingly. Consider the following example to illustrate how this process work.

Suppose we have constructed a visualization `v1`. We applied `vary` to it, adding a visualization `v1` within a new dimension A, which produces the following variational visualization.

```
Chc A v1 v2
```

Next we apply `vary` a second time to add a visualization `v3` within a new dimension B, which leads to:

```
Chc B (Chc A v1 v2) v3
```

Now we want to branch off the original `v1`. We pass the decision $\{A.l, B.l\}$ to `branch` to indicate this.

The branch function first encounters the outer B choice, performs a lookup, and sees the left alternative is selected. Accordingly, we traverse only the left branch. We then encounter the A choice, and traverse the left branch again. Next, suppose we encounter a `NextTo` constructor. This will trigger an application of `lacksDims`. That function sees that neither of the dimensions in the target decision occurs anywhere in the remainder of the visualization. This means we are finished and apply our transformation function. If either A or B had occurred, we would have needed to continue traversing before performing the branching transformation.

As we did for `vary`, we also define a version of the `branch` function for branching using a transformation of visual parameters.

```
branchVP :: (Dim,VPs -> VPs) -> Dec -> VVis -> VVis
branchVP (d,f) = branch (d,updVP f)
```

With this function we can produce branching visualizations, which effectively supports a visualization history mechanism. To illustrate, consider this simple example. Recall the `vBars` visualization from the previous example which encodes a bar chart as one variant and its square-root transformed version as the other. The choice structure of that visualization is the following (we use the name `sqrtBars` to refer to the result of the transformation `onHeight sqrt`).

```
vBars = Sqrt⟨myBars, sqrtBars⟩
```

Now suppose we decide that a square root transformation may have been a suboptimal choice. We want to create a new variant that transforms the original bars using a reciprocal transformation, without losing access to any existing variant. We can achieve it in the following way. First, we define the decision that corresponds to the variant we want (in this case the decision consisting of the only selector `("Sqrt",L)` to locate original bar chart `myBars`). Then we apply the transformation function to perform the reciprocal computation using Purescript's succinct notation for partially applied binary functions `1.0/_`) with `branchVP`.

```
> let orig = sel2dec ("Sqrt",L)

> let vvBars = branchVP ("Recip",1.0/_) orig vBars
```

This definition produces a new visualization with the following updated variation structure where `recipBars` denotes to the newly added visualization.

`vvBars = Sqrt⟨Recip⟨myBars, recipBars⟩, sqrtBars⟩`

Using a combination of `vary` and `branch` allows us to flexibly generate visualization variants through transformation.

6 Transforming and Maintaining Variation

The previous section introduced some ways in which we can systematically introduce variation into visualizations. Recall, however, that generation is only one of three general categories of tasks we want to support. The second category is for those tasks which transform variation without generating or removing it. That is, tasks which maintain the amount of variation.

6.1 Mutating and Replacing Variants

One straightforward task that does not affect the structure of the variation is to replace particular variants. In Section 5 we showed how to add variants using the `vary` and `branch` functions. Replacing a visualization is similar and equally useful. Suppose, for example, that we have produced a variational visualization but none of the variants include labels. Now we have changed our mind and would like to see all of the visualizations with labels.

We could use the `branch` function to transform each variant, adding a new dimension in which the right alternatives now contain the versions with labels. However, since adding global dimensions grows the number of variants exponentially, we may wish to avoid doing so. In cases when we do not care to preserve the original variants, we can instead turn to `mutate`, which is very similar to `branch`. Since `mutate` replaces variants of already existing choices, it doesn't need a dimension as an argument. In the definition we make good use of the `lacksDims` helper function from earlier.

```
mutate :: (VVis -> VVis) -> Dec -> VVis -> VVis
mutate f dec (Chc d l r) =
  case lookupDim d dec of
    Just L  -> Chc d (mutate f dec l) r
    Just R  -> Chc d l (mutate f dec r)
    Nothing -> if lacksDims dec (Chc d l r)
               then f (Chc d l r)
               else Chc d (mutate f dec l)
                          (mutate f dec r)
mutate f _ (Mark mps) = f (Mark mps)
mutate ...
```

Since in our example the goal is to mutate the visualization variants to add labels, and we want to do this to every variant rather than a subset of them, we can use the empty decision as an argument. This will ensure that no sub-tree is selected and so all variants are transformed. Using the empty decision in this way is, of course, no different than labeling the visualization directly with `label`, but it shows that our standard transformation functions are actually a special case of `mutate`.

```
> let myVis = ...

> let myLabVis = mutate (`label` ["A", ..., "G"])
                        emptyDec myVis
```

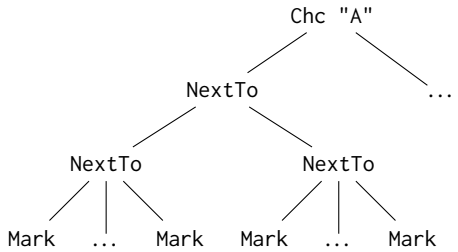
One simple but useful extension to `mutate` is the `replace` function. Rather than mutating an existing variant, `replace` simply overwrites it. We can easily define `replace` in terms of `mutate`.

```
replace :: VVis -> Dec -> VVis -> VVis
replace v = mutate (\_->v)
```

6.2 Conditional Mutation

One further kind of task we wish to support that does not add or remove variation is what we call *conditional mutation*. The idea is to allow predicates to be defined over visualizations and mutate or replace only those portions and variants which are matched by the predicate. As an example, suppose that we have generated a variational visualization in which each variant contains a number of composed bar charts of varying sizes and kinds. Several of the charts depict quarterly revenue, and we wish to transform only those bar charts into pie charts in order to better see the ratio of the individual quarters to the whole year. There are a number of ways to achieve this, but for this example we choose to define a predicate that matches those charts by counting the number of bars. Only the revenue charts have 4 data points, so it is an easy way to distinguish them. We could also write a predicate to check the color, labels, or other aspects of the structure as well.

Once again we can also make use of decisions to indicate where in the visualization we would like our predicate to be tested. However, there is one complication compared to `branch` and `mutate`. In addition to the check that `lacksDims` performs to see whether we have traversed far enough into the visualization structure, we also need to determine when to apply our predicate. Unfortunately, there is not an obviously correct solution to this. For example, suppose we have a visualization with the following structure.



Suppose further that we have used a decision to indicate we want the left branch of all *A* choices. At the root node we encounter an *A* choice and accordingly traverse the left

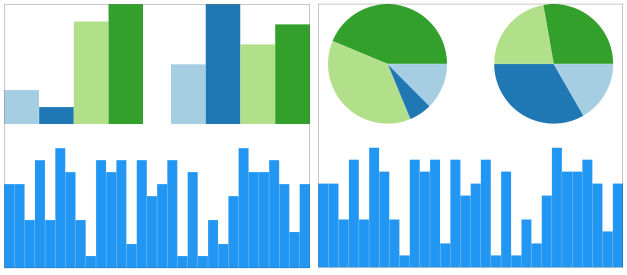
alternative, leaving the right alternative unchanged. Next we encounter a `NextTo` node. If we simply used `lacksDims`, we would discover that there are no more *A* choices remaining, indicating that the traversal is potentially finished. We could now test our predicate and, if appropriate, perform the mutation. However, it is not obvious that this is the correct approach. In this particular case, what remains is a horizontal composition of bar charts. This takes the form of nested `NextTo` constructors. If the user wants to transform these into pie charts, it would require a function which correctly handles a horizontal composition of exactly two charts, traversing the outer `NextTo` and then changing the inner ones. This is sometimes not possible to learn by visually inspecting the rendered output. For instance, by inspecting the output we cannot know if there is, perhaps, a `Cartesian` constructor layered between the `NextTo` constructors.

In fact, this would all but prevent the implementation of our desired check for four bars. We would need to actually be checking for a composition of two visualizations which each contain four bars, and moreover we would need to write a transformation function that deals with this multi-tiered composition.

Instead, we take the approach of not applying the predicate at this point, immediately after we exhaust the relevant choices. Instead, we continue traversing the visualization until we reach what we (heuristically) determine is a single, complete visualization. This results in the predicate being applied separately at each of the lower `NextTo` constructors. This work is done in the `isVis` function.

```
isMark VVis -> Boolean
isMark (Mark _) = true
isMark _ = false

isVis :: VVis -> Boolean
isVis (NextTo vs) = any isMark vs
isVis (Above vs)  = any isMark vs
isVis (Chc _ l r) = isVis l && isVis r
isVis ...
```



(a) (b)

Figure 5. Conditionally mutating parts of a visualization. In this case we globally mutate all charts containing exactly four bars into pie charts to better showcase their relationship to the whole.

This function makes use of some simple heuristics about what constitutes a complete visualization. Unfortunately, it is always possible to construct a complicated visualization that it fails to detect correctly. We believe this is a reasonable tradeoff, pending some practical evaluation of both options.

We can now return to the implementation of `condMutate`, and make use of both `lacksDims` and `isVis`.

```
condMutate :: (VVis -> Boolean) -> (VVis -> VVis)
           -> Dec -> VVis -> VVis
condMutate p f dec v@(Chc d l r) =
  case lookupDim d dec of
    Just L  -> Chc d (condMutate p f dec l) r
    Just R  -> Chc d l (condMutate p f dec r)
    Nothing -> if lacksDims dec v && p v
                 then f v
                 else v
condMutate p f dec v@(NextTo vs) =
  if lacksDims dec v && isVis v
     then if p v then f v else v
     else NextTo $ map (condMutate p f dec) vs
condMutate ...
```

Returning to the example, the final step is to define the predicate for `condMutate`. Here is a predicate that checks whether a visualization contains exactly four bars.

```
> fourBars :: VVis -> Boolean
  fourBars (NextTo vs) = case toUnfoldable vs of
    [Mark _, Mark _, Mark _, Mark _] -> true
    _ -> false
  fourBars _ = false
```

This function converts the nonempty list to an array for easy pattern matching and then checks whether we have a `NextTo` composition of exactly four `Marks`, corresponding to bars. Now we can finally apply it to the existing visualization to perform the mutation. The output is shown in Figure 5.

```
> let barAndPies =
      condMutate fourBars (Polar << reorient)
                 emptyDec allBars
```

The idea of using a predicate to perform mutation conditionally applies more generally in other transformation situations as well. We could define more general versions of functions such as `vary` and `branch` that also include such a conditional check. It is a completely orthogonal feature.

7 Aggregating Variation

The third and final category of tasks we support are those which reduce the amount of variation through aggregation. In the context of visual data analysis, this corresponds to situations in which we make decisions about design aspects we had been exploring.

7.1 Selecting and Ignoring

When we generate variation in a visualization, it is often because we are not sure what the optimal visual representation is for the task at hand. However, as the process continues

some of those uncertainties will be removed. In these situations, we may wish to commit to one alternative in a dimension of variation through selection (in the choice calculus sense, see Section 3). The definition of the function `select` provided earlier needs to be adapted slightly. That definition would only work if we had defined our visualizations to be of type `V Vis` rather than integrating `V` directly as a constructor. Here is a partial definition.

```
select :: VVis -> Dec -> VVis
select (Chc d l r) dec = case lookupDim d dec of
  Just L  -> select l dec
  Just R  -> select r dec
  Nothing -> Chc d (select l dec) (select r dec)
select ...
```

Suppose we have used the `vary` function to produce a variant of a chart with labels, which were originally omitted.

```
> let labVis = vary ("Labels",`label` myLabs) myVis
```

After doing this we see that the labels clutter the representation too much, making it difficult to read. We would like to commit to the representation without labels, effectively undoing the application of `vary`. All we need is selection. Since this pattern of ignoring a previously added variant occurs frequently, we add a function that does exactly this. This function also relieves the user from the need to remember that new variants are added as "right" variants and that the old visualizations are kept as "left" variants.

```
ignore :: VVis -> Dim -> VVis
ignore d = select (sel2dec (d,L))
```

With the help of `ignore`, we can now simply define the variation-reduced visualization as follows.

```
> let noLabVis = ignore "Labels" labVis
```

Note that since in this example we have a reference to `myVis`, we don't strictly need the function `ignore`, but that will not always be the case.

7.2 Flattening

One final kind of variation aggregation the DSL supports is flattening. The basic idea of flattening is that sometimes we want to eliminate variation but not by promoting one variant and eliminating the other. Instead, flattening allows us to specify exactly what should be done with the variants. Since this operation acts on choices directly rather than on visualizations, the traversal of the `flatten` function looks slightly different from the ones introduced so far.

```
flatten :: (VVis->VVis->VVis) -> Dec -> VVis -> VVis
flatten f dec (Chc d l r)
      | lacksDims dec l && lacksDims dec r = f l r
      | otherwise = Chc d (flatten f dec l)
                          (flatten f dec r)
flatten f dec (NextTo vs) =
        NextTo $ map (flatten f dec) vs
flatten ...
```

Again, we also provide a version of the function that takes as argument a function that manipulates visual parameters.

```
flattenVP :: (VPs->VPs->VPs) -> Dec -> VVis -> VVis
flattenVP f = flatten (updVP f)
```

This function has many uses. Suppose, for example, we are analyzing data collected from a set of redundant sensors. For each data point we have potentially more than one value due to the redundancy, that is, variational data. As shown in Figures 6a and 6b, we have two variant charts that show two of the values varying. Our goal now is to produce a third chart which will show the mean values everywhere that variation occurs. The first thing we need to do is define a simple helper function to produce a bar based on the average height of two existing bars.

```
avgHeight :: VPs -> VPs -> VPs
avgHeight vps1 vps2 =
    vps1 { height = (vps1.height+vps2.height)/2.0 }
```

Now we can apply that to our existing visualization and get the result shown in Figure 6c.

```
> let avgBars = flattenVP avgBars emptyDec myBars
```

Again, one reason for why one might want to transform the visualization directly instead of the underlying data is that the visualization might be part of a bigger variational visualization, which had to be reconstructed if the data were to be changed. Another reason is that we may want to see the resulting averages in the context of the original data, for example, to get a sense of where and how much the data varies. This is not clear from Figure 6c. We can create yet another version of the visualization by overlaying the averages with the bars for the original data (displayed using a lighter color). The result is shown in Figure 6d.

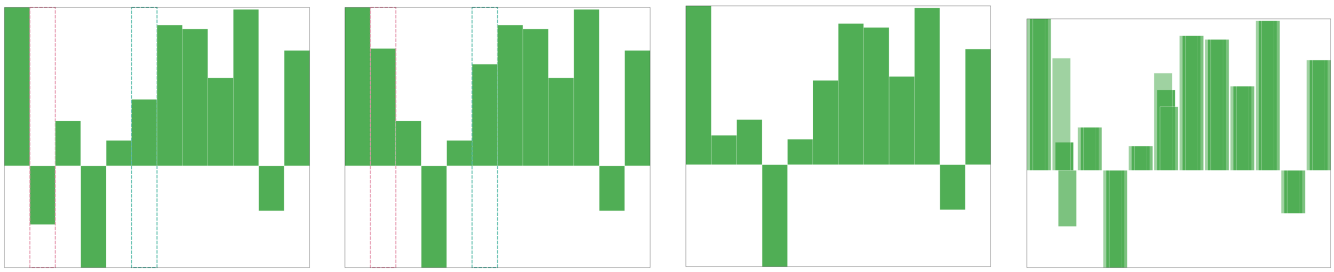
This example presents only a very simple case for flattening for the purpose of illustration. Many other, more convincing, use cases exist. For example, if in a variational visualization each of two variants has its advantages, and we want to combine the best of both such as when we want to take the frames from one alternative but the color from another. Or consider the case when we want to merge two variants into an overlay of two visualizations. Sometimes we might even need a conditional selection from different variants, for example, to promote the variant that is simplest (has the fewest marks) or has the highest maximum value. The function `flatten` can be employed to express all these variation transformations.

8 Related Work

The related work in this field can roughly be assigned to one of two categories: (1) DSLs and other visualization tools, and (2) applications of variation.

8.1 DSLs and Visualization

The grammar of graphics [20] and its most popular implementation in the form of ggplot2 [19] take an object-oriented



(a) (b) (c) (d)

Figure 6. Using flattening to aggregate variational data. Figures (a) and (b) show two variants of the same variational visualization where two of the data values vary. Figure (c) shows the visualization after using `flatten` to compute the average values of the variational bars and charting that value instead. Figure (d) shows the average visualization overlaid with the visualizations of the original data.

DSL approach to visualization, which supports some similar concepts to this work, including interpreting visualizations in different coordinate systems and transforming data in a post hoc fashion. However, the language is not designed to support exploratory visualization but instead primarily to describe and produce a single visual artifact.

Many visualization systems make use of domain-specific languages. For example, Protovis [3] is a Javascript-embedded DSL for constructing visualizations. It was based on a declarative domain-specific language which separated the specification of visualizations from the rendering process [9]. Protovis employed typical visualization abstractions such as scales (for example, for distinguishing quantitative and ordinal data) and layouts (such as clusters and stacks). The original authors have since moved on to D3 [4], which trades the domain-specific abstractions for more familiar web standards. While this has the advantage of being more flexible in terms of creating general documents driven by data, the large number of D3 wrappers developed in the visualization community demonstrates that a DSL approach is still desirable. As with most other visualization DSLs, the goal of Protovis was to create single artifacts rather than to support an iterative and incremental workflow.

DSLs have also been effective in visual domains outside of information visualization. For example, Rautek et al. [12] showed how multiple DSLs can be employed to perform scientific visualization tasks and Duke et al. [5] advocated for DSL approaches in general, using scientific visualization as a domain to demonstrate the potential value. The Diagrams DSL, described partially by Yorgey [21], supports the creation of mathematical diagrams and, like this work, makes use of composition and relative spacing.

8.2 Variation

Probably the work most directly related to this work form the perspective of working with variation is Side Views [15]. Side Views is a user interface extension specifically targeted at supporting open-ended tasks. It provides both dynamic previews of visual editing operations as well as interface elements for adjusting the parameters to those operations (for example, angle of rotation or degree of blur). It is also possible to compose operations to preview sequences of commands.

Parallel Paths [16] expands on Side Views by focusing on how users can be supported in navigating variants and promoting variants. While their specific model of variation is not discussed, the authors mention being able to show "slices" of variation which is analogous to variants in the choice calculus. They also have a model they call "selection" which supports a kind of projectional editing [1].

Finally, Hartmann et al. [8] took a variation-based approach to user interfaces and interactions which require comparison tasks, and Erwig and Smeltzer [6] describe applying the choice calculus to produce variational pictures.

9 Conclusion

This paper demonstrates a domain-specific embedded language in Purescript for creating and manipulating variational visualizations. Through the use of examples, we have demonstrated how the generation, transformation, and aggregation of variation contained in visualizations is useful in supporting the iterative workflow associated with visual data analysis.

In particular, the combination of variational visualization and visualization transformations allows analysts to delay committing to any particular design decisions regarding their visualizations. Instead, by modeling variation explicitly, they can simply keep all desired variants, navigate among them, transform them, and only remove extraneous ones when ready.

Acknowledgments

This work is partially supported by the National Science Foundation under the grants IIS-1314384 and CCF-1717300.

References

[1] Thorsten Berger, Markus Völter, Hans Peter Jensen, Taweesap Dangprasert, and Janet Siegmund. 2016. Efficiency of Projectional Editing: A Controlled Experiment. In *ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*. 763–774.

[2] Jacques Bertin. 1999. *Semiology of Graphics: Diagrams, Networks, Maps*. Morgan Kaufmann Publishers Inc. English translation.

[3] Michael Bostock and Jeffrey Heer. 2009. Protovis: A Graphical Toolkit for Visualization. *IEEE Transactions on Visualization and Computer Graphics* 15, 6 (2009), 1121–1128.

[4] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. $\mathbb{D}^3$: Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (2011), 2301–2309.

[5] D. J. Duke, R. Borgo, M. Wallace, and C. Runciman. 2009. Huge Data But Small Programs: Visualization Design via Multiple Embedded DSLs. In *Practical Aspects of Declarative Languages*, Andy Gill and Terrance Swift (Eds.). 31–45.

[6] Martin Erwig and Karl Smeltzer. 2018. Variational Pictures. In *Int. Conf. on the Theory and Application of Diagrams (LNAI 10871)*. 55–70.

[7] Martin Erwig and Eric Walkingshaw. 2011. The Choice Calculus: A Representation for Software Variation. *ACM Transactions on Software Engineering and Methodology* 21, 1, Article 6 (2011), 27 pages.

[8] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. 2008. Design As Exploration: Creating Interface Alternatives Through Parallel Authoring and Runtime Tuning. In *ACM Symp. on User Interface Software and Technology*. 91–100.

[9] Jeffrey Heer and Michael Bostock. 2010. Declarative Language Design for Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics* 16, 6 (2010), 1149–1156.

[10] Daniel Keim, Gennady Andrienko, Jean-Daniel Fekete, Carsten Görg, Jörn Kohlhammer, and Guy Melançon. 2008. Visual Analytics: Definition, Process, and Challenges. In *Information Visualization: Human-Centered Issues and Perspectives*, Andreas Kerren, John T. Stasko, Jean-Daniel Fekete, and Chris North (Eds.). 154–175.

[11] Ralf Lämmel and Simon Peyton Jones. 2003. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *ACM SIGPLAN Int. Workshop on Types in Language Design and Implementation*. 26–37.

[12] Peter Rautek, Stefan Bruckner, M Eduard Gröller, and Markus Hadwiger. 2014. ViSlang: A system for interpreted domain-specific languages for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (2014), 2388–2396.

[13] David Reinsel, John Gantz, and John Rydning. 2017. *Data Age 2025: The Evolution of Data to Life-Critical*. Technical Report. IDC.

[14] Karl Smeltzer, Martin Erwig, and Ronald Metoyer. 2014. A Transformational Approach to Data Visualization. In *Int. Conf. on Generative Programming: Concepts and Experiences*. 53–62.

[15] Michael Terry and Elizabeth D. Mynatt. 2002. Side Views: Persistent, On-demand Previews for Open-ended Tasks. In *ACM Symp. on User Interface Soft. and Tech.* 71–80.

[16] Michael Terry, Elizabeth D. Mynatt, Kumiyo Nakakoji, and Yasuhiro Yamamoto. 2004. Variation in Element and Action: Supporting Simultaneous Development of Alternative Solutions. In *SIGCHI Conf. on Human Factors in Comp. Systems*. 711–718.

[17] Edward R. Tufte. 2001. *The Visual Display of Quantitative Information* (2nd ed.). Graphics Press LLC.

[18] Jarke J. van Wijk. 2005. The Value of Visualization. In *IEEE Visualization*. 79–86.

[19] Hadley Wickham. 2016. *ggplot2: Elegant Graphics for Data Analysis* (2nd ed.). Springer.

[20] Leland Wilkinson. 2006. *The Grammar of Graphics*. Springer Science & Business Media.

[21] Brent A. Yorgey. 2012. Monoids: Theme and Variations (Functional Pearl). In *Haskell Symposium*. 105–116.