AN ABSTRACT OF THE DISSERTATION OF

Karl Smeltzer for the degree of Doctor of Philosophy in Computer Science presented on

June 15, 2018.

Title: Design and Application of Variational Representations

Abstract approved: ______________________________

Martin Erwig

Variability is an important and widely studied topic across domains such as version control, software product lines, and metaprogramming. This dissertation presents an investigation into the process of systematically adding variability to data structures and programs, leading to guidelines for variational data structures and implications for programs that create, manipulate, and otherwise consume them. In particular, it focuses on the tradeoffs between expressiveness and efficiency that must be managed when designing these structures and programs. While more general and expressive variational programs can simplify the work programmers need to do, they are inevitably less efficient than approaches tailored to specific scenarios and domains.

This means that these approaches to variability are more appropriate for a certain class of applications than others. While efficiently consuming variational data remains a challenge, concentrating on domains that focus on generating variational artifacts has been effective. We demonstrate this directly with two example application domains, namely variational pictures and variational data visualization.

©Copyright by Karl Smeltzer
June 15, 2018
All Rights Reserved

Design and Application of Variational Representations

by

Karl Smeltzer

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented June 15, 2018
Commencement June 2019

Doctor of Philosophy dissertation of Karl Smeltzer presented on June 15, 2018.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Karl Smeltzer, Author

ACKNOWLEDGEMENTS

Foremost, I would like to offer my sincere thanks to Martin Erwig not only for his role as advisor and mentor, but also for his tireless encouragement, thoughtfulness, and support well beyond the requirements of the job. Thank you also to everyone else who has kindly volunteered time to serve on my committee at some point, including Margaret Burnett, Eugene Zhang, Eric Walkingshaw, Ron Metoyer, Mike Rosulek, and Julia Jones.

Stacey, I am incredibly lucky to have you in my life, and the motivation you instill in me has been absolutely vital to me getting this dissertation finished. I can't wait to start the rest of my life together with you. To my parents and sister, whom I often take for granted, thank you for your unconditional support and setting me up for success in life.

To all of my friends, especially those who listened to me vent over beers, my heartfelt thanks. You all know who you are.

CONTRIBUTION OF AUTHORS

Many parts of this dissertation describe work performed jointly with other contributors and take content from co-authored pulications. Specifically:

- Chapters 3, 4, and 5 describe work performed jointly with Martin Erwig and borrow from Smeltzer and Erwig (2017) as well as a draft paper coauthored with Martin Erwig.
- Chapter 6 describes work performed jointly with Martin Erwig and borrows from Erwig and Smeltzer (2018).
- Chapter 7 describes work performed jointly with Martin Erwig and Ron Metoyer and borrows from Smeltzer et al. (2014).
- Chapter 8 describes work performed jointly with Martin Erwig and borrows from a draft paper.
- Chapters 1, 2, and 9 also borrow from all of these papers.

TABLE OF CONTENTS

Page

1 Introduction 1

1.1 Variational Data Structures and Programming 1

1.1.1 Implications for Application Domains 5

1.2 Variational Pictures 6

1.3 Variational Information Visualization 8

1.4 Contributions and Chapter Structure 10

2 Background and Related Work 13

2.1 Models of Variation 13

2.2 The Choice Calculus 14

2.3 Correctness of Variational Functions 17

2.4 Related Work: Variational Programming and Data Structures 18

2.5 Related Work: Variational Pictures 20

2.6 Related Work: Information Visualization 22

3 Variational Programming 25

3.1 Programming with the Choice Calculus 26

3.2 Lifting Functions and Control Structures 27

3.3 Generic Data Structure Interfaces 30

4 Variational Linked Lists 35

4.1 Named Choices and Choice Trees 35

4.2 Tag Formulas and Tag Maps 36

4.3 Visual Notation 37

4.4 Variational List Representations 38

4.4.1 Box Lists 39

4.4.2 Pointer Lists 40

4.4.3 Choice Lists 42

4.4.4 Tag Lists 45

4.4.5 Suffix Lists 47

4.4.6 Segment Lists 48

TABLE OF CONTENTS (Continued)

Page

4.5 Sharing . . . 49

4.6 Performance . . . 51

4.6.1 Specific List Operations . . . 52

4.6.2 Benchmarks . . . 53

4.6.3 Redesigning the Interface . . . 55

4.7 Conclusions . . . 57

5 Dependent Types for Variational Programming 59

5.1 Inefficiencies in Variational Programming . . . 59

5.2 Faster Applicative Traversals . . . 61

5.3 Variational Split Lists . . . 68

5.3.1 Speeding Up Operations with Split Lists . . . 70

5.3.2 Lazy Split Lists . . . 75

5.4 Conclusions and Lessons for Variational Applications . . . 78

6 Variational Pictures 80

6.1 Plain Pictures . . . 80

6.2 Adding Choices to Pictures . . . 81

6.3 Variability Types . . . 83

6.4 Variability Regions . . . 85

6.5 Distilling Variational Pictures . . . 87

6.6 Properties of Variational Pictures . . . 89

6.7 Maintenance of Variational Pictures . . . 92

6.8 Variational Area Trees . . . 94

7 Domain-Specific Language for Information Visualization 97

7.1 Some Basic Datatypes . . . 97

7.2 Creating Simple Charts . . . 99

7.3 Visualization Transformations . . . 101

7.4 Chart Composition and Layout . . . 105

7.5 Whitespace . . . 108

TABLE OF CONTENTS (Continued)

Page

7.6 Visualization Functor . . . 110
7.7 Visualization Monad . . . 113
7.8 Visualization Comprehensions . . . 118
7.9 Evaluation . . . 122
7.9.1 Applicable Evaluation Schema . . . 123
7.9.2 Conclusions . . . 127

8 Supporting Exploratory and Comparative Information Visualization with Variation 128
8.1 Redesign of the DSL . . . 128
8.2 Adding Variation to Visualizations . . . 131
8.3 Rendering and Navigating Variational Visualizations . . . 133
8.4 Variation for Exploratory Visualization . . . 134
8.4.1 Generating Variation . . . 135
8.4.2 Transforming and Maintaining Variation . . . 139
8.4.3 Aggregating Variation . . . 143
8.5 Variation for Comparative Visualization . . . 146
8.5.1 Example Comparative Visualizations . . . 146
8.5.2 Evaluation of Variation for Comparison . . . 149

9 Conclusion 154
9.1 Summary of Contributions . . . 156

Bibliography 156

Appendices 165
A Variational Programming Haskell Source Code . . . 166
B Variational Programming Idris Source Code . . . 171
C Variational Visualization Purescript Code . . . 176

LIST OF FIGURES

Figure		Page
1.1	A small snippet of source code annotated with C-preprocessor conditional compilation macros.	3
1.2	A sequence of variational pictures showing the design of a park. In (a) we have one area of variability for the potential removal of trees, in (b) we have an independent dimension for a pavilion area, and in (c) we have a nested dimension for a small fountain that can only exist if the trees are removed.	7
1.3	A variational visualization showing how a user might navigate among different configurations of the same visualization.	9
2.1	A commuting diagram showing the meaning of a correct variational function. For a given function $f : a \to b$ the variational analogue $vf : V(a) \to V(b)$ preserves the variation across function application. That is, selecting a plain value and applying f must produce the same result as applying vf followed by selection.	18
4.1	The unabbreviated visual notation used to demonstrate the general shapes that variational lists take.	37
4.2	Box list. A more concise representation of the list shown in Figure 4.1. Every variant list is stored in its entirety and nothing is shared.	38
4.3	Example variational lists containing conditional pre-processor annotated source code from the Introduction making use of the visual notation introduced in Section 4.3.	41
6.1	The variational picture after resizing the previously nested *Fountain* area to contain the *Trees* area, in order to depict the connecting water pipes.	93
6.2	A Variational Area Tree (VAT) that showing an entire variational picture at once. Here the VAT for the view decision $\{Trees.l, Fountain.l, Pavilion.r\}$ is shown.	94
6.3	Additional configurations of the park picture VAT. In (a) we see the view decision $\{Trees.r, Fountain.r, Pavilion.r\}$, and (b) shows the case after the commuting of the *Trees* and *Fountain* regions for the view decision $\{Trees.l, Fountain.l, Pavilion.r\}$.	95
7.1	Two simple visualizations showing (a) the basic bar chart with only default values and (b) a basic bar chart with a few simple aesthetic customizations.	100

LIST OF FIGURES (Continued)

Figure		Page
7.2	A bar chart showing passenger data from the RMS Titanic.	102
7.3	Two charts created by transforming an earlier bar chart. In (a) a rose chart in which the data is driving the radii of the wedges, and in (b) the same visualization recolored, relabeled, and reoriented so data drives the sector angles instead of the radii.	105
7.4	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.	108
7.5	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.	111
7.6	In (a) is a simple barchart of data that appear to show an exponential trend and in (b) is the same barchart after log-transforming each of the bars, showing what appears to be a linear trend.	112
7.7	Another example of a log-transformed visualization, as in Figure 7.6, except this time using the visualization `Monad` instance to generate new bars in place beside their counterparts.	115
7.8	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.	118
7.9	Estimated employee performance data created using a visualization comprehension to combine two existing bar charts. Labels identifying employees are omitted because of formatting limitations.	122
8.1	A screen capture of our prototype user interface showing possible configuration/selection options. On the left is the rendered visualization currently being constructed and on the right are the interface elements which allow the user to navigate among the variants.	135

LIST OF FIGURES (Continued)

Figure | Page

8.2 Generating variation. By using the `vApp` 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. 136

8.3 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. 141

8.4 Using a flattening approach to dealing with variational data. The left and center figures show two variants of the same variational visualization. Two of the data values vary. On the right is the visualization after using `flatten` to compute the average values of the variational bars and charting that value instead. Although subtle, the lack of dotted outline in (c) shows that the variation has been removed. . 144

8.5 Comparative visualizations for exploring visualization details; (a) Summary visualization that corresponds to the decision $\{West.l, East.l, South.l\}$; (b) Revealing details for the east with decision $\{West.l, East.r, South.l\}$. 147

8.6 Examples of comparative visualizations using hybrid designs. In (a) we have two charts on the left which are zipped together to produce the chart on the right by subtracting the heights of the lower chart from the top one. Note that each chart is scaled independently and so simply measuring the bars would be misleading. Figure (b) shows a small multiples rendering of a data set overlaid with its log-transformed data on the left and overlaid with its square-root-transformed data on the right. Finally, (c) shows a variational visualization in which the left variant shows the original data and the right variant shows the visualizations sorted after their creation. 149

LIST OF TABLES

Table | Page

4.1 A categorization of variational list representations according to their use of variational type constructors and how they are integrated into the list structure. 39

4.2 A summary of the type of sharing offered by each representation discussed in this work. 51

4.3 Benchmark results for applying `vFilter` to five CPP-annotated source code files for each of our list representations. Runtimes given in μs. 55

4.4 Benchmarks for applying `vFilter` on some list representations using a more granular approach. All runtimes given in μs. Compare to Table 4.3. 56

8.1 Matching our variational visualization approaches to comparative design categories. 153

Chapter 1: Introduction

This dissertation presents research on two major aspects of variation. First, we present an investigation of how variation can be systematically integrated into data structures and programs. Second, based on lessons learned during that investigation, we then present two example applications demonstrating effective uses of variation in practical domains.

The remainder of this chapter will motivate the need for variational data structures and programs, discuss the selection of suitable application domains, and outline some of the challenges posed by those domains. Additionally, we will summarize the specific contributions and the structure of the remaining chapters.

1.1 Variational Data Structures and Programming

Variability in software and data is ubiquitous. It directly underpins all research topics which are concerned with generating and managing variant artifacts. This includes, but is not limited to, version control systems, software product lines, and metaprogramming. Those areas are widely studied, but typically focus only on variation as it occurs when viewing source code as data. It is also possible, however, to think of a program that is executed repeatedly on different inputs as a variational program that operates on variational data. For example, consider searching for a specific string in all variants of a version control system (Lee et al., 2015), in the history of an editor's undo stack (Yoon and Myers, 2015), or in speculatively merged branches (Brun et al., 2011). Performing such a search repeatedly in each variant is potentially very inefficient. Since different variants often have large parts in common, a more promising approach would be to represent all documents as one document with

differences expressed locally, and then executing the search over this combined, variational document. This would help avoid unnecessary repeated searches over the same parts and could overall speed up the search considerably.

Finding a way to take advantage of this potential efficiency gain requires addressing several challenges. First, we need a representation for variational data. There are many different possible representations which all have advantages and disadvantages in different situations. Second, we need to consider the technical requirements for a actually working with this variational data and implementing new operations on them without negating any of the potential benefits. This is surprisingly challenging as many typical programming idioms do not lend themselves to efficient variational computation.

To that end, this dissertation presents variational lists as a fundamental data structure which can serve as a basis for many applications, including such a search over variational documents. A variational list is a mapping from configurations to *plain* (that is, non-variational) lists. A configuration is a structure that uniquely identifies a particular variant among all the variants in the variational list (or variational structure in general). This is best illustrated with a specific example.

When source code is annotated with C preprocessor (CPP) macros for conditional compilation (using `#ifdef` and family), we can view that code as a variational list of code sections. That is, each combination of macros defines a configuration that determines a particular variant list of sections in our variational list, and thus a particular variant of the code.[1] Consider the code snippet in Figure 1.1.

By just naively looking at the number of conditional statements in this snippet, we can see four separate macros that are checked: `WINDOWS`, `UNIX`, `MAC`, and `LINUX`. This means that even this unrealistically small code snippet encodes $2^4 = 16$ different possible configurations (although not all make sense in practice).

[1]Some other example sources of variational data include political polling from multiple sources, rankings of all sorts, weather data, and redundant sensors as in autonomous vehicles.

```
#ifdef WINDOWS
   printf("Hi Redmond.");
#elif defined UNIX
   printf("Hello");
   #ifdef MAC
       printf(" Cupertino.");
   #elif defined LINUX
       printf(" Helsinki.");
   #endif
#endif
printf("Goodbye.");
```

Figure 1.1: A small snippet of source code annotated with C-preprocessor conditional compilation macros.

If we wanted to analyze and query this code in a configuration-aware manner, say by trying to search for all those configurations which contain a particular string, then a naive approach will need to generate all 16 source code files and search each individually. With the large number of variants we end up repeating work unnecessarily. In this small example, the last `printf` line would be queried 16 times since it occurs in each variant. A real source code file would likely include hundreds or thousands of lines of code shared among variants, not to mention many more macros. Repeating work in that context quickly becomes unreasonable.

Instead, by taking the approach of first translating the code into a variational list where the elements store sections of the program text and then performing a variational search on that list, we stand to potentially avoid repeating all of this work. As mentioned, the first step to achieving this is to determine how to actually represent such a variational list containing all of these plain variants. One intuitive option might be to use a map in which each key is a particular configuration and the value stored is the corresponding variant. To keep the example small, we are using just the three possible configurations that produce distinct results, plus the empty configuration.

$$c_0 = \{\} \qquad c_1 = \{\texttt{WINDOWS}\} \qquad c_2 = \{\texttt{UNIX, MAC}\} \qquad c_3 = \{\texttt{UNIX, LINUX}\}$$

$$\begin{aligned}
\{\ c_0 &\mapsto [\texttt{printf("Goodbye.");}],\\
c_1 &\mapsto [\texttt{printf("Hi Redmond.");}, \texttt{printf("Goodbye.");}],\\
c_2 &\mapsto [\texttt{printf("Hello");}, \texttt{printf(" Cupertino");}, \texttt{printf("Goodbye.");}],\\
c_3 &\mapsto [\texttt{printf("Hello");}, \texttt{printf(" Helsinki");}, \texttt{printf("Goodbye.");}] \quad \}
\end{aligned}$$

Each plain list used in this map is called a *variant* of the variational list, and the keys in the map amount to *decisions* that one has to make in order to access any particular variant. We can observe that each subset of macros corresponds to a potential configuration that can appear as a key in the mapping. Any representation of variational values consists in some way of these two components: (a) the individual variants, and (b) the decisions to distinguish between the variants.

Another representation is to swap the usage of the list and map and instead make every list element a map from a configuration to a single plain value.

$$\begin{aligned}
[&\{\ c_0 \mapsto \epsilon, c_1 \mapsto \texttt{printf("Hi Redmond.");}, c_2 \mapsto \texttt{printf("Hello");}, c_3 \mapsto \texttt{printf("Hello");}\ \},\\
&\{\ c_0 \mapsto \epsilon, c_1 \mapsto \epsilon, c_2 \mapsto \texttt{printf(" Cupertino");}, c_3 \mapsto \texttt{printf(" Helsinki");}\ \},\\
&\texttt{printf("Goodbye.");}\ ]
\end{aligned}$$

This may not seem like a particularly consequential choice to make, but differences in representation can lead to surprisingly different performance depending on the task at hand. In the first representation the `printf("Goodbye.");` statement is copied in every map element. Suppose now we want to query a variational list for the last line of every configuration of a source code file. If we use the former representation, we are forced to repeatedly traverse lists until we reach the end, once per configuration.

If we instead use the latter representation, we only need to index into the outer list once to obtain the last element for every variant.

Conversely, suppose we want to compare differences between two specific configurations. Using the second representation we need to perform two map lookups (corresponding to our two chosen configurations) for every element in the variational list. The first representation only requires two map lookups total, after which we have the data we want.

This begins to show the serious ramifications that seemingly simple representation decisions have when it comes to variational lists. Furthermore, the design of such a data structure is only part of the necessary components for successful variational programming. Programmers will inevitably need to implement list operations and transformations that are not included in a library. This means that they need an interface to the data structure that enables efficient techniques for constructing and decomposing them. As we will show later, this turns out to be a surprisingly difficult problem.

In Chapter 3 we will introduce some of the components necessary for variational programming and the challenges they raise. Then, in Chapter 4 we will compare specific variational list representations in detail.

1.1.1 Implications for Application Domains

One of the fundamental findings from the work discussed in Chapters 3 and 4 is that it is, at best, difficult (or not possible, at worst) to design an optimal variational data structure that excels in all circumstances. This makes designing new variational data structures nuanced and challenging. Furthermore, even if the designer is able to conceive of a clever and efficient representation, there are cascading consequences for the efficiency of the programs that consume them. In general, trying to maximize expressiveness in representations of variation ultimately results in reduced efficiency. The tension between these characteristics, while somewhat unsurprising, has major consequences for

working with variation in practice. In particular, it means that not all domains make good candidates for integrating variation efficiently. Variation as it is discussed here is well suited for domains which focus on the generation variational artifacts in a systematic manner. Consuming and deconstructing variational artifacts, on the other hand, remains a challenge.

Based on this finding we demonstrate two practical application domains which are able to successfully navigate this tradeoff, namely variational pictures and variational information visualization.

1.2 Variational Pictures

Visual media such as diagrams and pictures are ubiquitous in the modern world. These take many forms and are employed by a variety of professionals, ranging from architects and industrial designers using CAD tools, graphic designers and photographers using photo editing tools, to scientists and business owners creating charts and technical diagrams to analyze and share data.

While the software tools for these applications are quite sophisticated, they offer little or no support for managing variation in the produced artifacts, forcing users to employ rudimentary techniques to manage multiple versions of their work. For example, a graphic designer who might want to showcase changes to a logo design might be forced to overuse the layer system in their editing tools or simply create multiple copies of the picture file. A data scientist generating a series of similar or related charts and tables might have to manually copy files or images and rename them meaningfully to be able to view and compare them. Architects and engineers frequently make revisions to their drawings and designs, and their tools offer minimal, if any support, forcing them to adopt ad hoc approaches.

The need for variation comes in two different forms. First, one might want to create several concurrent variants of an artifact. Second, one might need some kind of version control for pictures. We propose *variational pictures* as an underlying model to support both forms of picture variation. A variational picture is simply a picture (here a grid of pixel values) that offers an explicit way of

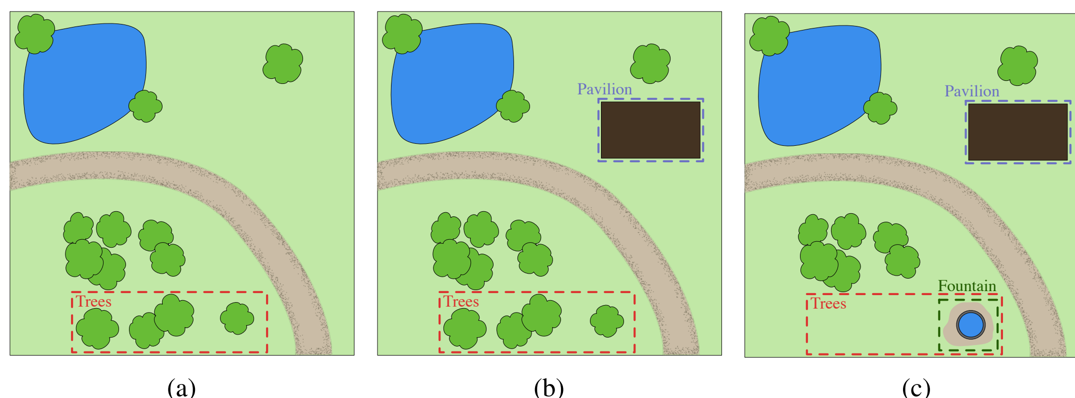


Figure 1.2: A sequence of variational pictures showing the design of a park. In (a) we have one area of variability for the potential removal of trees, in (b) we have an independent dimension for a pavilion area, and in (c) we have a nested dimension for a small fountain that can only exist if the trees are removed.

representing and selecting different variant pictures. For example, a picture along with its history (or undo) states can be viewed as a variational picture. Similarly, a collection of related but different designs is also a variational picture. It is even possible to view an animation as a variational picture in which the variants are the frames with a temporal ordering.

Consider a landscape architect working on the plan for a new city park, the features of which are not fully decided yet. For example, an additional wooded area may be cleared to make more space for the park if the budget is determined to allow for it. In another part of the park, a pavilion may be added to provide covered seating. Finally, on the condition that the trees are cleared, a small fountain may be installed where they were. Even with such a small number of undetermined features, this already means there are six possible park layouts. It is easy to imagine this growing out of control rather quickly with more options. By using a model of variational pictures, the architect could produce a single plan with the areas of variability clearly marked that also allows toggling between the options to show them to the final decision makers. We show a sequence of such variational pictures in Figure 1.2.

The example does not only show that variational pictures are useful, it also illustrates that managing variability is a nontrivial matter that requires a number of operations for creating, eliminating, and adapting variability. We will return to this example later, in Chapter 6.

1.3 Variational Information Visualization

Visualization has emerged as as a core component of modern data analysis. Data collection continues to grow exponentially (Reinsel et al., 2017) and with it the number of tools and systems for creating data driven graphics and visualizations has increased accordingly. While there are exceptions (see Chapter 2 for related work) many of these only support the creation of individual visualization artifacts with little attention paid to the iterative and exploratory nature of visual data analysis (Keim et al., 2008; van Wijk, 2005).

Unlike the hypothesis testing used in confirmatory data analysis, exploratory data analysis centers on extracting hypotheses from the data and making decisions about which techniques to apply next (Tukey, 1977). One of the characteristics inherent to exploratory data analysis is the analyst not knowing the characteristics of the data and how they might best be shown. Even powerful tools like ggplot2 and are targeted primarily at the creation of a single visualization artifact rather than an interative process and also do not support this kind of open-ended specification. Analysts are forced to commit to certain aspects of their visualizations before they can be confident about doing so.

For example, suppose we are analyzing a set of financial data 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.

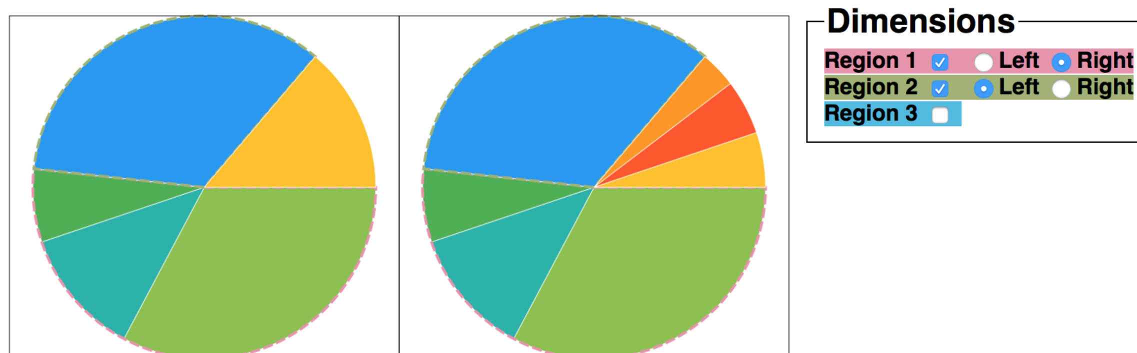


Figure 1.3: A variational visualization showing how a user might navigate among different configurations of the same visualization.

In advanced tools such as ggplot2 or $\mathbb{D}^3$, an experienced user could be able to find a way to parameterize the color and space information, but there is no obvious way to support data that take 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. A simplified example of this scenario, which only shows options for the visibility of subregions, is shown in Figure 1.3, along with a prototype interface for navigating the variants.

Variational visualizations give visualization authors a way to avoid prematurely comitting to certain design details and instead encode all of the different variants into a single visualization structure. We present a model of variational visualizations as well as a prototype implementation in Chapter 8.

It turns out that variational visualizations enable other visualization tasks as well, not just exploratory ones. One example is visual comparison tasks. Comparisons tasks feature prominently in all kinds of visualization history mechanisms (Heer et al., 2008), uncertainty visualization (Bonneau et al., 2014), software visualization (Holten and van Wijk, 2008), and many more areas.

One widely adopted approach to visual comparison is small multiples (Tufte, 2001) (roughly, composing the variant visualizations into a grid containing all possibilities). This approach is only a partial solution, however. Two immediate problems it faces are how to organize the charts into a grid and how to ensure they are simple enough to be read at a small size. More importantly, a grid layout is inherently only scalable in two dimensions. As the number of orthogonal parameters in the data grows we need exponentially many charts to keep up.

Yet another complication arises from the difficulty of context-dependent comparison. For example, suppose we are tasked with an analysis of profits per quarter for a business. During that process we might need to undertake some semantic zooming subtasks such as comparing only fourth quarter profits across years to see the impact of holiday bonuses, or perhaps exploring the data only for specific geographic regions. Such tasks can of course always be performed manually by returning to the data, subsetting or manipulating it, and then re-creating appropriate visualizations. However, it is highly inefficient to essentially start from scratch with each iteration. Variational visualizations offer a systematic way to create, transform, and compare visualizations. This will be demonstrated through numerous examples in Section 8.5

1.4 Contributions and Chapter Structure

This dissertation makes the following contributions.

1. A study of both the systematic integration of variability into data structures and the impact that this has on the requirements for variational programming, in which variational data structures

are created, modified, and otherwise consumed. This culminates in observing a tension between expressive power of a variational data structure and its efficiency. This study is a composition of several individual contributions.

(a) An approach to variational programming based on a variational type constructor class, including implications for the design of generic interfaces to data structures. This is detailed in Chapter 3.

(b) A thorough investigation of variational linked lists in Chapter 4, including specific representations, a comparison of their ability to share data between variants, discussion of theoretical efficiency, and practical benchmarks on real-world data. This also includes general guidelines for the design of other variational data structures and examples of additional complexities that need to be managed when doing so.

(c) An alternative approach to variational programming based on dependent types, described in Chapter 5, which concedes some flexibility to avoid performance pitfalls.

2. A demonstration of the suitability of adding variability to a certain class of applications, specifically variational pictures and variational information visualization. It consists of the following more specific contributions.

(a) A formal model of variational pictures, as well as practical examples and use cases, described in Chapter 6.

(b) A domain-specific language for non-variational information visualization, discussed in Chapter 7.

(c) An extended domain-specific language for creating and transforming variational visualizations, in Chapter 8.

Finally, Chapter 9 draws some conclusions, discusses potential future work, and summarizes the major contributions.

Chapter 2: Background and Related Work

This dissertation is principally about variation. Its contributions are focused on integrating variation into new domains, however, and therefore depend on existing work to actually model the variation. The purpose of this chapter is twofold. First, it will compare approaches and justify the model used in the remaining sections. Second, it will detail related work for variational programming, variational pictures, and variational information visualization.

2.1 Models of Variation

Techniques for modeling variational artifacts generally take either an annotational approach, a compositional approach, or a metaprogramming approach (Walkingshaw, 2013). Annotational approaches make use of a metalanguage to syntacticaly delineate which parts of an object language artifact belong to particular variants. The C preprocessor is one widely used annotative metalanguage. Compositional approaches generally decompose variational artifacts into features (Apel and Kästner, 2009) or aspects (Kiczales et al., 1997), and then provide a way to map configurations to sets of them in order to synthesize particular variants. Finally, metaprogramming approaches make use of macros (Kohlbecker et al., 1986) or templates (Sheard and Jones, 2002).

Each approach has advantages, but for the work described in this dissertation the most important is for the approach to be able to be applied to arbitrary object languages. Since our goal is to apply variation across several domains, using an approach that is tailored to a particular language or environment would not be suitable. Annotative approaches serve in this role more effectively than others.

There are a number of annotative approaches to select from. The C preprocessor, for example, is widely known and used. However, it has also been shown to be a suboptimal choice (Le et al., 2011). Another possible approach is one based on tags. For example, objects in a metalanguage or pieces of variational data can be tagged with a name corresponding to a particular configuration. It is also possible to extend this approach to use tag formulas for more flexibility. This would allow us to, for example, tag a value with the formula $c_1 \wedge c_2$ to indicate that the tagged value is only relevant if both configuration options c_1 and c_2 are enabled. One major downside to this approach is that such formulas require a SAT solver for many operations.

The tagging approach essentially maps a value to its configuration. Another possible approach is to reverse that relationship and instead map configurations to values. This is the approach used by the choice calculus (Erwig and Walkingshaw, 2011). This has the advantage of not requiring any advanced machinery to analyze and work with, but does sometimes require duplication. For the work described in this dissertation that is a reasonable tradeoff to make and so, with the exception of a brief consideration of tagging again in Chapter 4, all portions of this work are based on the choice calculus as a formal model of variation. For that reason, the next section is devoted to providing an overview of the choice calculus with enough detail to allow readers unfamiliar with it to follow all the remaining chapters.

2.2 The Choice Calculus

The choice calculus encodes variability in *choices*. Choices are lists of *alternatives* together with a *dimension*, or name. We can encode a variational number with two variants, 3 and 5, as a choice in dimension A by writing $A\langle 3,5\rangle$. Choices can also be nested, as in $A\langle B\langle 3,4\rangle,5\rangle$.

The choice calculus allows for alternative lists of arbitrary length, but in this work we use exclusively binary choices. This makes many definitions and implementations easier. An n-ary choice can always be encoded as $n-1$ nested binary choices. For example:

$$A\langle w,x,y,z\rangle \approx A\langle w,B\langle x,C\langle y,z\rangle\rangle\rangle$$

Nested choice structures are called *choice trees*. Although some work, such as Walkingshaw et al. (2014) has called them *tag trees*, we prefer choice trees to better distinguish them from tag-based approaches. A choice tree with n leaves contains $n-1$ dimensions.

Each choice gives rise to *selectors*, which allow us to indicate a particular alternative from the list contained in that choice. Since we are limited to binary choices, we write these selectors $D.l$ and $D.r$ for the left and right alternatives in dimension D, respectively. Sets of selectors are called *decisions* and can be used for *selection*, which is the process by which we can promote a particular variant in a choice to reduce or eliminate variation.

Selection, written $\lfloor\cdot\rfloor_s$, is defined as follows for a single selector.

$$\lfloor D\langle vx,vy\rangle\rfloor_s = \begin{cases} \lfloor vx\rfloor_s & \text{if } s = D.l \\ \lfloor vy\rfloor_s & \text{if } s = D.r \\ D\langle\lfloor vx\rfloor_s,\lfloor vy\rfloor_s\rangle & Otherwise \end{cases}$$

$$\lfloor x\rfloor_s = x$$

For example, $\lfloor A\langle 1,2\rangle\rfloor_{A.l} = 1$, $\lfloor A\langle B\langle 1,2\rangle,3\rangle\rfloor_{A.l} = B\langle 1,2\rangle$, and $\lfloor A\langle B\langle 1,2\rangle,3\rangle\rfloor_{B.r} = A\langle 2,3\rangle$. Choices that share a dimension are synchronized, meaning that if a selection is made in dimension D it is selected for every D choice.

Let us look at a slightly larger example. Consider the following two equally valid implementations of a function to double a number.

```
double1 x = x * 2

double2 x = x + x
```

Instead of defining two separate functions, we can define a single function annotated with the choice calculus where each variant corresponds to a different implementation. There are potentially many ways of encoding the same variational artifact with choices. The following is one way using a single choices in dimension *Impl*.

$$\texttt{double x =}\ \mathit{Impl}\langle \texttt{x * 2, x + x} \rangle$$

We can then add a second, nested choice which controls the name used for the parameter to the function, in dimension *Par*.

$$\texttt{double}\ \mathit{Par}\langle \texttt{x}, \texttt{y} \rangle\ \texttt{=}\ \mathit{Impl}\langle \mathit{Par}\langle \texttt{x}, \texttt{y} \rangle\ \texttt{* 2}, \mathit{Par}\langle \texttt{x}, \texttt{y} \rangle\ \texttt{+}\ \mathit{Par}\langle \texttt{x}, \texttt{y} \rangle \rangle$$

We now have the following four variants:

Decision	Code Variant
$\{\mathit{Impl.l}, \mathit{Par.l}\}$	`double x = x * 2`
$\{\mathit{Impl.r}, \mathit{Par.l}\}$	`double x = x + x`
$\{\mathit{Impl.l}, \mathit{Par.r}\}$	`double y = y * 2`
$\{\mathit{Impl.r}, \mathit{Par.r}\}$	`double y = y + y`

The choice calculus also defines a number of useful laws. A full listing is out of scope here, but there are two rules that are important. The first is a law that allows the elimination of *dominated* choices.

$$A\langle A\langle x, y\rangle, z\rangle \equiv A\langle x, z\rangle$$
$$A\langle x, A\langle y, z\rangle\rangle \equiv A\langle x, z\rangle$$

Since dimensions synchronize choices, selecting one variant in dimension A will select that variant for every choice. This means that if we have a choice in dimension A, having a nested choice in the same dimension will contain one unreachable alternative. In the first case above y is unreachable since a left selection of the outer A also selects the x from the inner A choice. For the same reason, y is unreachable in the second case as well.

Finally, we also have a transformation law for commuting choices, as follows.

$$A\langle B\langle x, y\rangle, z\rangle \equiv B\langle A\langle x, z\rangle, A\langle y, z\rangle\rangle$$
$$A\langle x, B\langle y, z\rangle\rangle \equiv B\langle A\langle x, y\rangle, A\langle x, z\rangle\rangle$$

Both the removal of idempotent choices and choice commutation prove to be useful in later chapters.

2.3 Correctness of Variational Functions

In many of the chapters that follow, we will talk about lifting functions that work on typical plain values to a variational analogue that works on variational values. The process of doing this can sometimes be complex, depending on the function and representation in question. For that reason, it is important to establish what is a correct implementation of a variational function based on its plain version. The correctness of a variational functions is determined according to the commuting diagram in Figure 2.1.

$$\begin{array}{ccc} V(a) & \xrightarrow{vf} & V(b) \\ \downarrow{\lfloor\cdot\rfloor_s} & & \downarrow{\lfloor\cdot\rfloor_s} \\ a & \xrightarrow{f} & b \end{array}$$

Figure 2.1: A commuting diagram showing the meaning of a correct variational function. For a given function $f : a \to b$ the variational analogue $vf : V(a) \to V(b)$ preserves the variation across function application. That is, selecting a plain value and applying f must produce the same result as applying vf followed by selection.

The diagrams shows that variation must be preserved across function application. Suppose the goal is to determine whether a variational function vf is correct according to its plain analogue f. We say that vf is correct with respect to f when, for any value $vx : V(a)$ and any selector s, $\lfloor vf(vx)\rfloor_s = f(\lfloor vx\rfloor_s)$. That is, selecting a plain value of type a and then applying f should lead to the same result as first applying vf and then performing the same selection.

2.4 Related Work: Variational Programming and Data Structures

Relatively few proposals exist for variational programming. Mezini and Ostermann (2004) proposed an approach combining features and aspects to mitigate issues in writing variational Java programs and Schaefer et al. (2010) espoused delta-oriented programming in which changes are conditionally applied to a core module based on the chosen configuration. An approach to variational programming inspired by the choice calculus was also proposed in Chen et al. (2016). Unlike that work, this dissertation focuses on the practical requirements of variational programming with existing languages rather a completely new one.

There is somewhat more work on variational data structures specifically. Walkingshaw et al. (2014) presents an overview of several approaches to variational lists, maps, and sets. It also inspired one of the designs in Chapter 4 and serves as a good complement to this dissertation since it also

considers a number of tradeoffs among representations. Another approach to variational lists, as well as a general strategy of adding variation to data types was developed in Erwig and Walkingshaw (2013). That work is, in some ways, a precursor to this dissertation since it addresses many of the challenges of programming with variation based on the choice calculus. It does not, however, attempt to compare approaches and extract guidelines for new applications.

Some work has also moved beyond the venerable list. Meng et al. (2017) designed and compared the performance of variational stacks in the VarexJ environment. The stack representations compared are analogous to the top-level and parametric approaches to applying variation discussed in Chapter 4, and some stack-specific optimizations are also discussed. Erwig et al. (2013) investigated variational graphs, presenting a tag-based approach much like the tagged list representation discussed here. Nodes and edges are tagged for inclusion in particular variants using tag formulas, and graph algorithms can be lifted using the Haskell functor type class, similar to the approach we take in Chapter 3.

Both TypeChef (Kenner et al., 2010) and SuperC (Gazzillo and Grimm, 2012) make use of variational data structures to perform variability-aware analysis on code annotated with the C preprocessor. However, the data structures are not the primary contribution of those projects and so their design and general usefulness are not discussed in detail.

Finally, work on model checking shares some goals with work on variational data structures. Just as variational data structures strive to share data among variants in order to eliminate unnecessary work, model checking systems look to avoid unnecessary duplication of analysis among models. (d'Amorim et al., 2008; Post and Sinz, 2008; Apel et al., 2011). Some model checking work has even gone so far as to effectively design variational data structures, even if not described as such directly, e.g., Classen et al. (2011) and Lauenroth et al. (2009).

2.5 Related Work: Variational Pictures

Although there are a large number of potential applications for variational pictures, the research in this area is limited in scope. The work most closely related to variational pictures is that on Side Views (Terry and Mynatt, 2002) and Parallel Paths (Terry et al., 2004). 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 (e.g., angle of rotation or degree of blur). It is also possible to compose operations to preview sequences of commands. No model of variation is detailed by the authors, but Side Views can be seen as a user-facing implmementation of variational pictures.

Parallel Paths 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, and also have a model they call “selection” which supports a kind of projectional editing. Again, this can be viewed as a particular implementation of a tool that makes direct use of variational pictures.

Hartmann et al. (2008) introduced Juxtapose to support exploration tasks performed by interface designers. Juxtapose provides mechanisms by which to edit and execute source code variants in parallel, as well as to adjust parameters at runtime. The linked editing approach partitions the source code into blocks based on the longest common subsequence. The authors do not formally detail their model of variation but one way of viewing the system is as a combination of of variational pictures, projectional editing (Walkingshaw and Ostermann, 2014), and a variational programming language (Chen et al., 2016).

Tangentially related, Foo et al. (2007) summarized existing work and propose new ideas in the challenge of identifying similar (in the sense of different resolutions, compression techniques,

fragments, etc.) images. This work focused on finding similar images rather than explicitly managing the variability among them.

There are also many tools design to offer digital image version control and asset management, although most are proprietary and do not describe their specific model. Examples of these include Adobe Drive and AutoDesk Vault. Some more general version control tools offer support for image diff operations including Perforce and Git via services such as Github.

This topic also emerges in the field of information visualization. For instance, Heer et al. (2008) performed a large survey of graphical history tools from the context of information visualization and exploratory analysis. The primary goal of that work was to design a history mechanism for Tableau. The focus was largely on details that are orthogonal to the model used to represent variants and change, to an extent. For example, it is concerned with aspects such as thumbnail generation, visual organization of histories, navigation, and so on. Their internal representation is ultimately a graph in which nodes are states and edges are actions. This is a less general representation than variational pictures, since we make no assumptions about the specific actions that should be modeled.

Chen et al. (2011) proposed a graph-based revision control system for images. Like the Tableau model, being based on graphs means the focus is on paths of editing operations rather than pixels or image objects, and challenges such as diff and merge are solved with graph operations. They also offer support for selective undo, which has also been achieved (for text artifacts) using a choice calculus approach (Erwig et al., 2014). Finally, Chen et al. (2011) also introduced the notion of "nonlinear exploration" in which the user can adjust parameters to operations that have already been performed. Again this requires tracking specific actions taken by the user which is less general than our target with variational pictures.

Gleicher et al. (2011) demonstrated comparative visualization as a fundamental idea, which can be viewed as an application of variability to pictures within the scope of information visualization. This

is explored in much greater detail in our application of variation directly to information visualization in Chapters 7 and 8.

Finally, model difference techniques overlap to an extent with the objectives of our variational picture model. Kolovos et al. (2009) provides an overview of the topic. Specific examples include Ohst et al. (2003), which described an approach to model difference in UML and Cicchetti et al. (2007) which showed a technique for representing differences between models independent of the metamodel. As models are generally expressed using graphs, none of this work describes a pixel-based approach.

2.6 Related Work: Information Visualization

The fundamental idea of desciribing visualizations as data bound to visual variables (or, as we prefer, parameters) comes from Bertin's seminal work (Bertin, 1999, 2005). Additionally, much of the vocabulary used here and in the wider information visualization community, such as graphical marks, is due to Bertin.

The grammar of graphics (Wilkinson, 2005) has been a direct incluence on many of our design decisions. Some specific examples include modeling operations such as rotation as coordinate transformations, frames to capture information about the extents of visualizations, and computing statistics directly as part of the visualization process rather than beforehand. There have been several implementations of the grammar of graphics, but by far the most successful is ggplot2 (Wickham, 2009), in part because it is able to leverage all of the additional tools available as packages for the R language.

Another tool, widely used for visualization, is $\mathbb{D}^3$ (Bostock et al., 2011). While powerful, $\mathbb{D}^3$ is properly a library for creating interactive and data-driven documents, rather than visualizations, and lacks visualization-specific abstractions. However, its predecessor Protovis (Bostock and Heer, 2009; Heer and Bostock, 2010) is much closer to our work. It provides a declarative domain-specific

language embedded in Javascript which allows for the construction of visualizations by defining functions that operate on data and specify the details of the corresponding graphical marks. Due to its choice of host language, Protovis lacks support for, e.g., being easily able to map functions across existing visualizations or otherwise changing them after they are defined. The move to $\mathbb{D}^3$ made this more feasible by choosing to use representations native to the browser environment rather than the domain. While this made it easier to inspect and transform elements, it also loses the advantages of a domain-specific approach.

The decision to address this problem with a domain-specific language is partially motivated by Mackinlay (1986), which introduced the idea that information visualizations are inherently sentences of a graphical language. DSLs have also been successful in other, related areas, including astrophysics visualization (Duke et al., 2009) and scientific visualization (Borgo et al., 2006), for example.

The Haskell community has also produced Diagrams (partially described in Yorgey (2012)) and its predecessor Chalkboard (Matlage and Gill, 2009) designed for image and figure creation. Diagrams is targeted at mathematical and artistic graphics rather than data-driven visualizations, but its core design principle of composing scalable and unitless elements in an infinite canvas was directly influential in our DSL. Some Haskell DSLs do exist for information visualization, most notably the Chart package[1]. These are primarily focused on statistical plotting, however, and do not support transformations or custom visualization types.

Stencil (Cottam, 2011) provides a Java-based DSL for creating information visualizations which shares the use of data-bound visual parameters with our work. It does not aim to support the traversal and transformation of visualizations that have already been created.

A number of tools offer a substantially higher-level abstraction than our work. Tableau (Mackinlay et al., 2007), 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

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

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, Improvise (Weaver, 2004), Prefuse (Heer et al., 2005), and Mondrian (Theus, 2003).

Chapter 3: Variational Programming

Before we endeavor to integrate variation into data structures, we need to first determine how programmers can interact with it more generally. Just as understanding how to work with a typical linked list structure requires understanding programming topics such as control flow and data type definitions, variational programming has its own fundamental requirements. The purpose of this chapter is to demonstrate some of the fundamental building blocks required for more general variational programming, including control structures and the design of generic interfaces.

While this chapter contains a great deal of code snippets, understanding the high-level ideas does not necessarily require mastering each of them. Depending on the intended application, it could be enough to understand the following summaries of the main contributions made here.

- To support variational programming we need a type constructor that implements the choice calculus model of variation discussed in Chapter 2. We call this `V`, and it can be applied to any existing types in order to create variational versions of them. For example, if we have a type `Nat` for natural numbers, we can apply `V` to it to obtain a type `V Nat` which represents variational natural numbers.

- Together with this type constructor, we also require functions which allow us perform choice calculus operations such as selection.

- The Haskell `Functor`/`Applicative`/`Monad` type class hierarchy provides a systematic way to lift any plain function to work on variational parameters. Since control flow in functional languages is generally provided by (higher-order) functions, this lifting also extends to control flow for free, allowing us to define, for instance, a variational if-then-else construct.

- We can generalize over the `V` type constructor using a type class to allow for other kinds of variational values.

The remainder of the chapter will avoid superfluous code definitions where possible. When noted, full definitions can be found in Appendix A.

3.1 Programming with the Choice Calculus

Since the variation applications discussed in this dissertation are based on the choice calculus, the first step to programming with variability is an implementation of the choice calculus. In turn, that begins with a data type definition for the notion of a choice. Note that elsewhere this type has been named `V` rather than `VChc`, but the need for this will become clear later in this chapter and again in Chapter 4.

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

type Dim = String
```

This `VChc` data type contains two constructors. The `Chc` constructor is for creating choices and takes a dimension as well as left and right recursive variational values. The `One` constructor is for lifting a plain value of the object language into a choice tree. We can then also define some additional types necessary for selection.

```
data Direction = L | R

type Selector = (Dim, Direction)

type Decision = Map Dim Direction
```

The `Dir` data type represents a *direction*, i.e., left or right, which is used to indicate the left and right alternatives of a binary choice. `Selector` pairs together a dimension and a direction, and `Decision` collects selectors together into a map structure. This is enough to implement choice calculus selection.

```
select :: Decision -> VChc a -> VChc a
select _ (One x) = One x
select dec (Chc d l r) = case lookup d dec of
  Just L -> select dec l
  Just R -> select dec r
  _      -> Chc d (select dec l) (select dec r)
```

The `select` function behaves just like the definition of $\lfloor\cdot\rfloor_s$ in Section 2.2. We traverse potentially nested `VChc` values, stopping at each choice to determine whether to keep the left, right, or both branches. Note that the `Just` constructors are part of the built-in `Maybe` type, which represents computations that could fail, such as `lookup` in a map. With the fundamentals out of the way, we can now turn to some simple programming tasks.

3.2 Lifting Functions and Control Structures

Programming in general requires functions (or methods or procedures, depending on your language of choice) and variational programming is no different. In general, supporting variational programming requires a variational function analogue to each plain function that would normally be used. That variational function should operate on variational parameters and behave according to the definition of correctness in Chapter 2.

This is easiest to understand by looking at an example. Suppose we want to define a variational function that checks whether or not an integer is negative. A plain version, relying on some built-in comparison operator `(<)`, might look like this.

```
negative :: Int -> Bool
negative x = x < 0
```

Let us try naively to create a variational version, `vNegative`. The first thing we know is that we need to variationalize the function parameters. That is, we expect our function to take a variational integer and return a variational Boolean value.

```
vNegative :: VChc Int -> VChc Bool
vNegative x = if … then … else …
```

We almost immediately run into several problems. Most obviously, the built-in comparison operator is not defined for variational integers, only plain ones. We will have to define this ourselves. Moreover, we cannot return typical `True` or `False` since those are of type `Bool` rather than `VChc Bool`. We can define a new function, analogous to `(<)`, that performs a less-than comparison on variational integers. We could stub out such a function in the following way.

```
vlt :: VChc Int -> VChc Int -> VChc Bool
vlt (Chc d l r) y       = Chc d (vlt l y) (vlt r y)
vlt (One x) (Chc d l r) = Chc d (vlt (One x) l) (vlt (One x) r)
vlt (One x) (One y)     = One (x < y)
```

While defining this function, experienced functional programmers may recognize that this kind of traversal of the choice structure is a candidate for factoring out into something more general. Successfully doing so eliminates the need to repeatedly define the same recursive traversal in each function we implement going forward. In principle, it is now possible to implement `vNegative` as follows.

```
vNegative x = vlt x (One 0)
```

We will take this opportunity to provide instance definitions for the Haskell `Functor`, `Applicative`, and `Monad` type classes to make these kinds of definitions simpler in the future. The full definitions are available in Appendix A.

```
instance Functor VChc where
  fmap f (One x) = One (f x)
  fmap f (Chc d l r) = Chc d (fmap f l) (fmap f r)

instance Applicative VChc where …

instance Monad VChc where …
```

A full introduction to the concepts underlying these type (constructor) classes is out of scope here. The following intuitive understanding should be enough. The `Functor` type class lets us apply an arbitrary function to all the variant values contained in a choice while preserving the structure. We could, for example, map the function `succ` (which increases a number by 1) across a variational integer and it will be applied to each variant. The `Applicative` instance is a bit more complicated, but in essence gives a way to lift plain values (including functions) to variational ones as well as to apply variational functions to variational values. Finally, the `Monad` class gives us a way to apply functions that create new variants to values that are already variational.

Returning to our intermediate goal of implementing a function that performs a less-than comparison of two variational values, we can now simplify the definition a great deal by making use of the `Applicative` instance.

```
vlt :: VChc Int -> VChc Int -> VChc Bool
vlt = liftA2 (<)
```

The `liftA2` function is a library function that lifts binary functions into any type that is an instance of `Applicative`. It has the type `Applicative f => (a -> b -> c) -> f a -> f b -> f c`. In this case we use it to apply the `(<)` function to types `VChc` types.

This machinery makes it trivial to lift arbitrary plain functions to work on variational values. The built-in Haskell applicative module also provides the functions `liftA` and `liftA3` for unary and ternary functions, respectively, and it is easy to define functions for other arities as necessary.

Since functional programming languages generally use higher-order functions for control flow, we can use this same boilerplate to create variational control flow constructs. For example, given a plain if-then-else function, we can easily define a variational version which will become useful soon.[1]

```
if' :: Bool -> a -> a -> a
if' True  t _ = t
if' False _ f = f

vIf :: VChc Bool -> VChc a -> VChc a -> VChc a
vIf = liftA3 if'
```

This also works for high-order functions such as the venerable `foldr`.

```
vFoldr :: (Foldable f) => V (a -> b -> b) -> V b -> V (f a) -> V b
vFoldr = liftA3 foldr
```

3.3 Generic Data Structure Interfaces

Since we can implement arbitrary functions on variational values, we now turn to how to make use of them to implement variational data structures. A good starting place is the linked list, since it is such a fundamental and useful data structure. As established already, the easiest way to create a variational type is to simply apply our type constructor `VChc`. We can therefore define a variational list as follows, where the square brackets `[]` indicate built-in Haskell lists.

```
type VList a = VChc [a]
```

[1]Note that the name `if'` is used instead of `if` because the latter is a reserved keyword in Haskell.

We can then make use of the same lifting mechanisms to lift list functions such as `head` and `tail`. Note that `fmap` would also work instead of `liftA`.

```
vHead :: VList a -> a
vHead = liftA head

vTail :: VList a -> VList a
vTail = liftA tail
```

Indeed, this is one of the six variational list representations that will be evaluated in Chapter 4. However, we are still missing one core component that would allow us to work with different variational data structure representations and compare them fairly. In order to benchmark the performance of list representations accurately, we need to write a list function just once and have it be polymorphic in the type of lists it acts on. That is, instead of writing six variational list reverse functions, we want to write one that works for all kinds of variational lists. Naturally it is possible to define separate functions for each list representation, but doing so increases the chances of testing them unfairly. For example, a small change or oversight in one function that causes extra misses could overshadow an otherwise more efficient representation.

One way to achieve this is to design a variational list interface which allows us to define functions that only depend on the interface without access to the implementation details of the specific data structure definitions.[2]

In Haskell, the way to define a generic interface like this is with a type class. Type classes, as we use them, are essentially like interfaces in Java. We can define such a type class as follows.

[2]To a degree, we are conflating data structures with abstract data types (ADTs) here. While linked lists are data structures, variational linked lists are better understood as ADTs. We will continue to refer to them as data structures for the sake of simplifying the language.

```
class VList (vl :: * -> *) where
  vCons :: VChc (Maybe a) -> vl a -> vl a
  vHead :: vl a -> VChc (Maybe a)
  vTail :: vl a -> vl a
  vNull :: vl a -> VChc Bool
  vEmpty :: vl a
```

This type class defines the core functionality we expect from a variational list, such as the ability to add elements and to deconstruct lists into their head and tail components. Even without creating instances we can use this interface to implement some new list functionality. For example, we can define a function to produce a singleton list.

```
vSingleton :: (VList vl) => VChc (Maybe a) -> vl a
vSingleton x = vCons x vEmpty
```

Unfortunately, we run in to a problem with some more complex functions. We will use list reversal as an example to illustrate. Consider the following incomplete definition.

```
vRev :: (VList vl) => vl a -> vl a
vRev xs = vIf (vNull xs) vEmpty …
```

Even at this incomplete stage the definition already contains a type error. We want to pass `vEmpty` to `vIf`, which is a variational list, but that function only accepts choices with the type `VChc a`.

What we need instead is to come up with a type that describes variational values more generally, including both applications of our variational type constructor as well as any kind of variational list. This, too, can be achieved with a new type class. Since our model of variation is the choice calculus, our type class essentially describes values that are choice-like, i.e., support the creation of choices and the lifting of plain values.

```
class V (v :: * -> *) where
  chc :: Dim -> v a -> v a -> v a
  one :: a -> v a
```

The two functions required to create an instance of the type class `V` are intended to be reminiscent of the two `VChc` constructors. This makes it trivial to create the `VChc` instance.

```
instance V VChc where
  chc = Chc
  one = One
```

Creating an instance for variational lists is more complex. We need to demonstrate that `V` is a superclass of `VList`, i.e., every `VList` is a `V`. To do this we need to provide an implementation for `chc` and `one` based only on the functions available in the `VList` type class. The complete definition is reproduced in Appendix A.

```
instance (VList vl) => V vl where
  chc d l r = if all vNull l && all vNull r
              then vEmpty
              else vCons (chc …) (chc …)
  one = vSingleton . one . Just
```

We have successfully defined a type that generalizes both the `VList` and `V` types. With it, can return to the `vIf` definition and modify it to work with values of this type instead of just `VChc`.

```
vIf :: (V v) => VChc Bool -> v a -> v a -> v a
vIf (Chc d l r) t f = chc d (vIf l t f) (vIf r t f)
vIf (One b)     t f = if b then t else f
```

Now, finally, we can implement a variational list reversal function.

```
vRev :: (VList vl) => vl a -> vl a
vRev vl = vIf (vNull vl)
          vEmpty
          (vCat (vRev (vTail vl))
                (vSingleton (vHead vl)))
```

This function works just like the plain list reverse function. At each iteration we check whether the variants are empty, indicating that we have finished traversing them. If so, we just return an empty (variational) list. If not, then we concatenate the head element onto the back of the recursively reversed list (using `vCat`, see Appendix A).

In addition to offering potentially useful features, this generic interface also enables meaningful benchmarks by ensuring that functions can be written generically for each list representation. As mentioned above, this reduces the risk of uneven performance due to details such as cache misses or superfluous allocations. We now move on to designing specific variational linked list representations in Chapter 3.

Chapter 4: Variational Linked Lists

This chapter describes variational lists. As mentioned in Chapter 1, variational lists are structures which encode potentially many plain lists. However, there is no canonical implementation of a variational list. Chapter 3 introduced one way of designing such a structure, but there are many potential implementations, and it is not immediately obvious which perform well and lead to less complex definitions and extensions. Here we introduce six variational list designs and compare them for performance and efficiency across a number of areas, eventually drawing conclusions about the design of variational data structures in general.

This section will also assume familiarity with the variational programming approach described in Chapter 3, as well as build upon it with some new additions. As in the previous chapter, some code definitions here are shortened or make use of some syntactic shortcuts to enhance readability. When noted, refer to Appendix A for the full listings.

4.1 Named Choices and Choice Trees

Of the six list representations detailed here, five make use of the choice calculus as a representation of variation, as introduced in Chapter 2. When applying the choice calculus, this chapter will make use of the data type definitions from Chapter 3. The sixth makes use of a tag-based approach which we will detail next.

It may seem intuitive to use a map directly instead of choices, as in the brief example in Section 1.1. However, such a map will grow exponentially with the number of configuration options and is therefore impractical for real scenarios.

4.2 Tag Formulas and Tag Maps

Since this chapter will also make use of tags as a model of variation, this section will provide more detail on the approach. Recall that the premise is to invert the direction of access to variants when compared with the choice calculus. That is, instead of decisions mapping to variants, variants map to their configurations.

Each plain value is tagged to indicate which variants include it. For example, the value 1^A indicates a value `1` which is included when tag (or dimension) A is selected, but not otherwise. Like the choice calculus, this relies on some labeling scheme in which a selection of labels forms a decision that identifies the desired variant. This approach can be extended to allow for tag formulas, analogous to boolean formulas. The advantages of this extension are twofold. First, identical plain values need not be duplicated. Instead of $[1^A, 1^B]$ we can simply write $[1^{A \vee B}]$. Second, it allows more complex relationships to be defined such as values which are only included when multiple selections are made. For example, $1^{A \wedge B}$ indicates that the plain value `1` is only included when both A and B are selected. With tag formulas, a variational value can be implemented as a map from plain values to tag formulas. We call this representation a *tag map*. A more complete definition is avilable in Appendix A.

```
data VTag a = VTag (Map a TagFormula)

instance V VTag where …
```

Both choice trees (refer to Chapter 2) and tag maps can serve as underlying implementations for a variational type constructor `V`, as discussed in Chapter 3. For example, the type `V Int` would represent variational integers and could correspond to either choices of integers or tagged integers. We distinguish between these two approaches, where necessary, by referring to them as `VChc` and `VTag`, respectively. In cases where the distinction is not necessary, we simply use `V`.

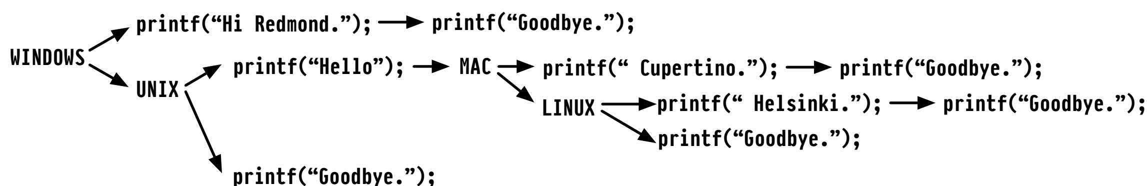


Figure 4.1: The unabbreviated visual notation used to demonstrate the general shapes that variational lists take.

It is worth noting that formulas can also be used as a condition for choices, as demonstrated in Hubbard and Walkingshaw (2016).

4.3 Visual Notation

In Section 4.4, we will make use of a compact visual notation when presenting examples. This notation is based on graphs whose nodes represent data and dimensions/tags and whose edges represent pointers in the data structure. In cases where both `VChc` and `VTag` can be used, we only show the structure of choices for brevity's sake. This notation, applied to the map of lists example from the Introduction, is shown in Figure 4.1.

Every binary choice is represented as a node with two outgoing edges corresponding to the choice's two alternatives. In general, rather than rendering this choice tree everywhere it occurs, we employ syntactic sugar which depicts each choice tree structure as a single, unlabeled node. The outgoing edges are labeled with configurations. We also add color to help distinguish particular variants. In this example, we would condense the graph to the one in Figure 4.2.

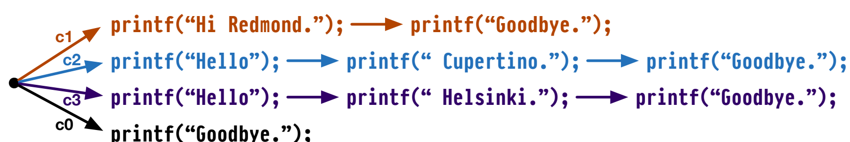


Figure 4.2: Box list. A more concise representation of the list shown in Figure 4.1. Every variant list is stored in its entirety and nothing is shared.

4.4 Variational List Representations

We can now turn to exploring the specific designs of variational linked list representations. While there is only one way to apply a type constructo to produce a typical variational value (such as integers or strings) there are several different approaches to do so with parameterized types such as data structures.

More specifically, we could simply apply the type constructor at the *top level*. That is, just as we might have a `V Int` for a variational integer, we could also define a `V [Int]` for a variational list of integers. But we could also distribute the variational type constructor *parametrically* throughout the list, giving us a list of variational integers `[V Int]`. Finally, we could take a *recursive* approach, where rather than reusing the `V` constructor, the variation definition is woven directly into the recursive list definition. This design decision has been identified in related work, as well. Liebig et al. (2013) suggests keeping variability as local as possible, for example. Similar ideas regarding localizing variation and the notions *late splitting* and *early joining* are expressed by both Kästner et al. (2012) and Apel et al. (2013).

By systematically combining the different implementations of `V` together with these approaches for integrating the variation into the list structure, we obtain six candidate variational list representations, namely the box, pointer, choice, tag, suffix, and segment lists. An overview of how each of the following representations is categorized is shown in Table 4.1.

<table>
<tr><th></th><th></th><th colspan="3">Use of V</th></tr>
<tr><th></th><th></th><th>Top Level</th><th>Parametric</th><th>Recursive</th></tr>
<tr><td rowspan="2">V</td><td>VChc</td><td rowspan="2">box, pointer</td><td>choice</td><td>suffix, segment</td></tr>
<tr><td>VTag</td><td colspan="2">tag</td></tr>
</table>

Table 4.1: A categorization of variational list representations according to their use of variational type constructors and how they are integrated into the list structure.

For each representation, we show the type definition, summarize the main idea, and explain how it impacts the implementation of typical list functions. When we talk about a variational version of a common function, it should be understood to mean the function obtained by *variationalizing* all of the inputs and outputs of that function. For example, where the plain version of `head` takes a plain list and produces a plain value, variational `head` takes a variational list as input and produces a variational value as output. For more details about programming with variation, refer to Chapter 3.

4.4.1 Box Lists

A straightforward variational list representation is obtained by applying the generic variational type constructor `V` to a list. Since this approach does not have access to the individual variant lists and stores each of them as whole, we call these lists *box lists*. The variation representation essentially provides slots into which only complete lists can be inserted. We use the generic `V` rather than `VChc` or `VTag` specifically, because either is adequate here.

```
type BList a = V [a]
```

The list in Figure 4.2 actually is the box list containing the conditional source code from the Introduction. This representation is essentially a brute-force approach which, while inefficient, does have advantages. Perhaps most importantly, it is trivial to lift plain funtions to their variational

analogues for box lists. We simply apply the original plain function to each variant in a brute-force approach. This is exactly the task achieved by the `Functor` instance for `V` shown in Appendix A.

With the functor `fmap` function, we can define many lifted variational functions by just applying it to plain, built-in functions such as `head` and `sort`.

```
vHead :: BList a -> V a
vHead = fmap head

vSort :: (Ord a) => BList a -> BList a
vSort = fmap sort
```

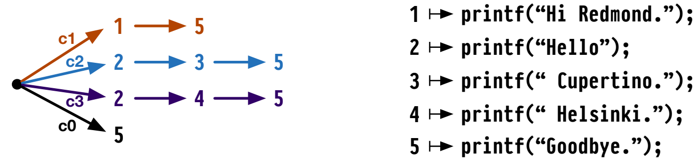
Generally, brute-force approaches are to be avoided since they do not allow any sharing of data. This has already been the subject of work on variationalizing interpreters (Kästner et al., 2012; Kim et al., 2012; Nguyen et al., 2014) as well as in dealing with privacy policies (Austin and Flanagan, 2012; Austin et al., 2013; Yang et al., 2012). However, since they are so straightforward, box lists serve as a base case for comparing the remaining representations. We will compare approaches to sharing in Section 4.5 and discuss performance in general in Section 4.6.

4.4.2 Pointer Lists

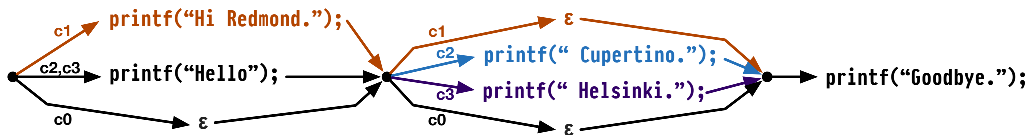
We can extend the box list by a level of indirection, which allows common elements to be shared but at the cost of increased overhead. We refer to lists using this extension as *pointer lists*. We apply `V` to keys, which index into a map storing the actual list elements, ensuring no duplication.

```
type PList a = (Map Int a, V [Int])
```

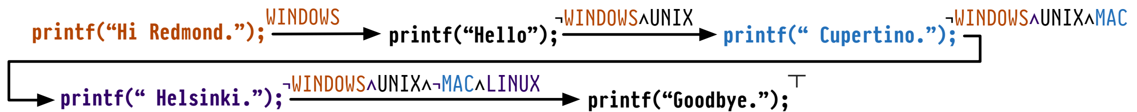
An example pointer list for the same variational source code data is shown in Figure 4.3a. The implementation of pointer list operations is similar to the box list case, since it is a direct extension,



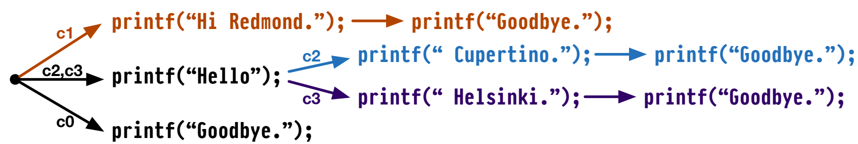
(a) Pointer list. Every element is stored in a hash table and only the indices are stored in the list.



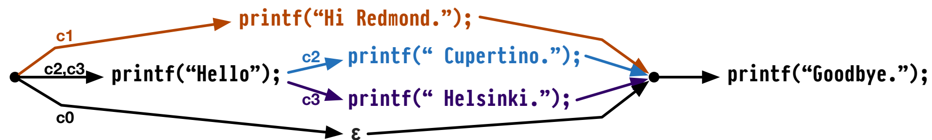
(b) Choice list



(c) Tag list



(d) Suffix list



(e) Segment list

Figure 4.3: Example variational lists containing conditional pre-processor annotated source code from the Introduction making use of the visual notation introduced in Section 4.3.

except that we often need to compose them with a lookup operation which extracts values from the map.

```
vLookup :: V Int -> Map Int a -> V a
vLookup v m = fmap (‘lookup‘ m) v

vHead :: PList a -> V a
vHead (m,vi) = vLookup (fmap head vi) m
```

We can sometimes avoid (or at least delay) performing the lookup operation in cases where we do not actually need to extract the values form a list. For example, if we just want to reverse the elements, the values pointed to are irrelevant.

```
vReverse :: PList a -> PList a
vReverse (m,vi) = (m, vReverse vi)
```

This approach adds overhead, especially since Haskell maps are implemented as trees, but enables sharing of all common elements in all variants. This complete sharing is unique in the representations considered here, which is discussed further in Section 4.5. Here we applied indirection to box lists, but it is worth noting that it could also be applied to any of the following representations.

4.4.3 Choice Lists

Since a list is itself a generic type constructor, we can move the `V` type constructor to its argument type and apply variability not on the level of whole lists, but on the level of elements. That is, we can swap the order in which we apply the type constructors so that instead of a variational list of elements we have a list of variational elements. This gives us a single large list in which each element is itself a choice tree of variational list elements.

Unfortunately, such an approach fails immediately because variant lists of different lengths cannot be modeled. Regardless of what decision is used for selection on a choice tree, it will produce a value. In order to model list variants with different lengths we need to have some choice trees which are only relevant to a subset of the variants, which is not possible.

Our workaround is to allow certain choice tree leaves to be empty. That is, we use yet another type constructor, the Haskell `Maybe` type, as part of our definition. Now we have a list containing variational elements, where the leaves of the choice trees are values which may or may not exist. The definition looks as follows.

```
type CList a = [VChc (Maybe a)]
```

These are called *choice lists*, and an example is shown in Figure 4.3b. As discussed in Section 4.5, choice lists are the first representation in which there is not a single canonical representation for a given set of data. That is, for any set of data, there are many choice lists we could construct to represent it. Consider the trivial example of the choice list representing the variants lists `[1,2]` and `[3]`. Either of the following representations is valid (there are actually infinitely many more since we can pad arbitrary empty values.)

$$[A\langle \texttt{Just 1}, \texttt{Just 3}\rangle,\ A\langle \texttt{Just 2}, \texttt{Nothing}\rangle]$$

$$[A\langle \texttt{Just 1}, \texttt{Nothing}\rangle,\ A\langle \texttt{Just 2}, \texttt{Just 3}\rangle]$$

Another complication is that all of the typical list operations will now need to perform some extra work to deal with the `Maybe` types. For example, consider how we might implement a function `vHead` which returns the first element in each list variant. We might be tempted to write something like the following.

```
vHead :: CList a -> VChc (Maybe a)
vHead (x:_) = x
```

At a glance this might seem to work, but in fact is not a correct implementation. Recall the definition of a correct variational function from Section 2.3. It says that a variational function followed by selection should produce the same result as selection followed by the analogous plain function. Now consider the following variational list again:

$$[A\langle \texttt{Just 1}, \texttt{Nothing}\rangle, A\langle \texttt{Just 2}, \texttt{Just 3}\rangle]$$

The two variant lists are `[1,2]` and `[3]`. Neither is empty, and applying the plain list head function to each variant gives us `1` and `3`, respectively. The version of `vHead` we have defined above returns the first element in the list, which is $A\langle \texttt{Just 1}, \texttt{Nothing}\rangle$. If we then select the right variant of dimension A, we can see that we get `Nothing` when we expect to get `3`. Therefore, the function is not correct.

A correct implementation needs to fill in `Nothing` values with `Just` values that may occur later in the list. One (inefficient) implementation follows.

```
vHead :: CList a -> VChc (Maybe a)
vHead [] = One Nothing
vHead (c:cs) = fillNothing c cs

fillNothing :: VChc (Maybe a) -> CList a -> VChc (Maybe a)
fillNothing (Chc d l r) cs = Chc d (fillNothing l cs) (fillNothing r cs)
fillNothing (One Nothing) (c:cs) = fillNothing c cs
fillNothing x _ = x
```

Whenever we encounter a `Nothing` value we simply fill it in with the (recursively filled) entire next choice in the list. This works because, although we are inserting superfluous elements, choice domination (see Section 2.2) will ensure that we only ever access the relevant elements. If the leaf already contains a value, we do nothing.

This implementation, and any other correct implementation, will need to potentially traverse the entire outer choice list to ensure that it finds a value (if one exists) for every variant. This

fundamentally diverges from what we expect about the runtime complexity of a list head function, which would normally be a constant time operation. Similar issues arise in many other cases, too. Section 4.6 discusses this aspect further.

Not all functions are complex to implement on choice lists, however. In fact, some functions are even simpler than they are for the base case box lists. Consider, for example, reversing a choice list. When we are not inspecting the values contained in the list, we do not need to worry about filling in `Nothing` values. This means that reversing a choice list simply requires reversing the choice tree elements. That means we can simply use the built-in list reverse function.

```
vReverse :: CList a -> CList a
vReverse = reverse
```

This simplicity is appealing and perhaps helps to counteract the added complexity of operations like `vHead`. The impact this has on performance will be disussed in Section 4.6.

4.4.4 Tag Lists

Tag lists use the same parametric application of `V` as choice lists, but with the tag formula implementation of `V` instead of choices. Tag formulas are paired with each element in a plain list, indicating which variants include those elements and which do not. See Figure 4.3c for a tag list example.

```
type TList a = [VTag a]
```

Somewhat unsurprisingly, since they share a similar structure, tag lists suffer from many of the same problems as choice lists. Any correct `vHead` function will once again not be a constant time function. Just as we had to traverse the choice list to fill in `Nothing` values, we now have to traverse the tag list to find values tagged for inclusion for every variant. Consider the list $[1^A, 2^A, 2^B]$. In order to find the first value for the B variant we need to scan all the way to the last element.

In general, we first need to extract a set of configurations from the list in order to know which variants we need to ensure we have handled. Then, for each, we have to scan the list solving the tag formulas until we encounter one that satisfies the current configuration. A particularly clever algorithm might be able to find the head element for multiple variants at once, but it is not clear that the effort would lead to a function that performs better in practice. Moreover, Haskell's lazy evluation means that we will only check for those values which we really use. One implementation is reproduced in Appendix A, which assumes a function `solve` for SAT solving.

Some functions get much more complex than this, even. Consider how we might implement a fold, for example. Since the tag map values we encounter as we traverse a list have no structure, we have to check the annotated tags and perform Boolean satisfiability computations to determine how to proceed at every step.

Again like choice lists, tag lists have a major advantage in that many plain list operations work as is. For example, the typical `reverse` and `sort` function work just fine.

```
vReverse :: TList a -> TList a
vReverse = reverse

vSort :: TList a -> TList a
vSort = sort
```

That we are able to use the built-in `sort` makes tag lists even simpler than choice lists, which we cannot sort without reordering the values contained inside the choice trees. Since tag lists contain only single values as elements we can just rely on the way Haskell compares tuples, which is by their first element. When the overall tag list is reversed or sorted, so is any particular variant.

This demonstrates that even simple, seemingly transparent changes to the way we build our variational data structures (in this case the implementation of `V` inside our list) can have far-reaching and non-intuitive effects, both good and bad.

4.4.5 Suffix Lists

As an alternative to using traditional linked lists as a base, variation can be integrated directly into the structure of the list. One such approach, first demonstrated in Erwig and Walkingshaw (2013), is to extend traditional cons lists with a nesting constructor that uses `V`. We call these *suffix lists*.

```
data SuffList a = Cons a (SuffList a)
                | Nil
                | Nest (VChc (SuffList a))
```

See Figure 4.3d for a suffix list containing example data. One thing to note about this approach is that it leads to tree-shaped structures rather than graphs or lists, and once two variants have split apart they will never be merged again. Because of this, suffix lists share common elements among variants in the prefix of variational lists. From the first element up until the first point of variation, we have something indistinguishable from a traditional cons list. Only after the first point of variation does the amount of sharing start to potentially decrease. The actual sharing achieved depends on the data being stored in the lists, and is therefore difficult to reason about in general. More details follow in Section 4.5.

As is becoming a theme, this representation encounters tradeoffs when implementing list operations. Generally, those which iterate through the list but do not modify the structure or reorder anything perform well on suffix lists. For example, `vHead` does not need to traverse the whole list as was required for choice and tag lists.

```
vHead :: SuffList a -> VChc (Maybe a)
vHead (Cons x _) = One $ Just x
vHead Nil = One Nothing
vHead (Nest v) = fmap vHead v
```

Conversely, functions that change the structure of the list such as `sort` and `reverse` are more challenging. The latter is a clear example. Since suffix lists are actually trees, which have one root

node and typically many leaf nodes, it is not possible to simply reverse the pointers. Instead, we need to find all of the leaf nodes and combine them into a root node, and then work backwards. Compared to the simple one-pass list reversal operations that work for other lists, this is inefficient. One implementation is shown in Appendix A. As with many of the other representations, the suitability of suffix lists depends on what operations are performed on it.

4.4.6 Segment Lists

The final variational list representation we will discuss here is the *segment list*, first proposed in Allen (2014). The idea of this representation to encode variational lists as a sequence of segments, where a segment is either a named choice of nested lists or a sequence of plain elements. See Figure 4.3e for an example segment list.

```
newtype SegList a = VL { segments :: [Segment a] }
data Segment a = Elems [a]
               | Split Name (SegList a) (SegList a)
```

The segment list is similar to the choice list, except more compact. We can reduce the amount of boilerplate necessary to encode sequences of values that do not vary by keeping them together in a built-in list. Empty elements, when they are necessary, can then be modeled by an empty list.

Because of this similarity to choice lists, it shares most the tradeoffs inherent to the parametric application of variation: We generally need to traverse the entire list in order to implement `vHead` function, and the extra structure makes for a somewhat complex definition. We have shown just the structure here, but the full definition is in Appendix A.

```
vHead :: SegList a -> VChc (Maybe a)
vHead (VL []) = …
vHead (VL (Elems [] : xs) = …
vHead (VL (Elems (a:_) : _)) = …
vHead (VL (Split d l r : xs)) = …
```

On the other hand, reversing the list is relatively simple since reversing each segment is sufficient to reverse each variant list.

```
vReverse :: SegList a -> SegList a
vReverse (VL segs) = VL $ reverse (fmap reverseSeg segs)

reverseSeg :: Segment a -> Segment a
reverseSeg (Elems xs) = Elems $ reverse xs
reverseSeg (Split d l r) = Split d (vReverse l) (vReverse r)
```

Having introduced the six representations that are discussed in this dissertation, we can now turn to evaluating how they do in terms of exploiting sharing and their practical performance.

4.5 Sharing

Sharing in the context of variational lists is valuable since it saves space and, in particular, since it helps to avoid unnecessarily duplicating expensive computations on common list elements. Variational lists provide relatively little value if they simply wrap a set of plain lists. Sharing is one major opportunity in which variational lists can offer an optimization above an ad hoc or brute-force approach.

In the representations discussed here, the sharing offered can be classified into one of four categories. Roughly in increasing order of power, these are (1) no sharing, (2) prefix sharing, (3) join sharing, and (4) total sharing. These are explained in turn, each with specific details of the representations that are categorized as such.

No Sharing Of the list representations described here, only the box list offers no sharing at all. It stores each variant completely independently and thus duplicates not only storage costs for each element that could, in theory, be shared, but also offers no mechanism to share computations on common elements.

Prefix Sharing Offering an incremental improvement over no sharing at all, prefix sharing enables sharing common elements only in list prefixes, that is, from the first element up until the first point of variation between two variants. This kind of sharing occurs in the suffix list representation discussed in Section 4.4.5.

Consider the example suffix list shown in Figure 4.3d. There `printf("Hello");` is shared, but `printf("Goodbye.");` is not. Prefix sharing is somewhat unpredictable, since it depends on the shape of the data being stored.

Join Sharing Join sharing is defined by the ability to share common list elements by aligning the separate variants so that common elements occur in the same position. This is the approach taken in choice, segment, and tag lists. Join sharing is sometimes more effective than prefix sharing and never worse.

Consider the example shown in Figure 4.3b. There, we inserted empty `Nothing` elements, rendered as ϵ in order to allow the variants to share the last token `printf("Goodbye.");`. Padding the lists with empty elements in this manner can increase the amount of sharing substantially.

Unfortunately, finding an alignment that maximizes sharing is difficult and the subject of much research in its own right, particularly in molecular biology (Carrillo and Lipman, 1988). Furthermore, a maximal alignment can be inefficient if too many empty values are padded. Consider aligning the list variants `[1, 2, …, 5000]` and `[5000, 5001, …, 10000]`. The maximum alignment ensures that the value `5000` is shared between both variants, but this results in needing to double the length of

Representation	Sharing Type			
	None	Prefix	Join	Complete
Box	×			
Pointer				×
Choice			×	
Tag			×	
Suffix		×		
Segment			×	

Table 4.2: A summary of the type of sharing offered by each representation discussed in this work.

the variational list structure with padded empty elements. This is almost certain to have a negative impact on performance.

Complete Sharing Pointer lists are the only variational lists we have presented which can share every common element regardless of the number of variants or the positions of particular values. Each element is stored exactly once. Although optimal in a sense, there is inherent overhead.

A summary of the kinds of sharing exhibited by our representations is shown in Table 4.2.

4.6 Performance

The runtime complexity of an algorithm is expressed as a function that depends on the size of its inputs. In the case of variational data structures, runtime not only depends on the total size of the input, but also on the variability (expressed, for example, as the number of variants) and exploitation of redundancy in the data (expressed, for example, by the degree of sharing).

As an example, consider finding the last element in a plain linked list. This takes $O(n)$ time where n is the length of the list. However, the variational analogue has to deal with potentially many variants of different lengths.

We opt to use $\hat{n}$ to denote the maximum length (or size, generally) of any variant list and then write $O(v\hat{n})$ to express that the runtime of an operation is linear with respect to the number of variants and the length of the longest one. While a tighter bound is sometimes possible, this is more succinct than using summation notation. Generally, if a function `f :: [a] -> [b]` has complexity $O(g(n))$ where n is the length of the input list, then the lifted version `vF :: V [a] -> V [b]` has time complexity $O(vg(\hat{n}))$.

Since the complexity can depend on the amount of sharing in a variational list, we use the variable S to indicate a metric depending on the redundancy in the data and the corresponding degree of sharing achieved by the data structure. We define $S = 0$ to mean no sharing (but arbitrary data redundancy) and $S = 1$ to mean completely redundant data and total sharing, i.e. every element is stored once. This metric accounts for actual sharing, rather than potential sharing.

For example, suppose we have two variant lists `[1,2,4]` and `[3,2,1]`. Under prefix sharing the ratio is $S = 0$. For a list representation with join sharing, we have $S = \frac{1}{3}$, since only the `2` will be shared. With a pointer list, we achieve $S = \frac{2}{3}$ since `1` and `2` can be shared.

4.6.1 Specific List Operations

Unsurprisingly, sharing can lead to improved performance. What is surprising, however, is that some representations are actually slower than a brute-force approach. We have seen this in the `head` function, which requires some representations to potentially traverse the entire length of the list. This effectively turns a constant time operation into one linear with respect to the length of the list. This kind of asymptotic slowdown can occur (for some representations) whenever the operation relies on the assumption that one variational element is equivalent to one plain element, i.e., that it contains one plain value for all variants.

Conversely, consider `reverse`. Both box lists and pointer lists have worst-case complexity $O(v\hat{n})$ and no improvement in the best case. This is not surprising for box lists, since we have no sharing, but the sharing offered by pointer lists does not provide any improvement because we still need to reverse all the copies of pointers.

For the other four representations, however, we need to factor in the amount of sharing that occurs as part of the measure. In these representations sharing actually reduces the number of elements we need to reverse. Using the measure S this gives us a runtime complexity $O(\hat{n} + \hat{n}(v - Sv)) = O(\hat{n}(1 + v(1 - S)))$ for choice, segment, suffix, and tag lists. Plugging in values for S between 0 and 1 gives back the expected results. When no sharing actually occurs (that is, $S = 0$), we end up with $O(v\hat{n} + \hat{n}) = O(v\hat{n})$. This is the same as the worst-case for box and pointer lists, which is correct since we do not suffer from the same asymptotic slowdown we did with `vHead`. On the opposite extreme, if variants are identical and only represented once, then $S = 1$. In that case we have runtime complexity $O(\hat{n})$. Again, this is intuitive because we only need to reverse each unique element once, which is the same as reversing a non-variational list. Note that in this case $\hat{n} = n$ since all the variants are the same. Finally, we can see that in more realistic situations where S lies between these two extremes, increases in sharing produce increasingly better performance. For example, when $S = 0.5$, then we have $O((1 + 0.5v)\hat{n})$. Although this is constant factor whenever $v \ll \hat{n}$ and easy to dismiss theoretically, the impact on pragmatic performance might still matter.

4.6.2 Benchmarks

While the computational complexity measures of operations are interesting, they can sometimes fail to fully portray the realities of practical performance. This is the case with our results.

Our benchmarking was performed on files from popular C/C++ projects for which the source is freely available, including Python, GCC, Gecko/Firefox, Linux, and Stockfish.[1] We selected files in each of those source code bases which contained a near-median number of CPP conditional compilation macros. Although we believe these files provide a good overview, it is not possible to choose files that precisely represent the general case, since the use of the preprocessor varies tremendously based on the needs and coding style of a project (Ernst et al., 2002).

In files with no or minimal CPP annotations, we would expect performance comparable to the base case for all representations, which is not insightful. On the other hand, files with the highest number of annotations are atypical and sometimes even too slow to benchmark reasonably.

The benchmarks reported here are based on a list filtering operation, a generally useful list operation that requires traversing the entire list. A filtering operation represents a realistic operation to be performed on a source code file, such as finding all fully configured sections in a file which satisfy some predicate (e.g., all parts of a file that apply a particular function or redefine a particular variable). Refer to Appendix A for the complete definition of the `vFilter` function.

We used the Criterion[2] package to benchmark our Haskell-based implementation of these list representations on 2.8Ghz Intel Core i5 with 16GB of memory. We use the default settings which run each benchmark 1000 times in order to ensure a dependable result in the presence of factors like garbage collection. Table 4.3 shows the benchmarked result for applying `vFilter` to one file from each of the five repositories with a predicate to filter out assert statements. It also includes information about the number of lines of code (*LoC*) and lines of relevant CPP annotations (*LoCPP*) for each of the files.

The runtimes show some unexpected results. First, we can see that not every representation is able to outperform the box list base case. Most notably, both pointer lists and tag lists occasionally

[1] `www.python.org`; `gcc.gnu.org`; `www.mozilla.org`; `www.kernel.org`; `https://stockfishchess.org`
[2] `hackage.haskell.org/package/criterion`

List Representation	`parser.c` (Python)	`bitmap.c` (GCC)	`FilterProcessing.cpp` (Firefox)	`raw.c` (Linux)	`bitboard.h` (Stockfish)
LoC/LoCPP	488/22	2280/16	262/26	1369/21	334/9
Box	1220	894	336	3610	1680
Pointer	341000	12800	321	3490	55600
Choice	1770	486	436	909	357
Suffix	1090	465	363	823	620
Segment	22	462	41	377	152
Tag	11000	438	2870	11300	6830

Table 4.3: Benchmark results for applying `vFilter` to five CPP-annotated source code files for each of our list representations. Runtimes given in μs.

perform significantly worse than the base case. Conversely, choice and suffix lists frequently (but not always) offer a noteworthy improvement over the base case. However, from the results, it is clear that the segment list representation is the top performer, beaten only in one benchmark and even then only by a small margin.

As mentioned, both choice and segment lists suffer from the potential for asymptotic slowdown in functions like `vHead`. The benchmarks here seem to suggest that this does not manifest in practice in at least some circumstances and that we should probably rely more on benchmarks than on theoretical performance.

4.6.3 Redesigning the Interface

There is one additional optimization that leads to interesting results. The current implementation makes use of the `VList` type class to support programming generically across the six different list representations. By making an instance of the type class for each list representation, we were able to write the `vFilter` and other functions just once.

While this has advantages (refer to Chapter 3), not all list representations are equally well suited to being accessed through the core variational list type class. We have already established how some

List Representations	parser.c (Python)	bitmap.c (GCC)	FilterProcessing.cpp (Firefox)	raw.c (Linux)	bitboard.h (Stockfish)
Pointer	1190	1000	420	3980	1940
Choice	176	456	169	473	158
Segment	173	451	168	473	162
Tag	172	490	166	470	161

Table 4.4: Benchmarks for applying `vFilter` on some list representations using a more granular approach. All runtimes given in μs. Compare to Table 4.3.

list representations have an asymptotic slowdown for `vHead`, for example, which is the main way of accessing elements through the type class. By changing the way we structure the genericity to make the constraints more granular, we can provide more efficient implementations for many of the list operations. For example, instead of a filter function based on the potentially slow `vHead` and `vTail` functions, we can define the following.

```
class VFilter vl where
  vFilter :: (a -> Bool) -> vl a -> vl a
```

Then we can instantiate this type class for all of the list representations, allowing us to directly manipulate their internal structure rather than working through a generic interface. A partial example for choice lists follows, and the complete definition is in Appendix A.

```
vFilterCList :: (a -> Bool) -> CList a -> CList a
vFilterCList p cs = fmap (chcFilter p) cs

chcFilter :: (a -> Bool) -> VChc (Maybe a) -> VChc (Maybe a)
chcFilter = ...

instance VFilter CList where
  vFilter = vFilterCList
```

We re-benchmarked this more granular use of type classes on the most inconsistent representations, the results of which are shown in Table 4.4. Performance is now generally improved and more

consistent. Not all the news is good, however. By comparing the two tables, we can see that in three of the five cases, the type class design actually worsens the performance of the segment lists. In the remaining two cases performance is more or less unchanged. Keep in mind, also, that while this approach seems to provide better performance, it also requires separate implementations for a great number of functions, giving the programmer more work to do in order to extend the library of list operations. Moreover, it clutters the type signatures of functions so that instead of just a `VList` constraint, we can sometimes have a long list of constraints for specific functionality.

4.7 Conclusions

Our exploration of variational list data structures has two main take-away messages. First, we have learned that the segment list representation (paired with the generic variational list interface) outperforms our other representations. This suggests that the best approach for a practical variational list library would be one based only on segment lists with no real need for the notion of a generic `VList` that can be instantiated to several different concrete types. This is somewhat surprising given that we know some segment list operations have asymptotically slower worst-case efficiency compared to even box lists. It could be that this worst case does not manifest frequently in practice and certainly it depends on exactly how the segment lists are constructed. If we construct them with relatively few holes, they will perform better. Additionally, we have only benchmarked one kind of list operation. While we believe it represents a good general case, some scenarios could potentially lead to different results. As a corollary, we recommend that designers of other variational data structures perform pragmatic benchmarking because theoretical worst-case analysis may not align with the results.

Second, we have learned a couple of lessons about designing variational data structures in general. Through the general failure of the pointer lists, we have learned that hash-consing (at least in pure

languages) is likely not worth the extra overhead. Hybrid approaches such as storing the indices in a segment list offer no improvement.

We have also seen that even though our segment lists ultimately outperformed everything else, many other representations saw significantly improved performance after reducing the genericity of the interface. This suggests that, in general, generic data structure interfaces should be used sparingly and with caution. If we were to move on to implementing variational trees or graphs and were unable to find a representation that clearly outperforms the others, we would expect the best approach to be supporting multiple representations and having one type class for every operation that is supported.

We can notice that there is generally no obvious best variational list implementation. While segment lists performed best in our benchmark, the implementation of list operations on these lists is sometimes quite complex, often dealing with mutually recursive functions and nested pattern matching. While this may be an acceptable tradeoff for expert programmers, it could also make writing and debugging programs that make use of these lists unnecessarily difficult.

Finally, a trend that will run throughout this dissertation begins to emerge here, namely that there exists a tension between efficiency and expressiveness in variational structures. Recall, for example, the original design of choice lists before adding the `Maybe` type in Section 4.4.3. Those lists were lacking severely in expressiveness, being unable to represent variational lists whose variants have different lengths. However, they are simple to program with, offer join sharing, and perform extremely well even when accessed with `vHead` and `vTail`. The final representation of choice lists with the `Maybe` type, conversely, is asymptotically less efficient but is able to represent many more variational lists. We will revisit this theme in later chapters.

Chapter 5: Dependent Types for Variational Programming

This chapter discusses one possible extension to variational programming and variational list approaches discussed in Chapters 3 and 4, respectively. Both chapters describe an approach that is flexible, but ultimately inefficient. By using dependent types to restrict the structures of variation allows us to reduce some of these inefficiencies. The code snippets in this chapter are written in Idris, which was chosen for its syntactic similarity to Haskell. Longer Idris definitions will be presented in Appendix B.

5.1 Inefficiencies in Variational Programming

Close examination of the variational programming approach described in Chapter 3 reveals at least one major inefficiency. Recall the `Functor` and `Applicative` instances that were used to lift plain functions to work in the variational case. For a binary function lifted in this way, such as the addition of variational integers, the `Applicative` instance will first traverse the choice tree of the left operand and, each time it reaches a leaf, will then traverse (with `fmap`) the entire right choice tree. For example, consider adding the following two variational numbers.

$$A\langle B\langle 1,2\rangle, B\langle 5,6\rangle\rangle$$

$$B\langle A\langle 3,8\rangle, A\langle 0,10\rangle\rangle$$

Implemented as described in Chapter 3 addition will proceed as follows. The first choice is traversed until we reach the value 1, then we use `fmap` to apply `(+ 1)` to each value in the second

choice. We then return to traversing the first choice and navigate to the value 2. Then `(+ 2)` is again mapped across every value in the right choice. There are two inefficiencies at work currently.

First, we are actually applying the function to values that are irrelevant. Each leaf is replaced by a copy of the entire second choice (this is not just a pointer to the second choice since we are actually modifying each leaf value, forcing the entire tree to be copied.) Fortunately, this is easy to avoid by adding a selection operation to the `Applicative` instance so that the extra elements are removed. This approach is not without a downside however, as the selection process itself requires traversing the choice tree and performing map lookups to determine which branches to keep and which to eliminate, which may not always be a desirable tradeoff.

Second, we have to start the traversal of the second choice tree over from the root node for each leaf in the first choice tree. Being able to traverse each tree only once is superior, especially with many variants. Section 5.2 will show one solution to this problem based on dependent types.

We also have a source of inefficiency when programming with variational lists. The findings in Chapter 4 found segment lists to be the best performing list representation. Consider how we might go about implementing a function such as `zipWith` for segment lists, which should take two segment lists and apply a function pairwise to them to produce a new list. The plain version of this function traverses each list only once.

Using our generic list interface, we could define a `zipWith` function without too much work based on `vHead` and `vTail`.

```
vZipWith :: (VList vl) => (a -> b -> c) -> vl a -> vl b -> vl c
vZipWith f xs ys | vAllNull xs || vAllNull ys = …
vZipWith f xs ys = vCons (…) (…)
```

As we have already established, accessing lists with `vHead` and `vTail` is often a suboptimal approach, since those functions are often much slower than their plain analogues. To avoid that we could instead

try taking the approach specifically mentioned in Section 4.6.3. That is, we could define the following type class.

```
class VZipWith vl where
  vZipWith :: (a -> b -> c) -> vl a -> vl b -> vl c
```

Now we need to provide an instance for segment lists. In this case, however, it turns out that implementing such a function requires exactly the same logic as `vHead` and `vTail`. With segment lists we could, at any time while traversing one, encounter empty lists. This means that we always need to scan ahead to ensure that we have found the first values for each variant in order to apply the zipping function to them. That is the exact behavior of `vHead`, and so this approach is doomed to be no better than the generic version.

Based on the dependently typed approach described in Section 5.2, we propose a new representation of variational lists that can partially eliminate this overhead by forcing the lists into a structure of a particular shape. This will be discussed in Section 5.3.

Dependent types have been used in other applications for challenges related to performance. For instance, Xi and Pfenning (1998) showed an early use of a dependent type system to eliminate the need to check array bounds, and Xi (2003) showed how pattern matching performance can be improved in Dependent ML. Nilsson (2005) used GADTs to enforce static invariants to improve performance in a functional reactive programming library. Finally, if less directly, Danielsson (2008) used dependent types to encode metadata in order to reason about runtime complexity.

5.2 Faster Applicative Traversals

This section demonstrates an approach using dependent types to encode information about the structure of variational values. This helps to mitigate some of the efficiency concerns mentioned in

the above examples as well as to free programmers from the need to concern themselves with some of the implementation details of variational programming library functions.

The core idea behind the solution is to give the programmer the option to enforce a more rigid choice structure in exchange for runtime performance gains. To do so, we can introduce a new, alternative data type that represents a kind of normal form for choices. (The name `NV` is a mnemonic for "normalized" variational values.)

```
data NV : List Dim -> Type -> Type where
  ChcN : (d : Dim) -> NV ds a -> NV ds a -> NV (d::ds) a
  OneN : a -> NV [] a
  InjN : d : Dim -> NV ds a -> NV (d::ds) a
```

This definition is different from the original `VChc` in a number of ways, so let us examine it step by step. First and most importantly, the type now has an additional index in the form of a `List Dim`. The idea behind this type index is that we can use it to collect information about which dimensions are included in the dimension basis of a variational value. (The *dimension basis* of a variational value v is the set of all dimensions contained in v, and the *configuration space* of v is the set of all complete decisions that can be formed from its dimension basis.)

For example, the choice $A\langle B\langle 1,2\rangle, B\langle 3,4\rangle\rangle$, represented as a value of type `NV`, would have the type `NV ["A","B"] Nat`. Crucially, the type index encodes not just which dimensions occur in the choice, but also the order in which they are nested. The reason for this will become clear soon.

Another, seemingly strong option would be to index `NV` by a set rather than a list, but using non-free data types such as sets can complicate function definitions considerably. Furthermore, that would require us to invent a way to track the order information contained automatically in a list.

The `ChcN` constructor conceptually still takes the same three inputs as `Chc`, but now also encodes the relation on the type indexes which track the dimension basis. The `OneN` constructor is mostly

unchanged from `One`, with the exception that it produces values with an empty dimension list in the type. The third constructor, `InjN`, is completely new.

The new constructor is purely for optimization by helping to reduce duplication in choices. If we look at the `ChcN` constructor again, we can see that both child nodes must have identical type indexes, which is indicated by the use of `ds` in both. Thus we can construct choices only of alternatives that have identical dimensions.

While this will soon be advantageous in terms of performance, it has the unfortunate side effect of sometimes introducing unnecessary duplication. Consider the choice $A\langle B\langle 1, 2\rangle, 3\rangle$. The left alternative of A contains a nested choice while the right alternative does not. However our new `NV` type does not allow this, because the two children of the A choice would not have the same type. This means that, without the additional constructor, we would be forced to duplicate the `3` value as follows.[1]

```
ChcN "A" (ChcN "B" (OneN 1) (OneN 2))
         (ChcN "B" (OneN 3) (OneN 3))
```

But by introducing the `InjN` constructor, we are able to avoid the overhead.

```
ChcN "A" (ChcN "B" (OneN 1) (OneN 2))
         (InjN (OneN 3))
```

Although it has only saved us a single duplicated value, depending on the ordering of the dimensions, it can frequently eliminate the need for copying entire subtrees.

It may seem that we have made the structure of choices more inflexible and added some overhead with nothing obvious gained. However, recall that in our `Applicative` instance for `V` we had to traverse the left choice and then every time we reached a leaf, traverse the entire right choice. With this new `NV` type, however, we can encode in a type signature that two choices have exactly the same

[1]Choices are idempotent, that is, $A\langle x, x\rangle \equiv x$.

dimension basis. In turn, this lets us prove to the type checker that it is safe to traverse both choices simultaneously and that they will be aligned. We can see this in the new `Applicative` instance by virtue of the fact that when a dimension `d` is matched as part of one constructor, it always occurs in the other as well. The full listing is in Appendix B.

```
Applicative (NV ds) where
  pure x = …
  (ChcN d f g) <*> (ChcN d x y) = …
  (OneN f)     <*> (OneN x)     = …
  …
```

The main thing to note here are the cases for `<*>` where we match some combination of `ChcN` or `InjN` constructors. Because the list of dimensions `ds` is part of our instance, we know statically that the dimension encountered for both parameters will be the same. This means that we do not need to match one side and traverse the other but can instead traverse both parameters at once. This provides an asymptotic improvement over the implementation for `V`. For two choice tree operands of size n and m, the original approach would take $O(n\ m)$ (or $O(n\ log\ m)$ if we perform selection in the `Applicative` instance). The new approach takes time $O(n+m)$ in the case that the two operands do not have any dimensions in common.

We can then define functions based on this `Applicative` instance that take advantage of the speedup, such as `vPlus` for adding variational numbers.

```
nvPlus :: NV ds Int -> NV ds Int -> NV ds Int
nvPlus = liftA2 (+)
```

We have one more challenge to overcome in order to actually use these lifted functions effectively. Consider applying `nvPlus` to the following two arguments.

$$A\langle B\langle a,b\rangle,B\langle c,d\rangle\rangle \qquad B\langle A\langle u,v\rangle,A\langle x,y\rangle\rangle$$

When using `NV`, these will have different types, namely `NV ["A","B"]` and `NV ["B","A"]`, respectively. If we try to apply `nvPlus` to them, it will lead to a type error.

One way to overcome this issue is by transforming the structure (and therefore the dimension basis and corresponding type) of the choices, which can be done with the `rebase` function. First we need an updated selection function. The full definition is in Appendix B.

```
selectNV : Decision -> NV ds a -> NV ds a
selectNV dec (ChcN d l r) = case lookupDim d dec of
  ...
```

Now we can define `rebase`. The most important part of this definition is in the parameters `dsI` (input dimension basis) and `dsO` (output dimension basis). Again, the complete listing is in Appendix B.

```
rebase : NV dsI a -> NV dsO a
rebase (ChcN d l r) {dsI = d::_} {dsO = d'::_} =
    if d == d'
    then ChcN d (rebase l) (rebase r)
    else ChcN d' (rebase (selectNV ...)) (rebase (selectNV ...))
rebase (OneN x) {dsI = []} {dsO = d'::_} = InjN (rebase (OneN x))
rebase (InjN x) {dsI = (d::_)} {dsO = d'::_} = InjN (rebase x)
...
```

Rebase works by iterating over the input parameter and the desired output dimension basis at the same time. When we encounter a choice (or the lifting constructor `InjN`), we check whether the current dimension is the same as the current desired output. If it is, we continue recursively. If it is not, we build up a larger choice expression by adding `InjN` constructors before continuing. In the case that the output dimension basis is exhausted but we still have choices remaining, we drop them and keep only the left alternative.[2]

[2]Choosing the left alternative is arbitrary, but motivated by our convention of using the left alternative to represent a disabled feature and the right to represent and enabled one.

We also do not want to try to predict when a programmer will want to use `NV` over `VChc`. Instead we can provide a mechanism to switch between the two. Going from `NV` to `V` is fairly simple.

```
relax : NV ds a -> VChc a
relax (OneN x)     = One x
relax (ChcN d l r) = Chc d (relax l) (relax r)
relax (InjN v)     = relax v
```

The corresponding function `norm` is a bit more complicated because, in order to be useful, it must include some of the functionality of `rebase`. We could simplify this by making use of `rebase`, but that would require an extra traversal and add unnecessary overhead. The complete listing is in Appendix B but the structure is as follows.

```
norm : VChc a -> NV ds a
norm (Chc d l r) {ds = d'::_} =
    if d == d'
    then ChcN d (norm l) (norm r)
    else ChcN d' (norm (select …)) (norm (select …))
norm …
```

As with `rebase`, we bring the output dimension basis into scope and dispatch accordingly. The bulk of the work is done by the case where the input is a choice and we have not yet traversed the entire output dimension basis. If the input and output dimensions match we just convert to the new `ChcN` constructor and continue traversing the input choice. If the dimensions do not match, then we create a new choice matching the desired output and recurse. We perform a selection on the two branches in order to ensure that we do not generate dominated choices.

With these conversion functions, programmers can choose the most appropriate type for their needs without worrying about being stuck permanently with one or the other. Most importantly of all, we can now apply functions like `vPlus` and know that it will only need to traverse our choices once, as in the following example:

```
nvn : NV ["A","B"] Nat
nvn = norm (Chc "A" (One 1) (One 2))

nvm : NV ["A","B"] Nat
nvm = rebase (ChcN "B" (OneN 3) (OneN 4))

> nvPlus nvn nvm
ChcN "A" (ChcN "B" (OneN 4) (OneN 5))
         (ChcN "B" (OneN 5) (OneN 6))
```

We have successfully managed to use dependent types to trade some flexibility in the shape of our choices for improved efficiency. This application of `nvPlus` will traverse each choice once, in parallel.

However, we still have one small issue to address. Currently, we are forcing programmers to distinguish between using `VChc` and `NV` and applying the correct version of the function. One of our stated goals was to free programmers from this type of concern insofar as it is possible.

We propose to use interfaces (the Idris alternative to Haskell type classes) to abstract over these two types. For example, instead of having `vPlus` and `nvPlus` as visible functions, we define a `VPlus` interface and instantiate it with both kinds of choices.

```
interface VPlus (v : Type -> Type) where
  plus : Num a => v a -> v a -> v a

VPlus VChc where
  plus = vPlus

VPlus (NV ds) where
  plus = nvPlus
```

If we take this approach throughout a variational programming library, the programmer can define new functions without worrying about the details of which types are in use. A simple example might be a function like `twice`, as in the following.

```
twice : (VPlus v, Num a) -> v a -> v a
twice v = plus v v
```

This style of library design is also supported by the benchmark results in Chapter 4.

5.3 Variational Split Lists

Ideally we could also use this approach when representing variational data structures, such as lists. Like `V`, the `NV` type constructor can be applied to any type. Doing so with lists, as in the box list representation in Chapter 4, leads to generally poor performance. Another option, as used in the segment and choice lists allows us to represent variational lists more efficiently, but has some unexpected implications for common list library functions. Recall, for instance, that the typically constant time list head function needed to traverse the entire variational list in the worst case.

Another approach is to represent lists as two distinct parts. Such a definition could define a front part of the list which is performant if inflexible, based on the choice lists prior to adding the `Maybe` type to them.

```
data Front : (ds : List Dim) -> Type -> Type where
  F : List (NV ds a) -> Front ds a
```

Recall that the only reason this representation is not suitable for a general-purpose variational list is that it is unable to represent variants of different lengths. The idea of splitting the list into parts is that we can represent the first n values of all the variants where n is the length of the shortest variant. We can then pair this representation together with another list representation for the remaining

elements. The following definition uses a choice list purely for simplicity, but a segment list or another representation is equally valid.

```
data Back : Type -> Type where
  B : CList a -> Back a

data SList : (ds : List Dim) -> Type -> Type where
  SL : Front ds a -> Back a -> SList ds a
```

The whole *split list* representation is defined as the concatenation of these two parts. Conceptually, this lets us represent lists as a combination of a fast part and a slow part. Returning to the challenge of list head, we can now give the following definition:

```
vHead : SList ds a -> VChc (Maybe a)
vHead (SL (F (x::_)) _) = map Just (relax x)
vHead (SL _ (B b))      = slowVHead b
```

This function can now return the list head in constant time whenever the front list is not empty. The slow version, assumed to be implemented in `slowVHead`, is only used when the front list is empty.

In a broad sense this is similar to functional queues (Okasaki, 1999), which also use two lists—a front and a back—as an implementation, although with a different effect. In that work, amortized analysis showed that both `head` (sometimes called peek or front) and `snoc` (sometimes called add or enqueue) could be achieved in constant time. Unfortunately, we do not inherit this propensity towards a nice amortized analysis. This is because, in general, we are not able to convert the back list to a front list in the same number of steps that it takes to iterate through the front list. In fact, we are not able to convert the back list to a front list at all sometimes. We can still gain an improvement even in many functions where the entire list needs to be traversed however.

5.3.1 Speeding Up Operations with Split Lists

Recall the variational `zipWith` function discussed in Section 5.1. With split lists we can actually achieve two separate speedups. The first is when zipping together two variational non-list values, i.e. zipping the individual values in a choice. We gain this speedup for free simply by virtue of the improvements to the `Applicative` instance from Section 5.2.

```
vZipWithNV : (a -> b -> c) -> NV a -> NV a -> NV a
vZipWithNV = liftA2
```

The second speedup is obtained from a more efficient traversal of the front and back lists. As we have already seen, `vHead` is fast when operating on front lists and potentially much slower on back lists. The same is true for `vTail`, since we can simply drop a single choice for front lists but not for back lists. This means that, at least when the front list contains values, traversing the list in order is simply more efficient.

The ability to zip choices means we have all of the machinery to implement `zipWith` for split lists and complete the example. This requires a fair amount of boilerplate, so we will separate the definitions into three parts. First let us focus on front lists. We need two functions, one which actually does the zipping of pairs of elements and another which handles the remaining elements in the case that the two front lists are not the same length. Unfortunately, we generally cannot drop those elements, since we still have all the elements in the back list too. See Appendix B for the complete listing.

```
vZipWithFWork : (a -> b -> c) -> List (NV ds a) ->
                List (NV ds b) -> List (NV ds c)
vZipWithFWork f (x::xs) (y::ys) = vZipWithNV f x y :: vZipWithFWork f xs ys
…

vZipWithF : (a -> b -> c) -> Front ds a -> Front ds b ->
            (Front ds c, Either (Front ds a) (Front ds b))
vZipWithF f (F xs) (F ys) = if length xs < length ys …
```

The `vZipWithFWork` function, which does the actual work of zipping front lists together, is more or less identical to a standard `zipWith` function for plain linked lists. The `vZipWithF` function is predominantly boilerplate for dealing with the leftover elements of the longer front list. We use an `Either` type, since they can have different element types.

Now we also need to handle zipping together two back lists. Fortunately, this is fairly straightforward using all of the supporting functions we have been defining until now. First, we define a helper function to break a list into its head and tail components.

```
uncons : Back a -> (VChc (Maybe a), Back a)
uncons bs = (slowHead bs, slowTail bs)
```

Next we can define `vZipWithBWork`, which is the function that does the actual work of zipping together two back lists. The complete listing is in Appendix B.

```
mutual
  vZipWithBWork : (a -> b -> c) -> Back a -> Back b -> Back c
  vZipWithBWork f xs ys = let (x,xt) = uncons xs
                              (y,yt) = uncons ys
                          in …

  vZipWithB : (a -> b -> c) -> Back a -> Back b -> Back c
  vZipWithB f xs ys = if vAllNull xs || vAllNull ys
                      then B []
                      else vZipWithBWork f xs ys
```

These functions are mutually recursive, iterating over the values of each until one is empty, at which point we are finished. Finally, we are able to move on to the actual `vZipWith` function for split lists. This is much easier, since we already have functions to work with the individual parts. We just need to apply them. Once again, the complete definition can be found in Appendix B.

```
relaxL : Front ds a -> VList a
relaxL (F fs) = map (… relax) fs

vZipWith : (a -> b -> c) -> SList ds a -> SList ds b -> SList ds c
vZipWith f (SL fr1 (B b1)) (SL fr2 (B b2)) =
  case vZipWwithF f fr1 fr2 of
    (fr, Left rest)  => …
    (fr, Right rest) => …
```

First we have defined `relaxL` which takes a front list, applies our existing `relax` function to each element, and then maps the `Just` constructor to create `Maybe` values. The `vZipWith` function is our primary entry point for zipping split lists with arbitrary functions. We first zip the front parts, which gives us a new front list as well as zero or more remainder elements. We need to track which list they

come from (the second parameter to the function or the third), since they have different types. We then relax the remainder front list, append it to the appropriate back list, and then zip the back lists together.

What has all of this extra work achieved? Zipping together the front lists is fast. We can find the head and tail of each in constant time, and we can zip the two head elements together in a single traversal. Conversely, the back lists are slow. Every application of head or tail could, in the worst case, need to traverse the entire list. Furthermore, every zipping operation on two elements requires many repeated traversals of the choice structures. If we can generate and work with lists that have as many elements as possible in the front, we can reduce the amount of work we need to do considerably. This suggests that we need a way for programmers to move as many elements into the front list as possible. In turn, this requires us to be able to change the dimension basis of the list depending on what values it contains.

The first step is implementing a normalization function that takes a split list as input and maximizes the number of elements in the front part, moving as many as possible from the back. One of the tasks that this requires is trying to remove `Maybe` types, so we begin by defining a helper function.

```
unM : VChc (Maybe a) -> Maybe (V a)
unM (Chc d l r) = case (unM l, unM r) of
  (Just l', Just r') => Just (Chc d l' r')
  (_, _)             => Nothing
unM (One Nothing)  = Nothing
unM (One (Just x)) = Just (One x)
```

Now we can define a function `split` which tries to do the normalization process. The complete definition is in Appendix B.

```
split : SList ds a -> SList ds a
split sl@(SL (F xs) bs) = let (hd,tl) = unconsB bs
  in case unM hd of
       Just v => …
       Nothing => sl
split l = l
```

Normalizing is a relatively straightforward process of examining the head element of the back list and applying `unM` to it. If the process fails, we know that there are only partial elements remaining, and so our normalization process is done. If it succeeds, we move our new element to the end of the front list (concatenating each element individually is suboptimal, since Idris evaluates strictly, but is done for the sake of clarity here), and we recursively repeat the process.

Since normalization can be an expensive operation, we want to use it sparingly. We discuss this in more detail in Section 5.3.2. Before we get to that, we also need functions to modify the dimension bases of split lists. We may want to zip together a list containing choices only in dimension *A* with another list which has choices in *A* and *B* dimensions. Currently this is not possible.

Fortunately, we can build on the `rebase` definition provided in Section 5.2 to accomplish this. Recall that `rebase` allowed us to restructure `NV` types to modify their dimension bases. We can simply map this transformation across all the elements in the front part of a split list.

```
rebaseFront : Front dsI a -> Front dsO a
rebaseFront (F xs) = F (map rebase xs)

rebaseSList : SList dsI a -> SList dsO a
rebaseSList (SL f b) = SL (rebaseFront f) b
```

Note that we do not need to modify the back part of the list, because this does not impact the index in the type. We can use this function as follows to dictate the shape of the output list.

```
l1 : SList ["A","B","C"] Nat


l2 : SList ["A"] Nat


l3 : SList ["A","B"] Nat
l3 = vZipWith (+) (rebaseSList l1) (rebaseSList l2)
```

Here we have used the `rebaseSList` operation to grow the dimension basis of one list and shrink the other. Recall that the original `rebase` grows the dimension basis by adding `InjN` constructors where necessary and shrinks the dimension basis by selecting the left alternative of the choice.

5.3.2 Lazy Split Lists

The split list data type is quite useful and in many circumstances can provide a performance improvement over a more naive representation. However, it is far from a perfect solution. We have conveniently avoided the discussion of some crucial simple functions, such as the consing of elements, for which performance is actually worsened compared to an approach based on `VList`. A `vCons` function needs to support the case where we only want to insert values for some variants of the variational list. This means that the type of the input to `vCons` needs to be `VChc (Maybe a)`. However, the front part of an `SList` does not hold values of this type, nor can we reliably convert the input to the appropriate type because it could contain `Nothing` values.

We also obviously cannot insert the new value in the back list without doing something about the elements in the front list, since they need to come after the new one. We could try to reconstruct the front list moving elements around one at a time, but it is simpler to just move elements into the back list and then normalize both parts.

This process is obviously slow and best done as infrequently as possible. Unfortunately, repeated consing of elements would result in this computation being done at each interval, almost certainly eliminating any performance increases we hope to achieve.

We have adopted one final tweak to the variational list definition to help mitigate this cost. Instead of enforcing the shape of exactly one front portion and exactly one back portion, we relegate that to a shape we call a *tidy* split list, and extend the `SList` definition to also allow *messy* split lists. A messy split list is one or more back lists concatenated to the front of a tidy split list. The new data type looks like this, which is called `LList`, short for *lazy* split list.

```
data LList : (ds : List Dim) -> Type -> Type where
  Tidy : Front ds a -> Back a -> LList ds a
  Messy : Back a -> LList ds a -> LList ds a
```

Now we can repeatedly cons values onto our list by simply marking it as messy. Not until we have to access the first element of a list do we have to normalize. Here is a correspondingly updated definition of the `vHead` function.

```
vHead : LList ds a -> V a
vHead (Tidy (F []) (B b)) = slowHead b
vHead (Tidy (F (x::_)) b) = map Just (relax x)
vHead ll@(Messy _ _)      = vHead (split ll)
```

Finally, the `split` function also needs to be updated to handle these messy lists.

```
split : LList ds a -> LList ds a
split (Messy (B xs) (Tidy (F fs) (B zs))) =
  split … (ls ++ relaxL fs ++ bs) …
split (Messy (B xs) (Messy (B ys) rest)) =
  split (Messy (B $ xs ++ ys) rest)
…
```

The `Tidy` cases are unchanged from before aside from the constructor names. The `Messy` cases push all of the messy values and the front list into the back of a new `Tidy` list, and then recursively apply `split`. Some of the list concatenation could probably be avoided, but making the definition easier to understand takes priority here. Of course, we also need to update the other list functions, but this is done predictably and not reproduced here.

Now we have a variational list definition that can offer an asymptotic improvement in runtime efficiency for all the values contained in the front list while also minimizing the slowdown caused by repeated consing of values. This speedup is provided in a way that is mostly transparent to the programmer, allowing them to focus on the task at hand instead of the details of their list implementation.

Although we cannot accurately predict the data that will be used by programmers, we do conjecture that in many cases the front list will contain the majority of the list elements. The reason for this is that, in most real world data, the list variants will not differ significantly in length. For example, sports ranking data such as the United States college football ranking composite[3] will generally have all lists be precisely the same length. This means we have no real need for partial variational values and every single element can be placed in the front list. While this may seem to suggest using a simple list of choices instead of split lists, we believe that a general solution that works well in both situations is ideal.

Weather data, too, will likely contain data samples at periodic intervals for every individual sensor, meaning for each time step we are likely to have a value for all variants. This is also the case for many other sources of variational data. Of course, there will be exceptions to this, but in many cases we should be able to significantly reduce the amount of computation needed.

[3]`http://www.masseyratings.com/cf/compare.htm`

5.4 Conclusions and Lessons for Variational Applications

This chapter has introduced a representation for a variational type constructor and for variational list based on dependent types. These representations are able to mitigate some of the inefficiencies inherent in the earlier representations from Chapters 3 and 4. However, the dependently typed approach is not a panacea. In exchange for the improved efficiency, we lose flexibility in the representation. Choices, both in lists and standalone, are forced to adhere to a particular structure. It is not difficult to conceive of applications where the structure and ordering of choices encodes useful information, such as tracking temporal ordering in order to support a history mechanism. Reordering the choices prevents this type of application without tracking extra information beyond what is contained in the choices.

This kind of tradeoff occurs repeatedly when programming with variation. Consider, for example, our original definition of choice lists which was just a plain list of choices. This variational list representation is extremely limited because it prevents the programmer from encoding list variants with different lengths. However, in a hypothetical application domain where list variants are always fixed, such a list representation can offer a representation that is both efficient and easy to work with. All kinds of list functions such as `head`, `tail`, `reverse`, and so defined for plain lists would work just as well for variational lists.

Understanding this tradeoff is crucial to understanding what kinds of applications are good candidates for integrating variation into. Since one of the major goals of this dissertation is to demonstrate practical applications of variation and variational programming, the choice of domains must be driven by these findings. Both of the applications described in the upcoming chapters, variational pictures and variational information visualization, allow us to limit the scope of the variability introduced. In turn, both are able to achieve straightforward and compact representations.

The details of variational pictures follow in Chapter 6 and variational visualization will follow that in Chapters 7 and 8.

Chapter 6: Variational Pictures

This chapter presents a model of variational pictures. As motivated in the Introduction, a variational picture is a structure that encodes potentially many plain pixel-based pictures while also providing a mechanism by which to navigate among the pictures and select particular variants. This chapter makes three contributions: (1) a formal model of variational pictures, (2) a set of properties that describe the generality and usefulness of the model, and (3) variational area trees, a visual language for exploring and navigating variational pictures.

The definition of variational pictures is based on the choice calculus (see Chapter 2) which we apply to a model of plain pictures described in Section 6.1. The resulting model of variational pictures is then presented in Section 6.2. The different degrees of variability in a variational picture give rise to a notion of variation type, introduced in Section 6.3, and a corresponding notion of region, which is discussed in Section 6.4. Finally, in Section 6.5 we introduce an operation that can create variational pictures automatically from a sequence of plain pictures.

6.1 Plain Pictures

We base our model of variational pictures on a model of plain pictures that is basically defined as a set of pixels. Specifically, given a finite domain of locations *Loc*, a *picture* is a finite mapping from *Loc* to some type T. Here T typically is a set of colors, but it can be any type that has an equality predicate defined on it. Moreover, in most cases pictures are given by fixed, rectangular grid of pixels, which means that the type of locations is of the form $Loc^{n,m} = \{1, \ldots, n\} \times \{1, \ldots, m\}$. However,

the definitions that follow do not depend on this particular structure, so that we can simply assume a finite set *Loc* of elements on which equality is defined.[1]

Thus the type of T pictures over the domain *Loc* is defined as $Pic_T = Loc \to T$. A picture is an element of that type, and a *pixel* of a picture $p \in Pic_T$ is given by a pair $(l, x) \in p$ with $l \in Loc$ and $x \in T$.

Here is a small example of a 2×2 picture over a type of symbols $S = \{\circ, \bullet, \star\}$: $p = \begin{smallmatrix}\circ & \bullet \\ \bullet & \circ\end{smallmatrix}$. Since the structure of T doesn't really matter, we will mostly omit it in the following and consider this parameter implicitly fixed, that is, we simply use the type *Pic*.

6.2 Adding Choices to Pictures

A *variational picture* of type T is a mapping from locations to variational T values, that is, $VPic_T = Loc \to V(T)$. One could consider a more general definition that also allows variability in the location domain. However, such a type would complicate the following definitions considerably without gaining much. In fact, if the type T contains some unit or null value, one can simulate differently sized pictures using such a designated value. On the other hand, the chosen definition still facilitates the local application of variability, which is an important feature of our model to be discussed later.

Corresponding to plain pictures, a *variational pixel* is an element of a variational picture $vp \in VPic_T$ where $vp = (l, vx)$ with $l \in Loc$ and $vx \in V(T)$. Again, we omit the T parameter from the type in the following.

Here is a variational 2×2 picture over type S that varies the pixels in *vp*'s second column in dimension A: $vp_A = \begin{smallmatrix}\circ & A\langle \bullet, \circ \rangle \\ \bullet & A\langle \circ, \circ \rangle\end{smallmatrix}$. Since vp_A contains only choices in one dimension, it encodes two plain variant pictures that can be extracted using selection, that is, $\lfloor vp_A \rfloor_{A.l} = p = \begin{smallmatrix}\circ & \bullet \\ \bullet & \circ\end{smallmatrix}$ and $\lfloor vp_A \rfloor_{A.r} = \begin{smallmatrix}\circ & \circ \\ \bullet & \circ\end{smallmatrix}$.

[1]This generality follows from the fact that our picture model doesn't require a notion of connectedness.

In general, a variational picture may contain multiple choices in different dimensions. Here is a picture that varies the upper right pixel in vp_A again using a nested B choice in A's left alternative: $vp_{AB} = \begin{smallmatrix} \circ & A\langle B\langle \bullet,\star\rangle,\circ\rangle \\ \bullet & A\langle \circ,\circ\rangle \end{smallmatrix}$. Selecting the left alternative of A in vp_{AB} now does not produce a plain picture, since the B choice has not been eliminated: $\lfloor vp_{AB} \rfloor_{A.l} = \begin{smallmatrix} \circ & B\langle \bullet,\star\rangle \\ \bullet & \circ \end{smallmatrix}$. This means that we need to also perform a selection for B. In the example we get $\lfloor vp_{AB} \rfloor_{\{A.l,B.l\}} = p$ and $\lfloor vp_{AB} \rfloor_{\{A.l,B.r\}} = \begin{smallmatrix} \circ & \star \\ \bullet & \circ \end{smallmatrix}$. On the other hand, selecting only the right alternative still produces a plain picture $\lfloor vp_{AB} \rfloor_{A.r} = \begin{smallmatrix} \circ & \circ \\ \bullet & \circ \end{smallmatrix}$.

The semantics of a variational picture is a mapping from decisions to plain pictures. The semantics definition iterates over all variational pixels and extracts and lifts the decisions to the level of pictures, effectively commuting decisions and locations.

$$
\begin{aligned}
[\![\cdot]\!] &: \mathit{VPic} \to V(\mathit{Pic}) \\
[\![vp]\!] &= \{(\delta, (l, x)) \mid (l, vx) \in vp, (\delta, x) \in vx\}
\end{aligned}
$$

The type of the semantic function helps explain its definition: since $V(X) = \mathit{Dec} \to X$, $\mathit{Pic} = \mathit{Loc} \to T$, and $\mathit{VPic} = \mathit{Loc} \to V(T)$, the type of the semantic function reads in expanded form as $(\mathit{Loc} \to (\mathit{Dec} \to T)) \to (\mathit{Dec} \to (\mathit{Loc} \to T))$.

As the examples vp_A and vp_{AB} illustrate, the ability to apply choices to individual pixels makes variability a localized feature. This is an important property, since it allows only those parts to be varied that need it and keeps non-variable picture parts constant across different variant pictures, which supports editing by avoiding update anomalies Date (2005). For example, if we change the upper left pixel in vp_{AB} from $\circ$ to $\bullet$, this change has to be done only once and will still correctly affect all variants of the picture.

In order to support precise operations to create and modify the variability in a variational picture, we employ the notion of a *view decision*, which is simply a choice calculus decision that specifies the plain picture variant that is currently active. View decisions are always complete, that is, they contain

exactly one selector for each unique dimension contained anywhere in the variational picture. For vp_{AB}, we have the three possible view decisions $\{A.l, B.r\}$, $\{A.l, B.r\}$, and $\{A.r\}$.

6.3 Variability Types

The inclusion of choices in pictures suggests a classification of pixels according to their variability and the grouping of pixels with the same variability into regions. To formalize this idea we first define the notion of a *variability type*. The type of a plain, non-variational value is $\diamond$ (called *unit*), and the type of a variational value is given by its choice structure, which is obtained by replacing all plain values in it by $\diamond$. The variability type of a value can be determined by the function φ, which is defined as follows.

$$\begin{aligned} \varphi &: V(T) \to V(\{\diamond\}) \\ \varphi(D\langle vx, vy\rangle) &= D\langle \varphi(vx), \varphi(vy)\rangle \\ \varphi(x) &= \diamond \end{aligned}$$

The type of a pixel is given by the type of its value, that is, $\varphi(l, vx) = (l, \varphi(vx))$.

With this definition, all pixels in $\begin{smallmatrix}\circ & \bullet \\ \bullet & \circ\end{smallmatrix}$ have type $\diamond$. In $vp_{AB} = \begin{smallmatrix}\circ & A\langle B\langle \bullet, \star\rangle, \circ\rangle \\ \bullet & A\langle \circ, \circ\rangle\end{smallmatrix}$, the pixels in the left column have type $\diamond$, the lower right pixel has type $A\langle \diamond, \diamond\rangle$, and the upper right pixel has type $A\langle B\langle \diamond, \diamond\rangle, \diamond\rangle$. For notational convenience we also write more succinctly A for a type $A\langle \diamond, \diamond\rangle$. With this abbreviation we can say that the lower right pixel has type A and the upper right pixel has type $A\langle B, \diamond\rangle$.

A variability type tells us exactly what decisions are needed to extract all plain values from a variational value. For example, the set of plain values contained in vp_{AB} is given by

$\{\lfloor vp_{AB}\rfloor_{\{A.l,B.l\}}, \lfloor vp_{AB}\rfloor_{\{A.l,B.r\}}, \lfloor vp_{AB}\rfloor_{A.r}\}$. We can compute the set of decisions required for extracting all plain values from a variational value of a particular type with the following function *decs*.

$$
\begin{aligned}
decs &: V(\{\diamond\}) \to 2^{Dec} \\
decs(D\langle vx, vy\rangle) &= \{\delta \cup \{D.l\} \mid \delta \in decs(vx)\} \cup \{\delta \cup \{D.r\} \mid \delta \in decs(vy)\} \\
decs(x) &= \varnothing
\end{aligned}
$$

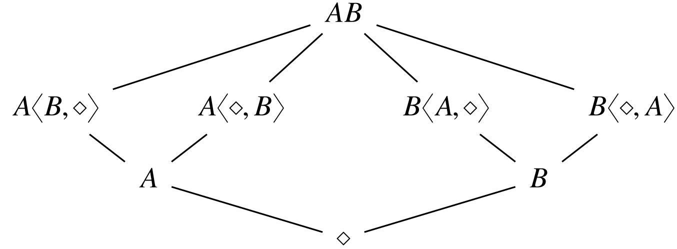
Based on the function *decs* we can define a *variability type equivalence* that holds for types that describe the same variability.

$$\phi \sim \phi' :\Longleftrightarrow decs(\phi) = decs(\phi')$$

Note that $\sim$ is not a simply derived from $\equiv$ (see Chapter 2). For example, whereas $D\langle x, x\rangle \equiv x$ (due to idempotency), $\varphi(D\langle x, x\rangle) = D \nleftarrow \diamond = \varphi(x)$. We can generalize the notion to type equivalence naturally to a *refinement ordering* on variability types, again based on the decisions that are represented by the variability types.

$$\phi \gtrsim \phi' :\Longleftrightarrow \forall \delta' \in decs(\phi') : \exists \delta \in decs(\phi) : \delta \supseteq \delta'$$

We have, for example, $A\langle B, B\rangle \gtrsim A\langle B, \diamond\rangle$ and $A\langle \diamond, B\rangle \gtrsim A$. The variability refinement is a partial order that give rise to a lattice structure. Here is a small excerpt of this lattice involving the two dimensions A and B. Note the two types $A\langle B, B\rangle$ and $B\langle A, A\rangle$ are equivalent. This means that the nesting is not relevant, and we can write the type more accurately as AB to indicate that neither A nor B is in any way privileged over the another.



This diagram indicates that the nesting of dimensions does not matter in fully variationalized values, since the types are equivalent. This is an important property we will come back to in Section 6.6.

We write more shortly $D \in \delta$ whenever $D.l \in \delta$ or $D.r \in \delta$, and say that a dimension D *depends* on dimension D' in a type ϕ, written as $D' \leftarrow_{\phi} D$, if $D \in decs(\phi) \implies D' \in decs(\phi)$, that is, in order to make a selection in D one also has to make a selection in D'. For example, B depends on A in $A\langle B, \diamond\rangle$. We notice that in the type AB (which is equal to $A\langle B, B\rangle$ and $B\langle A, A\rangle$), B depends on A and A depends on B. In such situations, when two dimensions D and D' depend on one another in a type ϕ, we say that D and D' are *co-dependent* in ϕ, written as $D' \leftrightarrow_{\phi} D$.

6.4 Variability Regions

Based on variability types we can define a notion of regions that have the same variability. All pixels with the same type have the same variability structure, which means that they are mapped to plain variants by the same set of decisions. This property partitions the set of all pixels (or more precisely, their locations) into a set of disjoint regions. Specifically, for each variation type $\phi \in V(\{\diamond\})$ we define the *region of ϕ-variability* (or *ϕ-region* for short) as follows.

$$R_{\phi}(vp) = \{(l, vx) \in vp \mid \varphi(vx) \sim \phi\}$$

In the park example from the Introduction, the part of the image affiliated with the fountain is given by the region $R_{Trees\langle\diamond,Fountain\rangle}$ and the part affiliated with the pavilion is given by the region $R_{Pavilion}$.

The following lemma is a direct consequence of the definition of ϕ-regions.

Lemma 1. *For every variational picture vp, the set of (non-empty) regions $R_\phi(vp)$ forms a partition of vp.*

Since regions are identified by the common types of their pixels/locations, we can derive a refinement relation for regions based on the type refinement defined earlier. Specifically, $R_{\phi'} \gtrsim R_\phi$ if and only if $\phi' \gtrsim \phi$.

As indicated by the example scenario in the Introduction, (variational) pictures are typically the result of a sequence of operations performed in an editor. In particular, variability is introduced into a picture by marking an area and assigning a dimension to it. After that the resulting two alternatives can be edited in different ways and will generally contain different content.

In many cases, however, not every pixel in the marked area will be different in both alternatives. Consider again the picture $vp_A = \begin{smallmatrix}\circ & A\langle\bullet,\circ\rangle \\ \bullet & A\langle\circ,\circ\rangle\end{smallmatrix}$. Both pixels in the right column are variational: they are both of type A and thus belong to the same region. However, only the upper pixel differs in its alternatives. Since both alternatives of the lower pixel are equal, we could apply the idempotency law of the choice calculus and replace $A\langle\circ,\circ\rangle$ by $\circ$ without changing the semantics of the variational picture. Such a change would, however, change the region partition for vp_A. By systematically applying transformations for eliminating idempotent choices ($D\langle x,x\rangle \mapsto x$) as well as dominated choices ($D\langle D\langle x,y\rangle,z\rangle \mapsto D\langle x,z\rangle$ and $D\langle x,D\langle y,z\rangle\rangle \mapsto D\langle x,z\rangle$) to all pixels in a variational picture, we can shrink regions and thus increase the sharing in the picture. We can define a corresponding region shrinking operation as follows. First, we define the operations on variational values. Note that pattern matching on dominated choices is insufficient, since they can occur at arbitrary depths, so we use selection to avoid them instead.

$$\dot{\rho}(D\langle vx, vy\rangle) = \begin{cases} \dot{\rho}(vx) & \text{if } \dot{\rho}(vx) = \dot{\rho}(vy) \\ D\langle \dot{\rho}(\lfloor vx \rfloor_{D.l}), \dot{\rho}(\lfloor vy \rfloor_{D.r})\rangle & \text{otherwise} \end{cases}$$

$$\dot{\rho}(x) = x$$

Region shrinking of variational pictures then simply works by applying the operation $\dot{\rho}$ to all values in all pixels.

$$\rho : VPic \to VPic$$

$$\rho(vp) = \{(l, \dot{\rho}(vx)) \mid (l, vx) \in vp\}$$

Note that a variational picture obtained by ρ is *not* guaranteed to have maximized the sharing of values and thus is not necessarily minimal. Consider, for example, the variational value $vx = A\langle B\langle x, z\rangle, B\langle y, z\rangle\rangle$. The definition of $\dot{\rho}$ cannot extract and share the value z, since $\dot{\rho}(vx) = vx$. However, the variational value $vy = B\langle A\langle x, y\rangle, z\rangle$, which is equivalent to vx, is smaller than vx and does share z.

Note that the redundant value could occur deeply nested in the two alternatives, which means that a simple one-level dimension rotation is, in general, insufficient to expose redundant values. Nevertheless, redundant choices in idempotent or dominated choices will be eliminated by the region shrinking operation. It checks explicitly for identical alternatives and, at every step, uses selection on both branches to ensure no nested choices in matching dimensions.

6.5 Distilling Variational Pictures

Given two plain pictures p and p', we can automatically merge them and produce a variational picture that captures the differences between p and p' in choices of some dimension D and keeps all common

parts as plain values. First, we define an operation $\dot{\Delta}$ for comparing individual pixel values. If we only needed $\dot{\Delta}$ for generating one variational picture from two plain pictures, its type could be $T \times T \to V(T)$, but since we actually want to apply the merge operation repeatedly to a number of pictures, its type should be $T \times V(T) \to V(T)$, which allows a plain picture to be merged with a variational picture. The operation can, of course, still be used with two plain pictures since a plain picture is just a special case of a variational one.

$$\dot{\Delta} : T \times V(T) \to V(T)$$

$$\dot{\Delta}(x, vx) = \begin{cases} x & \text{if } x = vx \\ D\langle x, vx \rangle & \text{otherwise} \end{cases}$$

Note that the equality between x and vx can only hold when vx is a plain value. Note also that we have not specified which dimension D is to be used in the definition, since the concrete name does not really matter. We have to postulate however, that D has not been used in any of the vx values that $\dot{\Delta}$ is applied to. Effectively, we want to use a new dimension for every new picture that is merged into a variational picture.

Using $\dot{\Delta}$, the operation Δ for merging a plain picture into a variational one and capturing their differences in choices can be defined as follows. We assume that both pictures are defined over the same domain of locations. Note that we use the same l in both qualifiers of the set comprehension to express a parallel iteration over all pixels in both pictures.

$$\Delta : Pic \times VPic \to VPic$$

$$\Delta(p, vp) = \{(l, \dot{\Delta}(x, vx)) \mid (l, x) \in p, (l, vx) \in vp\}$$

We can generalize the definition of Δ to work on not just two but a whole set of pictures in a straightforward way. The only side condition, which is not formalized here, is that the a fresh dimension is used in each new call to $\dot{\Delta}$. This could be formalized by threading a set of dimension names through the successive applications of Δ and $\dot{\Delta}$, but this doesn't contribute much to the understanding of the operations, and we therefore omit it here for brevity.

$$\begin{aligned} \Delta^* &: 2^{Pic} \to VPic \\ \Delta^*(\{p\}) &= p \\ \Delta^*(\{p\} \cup P) &= \Delta(p, \Delta^*(P)) \end{aligned}$$

We can see this operation in action using the small plain pictures $p_1 = \begin{smallmatrix}\circ & \bullet \\ \bullet & \circ\end{smallmatrix}$, $p_2 = \begin{smallmatrix}\circ & \circ \\ \bullet & \bullet\end{smallmatrix}$, and $p_3 = \begin{smallmatrix}\circ & \star \\ \bullet & \bullet\end{smallmatrix}$. Now we need to evaluate $\Delta^*(\{p_1, p_2, p_3\})$, which we can expand to $\Delta(p_3, \Delta(p_1, p_2))$. Evaluating $\Delta(p_1, p_2)$ produces $p_{12} = \begin{smallmatrix}\circ & A\langle\bullet,\circ\rangle \\ \bullet & A\langle\circ,\bullet\rangle\end{smallmatrix}$. Finally, we can evaluate $\Delta(p_3, p_{12})$ and get the variational picture $\begin{smallmatrix}\circ & B\langle\star, A\langle\bullet,\circ\rangle\rangle \\ \bullet & B\langle\bullet, A\langle\circ,\bullet\rangle\rangle\end{smallmatrix}$.

6.6 Properties of Variational Pictures

In this section we collect a number of general properties of variational pictures that serve as additional characterizations of the concept and also justify the chosen design.

The first observation is that variational picture distillation and semantics are in some sense inverse operations of each other. Distillation is, in fact, what users do when they have a set of pictures in mind that they want to represent in a variational one. Of course, users typically won't encode the differences as efficiently as the operation Δ^*, which will often result in a variational picture with

maximal sharing. Now we can show that the semantics of a variational picture that is generated by Δ^* from a set of plain pictures produces exactly the original plain pictures.

Theorem 1. $\forall P \in 2^{Pic} : range(\llbracket \Delta^*(P) \rrbracket) = P$

This theorem states that the semantics of variational pictures are correct; it says that distilling a variational picture from a set of plain pictures does not lose any information, because the original set of pictures can be extracted by the semantics.

Since a choice tree with n leaves contains $n-1$ dimensions (as mentioned in Section 2.2), that also means that n pictures are distilled into a variational picture with $n-1$ dimensions.

A closely related result is that from the plain pictures encoded in a variational picture we can recover an equivalent variational picture using Δ^*. Note that the reconstructed variational picture may not be identical to the original one in terms of how the variation is represented, that is, in general we have $\Delta^*(range(\llbracket vp \rrbracket)) \neq vp$; all we can guarantee is that the reconstructed picture has the same semantics, up to a renaming of dimensions.

Theorem 2. $\forall vp \in VPic : \llbracket \Delta^*(range(\llbracket vp \rrbracket)) \rrbracket =_\alpha \llbracket vp \rrbracket$

As the example scenario in the Introduction has shown, areas of variability are often nested, which leads to correspondingly nested choices in the pixel values. This is the case, for example, for the optional fountain. The fountain itself is an area of variability, but it is also nested inside one alternative of another area in which a wooded area is cleared. If we call these dimensions *Trees* and *Fountain*, we would expect to see many pixel values with the type $Trees\langle \diamond, Fountain \rangle$.

One concern is that the initially chosen area and thus choice nesting commits a user to a particular nesting that cannot be changed at a later stage of editing. The next theorem shows that this is actually *not* the case and that variation creation, in particular, the nesting of choices, does not lead to a premature commitment to a particular variation structure.

Consider the situation that an area for choice B has been created inside an area for choice A, and let's assume that the B choice lives inside A's left alternative (just as in the example vp_{AB} from Section 6.2). The type of the pixels inside the B area is $A\langle B, \diamond\rangle$, and the type of the pixels outside of B but inside of A is simply A. We call a pair of regions such as R_A and $R_{A\langle B,\diamond\rangle}$ a *region refinement pair*, since $A\langle B, \diamond\rangle \gtrsim A$ (and thus $R_{A\langle B,\diamond\rangle} \gtrsim R_A$), and write such a pair as $R_A[R_{A\langle B,\diamond\rangle}]$ to indicate that it was probably the result of creating a B choice in an A area (or creating an A choice around a B area).[2]

We can observe that for the pixels in $R_{A\langle B,\diamond\rangle}$, B depends on A. If we consider the dual case of a region refinement pair $R_B[R_{B\langle A,\diamond\rangle}]$, we can see that the dependency is the other way around, since for the pixels in $R_{B\langle A,\diamond\rangle}$, A depends on B. So it seems that the chosen nesting for the areas and dimensions determines two incompatible dependencies among the dimensions. However, there is a straightforward way to reconcile these different refinement pairs by refining the region $R_{A\langle B,\diamond\rangle}$ to $R_{A\langle B,B\rangle} = R_{AB}$ (or dually refining $R_{B\langle A,\diamond\rangle}$ to $R_{B\langle A,A\rangle} = R_{AB}$).

A region refinement $R_{A\langle B,\diamond\rangle}$ to R_{AB} can be achieved by simply expanding the right alternative of all A choices in the region's pixels into idempotent B choices, that is, by replacing $A\langle B\langle x, y\rangle, z\rangle$ with $A\langle B\langle x, y\rangle, B\langle z, z\rangle\rangle$.[3]

We can then apply a similar region refinement to R_A, turning it as well into a region of type AB. This automatically merges the region refinement pair $R_A[R_{A\langle B,\diamond\rangle}]$ into one region R_{AB}, and if we assume that the region is not nested within another area, we can consider it together with the surrounding non-variational pixels of type $\diamond$ as a new, bigger region refinement pair $R_\diamond[R_{AB}]$. Next we can refine (part of) the region $R_\diamond$ to R_B, leading to $R_B[R_{AB}]$. We can summarize the sequence of transformations as follows.

[2]Since regions are derived from the pixel types, such region pairs do not necessarily occur in any particular geometric relationship. However, such region pairs typically result from editor actions that create one choice area within another.

[3]There are situations in which other transformations, such as $A\langle B\langle x, y\rangle, B\langle z, y\rangle\rangle$ may be more appropriate, but the point is that a region refinement is easy to achieve.

$$R_A[R_{A\langle B,\diamond\rangle}] \rightsquigarrow R_A[R_{AB}] \rightsquigarrow R_{AB}[R_{AB}] \rightsquigarrow R_{AB} \rightsquigarrow R_\diamond[R_{AB}] \rightsquigarrow R_B[R_{AB}]$$

We can perform a similar transformation for the region pair $R_B[R_{A\langle B,\diamond\rangle}]$:

$$R_B[R_{A\langle B,\diamond\rangle}] \rightsquigarrow R_B[R_{AB}] \rightsquigarrow R_{AB}[R_{AB}] \rightsquigarrow R_{AB} \rightsquigarrow R_\diamond[R_{AB}] \rightsquigarrow R_A[R_{AB}]$$

This shows that while we can't turn $R_A[R_{A\langle B,\diamond\rangle}]$ into $R_B[R_{A\langle B,\diamond\rangle}]$ (or vice versa) without removing information, we can invert the nesting of choices if we refine the innermost region sufficiently.

Theorem 3. $R_A[R_{A\langle B,\diamond\rangle}] \rightsquigarrow^* R_B[R_{AB}]$ *and* $R_B[R_{A\langle B,\diamond\rangle}] \rightsquigarrow^* R_A[R_{AB}]$

We consider an application of this theorem in the next section.

6.7 Maintenance of Variational Pictures

It is unrealistic to expect variational picture authors to know the precise and final locations of variability they will need from the outset. This means that our model of pictures should support changing areas of variability that have already been defined.

Consider the park example from the Introduction. Suppose that, after creating the design shown, the architect is told that the *Fountain* area needs to include pipes connecting from the water main. This means that the associated area needs to not only grow, but grow such that the nesting order with the *Trees* area is reversed. The final result can be seen in Figure 6.1.

Fortunately, we have already seen in Section 6.6 that region refinement allows us to change the nesting order without issue. We just need to transform the appropriate pixels by expanding them all with an outer *Fountain* choice. There are three transformations to make, namely for those pixels

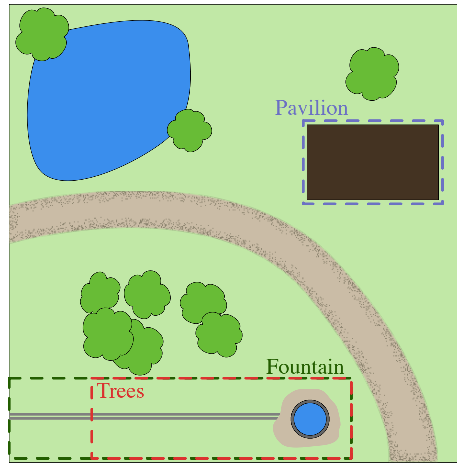


Figure 6.1: The variational picture after resizing the previously nested *Fountain* area to contain the *Trees* area, in order to depict the connecting water pipes.

originally non-variational which are now contained in *Fountain*, those which were originally in *Trees* but outside of *Fountain*, and those inside of *Fountain*. They are transformed as follows.

$$x \mapsto \mathit{Fountain}\langle x, x\rangle$$

$$\mathit{Trees}\langle x, y\rangle \mapsto \mathit{Fountain}\langle \mathit{Trees}\langle x, y\rangle, \mathit{Trees}\langle x, y\rangle\rangle$$

$$\mathit{Trees}\langle x, \mathit{Fountain}\langle y, z\rangle\rangle \mapsto \mathit{Fountain}\langle \mathit{Trees}\langle x, y\rangle, \mathit{Trees}\langle x, z\rangle\rangle$$

Although we do not depict it here, we could similarly envision scenarios where we want to shrink regions. The swapping works in exactly the opposite way to expanding. The only difference is we must remove part of the variability by performing a selection of the dimension that we are shrinking. Since we need to choose one alternative or the other to select, we defer to the value of the view decision. This gives the user control over the different possibilities.

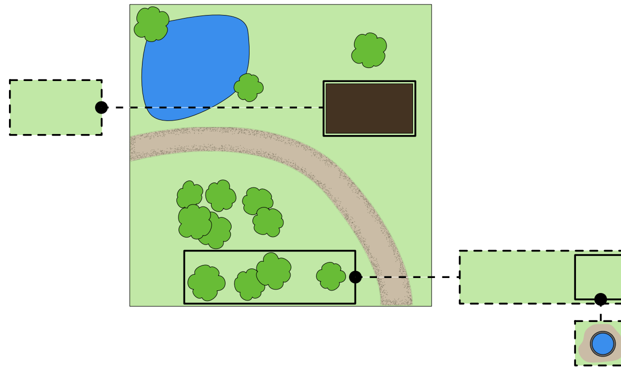


Figure 6.2: A Variational Area Tree (VAT) that showing an entire variational picture at once. Here the VAT for the view decision $\{Trees.l, Fountain.l, Pavilion.r\}$ is shown.

6.8 Variational Area Trees

Although the described model of variational pictures is sufficiently flexible to build variational pictures, so far we have limited ourselves to viewing a single variant at a time. While this simplifies editing operations, there is still a need to produce overviews to better understand and navigate the pictures, which calls for a visual language with a *graphical* domain (Erwig et al., 2017). To this end, we present *variational area trees* (VATs), a diagrammatic approach to viewing an entire variational pictures simultaneously.

VATs show the currently selected variant in its entirety as a root node, and then also all of the other possible selections as branches. Left and right alternatives are always connected via a dotted line and shown either to the left and right of one another or above and below, in order to use space more efficiently (we assume a reasonable layout algorithm). Dotted outlines show unselected variants and solid outlines indicate selected ones. A variational area tree for the view decision $\{Trees.l, Fountain.l, Pavilion.r\}$ the park example are shown in Figure 6.2.

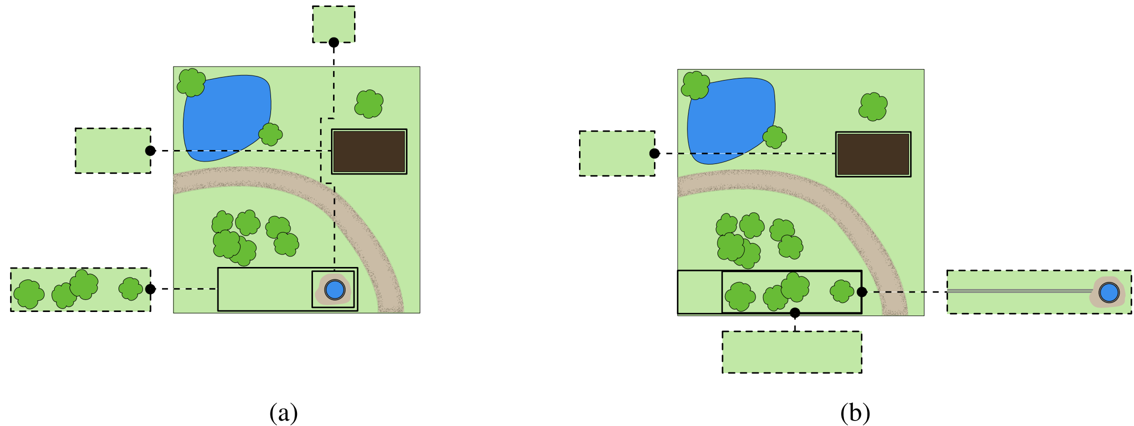


Figure 6.3: Additional configurations of the park picture VAT. In (a) we see the view decision $\{Trees.r, Fountain.r, Pavilion.r\}$, and (b) shows the case after the commuting of the *Trees* and *Fountain* regions for the view decision $\{Trees.l, Fountain.l, Pavilion.r\}$.

VATs have a number of use cases. Obviously, they provide a concise overview of an entire variational pictures including all of its variants. In addition, the design of VATs has a number of useful properties.

First, the total number of regions shown gives the total number of different drawing areas and provides a clue to the variability of the picture. Second, since unselected areas are never nested and thus all unselected areas are always placed on the top level, counting all top-level unselected areas provides a concrete measure of what is hidden in the current view. For example, in Figure 6.2 there are three top-level unselected areas indicating that there are exactly three hidden portions of the picture. Third, the number of boundaries that are crossed by a (horizontal or vertical) connector line indicates the nesting level of that choice/variational area, and the types of the crossed boundaries (unselected vs. selected) tell immediately what decisions have to be flipped (at minimum) to make the area visible. For example, in Figure 6.3a, the unselected fountain area is connected by a line that crosses two boundaries (the trees area boundary and the main picture boundary) since it is nested two levels deep. Fourth, VATs illustrate nicely where in the choice tree the current variant is located. For

example, the selected part of the VAT in Figure 6.3(a) is located rightmost/bottommost, which means that the right alternative must be selected in all dimensions.

VATs can also serve as quick navigation tools. It is easy to conceive of an interface in which a user can zoom out to a VAT and then change the selection of arbitrary dimensions quickly. Finally, VATs could also support compound operations or queries. For example, a picture creator may want to view the parts of the picture that are not variational or all the areas affected by a particular dimension. These kinds of operations could be supported by a simple filtering operation on VATs. Figure 6.3 demonstrates some additional examples for different selections and after expanding the *Fountain* region as done in Section 6.7.

Chapter 7: Domain-Specific Language for Information Visualization

This chapter will introduce a Haskell domain-specific language (DSL) for information visualization. Initially it will only support plain visualizations. By first establishing the language constructs for constructing plain visualizations, we can treat the variation as an orthogonal concern. This helps to clarify what functionality is supported by the language itself and what variation adds to that. This chapter will describe the basic model implemented by the DSL, and in Chapter 8 we will describe updating it to, among other things, support variation.

7.1 Some Basic Datatypes

The purpose of the DSL is not just to support visualization, but also to serve as a tool for the efficient and flexible information visualization in general. 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, this is usually 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. It is also worth noting that the definitions are not

intended to support completely comprehensive visualization construction. As already summarized in Section 1.3, this DSL implements a model of scalable, unitless, and composable visualization elements. This is intended to show how such an approach works and could, in principle, be extended to support more graphical elements.

```
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
```

It is always possible to construct visualizations by manually building structures out of these basic data types For many 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, making the manual construction innecsesary. Examples of such combinators include the following. Some boilerplate is omitted. More detail will follow in later sections.

```
barchart :: [Double] -> Vis (Mark Double)
barchart vs = NextTo $ map (\v -> Fill …) vs

piechart :: [Double] -> Vis (Mark Double)
piechart vs = NextTo $ map (\v -> Fill …) vs
```

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 chapter will demonstrate practical examples of these and how they can be used to visualize data.

7.2 Creating Simple Charts

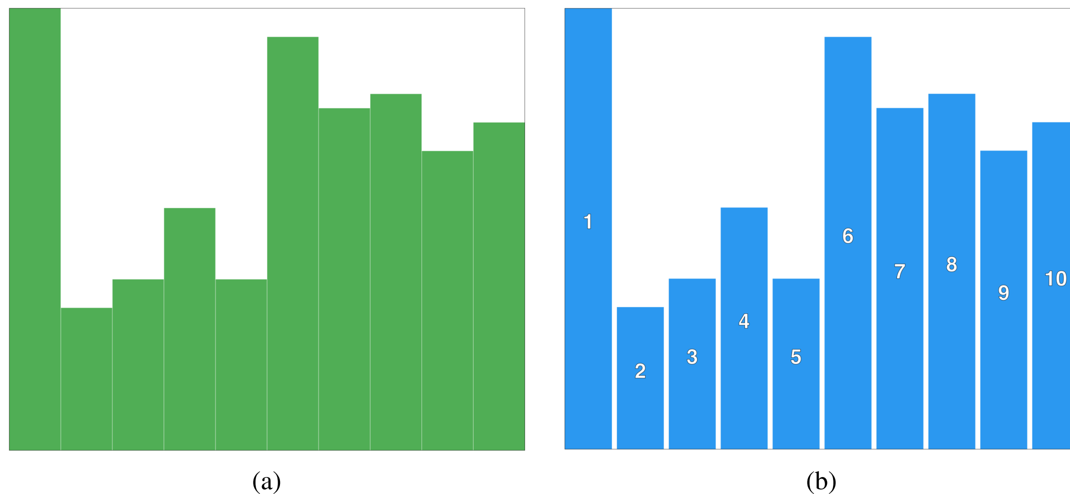
Bar charts are one of the simplest and most common visualization types, making them a good way to demonstrate the practical use of this DSL. As mentioned in the previous section, a high-level function `barchart` is appropriate for the most straightforward use cases. It works by choosing a set of default options without prompting the user. Given some data, `myData`, we can construct a chart as follows.

```
myBars = barchart myData
```

The output of this simple visualization is shown on the left of Figure 7.1. 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.

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

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

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.



(a) (b)

Figure 7.1: Two simple visualizations showing (a) the basic bar chart with only default values and (b) a basic bar chart with a few simple aesthetic customizations.

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 7.4.

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 7.1.

Note that these modifications are expressed by transformations of the existing visualization; it is not required to edit or rewrite the original visualization. Going through this process repeatedly could still become tedious, which can be mitigated to some extent by saving visualization customizations for later reuse. For example, we could write and keep the following function.

```
faveBars = (‘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 and spacing.

7.3 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. First we do a bit of data mangling. We have a sequence of values indicating the ticketed class (or crew status) of each passenger. We want to count the number in each class.

```
classDat = [1, 1, 1, …, 4, 4]

classSize = map length (group classDat)
```

Now we can begin by creating a bar chart showing this information, as in the previous example. This intermediate result is shown in Figure 7.2.

```
classLabels = map show classSize

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

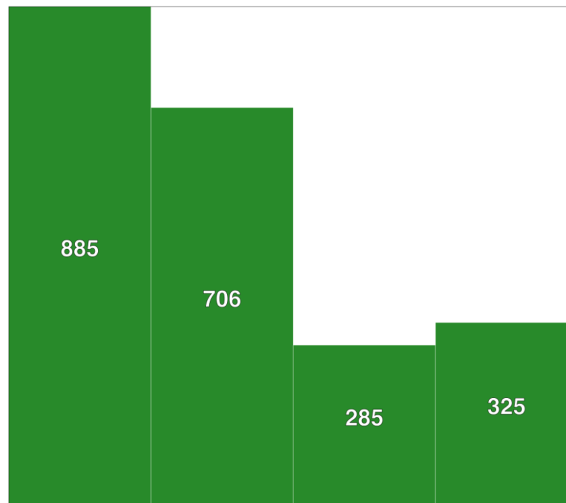


Figure 7.2: A bar chart showing passenger data from the RMS Titanic.

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

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

[1]The use of pie charts is sometimes considered poor practice because it requires readers to perform area estimates, which can be inaccurate (Cleveland and McGill, 1984). However, pie charts remain popular and widespread, and thus we support them.

To fully understand how these functions work, it is helpful to introduce some additional details. First, let us look more closely at the definition for the `Vis` data type that was reproduced in Section 7.1.

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 7.3.

```
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.

That means the orientation of the sectors needs to be flipped. This is exactly the purpose of the `reorient` function. By itself, `reorient` will toggle the orientation of all the marks in the visualization it is applied to by swapping the values bound to the width and the height. Composed with the `circular` function it will transform a vertical bar chart directly into a pie chart.

The complete code for the pie chart is shown below, and the output can be seen on the right in Figure 7.3. 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
```

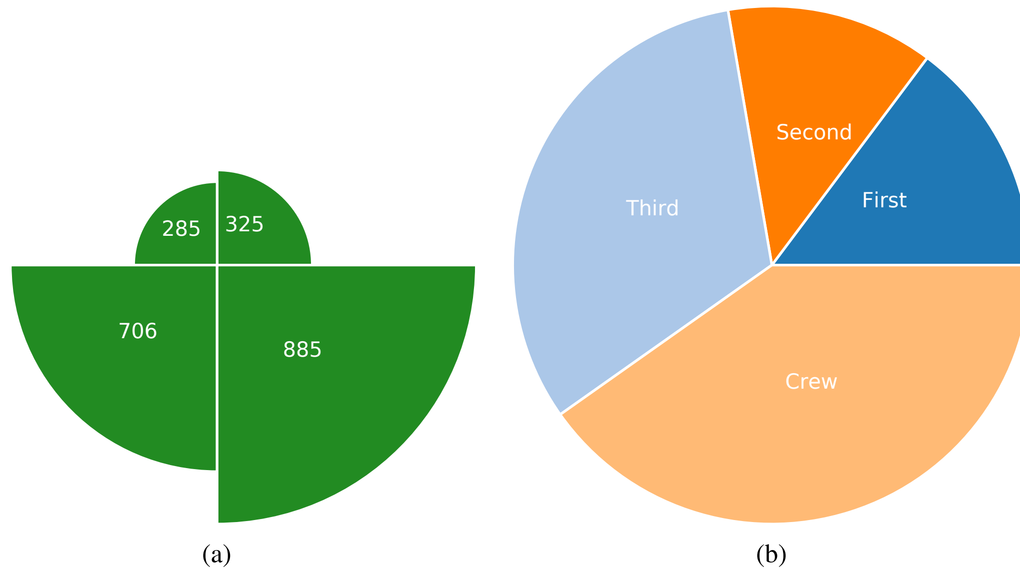


Figure 7.3: Two charts created by transforming an earlier bar chart. In (a) a rose chart in which the data is driving the radii of the wedges, and in (b) the same visualization recolored, relabeled, and reoriented so data drives the sector angles instead of the radii.

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 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.

7.4 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 :: Vis (Mark a) -> Vis (Mark a) -> Vis (Mark a)

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, users can create and save their own. Two of the built-in examples are listed below.

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

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 need to do a bit more data manipulation first in order to filter the passengers by survival status and then group them.

```
titanicDat :: (Int,Int,Int,Int)

groupClass = groupBy (\(c1,_,_,_) (c2,_,_,_) -> c1 == c2)

survDat = groupClass $ filter (\(_,_,_,x) -> x == 1) titanicDat

decDat =  groupClass $ filter (\(_,_,_,x) -> x == 0) titanicDat
```

Now we can construct bar charts for each.

```
survived = barchart survDat `color` blue

deceased = barchart decDat `color` orange
```

Finally, we compose them using both `grouped` and `stacked` (just for illustrative purposes) and then compose the results into a single, more complex visualization.

```
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 7.4. 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.

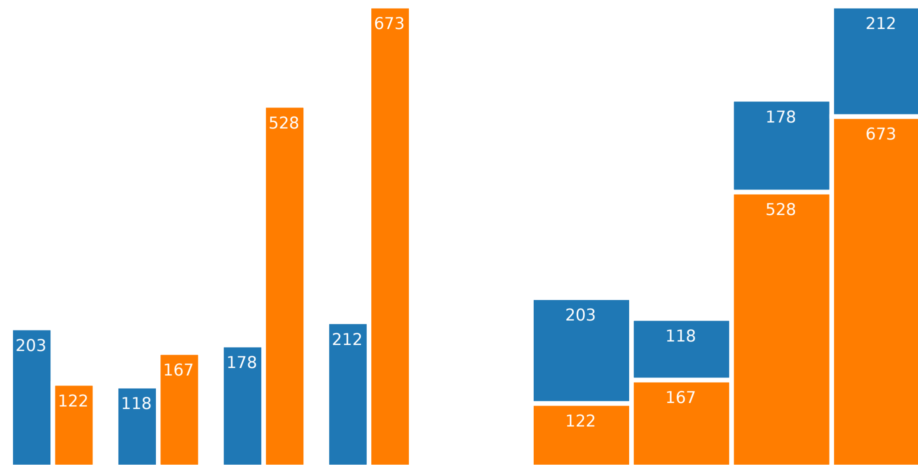


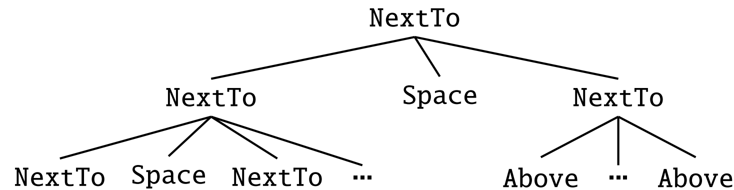
Figure 7.4: A single visualization showing two different representations of the same information. The blue bars show the number of survivors for the given travel class aboard the RMS Titanic and the orange show the number of deceased. This example demonstrates three types of composition: composing marks horizontally to construct bar charts, composing bar charts by stacking and grouping, and composing complete visualizations by dividing the canvas.

7.5 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.



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 (see Metoyer et al. (2012)) 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.

7.6 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 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 $0 \ldots 1$, 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 = getHeight m
                c | h > 0.75  = red
                  | h > 0.5   = orange
                  | otherwise = green

conditionalBars = fmap redHigh $ barchart myData
```

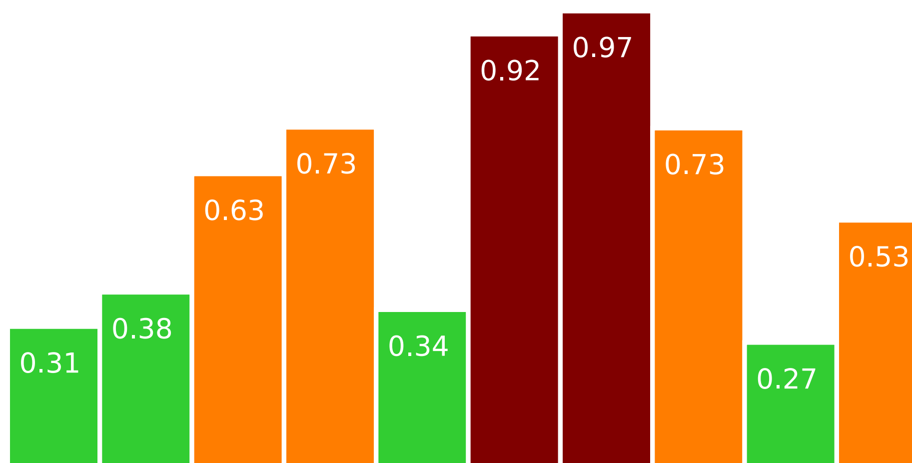


Figure 7.5: Conditional visualization formatting applied using `fmap` from the Haskell `Functor` type class. Bars with a height above 0.75 are colored red, bars with heights between 0.5 and 0.75 are colored orange, and the rest are colored green.

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 7.5. 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.

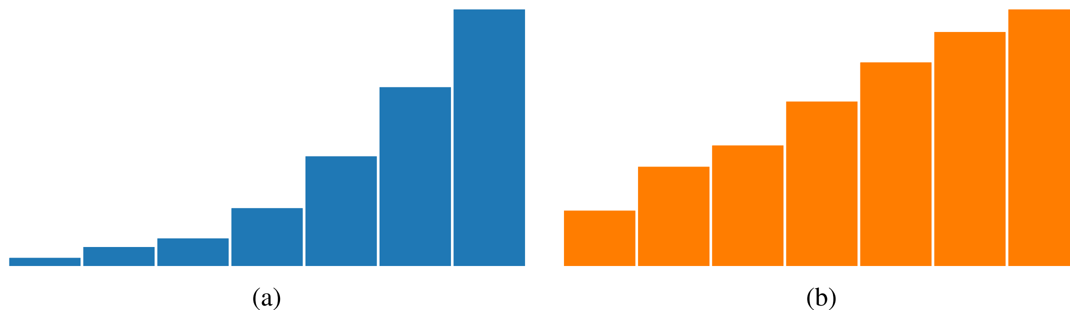


Figure 7.6: In (a) is a simple barchart of data that appear to show an exponential trend and in (b) is the same barchart after log-transforming each of the bars, showing what appears to be a linear trend.

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 ‘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 7.6. From this we 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.

7.7 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 7.6, which will show both the original and log-transformed bars at the same time. This is demonstrated in the following code, which assumes the definitions from that example are still available.

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

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

The previous definition of `markLog` 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 7.7. 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. Note that it is also possible to

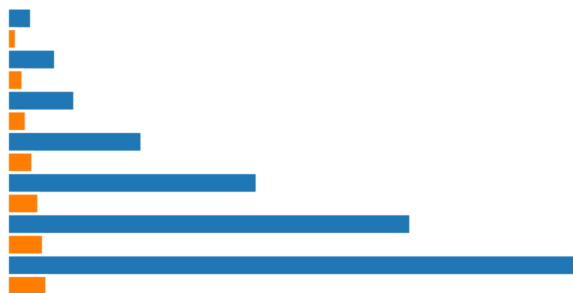


Figure 7.7: Another example of a log-transformed visualization, as in Figure 7.6, except this time using the visualization `Monad` instance to generate new bars in place beside their counterparts.

construct this example by generating the bar charts individually and then useding `grouped`, but this will not always be try as we will see in later examples. 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 7.6.

```
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 7.6, but also another operation to somehow zip together and merge the two data columns. This is true even for tools such as ggplot2 (Wickham, 2009), 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 zooming operations (sometimes called drill-down and roll-up for zooming in and out, respectively). Switching between an overview and a more detailed, granular view is a crucial component in visual data analysis (Shneiderman, 1996). 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 7.8.

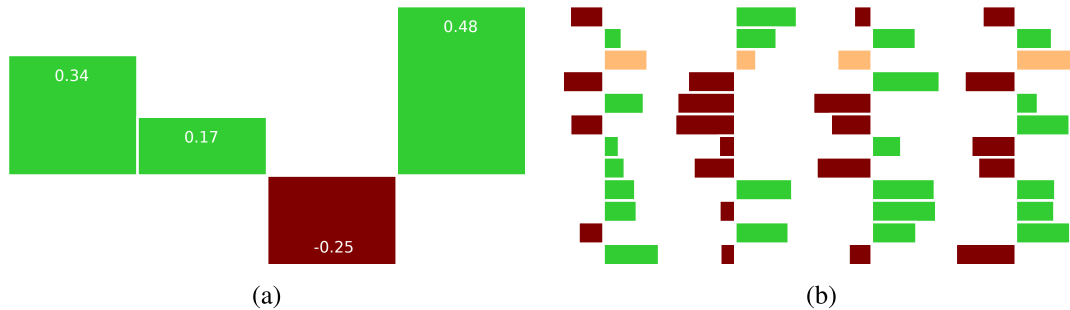
After seeing this subset of data we decide that we also want to see context. That is, we want to see additional months to see how the March data fits in by performing a zoom-out (or roll-up) of the visualization. We can make use of monadic functions in the following way.

```
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 7.8.

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



(a) (b)

Figure 7.8: 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.

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.

7.8 Visualization Comprehensions

By using the GHC[2] 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.

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

As an example, consider the situation where we want to analyze data related to employee performance in an office and choose appropriate training courses. Our data contains 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 thirty employees.

The second data set contains data for five possible training courses. The specific values show the estimated improvement in employee performance that attending the course provides. That is, a high value means that an employee attending the course is more likely to see a higher increase in performance.

Our goal, for bureaucratic reasons, is to ensure as many employees as possible meet the standard by sending underachievers to training courses. That is, we want to ensure every employee's performance to be rated as a non-negative value. 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. That is, we want to find the cheapest course for each under-performing employee that is estimated to improve that employee's performance to a non-negative value. To support this process, we want to see all possible, relevant combinations at once.

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, namely two bar charts composed vertically.

```
employees = barchart emplData ‘color‘ orange

courses = barchart courseData ‘color‘ blue

ecBars = employees ‘above‘ courses
```

This visualization is not sufficiently helpful for our task. Since it shows two separate charts, we still need to manually estimate the effect of each course on each employee. Instead, we want to visualize the computed final, estimated end result of each possible employee/training course pair.

Our desired final result should be a composition of bar charts. There will be one chart per underperforming employee (some number no larger than, and likely much less than, thirty). Each of those composed charts should contain five bars, one per potential training course. The height of the bars will show the employee's current performance plus the estimated improvement. That is, if an employee has a performance value of -15 and the paired training course has an estimated improvement of 30, we expect the bar for this pairing to have a height of 15. Negative bars will show cases where the improvement is not enough to cross the zero threshold. From that result we will be able to 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 is a complex scenario, but is well suited to an approach based on comprehensions. Before we get to those details, we need some helper functions.

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

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

We begin by defining the `courseColor` function. This is not shown because it is boilerplate code similar to the `redHigh` function from Section 7.6. Its purpose is to assign colors uniquely to each training course. That is, course 1 is colored blue, course 2 is colored orange, and so on. 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 into a single bar. This allows us to compute the estimated final performance for employees that attend training courses by adding their original performance value to the estimated result of the various courses.

Now we are ready to use a visualization comprehension to generate the new chart. The comprehension works as follows. First, we have two generators to get the values for all possible pairings of employees and courses. We then filter those results to only those pairings for employees who are underperforming. Employees with good performance will not be sent to training. The paired performance and course values are combined using `addBars` and a color is assigned with `courseColor`.

```
perfBars = [courseColor c (addBars e c) | e <- employees
                                        , c <- courses
                                        , mGetHeight e < 0]
```

We also need to 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 also want to add space between each group of bars to visually separate them.

```
clean $ perfBars ‘space‘ 0.6
```

The rendered output is shown on the right side of Figure 7.9. 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.

This kind of operation can be even more 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.

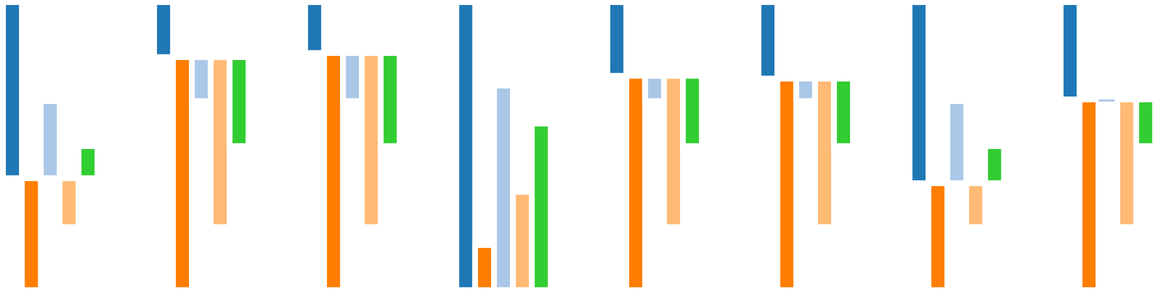


Figure 7.9: Estimated employee performance data created using a visualization comprehension to combine two existing bar charts. Labels identifying employees are omitted because of formatting limitations.

7.9 Evaluation

Evaluating visualization tools is inherently difficult (Plaisant, 2004). 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., Bostock and Heer (2009)) have used Cognitive Dimensions (Green and Petre, 1996) 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 (Bostock et al., 2011; Viegas et al., 2007). The most common option is to forego typical evaluation criteria completely and to instead demonstrate functionality equivalent with, or superior to, established tools (Cottam, 2011; Fekete, 2004; Stolte et al., 2002; Weaver, 2004; Wickham, 2009).

Some work has proposed visualization-specific evaluation techniques and criteria, or proposed task-based taxonomies that could be used to extract functional requirements (Amar and Stasko, 2005; Amar et al., 2005; Brehmer and Munzner, 2013; Kienle and Muller, 2007; Shneiderman and Plaisant, 2006). Many of these are still designed for interactive, graphical tools and are difficult to apply to a language-based solution, however.

7.9.1 Applicable Evaluation Schema

In this work, we turn to the typology of visualization tasks in Brehmer and Munzner (2013) 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. By evaluating the DSL according to this framework, we can establish whether it is, in principle, technically capable of functioning as a model for a general-purpose visualization tool.

However, we make one important distinction. The DSL is not intended to be a complete, user-facing product in itself. As is the case with Tableau/Polaris (Stolte et al., 2002), 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 7.7. 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 7.7), for sorting visualizations (via `vSort`, not demonstrated), for grouping and merging (as in Section 7.4), and for unzipping (via `vUnzip`, described in Section 7.7). 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 7.6, transformation of scales is shown in Sections 7.6 and 7.7, and the transformation and composition of bar charts is shown in Section 7.4. We also support changing visualization types completely in Section 7.4.

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 7.8 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 7.7.

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 7.4. 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 use 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 7.6 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 (Stolte et al., 2002) and ggplot2 (Wickham, 2009).

7.9.2 Conclusions

According to this set of criteria, our work meets nearly all criteria for the implementation of a visualization tool. The only real shortcoming 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, text handling for things like legends, guides and axes, and more.

The DSL presented in this chapter is suitable for creating and transforming data visualizations. Also, by virtue of the parameterization and type class hierarchy offered by Haskell, our model supports the transformation and modification of existing visualizations rather than creating new ones from scratch, potentially supporting 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. In the next chapter we will turn to an updated version of the DSL which supports variation.

Chapter 8: Supporting Exploratory and Comparative Information Visualization with Variation

This chapter presents a model for variational visualizations based on an expanded version of the domain-specific language first presented in Chapter 7. Additionally, we will present numerous examples of practical use cases for variational visualizations demonstrating their effectiveness for tasks related to exploration (Section 8.4) and visual comparison (Section 8.5).

The DSL presented in this chapter is extended from what was presented before. First, it is embedded in Purescript rather than Haskell which is similar but not identical. This is primarily because creating scalable graphics interactively from a REPL environment is easier in Purescript. We will document the relevant differences where they arise. Second, some of the main data types have undergone refactorings for simplicity, since variation can supplant some of the complexity of the original design. Section 8.1 will detail these before later sections describe the application of variation.

8.1 Redesign of the DSL

The first update to the core DSL is to the `Vis` data type. We have removed the parameterization and added several new constructors. In some ways these new constructors deepen the embedding of the language. This simplifies many tasks, since rendering a visualization is not obviously compositional (Gibbons and Wu, 2014). That is, a visualization mark alone does not contain enough information to render itself. For instance, in a vertical bar chart, the relative height of a bar will

depend on the height of the tallest and shortest bars in the chart. The parameterization will be made dispensible by the inclusion of variation in later sections. Tasks such as conditional formatting will be easily representable using variation. Below is the updated `Vis` data type.

```
data VVis = Mark MarkRec
          | V Dim VVis VVis
          | NextTo (NonEmptyList VVis)
          | Above (NonEmptyList VVis)
          | Cartesian VVis
          | Polar VVis
          | Overlay (NonEmptyList VVis)
          | Stacked (NonEmptyList VVis)
```

There are several things to note. First, we have replaced the list of child visualizations with non-empty lists. Most Purescript functions are total, unlike much of the Haskell prelude. For example, the `head` and `tail` functions in Purescript return `Maybe` values. Since there is no obvious advantage to supporting empty lists of visualizations, switching to a non-empty list representation is useful. The `Mark` data type is essentially embedded directly in the `Vis` type, and its components are all contained in the `MarkRec` type synonym.

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

```
data VPs = VPs
  { height :: Number
  , width :: Number
  , color :: Color
  , visible :: Boolean
  , orientation :: Orientation
  }
```

The new constructors `Cartesian` and `Polar` take on the role of the `rectangular` and `circular` functions from the previous version, and `Overlay` and `Stacked` replace the functions of the same name. The functionality remains mostly unchanged, except it becomes simpler to create charts in which many visualizations are stacked or overlaid since the original functions only operated on two at a time.

The `VPs` record type is an expanded analogue to the `VisualParameter` type. It contains some of the components of a visualization mark that can be driven by data (Bertin, 1999).

Since Purescript lacks a Scrap Your Boilerplate (Lämmel and Jones, 2003) library, we have also defined several functions to help make some routine traversals of visualization structures easier. For example, we can map across `Mark` constructors as follows (some boilerplate omitted).

```
mapMark :: (MarkParam -> MarkParam) -> Vis -> Vis
mapMark f (Mark fp) = Mark $ f fp
mapMark f (Cartesian v) = Cartesian $ mapMark f v
mapMark f (Polar v) = Polar $ mapMark f v
mapMark f (… vs) = … $ map (mapMark f) vs

mapVPs :: (VPs -> VPs) -> Vis -> Vis
mapVPs f = let appVP fp = fp { vps = f fp.vps }
           in mapMark appVP
```

Finally, many of the combinator functions like `barchart` and `piechart` now operate on arrays of values rather than lists. In Haskell, lists are given a special, convenient syntax using square brackets. Purescript has this same syntax but for arrays. For that reason we define many helper functions to work on arrays so that they are easy to specify even though the internal structures tend to be non-empty lists.

8.2 Adding Variation to Visualizations

Like the other chapters of this dissertation, our model of variation is based on the choice calculus. Recall the initial definition of the `VChc` type constructor from Chapter 3, which we have reproduced below.

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

This definition allows the top-level application of `VChc` to the `Vis` type, that is, a variational visualizations would have the type `V Vis`. For example, we can construct the variational chart PickColor⟨`blueChart`, `greenChart`⟩, which allows the selection between two charts as a whole with the following code.

```
Chc "PickColor" blueChart greenChart
```

Just as with the design of variational lists in Chapter 4, there are two other ways to integrate `VChc` into structures in addition to the top-level application, namely at the leaves or recursively. Application of the `VChc` type constructor at the leaves involves moving the variation into the visualization structure, applying it directly to the marks. We could redefine our visualization type to be the following.

```
data VVis = VMark (V Mark)
          | NextTo (NonEmptyList VVis)
          | ...
```

Since we likely want to avoid constructing all of our variational marks manually, we would need to update many of the built-in operations such as `barchart`. In order to construct a variational visualization where the marks are the parts that vary, we need to provide `barchart` with variational data as input. That is, instead of the type

```
barchart : Array Number -> VVis
```

we would have something of the form

```
barchart : VChc (Array Number) -> VVis
```

This new `barchart` function would then be responsible for creating a mark for each variant data value.

By pushing the type constructor down into the marks we are able to eliminate two of the major drawbacks from the top-level approach: First, redundant visualization structures can be avoided because the variation can only occur at the innermost level, and second, we can also determine exactly where the variation occurs by observing the marks and their variation directly.

However, the leaf-level application of `VChc` prevents many kinds of useful variations in visualizations. For example, we are not able to represent a bar chart that is either a vertically or a horizontally oriented bar chart depending on the selection. These have subtly different structures (`Above` as opposed to `NextTo`), and since we only allow marks to vary, and not the composition operators, this is not possible. Therefore, this approach is obviously too limited to be useful in the general case.

A final possibility is to integrate the variability directly into the recursive structure of the visualizations, allowing it to occur wherever is most appropriate for the desired effect. We can add a new case to the visualization definition as follows.

```
data VVis = ...
          | V Dim VVis VVis
```

The added flexibility of this recursive application of `VChc` allows us to avoid any issue with being unable to represent particular kinds of variation. Moreover, assuming the variation is allocated judiciously, we can also avoid unnecessary redundancy.

However, the burden of choosing where to integrate variation is shifted onto the visualization author. Given such a system, the user must have a sufficient understanding of its inner workings to not only know when it is preferable to move the variability inward or outward in the visualization structure, but also how to actually achieve this by using and defining operations. Still, given the drawbacks of the top-level and leaf-level approaches, the additional demands on the user seem to be reasonable.

8.3 Rendering and Navigating Variational Visualizations

Having found a suitable way to integrate variability with our visualizations, we still need to decide how they could be presented to and navigated by users. We have implemented a prototype, which renders variational visualizations according to the model of variational pictures described in Chapter 6, that is, it produces a variational picture where each picture variant is one of the visualizations variants. All of the figures in this chapter have been generated by this prototype.

A tool for displaying variational visualizations must allow users to navigate among the different variants. For simplicity, we chose an approach based on standard browser form elements. We extract all of the dimensions that a variational visualization contains and produce a checkbox for each. This checkbox toggles the selection in that dimension. When one of the dimensions is toggled on, radio buttons are shown to select between either the left or right alternatives. This scheme allows users to specify any decision, whether partial or total, and see the rendered result.

When a decision is not total, and variation still exists in the visualization being rendered, it is not obvious what should be drawn. One possibility is to draw nothing for the parts that are unselected and

just indicate that a selection must be made first, such as by a box or outline. Another, less limiting approach, is to show all variants of the current visualization at once, side-by-side, using a small multiples approach. This not only supports comparison but also gives users the ability to see visually how much variation remains unselected in their visualization.

We have also used colors to map the navigation interface to the portions of the rendering canvas that they affect. For example, if a particular dimension toggles the height of a bar, we outline that bar's space with a colored, dashed outline and color the corresponding UI elements with the same color. These colors are assigned automatically. A screenshot of the prototype is shown in Figure 8.1. The visualization in the figure is a pie chart with three dimensions of variation. For each dimension, the left alternative hides details related to a particular geographic region while the right alternative shows the full details. Specifically, the left alternative of the Region 1 dimension is selected, which corresponds to the green wedge. The right alternative of the Region 2 dimension is selected, and so the three blue/grey wedges are shown rather than a single composite. Finally, Region 3 is unselected and so both variants are shown at once, corresponding to the remaining wedges.

8.4 Variation for Exploratory Visualization

As motivated in Section 1.3, there are two main analysis tasks that variational visualizations can help support, namely exploratory visualization and comparative visualization. This section discusses the former and Section 8.5 will cover the latter.

There are many ways that variation can support users in performing exploratory visualization tasks. We demonstrate this through a series of examples belonging to one of three separate categories: generating variation, transforming and maintaining variation, and aggregating variation.

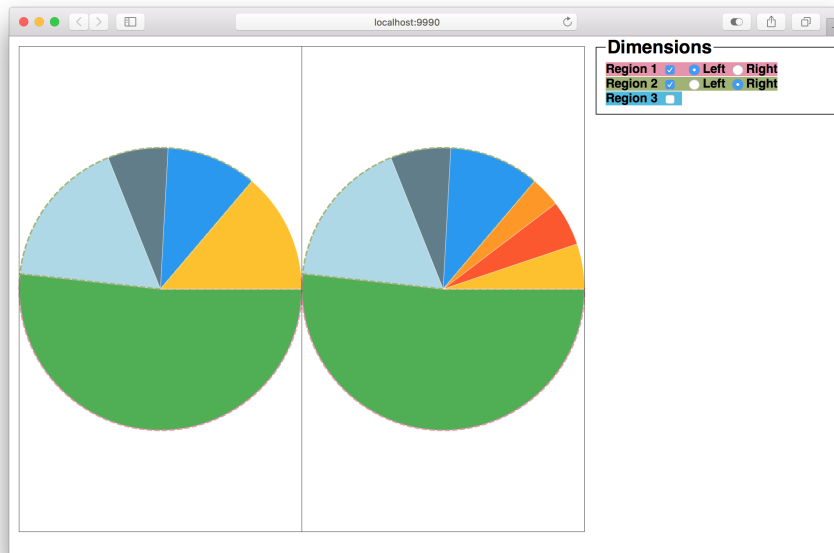


Figure 8.1: A screen capture of our prototype user interface showing possible configuration/selection options. On the left is the rendered visualization currently being constructed and on the right are the interface elements which allow the user to navigate among the variants.

8.4.1 Generating Variation

Visualization Provenance The examples in this section are focused on ways programmers can generate variation in their visualizations. Aside from constructing the data types manually, one simple and useful way of generating variation is with the `vApp` function:

```
vApp :: (VVis -> VVis) -> Dim -> VVis -> VVis
```

This function is reminiscent of function application, with good reason. Its purpose is to apply a visualization transformation function in a way that preserves both the original and the transformed result. As an example, let us create a simple barchart.

```
myBars :: VVis
myBars = barchart myData
```

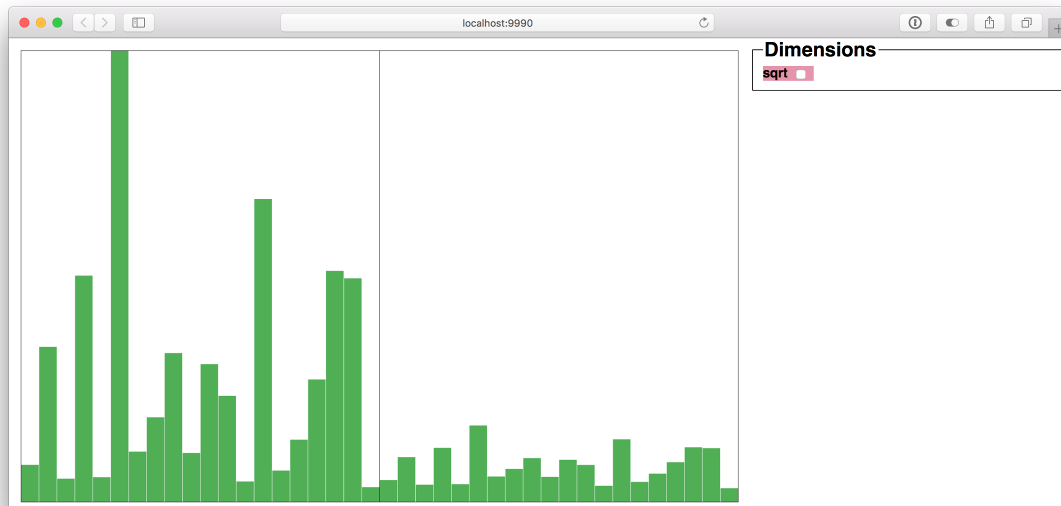


Figure 8.2: Generating variation. By using the `vApp` function we can transform a visualization and keep the original as an alternative. By chaining applications, it is possible to track the provenance of a visualization completely, ensuring every variant is always reachable.

After viewing it, we decide that it may be valuable to see the same data after applying a square root transformation. We can first define a function to transform the height the way we expect and then map it across the visualization using `vApp`. Without variation, we could have perform this transformation but would have needed to manually compose the original and transformed version. Using variation handles this automatically and systematically. The two variants are shown in Figure 8.2 with the prototype user interface. Since we have not selected either variant, both are shown as small multiples.

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

vBars :: VVis
vBars = vApp (mapVPs (onHeight sqrt)) "Sqrt" myBars
```

By chaining applications of this function 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 navigate through the visualization in order to revisit older states.

Branching Visualizations In the interest of genuinely supporting visualization provenance, we need to be able to do more than just create a linear sequence of changes. The `vApp` function just creates top-level choices, which is somewhat restrictive. Instead, we need a way to create new variants by branching off from arbitrary points in an existing choice tree. For that, we can use the `branch` function.

```
branch :: (VVis -> VVis) -> Decision -> Dim -> VVis -> VVis
```

The implementation for this function is not quite as simple as `vApp` however. Since a decision often does not correspond directly to a path through the choice tree, we need to calculate when to continue traversing to the branch point, and when to stop. We want to keep traversing as long as there are still nested `V` constructors that have to be selected one way or another by our decision. As soon as we run out of (relevant) `V` constructors, we want to stop, typically before reaching a leaf node. We need a helper function to check for this.

```
lacksDims :: Decision -> VVis -> Boolean
lacksDims dec (V d l r) = case lookupDim d dec of
  Nothing -> lacksDims dec l && lacksDims dec r
  _       -> false
lacksDims _   (Mark _)  = true
…
```

The remaining cases are just boilerplate to recurse on their child visualizations and so are not reproduced. With this, we can return to `branch`.

```
branch :: (VVis -> VVis) -> Decision -> Dim -> VVis -> VVis
branch f dec newDim (V dim l r) =
  case lookupDim dim dec of
    Just L  -> V d (branch f dec newDim l) r
    Just R  -> V d l (branch f dec newDim r)
    Nothing -> if branchDone dec (V dim l r)
               then vApp f newDim (V dim l r)
               else V d (branch f dec newDim l)
                        (branch f dec newDim r)
branch f dec newDim ...
```

Again, the remaining cases are boring and are essentially the same as the `Nothing` branch of the above case statement. With this machinery, we can now produce branching visualizations. Lets look at a 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:

$$Sqrt\langle \ldots,\ \ldots \rangle$$

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. We can achieve it as follows. Unlike Haskell, Purescript does not have special syntactic sugar for tuples.

```
heightRecip :: Vis -> Vis
heightRecip = mapVPs $ onHeight (\x -> 1.0 / x)

dec :: Decision
dec = singleDec (Tuple "Sqrt" L)

recipBars :: VVis
recipBars = branch heightRecip dec "Recip" vBars
```

This produces a new visualization with the following updated variation structure.

$$\mathit{Sqrt}\langle \mathit{Recip}\langle \ldots,\ \ldots \rangle,\ \ldots \rangle$$

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

8.4.2 Transforming and Maintaining Variation

The previous section introduced some ways in which we can systematically introduce variation into our 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.

Replacing Variants One straightforward task that does not affect the structure of the variation is to replace particular variants. In Section 8.4.1 we showed how to add variants using the `branch` function. 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.

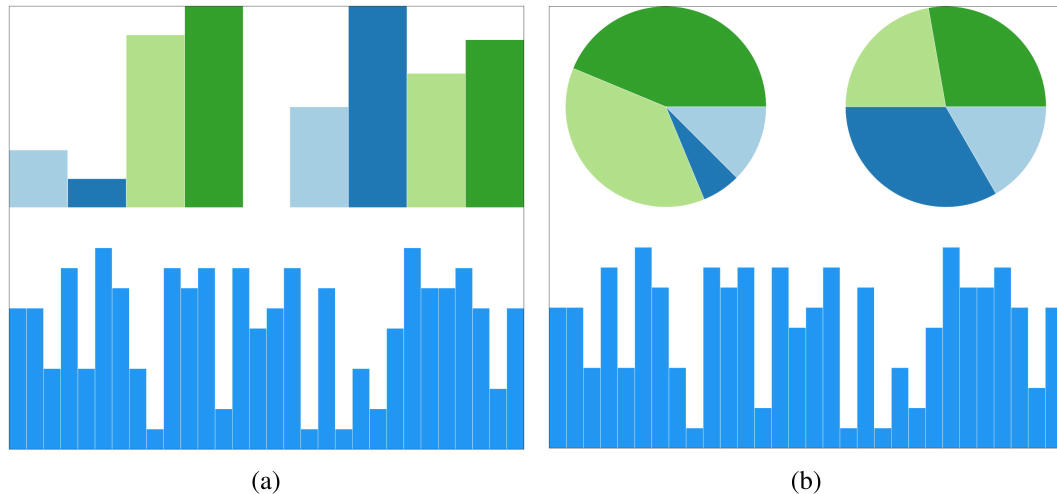
We could use the `branch` function to transform each variant, adding a new dimension in which the right alternatives now contain the versions with labels. However, since adding global dimensions grows the number of variants exponentially, we may wish to avoid doing so. In cases when we do not care to preserve the original variants, we can instead turn to `mutate`. The definition can make good use of the `lacksDims` helper function from earlier.

```
mutate :: (VVis -> VVis) -> Decision -> VVis -> VVis
mutate f dec (V d l r) = case lookupDim d dec of
  Just L  -> V d (mutate f dec l) r
  Just R  -> V d l (mutate f dec r)
  Nothing -> if lacksDims dec (V d l r)
             then f (V d l r)
             else V d (mutate f dec l) (mutate f dec r)
mutate …
```

Recall that our goal was to mutate visualization variants to add labels. Since we want to do this to all of the variants, rather than just some, we can pass in an empty decision as the argument which will ensure that no sub-tree is selected.

```
myVis :: VVis
myVis = Above [NextTo …, …] …

myLabVis :: VVis
myLabVis = mutate (\v -> v ‘label‘ ["One", "Two", …, "Six"]) 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 this in terms of `mutate`. Note that `(<<<)` is the Purescript operator for function composition, since the typical dot is used for record access.



(a) (b)

Figure 8.3: 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.

```
replace :: VVis -> Decision -> VVis -> VVis
replace = mutate <<< const
```

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 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 for those we wish to change the type to a pie chart in order to better see the ratio of the individual quarter 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.

Once again we can make use of decisions to indicate where in the visualization we would like our predicate to be tested. First let us define the `condMutate` function, which works like `mutate` but with the additional machinery to check whether the predicate applies.

```
condMutate :: (VVis -> Boolean) -> (VVis -> VVis) -> Decision -> VVis -> VVis
condMutate p f dec v@(V d l r) = case lookupDim d dec of
    Just L  -> V d (condMutate p f dec l) r
    Just R  -> V 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 ...
```

The `isVis` function is a function that checks whether a `VVis` is a self-contained visualization or a composition of several visualizations. In the latter case we do not yet want to apply our predicate. Now we can turn to defining our predicate to check for 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`. Now we can apply it to our existing visualization to perform the mutation. The output is shown in Figure 8.3.

```
pies :: VVis
pies = condMutate fourBars (Polar << reorient) emptyDec myVis
```

It is also possible to apply this generalization to `vApp` and `branch`, that is, we can also use a predicate to add conditions to operations such as `vApp` and `branch`. It is not difficult to envision a function, for example, to branch on visualizations indicated by a particular decision only if they also meet some secondary condition.

8.4.3 Aggregating Variation

The third and final category of tasks we want to 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.

Selection When we generate variation in a visualization, often it is 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). The definition 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` in directly as a constructor. Here is a partial definition since it is primarily boilerplate.

```
selectVis :: VVis -> Decision -> VVis
selectVis (V d l r) dec = case lookupDim d dec of
  Just L  -> selectVis l dec
  Just R  -> selectVis r dec
  Nothing -> V d (selectVis l dec) (selectVis r dec)
selectVis …
```

Supposing we had used the `vApp` function to produce a variant of our chart with labels, which were originally omitted.

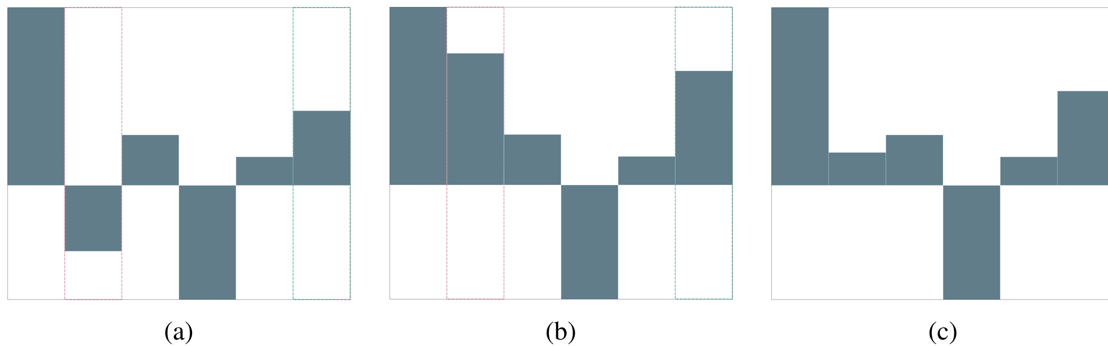


Figure 8.4: Using a flattening approach to dealing with variational data. The left and center figures show two variants of the same variational visualization. Two of the data values vary. On the right is the visualization after using `flatten` to compute the average values of the variational bars and charting that value instead. Although subtle, the lack of dotted outline in (c) shows that the variation has been removed.

```
labVis :: VVis
labVis = vApp (\v -> label v myLabels) "labels" 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 `vApp`. All we need is selection. Note that in this case we have a reference to `myVis` so this is not strictly necessary, but that will not always be the case.

```
noLabVis :: VVis
noLabVis = let dec = singleDec (Tuple "labels" L)
           in selectVis labVis dec
```

We could also extend this idea with conditionals, just as we did with `condMutate`.

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) -> Decision -> VVis -> VVis
flatten f dec (V d l r)
    | lacksDims dec l && lacksDims dec r = f l r
    | otherwise = V d (flatten f dec l) (flatten f dec r)
flatten f dec (NextTo vs) = NextTo $ map (flatten f dec) vs
flatten …
```

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, i.e., variational data. As shown in Figures 8.4a and 8.4b, 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 :: VVis -> VVis -> VVis
avgHeight (Mark f1) (Mark f2) =
  let h1 = getHeight f1.vps
      h2 = getHeight f2.vps
   in Mark (f1  vps = setHeight f1.vps ((h1 + h2) / 2.0) )
```

Now we can apply that to our existing visualization and get the result shown in Figure 8.4c.

```
avgBars :: VVis
avgBars = flatten avgBars emptyDec myBars
```

Admittedly, in this small example it may have been simpler to perform this transformation directly on the data rather than by flattening. That will not be true in general, however. It is not difficult to envision stacking or overlaying the variant bars instead of averaging them.

8.5 Variation for Comparative Visualization

Having demonstrated examples of using variation for exploration, we now turn to comparative visualization. This section will begin by showing some relatively simple examples of variational visualizations and then, in Section 8.5.2, will show how those examples and the fundamental approaches they represent satisfy the requirements of comparative visualization.

8.5.1 Example Comparative Visualizations

One way variation supports comparison is by controlling the level of detail shown for a set of data. For example, suppose we want to produce a pie chart showing a breakdown of some costs for geographic regions in the United States much like the example from Section 8.3. We might want to show an overview for areas such as West coast, East coast, South, etc. (Figure 8.5a). However, we also want to make the details of the states comprising each region available on demand by selecting corresponding variants (Figure 8.5b). We can encode the zoomed out and zoomed in versions of each pie wedge in a choice, allowing variation to take care of the exponentially many different versions.[1] The rendered output is shown in Figure 8.5.

```
Polar $ NextTo [ Chc "West"  (… 2.0) (… 1.0, … 0.4, … 0.6)
               , Chc "East"  (… 0.8) (… 0.2, … 0.3, … 0.3)
               , Chc "South" (… 3.0) (… 1.9, … 0.7, … 0.4)]
```

Variation is not just useful for comparing aesthetic options, however. One of the major advantages that an approach based on variation gives us is the ability to work directly with variational data. Suppose we want to examine source code that is annotated with C-preprocessor macros such as `#ifdef` to to chart the number of lines of code in each block, similar to the example used for variational lists

[1] A few details are omitted here which are shown in full in Appendix C.

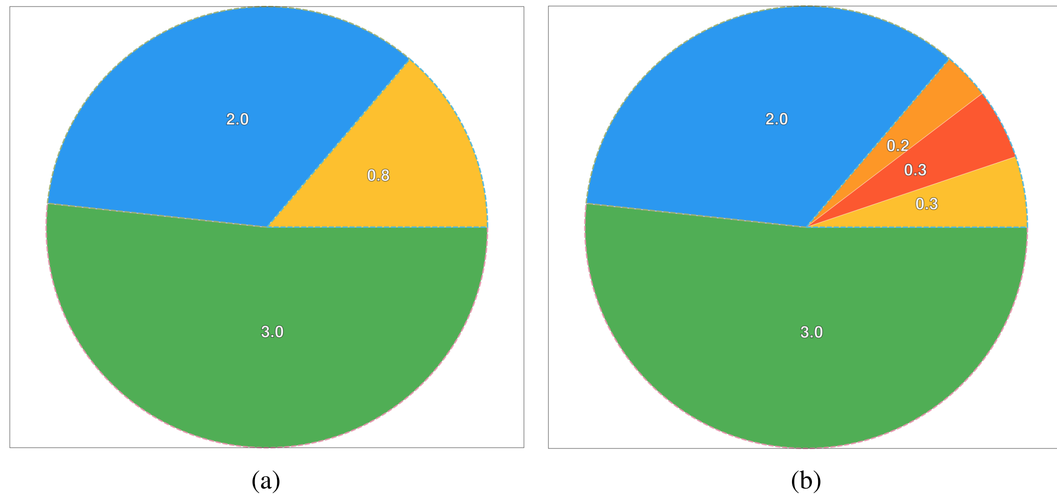


Figure 8.5: Comparative visualizations for exploring visualization details; (a) Summary visualization that corresponds to the decision $\{West.l, East.l, South.l\}$; (b) Revealing details for the east with decision $\{West.l, East.r, South.l\}$.

in Chapter 4. We could produce each of the 2^n possible configurations (where n is the number of configuration options), count the relevant lines of code in each, and then produce a separate chart for each configuration.

This, however, is generally infeasible since large software projects have hundreds or even thousands of configuration options (Liebig et al., 2010). However, if we instead count the number of lines directly, but encode the values as variational numbers corresponding to the preprocessor macros, we can perform all of the work in a single pass. Since the data and the visualization tool would be making use of the same variation representation, we can then just chart the results directly. For example, we could make use of the `vbarchart` function, which is the variational equivalent of `barchart`.

```
vbarchart : Array (VChc Number) -> Vis
```

The viewer of the visualization can then navigate the different configurations to compare the results for each.

Finally, we can compute modified versions of visualizations to compare between using variation. Suppose, for example, we have two bar charts showing monthly earnings over the past two years for a business. We would now like to compare the two years directly by charting the change from the first year to the second for each month. We can use the `vZipWith` function, which accepts two visualizations as input, traverses them in parallel, and computes a new visualization element based on a binary operation. In this example, we want to subtract the height of the bars in the second chart from the height of the bars in the first. We can simply use `vZipWith` together with the `minusHeight` function (refer to Section 8.4 for details), which calculates the difference between the heights of the individual bars.

```
vZipWith minusHeight bars1 bars2
```

An example of this computed visualization, composed next to the two original charts, is shown in Figure 8.6a.

Similarly, we can also compute data transformations directly on visualizations. For example, we may have a set of data already charted for which we also want to see both the log-transformed version and a square-root-transformed version. Moreover, we want to see the original and transformed data at the same time, overlaid using transparency. We can achieve the result shown in Figure 8.6b in the following way. Note that the figure shows the output when we have not selected either alternative, meaning both are shown using a small multiples approach.

```
Chc "MapType" (Overlay [bar1 `alpha` 0.5, mapVPs (onHeight log) bar1])
              (Overlay [bar1 `alpha` 0.5, mapVPs (onHeight sqrt) bar1])
```

Using the `mapVPs` and `onHeight` helpers from Section 8.4, we can apply the transformations directly to the visualization elements and then compose them into an overlaid variational chart.

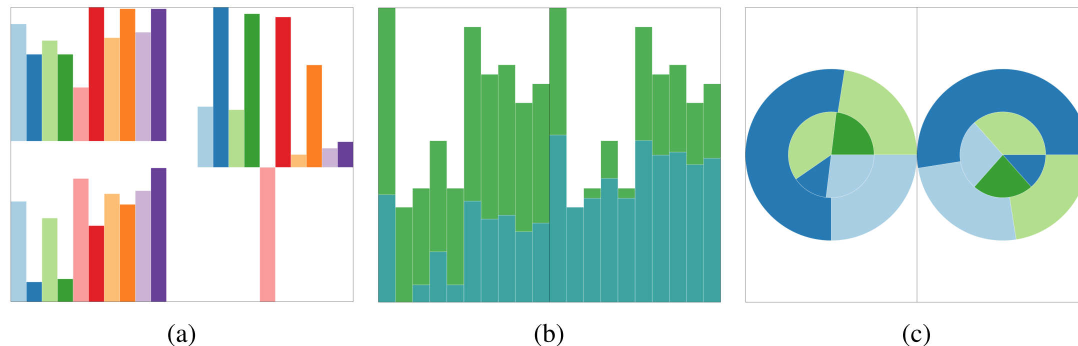


Figure 8.6: Examples of comparative visualizations using hybrid designs. In (a) we have two charts on the left which are zipped together to produce the chart on the right by subtracting the heights of the lower chart from the top one. Note that each chart is scaled independently and so simply measuring the bars would be misleading. Figure (b) shows a small multiples rendering of a data set overlaid with its log-transformed data on the left and overlaid with its square-root-transformed data on the right. Finally, (c) shows a variational visualization in which the left variant shows the original data and the right variant shows the visualizations sorted after their creation.

Another operation useful for comparison is to perform computations across entire visualizations, such as when sorting elements. Perhaps, for example, we have created some donut charts and realize now that they may be easier to read when sorted. Again, we can avoid having to copy and paste code or start from scratch by directly sorting the elements of an existing visualization. Figure 8.6c shows the result.

```
Chc "Sorted" (Polar $ Above [pie1, pie2])
             (Polar $ Above [vSort pie1, vSort pie2])
```

8.5.2 Evaluation of Variation for Comparison

To evaluate how well variation and parameterization is able to serve as a model for comparative visualization, we need to know what features are required. Gleicher et al. (2011) proposed a taxonomy

of visual designs used for such comparison tasks. The taxonomy is validated through a significant survey of work.

Their taxonomy of comparative designs categorizes all of the work surveyed into three main categories (as well as pairs of categories) *juxtaposition*, *superposition*, and *explicit representation of the relationships*. Additionally, there are hybrid categories which combine two of these approaches into one design. We explore each of these options in turn and demonstrate which parts of our model can be used to express them.

Juxtaposition The core idea of juxtaposition is to support comparison tasks by placing the objects to be compared into separate spaces. The objects are always shown independently and in their entirety. One common form is spatial juxtaposition, also often called small multiples, in which the objects to be compared are all shown and arranged (often as a grid) in the available space. The taxonomy also allows for juxtaposition in time, in which objects are displayed one after another in sequence.

Our model of variational visualizations supports juxtaposition in more than one way. The easiest way to achieve it is to use variation to encode the visualizations we want to compare, and then rely on the default behavior of our prototype tool. This renders a small multiples view of all visualization alternatives that are not explicitly selected. Another approach is to use the spatial composition operators explicitly, such as `Above` and `NextTo`. These juxtapose visualizations geometrically by dividing up the available space equally. However, plain spatial composition does not support the selective display and navigation of alternatives provided by variations.

Finally, juxtaposition can also be temporal, as in an animation that cycles through a set of visualizations. Our prototype does not support animation directly, but it is trivial to replicate this behavior using choices. We simply encode the variants as part of a variational visualization, as in the first example, and map each step of the animation to a particular selection. It is then easy to envision

a tool which allows the user to define an animation by setting a timer which navigates among the desired selections.

Therefore, we can say that juxtaposition can be modeled by a combination of variation and spatial composition.

$$Juxtaposition \approx \mathtt{V} \oplus SpatialComposition$$

Superposition The superposition category includes designs in which the objects being compared all share a single space. In general, this is realized by composing visualizations via an overlay operation. Aesthetic tweaks such as transparency and small shifts to avoid totally obscuring some objects are common. Superposition also frequently requires some computation to determine an alignment for different objects. In Section 8.5 we showed examples making use of overlays, transparency, and spacing directly.

Because our model offers the ability to define functions over visualizations, realizing a general mechanism for parameterization, we can employ computation at essentially all levels. We have shown examples that include sorting values. We can also envision more sophisticated scenarios such as changing the order of overlaid charts or organizing a small multiples layout based on some derived value from a set of charts.

It is clear from these examples that superposition can be modeled by parameterization/computation and *Overlay*.

$$Superposition \approx Parameterization \oplus Overlay$$

Explicit Representation of Relationships The final category of the taxonomy includes designs which encode the relationships among the objects being visualized directly. One example we have already seen of this is charting the difference between two ordered data sets (using `zipWith`) and visualizing that result rather than showing both original data sets. A design in this category always

involves the extra step of computing the relationships among objects before anything can be visualized. In general, visualizations following these design principles do not require variation techniques in our model, since we have access to computation. We have also shown how we can apply data transformations to individual visualizations, including log and square-root transformations.

Finally, in some cases variation can also be used to explicitly encode relationships. One example of this is the pie chart in Section 8.5 in which the variation controls the visible level of detail.

Based on these examples, we see that the explicit encoding approach can be modeled by a combination of variation and parameterization.

$$\mathit{ExplicitEncoding} \approx \mathit{Parameterization} \oplus \mathtt{V}$$

Hybrid Categories The taxonomy also includes designs that take hybrid approaches. Combining the three original categories into pairs results in three new hybrid categories, juxtaposition and superposition, juxtaposition and explicit encoding, and superposition and explicit encoding. Each of these is manifested in designs included in the survey and so are necessary to include.

Due to our compositional design of visualizations, all of the functionality discussed so far is essentially orthogonal and all techniques can be composed. For example, to support juxtaposition and superposition at the same time, we can use any of the approaches mentioned above to produce visualizations making use of superposition. With those results, we can then compose those charts together (using language constructs or variation) to produce a hybrid visualization.

Analogously, for juxtaposition and explicit encoding we can apply any desired computations to explicitly visualize relationships among objects and data and then compose those into larger, hybrid visualizations. One example of this would be to juxtapose (using variation) two charts which are themselves variational, as we did with the log and square-root transformation example in Section 8.5.

Finally, for a hybrid approach involving superposition and explicit encoding we can compute any desired relationships among objects and then add them to the overlay composition used in

	`VChc`	*Parameterization*	*Spatial Composition*	*Overlay*
Juxtaposition	×		×	
Superposition		×		×
Explicit Representation	×	×		

Table 8.1: Matching our variational visualization approaches to comparative design categories.

superposition. Conveniently, the same log and square-root transformation example also demonstrates an example of this category.

Evaluation Conclusions Since our model is intentionally limited to a small subset of visualization types we do not claim to be able to reproduce most of the actual designs surveyed to produce the taxonomy. However, we have shown how an approach based on parameterization and variation can, in principle, support any combination of identified comparative designs (see the summary in Table 8.1). Our prototype implementation is able to handle all of the core ideas underlying the comparative designs.

Chapter 9: Conclusion

Variation can provide a lens through which to view many topics. One of the high-level goals of this dissertation is to demonstrate some of the practical challenges of doing so, as well as some of the potential benefits. To this end, we first demonstrated how the choice calculus can serve as a formal basis for variational programs and variational data structures. Second, we applied those ideas to two distinct application domains, namely variational pictures and variational information visualization.

When viewed as a whole, this dissertation reveals several themes that are not apparent from any individual chapter. The first such theme is the permeating nature of variation. In general, it is not possible to variationalize just one part of a domain without it impacting the rest. For example, by defining a variational list type, we immediately also require a large set of variational functions and types. We not only need variational version of list library functions like `head` and `tail`, but also other kinds of variational values so we can extract whatever our variational list contains, and variational control flow in order to dispatch based on variational list predicates.

The same is true for the other applications as well. Implementing variation in the visualization DSL, for example, is useless without functions to work with variational data and transform variational visualizations. Variational pictures are not useful without being able to add new variation, navigate among variants, and so on.

There are several ways to interpret this permeation of variation. One is that we should be judicious in choosing to variationalize new domains, since they must be reinvented to some extent. Another, more optimistic interpretation, is that because variation is useful and impossible to localize, it should be made a core, built-in component of future programming languages. Plain, non-variational

applications would just be a special case. Some work on this exists (e.g., Chen et al. (2016)) but such a language does not exist yet in practice.

A second theme is the recurrence of tree-shaped variational data structures where sharing occurs in the prefix of the structure, i.e., from the root up until the first point of variation. The suffix list, introduced in Chapter 4, was the first such data structure. However, the same general idea emerged from both the work on variational pictures and variational visualizations independently. In variational pictures the prefix is essentially those portions of the picture which do not vary. We localize the choices, nesting them inside the shared parts. For variational visualizations, we took a prefix-based approach by adding variation in the form of a new constructor rather than applying it only at the top-level or to the leaves. Speculatively, this might suggest a certain harmony in this type of representation, being neither too inflexible nor too complex.

Some existing work has already explored variational data structures such as graphs (Erwig et al., 2013) and stacks (Meng et al., 2017), but there are many more that could be investigated more thoroughly. If a prefix-based approach turns out to be generally useful, as we speculate here, then one future approach could be to define such representations automatically for arbitrary data structures. Just as Swierstra (2008) defined a fix-point data type `Fix` for treating data structures generically, it is also possible to define a variational fix-point type.

```
data VFix s a = VIn { vOut :: V (s a (VFix s a))}
```

Such an approach might allow the automatic definition of many reasonably efficient variational data structures.

As a final note, there are undoubtedly many more application areas which stand to benefit from being informed in their design by a systematic model of variation. Speculatively, it is not difficult to imagine using variation to capture topics such as uncertainty in computer vision, personalized user

interfaces that vary based on the context or user, or even variant move sequences in games such as chess.

9.1 Summary of Contributions

This dissertation made the following specific contributions. Chapter 2 compared existing approaches to modeling variation and justified the decision to ultimately use the choice calculus. It also summarized related work in variational programming and data structures, variational pictures, and variational information visualization. The remaining chapters can be separated into two parts.

The first part began with Chapter 3, which demonstrated one approach to using the choice calculus to write variational programs, including some of the challenges this creates for writing programs that work generically on different variational structures. This is followed by Chapter 4, which compared six different variational list representations, and Chapter 5, which made use of dependent types to help mitigate some of the efficiency issues that arise in the preceding chapters.

The second part of the dissertation focused on applying the techniques from the first part to suitable domains. Chapter 6 introduced variational pictures which can serve not only as an exploration tool for designers and artists, but also as a model of version control for images. Finally, Chapter 7 described a domain-specific language for creating information visualizations, and Chapter 8 expanded upon it to allow for the creation of variational visualizations, which support both exploration and comparison analysis tasks.

Bibliography

Wyatt Allen. Variational parsing with the choice calculus. Master's thesis, Oregon State University, 2014.

Robert Amar and John Stasko. Knowledge precepts for design and evaluation of information visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 11(4):432–442, July 2005.

Robert A. Amar, James Eagan, and John Stasko. Low-level components of analytic activity in information visualization. In *IEEE Symposium on Information Visualization*, pages 111–117, Oct 2005.

Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Detection of feature interactions using feature-aware verification. In *International Conference on Automated Software Engineering*, pages 372–375, 2011.

Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Grösslinger, and Dirk Beyer. Strategies for product-line verification: Case studies and experiments. In *IEEE International Conference on Software Engineering*, pages 482–491, 2013.

Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 165–178, 2012.

Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. Faceted execution of policy-agnostic programs. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 15–26. ACM, 2013.

Jacques Bertin. *Graphics and Graphic Information Processing*. Morgan Kaufmann Publishers Inc., 1999. English translation.

Jacques Bertin. *Semiology of Graphics: Diagrams, Networks, Maps*. University of Wisconsin Press, 2005. English translation.

Georges-Pierre Bonneau, Hans-Christian Hege, Chris R. Johnson, Manuel M. Oliveira, Kristin Potter, Penny Rheingans, and Thomas Schultz. Overview and state-of-the-art of uncertainty visualization. In Charles D. Hansen, Min Chen, Christopher R. Johnson, Arie E. Kaufman, and Hans Hagen,

editors, *Scientific Visualization: Uncertainty, Multifield, Biomedical, and Scalable Visualization*, pages 3–27. Springer London, 2014.

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

Michael Bostock and Jeffrey Heer. Protovis: A graphical toolkit for visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1121–1128, 2009.

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

Matthew Brehmer and Tamara Munzner. A multi-level typology of abstract visualization tasks. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2376–2385, Dec 2013.

Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering*, pages 168–178, 2011.

Humberto Carrillo and David Lipman. The multiple sequence alignment problem in biology. *SIAM Journal on Applied Mathematics*, 48(5):1073–1082, 1988.

Hsiang-Ting Chen, Li-Yi Wei, and Chun-Fa Chang. Nonlinear revision control for images. *ACM Transactions on Graphics*, 30(4):105:1–105:10, 2011.

Sheng Chen, Martin Erwig, and Eric Walkingshaw. A calculus for variational programming. In *European Conference on Object-Oriented Programming*, pages 6:1–6:28, 2016.

Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology*, 6(9):165–185, 2007.

Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *ACM International Conference on Software Engineering*, pages 321–330. ACM, 2011.

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

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

Marcelo d'Amorim, Steven Lauterburg, and Darko Marinov. Delta execution for efficient state-space exploration of object-oriented programs. *IEEE Transactions on Software Engineering*, 34(5): 597–613, 2008.

Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 133–144. ACM, 2008.

Chris J. Date. *Database in Depth: Relational Theory for Practitioners*. O'Reilly Media, Inc., 2005.

David J. Duke, Rita Borgo, Malcolm Wallace, and Colin Runciman. Huge data but small programs: Visualization design via multiple embedded DSLs. In *International Symposium on Practical Aspects of Declarative Languages*, pages 31–45, 2009.

Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of c preprocessor use. *IEEE Trans. on Software Engineering*, 28(12):1146–1170, Dec 2002.

Martin Erwig and Karl Smeltzer. Variational pictures. In *International Conference on the Theory and Application of Diagrams*, 2018. To appear.

Martin Erwig and Eric Walkingshaw. The choice calculus: A representation for software variation. *ACM Transactions on Software Engineering and Methodology*, 21(1):6:1–6:27, 2011.

Martin Erwig and Eric Walkingshaw. Variation programming with the choice calculus. In *Generative and Transformational Techniques in Software Engineering*, pages 55–100. 2013.

Martin Erwig, Eric Walkingshaw, and Sheng Chen. An abstract representation of variational graphs. In *ACM International Workshop on Feature-Oriented Software Development*, pages 25–32, 2013.

Martin Erwig, Karl Smeltzer, and Keying Xu. A notation for non-linear program edits. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 205–206, 2014.

Martin Erwig, Karl Smeltzer, and Xiangyu Wang. What is a visual language? *Journal of Visual Languages and Computing*, 38(C):9–17, 2017.

Jean-Daniel Fekete. The InfoVis toolkit. In *IEEE Symposium on Information Visualization*, pages 167–174, 2004.

Jun Jie Foo, Ranjan Sinha, and Justin Zobel. Discovery of image versions in large collections. *Lecture Notes in Computer Science*, 4352:433, 2007.

Paul Gazzillo and Robert Grimm. SuperC: Parsing all of C by taming the preprocessor. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 323–334, 2012.

Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In *ACM SIGPLAN International Conference on Functional Programming*, pages 339–347, 2014.

Michael Gleicher, Danielle Albers, Rick Walker, Jusufi Ilir, Charles D. Hansen, and Jonathan C. Robers. Visual comparison for information visualization. *Information Visualization*, 10:289–309, 2011.

Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.

Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer. Design as exploration: Creating interface alternatives through parallel authoring and runtime tuning. In *ACM Symposium on User Interface Software and Technology*, pages 91–100, 2008.

Jeffrey Heer and Michael Bostock. Declarative language design for interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1149–1156, 2010.

Jeffrey Heer, Stuart K. Card, and James A. Landay. Prefuse: a toolkit for interactive information visualization. In *ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 421–430, 2005.

Jeffrey Heer, Jock Mackinlay, Chris Stole, and Maneesh Agrawala. Graphical histories for visualization: Supporting analysis, communication, and evaluation. *IEEE Transactions on Visualization and Computer Graphics*, 14:1189–1196, 2008.

Danny Holten and Jarke J. van Wijk. Visual comparison of hierarchically organized data. In *Joint Eurographics / IEEE VGTC Conference on Visualization*, EuroVis '08, pages 759–766, 2008.

Spencer Hubbard and Eric Walkingshaw. Formula choice calculus. In *International Workshop on Feature-Oriented Software Development*, pages 49–57, 2016.

Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward variability-aware testing. In *ACM Int. Work. on Feature-Oriented Soft. Dev.*, pages 1–8, 2012.

Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward variability-aware testing. In *GPCE Workshop on Feature-Oriented Software Development*, pages 1–8, 2012.

Daniel Keim, Gennady Andrienko, Jean-Daniel Fekete, Carsten Görg, Jörn Kohlhammer, and Guy Melançon. Visual analytics: Definition, process, and challenges. In Andreas Kerren, John T. Stasko, Jean-Daniel Fekete, and Chris North, editors, *Information Visualization: Human-Centered Issues and Perspectives*, pages 154–175. Springer, Berlin, Heidelberg, 2008.

Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. Typechef: Toward type checking # ifdef variability in c. In *Int. Work. on Feature-Oriented Soft. Dev.*, pages 25–32. ACM, 2010.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *European Conference on Object-Oriented Programming*, pages 220–242. Springer Berlin Heidelberg, 1997.

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

Chang Hwan Peter Kim, Sarfraz Khurshid, and Don Batory. Shared execution for efficiently testing product lines. In *IEEE International Symposium on Software Reliability Engineering*, pages 221–230, 2012.

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *ACM Conference on LISP and Functional Programming*, pages 151–161, 1986.

Dimitrios S. Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *ICSE Workshop on Comparison and Versioning of Software Models*, pages 1–6, 2009.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 26–37, 2003.

Kim Lauenroth, Klaus Pohl, and Simon Töhning. Model checking of domain artifacts in product line engineering. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 269–280, 2009.

Duc Le, Eric Walkingshaw, and Martin Erwig. #ifdef confirmed harmful: Promoting understandable software variation. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 143–150, 2011.

Yun Young Lee, Darko Marinov, and Ralph E. Johnson. Tempura: Temporal dimension for IDEs. In *ACM/IEEE International Conference on Software Engineering*, pages 212–222, 2015.

Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *ACM/IEEE International Conference on Software Engineering*, pages 105–114, 2010.

Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable analysis of variable software. In *ACM Symposium on the Foundations of Software Engineering*, pages 81–91, 2013.

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

Jock Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 5(2):110–141, 1986.

Kevin Matlage and Andy Gill. ChalkBoard: Mapping functions to polygons. In *Symposium on Implementation and Application of Functional Languages*, 2009.

Meng Meng, Jens Meinicke, Chu-Pan Wong, Eric Walkingshaw, and Christian Kästner. A choice of variational stacks: Exploring variational data structures. In *ACM International Workshop on Variability Modelling of Software-intensive Systems*, pages 28–35, 2017.

Ronald Metoyer, Bongshin Lee, Nathalie Henry Riche, and Mary Czerwinski. Understanding the verbal language and structure of end-user descriptions of data visualizations. In *ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 1659–1662, 2012.

M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 29(6): 127–136, 2004.

Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *ACM International Conference on Software Engineering*, pages 907–918, 2014.

Henrik Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *ACM SIGPLAN International Conference on Functional Programming*, pages 54–65, 2005.

Dirk Ohst, Michael Welle, and Udo Kelter. Differences between versions of uml diagrams. In *European Software Engineering Conference Held Jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 227–236, 2003.

Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

Catherine Plaisant. The challenge of information visualization evaluation. In *ACM Working Conference on Advanced Visual Interfaces*, pages 109–116, 2004.

Hendrik Post and Carsten Sinz. Configuration lifting: Verification meets software configuration. In *Int. Conf. Automated Soft. Eng.*, pages 347–350, 2008.

David Reinsel, John Gantz, and John Rydning. Data age 2025: The evolution of data to life-critical. Technical report, IDC, 2017.

Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Software Product Lines: Going Beyond*, pages 77–91. 2010.

Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *ACM SIGPLAN Workshop on Haskell*, pages 1–16, 2002.

Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *IEEE Symposium on Visual Languages*, pages 336–343, 1996.

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

Karl Smeltzer and Martin Erwig. Variational lists: Comparisons and design guidelines. In *ACM SIGPLAN International Workshop on Feature-Oriented Software Development*, pages 31–40, 2017.

Karl Smeltzer, Martin Erwig, and Ronald Metoyer. A transformational approach to data visualization. In *International Conference on Generative Programming: Concepts and Experiences*, pages 53–62, 2014.

Chris Stolte, Diane Tang, and Pat Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):52–65, 2002.

Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423—436, 2008.

Michael Terry and Elizabeth D. Mynatt. Side views: Persistent, on-demand previews for open-ended tasks. In *ACM Symp. on User Interface Soft. and Tech.*, pages 71–80, 2002.

Michael Terry, Elizabeth D. Mynatt, Kumiyo Nakakoji, and Yasuhiro Yamamoto. Variation in element and action: Supporting simultaneous development of alternative solutions. In *SIGCHI Conf. on Human Factors in Comp. Systems*, pages 711–718, 2004.

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

Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press LLC, 2nd edition, 2001.

John Tukey. *Exploratory Data Analysis*. 1977.

Jarke J. van Wijk. The value of visualization. In *IEEE Visualization*, pages 79–86, 2005.

Fernanda B. Viegas, Martin Wattenberg, Frank Van Ham, Jesse Kriss, and Matt McKeon. Manyeyes: A site for visualization at internet scale. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1121–1128, 2007.

Eric Walkingshaw. *The Choice Calculus: A Formal Language of Variation*. PhD thesis, Oregon State University, 2013.

Eric Walkingshaw and Klaus Ostermann. Projectional editing of variational software. In *International Conference on Generative Programming: Concepts and Experiences*, pages 29–38, 2014.

Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. Variational data structures: Exploring tradeoffs in computing with variability. In *ACM Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming & Software*, pages 213–226, 2014.

Chris Weaver. Building highly-coordinated visualizations in improvise. In *IEEE Symposium on Information Visualization*, pages 159–166, 2004.

Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer, 2009.

Leland Wilkinson. *The Grammar of Graphics*. Springer-Verlag, 2nd edition, 2005.

Hongwei Xi. Dependently typed pattern matching. *Journal of Universal Computer Science*, 9(8): 851–872, 2003.

Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 249–257. ACM, 1998.

Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 85–96, 2012.

YoungSeok Yoon and Brad A. Myers. Supporting selective undo in a code editor. In *ACM/IEEE International Conference on Software Engineering*, pages 223–233, 2015.

Brent A Yorgey. Monoids: theme and variations (functional pearl). In *Haskell Symposium*, pages 105–116, 2012.

APPENDICES

Appendix A: Variational Programming Haskell Source Code

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

type Dim = String

data Dir = L | R

type Sel = (Dim, Dir)

type Dec = [Sel]

instance Functor VChc where
  fmap f (Chc d l r) = Chc d (fmap f l) (fmap f r)
  fmap f (One x) = One $ f x

instance Applicative VChc where
  pure = One
  Chc d l r <*> x = Chc d (l <*> x) (r <*> x)
  One f <*> x = fmap f x

instance Monad VChc where
  return = pure
  One x >>= f = f x
  Chc d l r >>= f = Chc d (l >>= f) (r >>= f)
```

```
instance Foldable VChc where
  foldMap f (One x) = f x
  foldMap f (Chc d l r) = foldMap f l `mappend` foldMap f r

newtype VTag a = VTag [(a,TagFormula)]

data TagFormula = And TagFormula TagFormula
                | Or TagFormula TagFormula
                | Not TagFomula
                | T Dim
                | B Bool

class V v where
  chc :: Dim -> v a -> v a -> v a
  one :: a -> v a

instance V VChc where
  chc = Chc
  one = One

instance V VTag where
  chc d (VTag ls) (VTag rs) =
    let ls' = map (second (And (Not (T d)))) ls
        rs' = map (second (And (T d))) rs
    in VTag (ls ++ rs)
  one x = VTag [(x, B True)]
```

```
instance (VList vl) => V vl where
  chc = chcVL
  one = vSingleton . one . Just

chcVL :: (VList vl) => Dim -> vl a -> vl a -> vl a
chcVL d l r = if vAllNull l && vAllNull r
              then vEmpty
              else vCons (chc d (vHead l) (vHead r))
                         (chVL d (vTail l) (vTail r))

vAllNull :: (VList vl) => vl a -> Bool
vAllNull = all (== True) . vNull

vHead :: TList a -> VChc (Maybe a)
vHead ts = let tags = getAllTagPermutations ts
           in tagToChc $ fmap (findHead ts) tags

findHead :: TList a -> TagSet -> Maybe a
findHead [] _ = Nothing
findHead ((x, tf) : ts) tags = if solve tags tf
                               then Just x
                               else findHead ts tags

-- Code adapted from Erwig and Walkingshaw (2013)
vReverse :: SuffList a -> SuffList a
vReverse Nil = Nil
vReverse (Cons x xs) = vReverse xs ‘vCat‘ vSingleton x
vReverse (Nest sl) = Nest $ fmap vReverse sl
```

```
vCat :: SuffList a -> SuffList a
vCat Nil rs = rs
vCat (Cons l ls) rs = Cons l (ll `vCat` rs)
vCat (Nest l) rs = Nest $ fmap (`vCat` rs) l

-- Code adapted from Allen (2014)
vHead :: SegList a -> VChc (Maybe a)
vHead (VL []) = One Nothing
vHead (VL (Elems [] : xs) = vHead xs
vHead (VL (Elems (a:_) : _)) = One $ Just x
vHead (VL (Split d l r : xs)) = Chc d l' r'
    where l' = vHead $ l ++ (VL xs)
          r' = vHead $ r ++ (VL xs)

vFilter :: (VList v) => (a -> Bool) -> v a -> v a
vFilter p l = vif (vNull l) vEmpty (vFilterV p l)

vFilterV :: (VList v) => (a -> Bool) -> v a -> v a
vFilterV p l = vCons (ccFilter p (vHead l)) (vFilter p (vTail l))

vFilterCList :: (a -> Bool) -> CList a -> CList a
vFilterCList p cs = fmap (chcFilter p) cs

chcFilter :: (a -> Bool) -> VChc (Maybe a) -> VChc (Maybe a)
chcFilter p (Chc d l r) = Chc d (chcFilter p l) (chcFilter p r)
chcFilter p (One (Just x)) | p x = One $ Just x
chcFilter _ _ = One Nothing
```

```
instance VFilter CList where
  vFilter = vFilterCList
```

Appendix B: Variational Programming Idris Source Code

```
vZipWith :: (VList vl) => (a -> b -> c) -> vl a -> vl b -> vl c
vZipWith f xs ys | vAllNull xs || vAllNull ys = vEmpty
vZipWith f xs ys = vCons (zipChcM f (vHead xs) (vHead ys))
                         (vZipWith f (vTail xs) (vTail ys))


zipChcM :: (a -> b -> c) -> VChc (Maybe a) -> VChc (Maybe b) -> VChc (Maybe c)
zipChcM = liftA2 . liftA2

Applicative (NV ds) where
  pure x {ds = []}       = OneN x
  pure x {ds = (d::_)} = InjN (pure x)
  (ChcN d f g) <*> (ChcN d x y) = ChcN d (f <*> x) (g <*> y)
  (ChcN d f g) <*> (InjN x)     = ChcN d (f <*> x) (g <*> x)
  (OneN f)     <*> (OneN x)     = OneN (f x)
  (InjN f)     <*> (ChcN d x y) = ChcN d (f <*> x) (f <*> y)
  (InjN f)     <*> (InjN x)     = InjN (f <*> x)
```

```
selectNV : Decision -> NV ds a -> NV ds a
selectNV dec (ChcN d l r) = case lookupDim d dec of
  Just L => InjN $ selectNV dec l
  Just R => InjN $ selectNV dec r
  Nothing => ChcN d (selectNV dec l) (selectNV dec r)
selectNV dec (InjN x) = InjN $ selectNV dec x
selectNV _ x = x

rebase : NV dsI a -> NV dsO a
rebase c@(ChcN d l r) {dsI = d::_} {dsO = d'::_} =
  if d == d'
  then ChcN d' (rebase l) (rebase r)
  else ChcN d' (rebase (selectNV (dec [(d',L)]) c))
               (rebase (selectNV (dec [(d',R)]) c))
rebase (ChcN d l r) {dsI = d::_} {dsO = []} = rebase l
rebase (OneN x) {dsI = []}        {dsO = d'::_} = InjN (rebase (OneN x))
rebase (OneN x) {dsI = []}        {dsO = []}    = OneN x
rebase (InjN x) {dsI = (d :: ds)} {dsO = d'::_} = InjN (rebase x)
rebase (InjN x) {dsI = _::_}      {dsO = []}    = rebase x
```

```
norm : VChc a -> NV ds a
norm (Chc d l r) {ds = []} = norm l
norm c@(Chc d l r) {ds = d'::_} =
  if d == d'
  then ChcN d' (norm l) (norm r)
  else ChcN d' (norm (select (mkDec [(d', L)]) c))
               (norm (select (mkDec [(d', R)]) c))
norm (One x) {ds = []}   = OneN x
norm (One x) {ds = _::_} = InjN (norm (One x))

vZipWithFWork :
    (a -> b -> c) -> List (NV ds a) -> List (NV ds b) -> List (NV ds c)
vZipWithFWork f (x::xs) (y::ys) = vZipWithNV f x y :: vZipWithFWork f xs ys
vZipWithFWork _ _ _ = []

vZipWithF : (a -> b -> c) -> Front ds a -> Front ds b ->
            (Front ds c, Either (Front ds a) (Front ds b))
vZipWithF f (F xs) (F ys) =
    let (lx,ly) = (length xs, length ys)
    in if lx < ly
        then (F $ vZipWithFWork f xs (take lx ys), Right (F $ drop lx ys))
        else (F $ vZipWithFWork f (take ly xs) ys, Left (F $ drop ly xs))
```

```
mutual
  vZipWithBWork : (a -> b -> c) -> Back a -> Back b -> Back c
  vZipWithBWork f xs ys = let (x,xt) = uncons xs
                              (y,yt) = uncons ys
                              (B rest) = vZipWithB f xt yt
                          in B $ vZipWithV f x y :: rest

  vZipWithB : (a -> b -> c) -> Back a -> Back b -> Back c
  vZipWithB f xs ys = if vAllNull xs || vAllNull ys
                      then B []
                      else vZipWithBWork f xs ys

relaxL : Front ds a -> VList a
relaxL (F fs) = map (map Just . relax) fs

vZipWith : (a -> b -> c) -> SList ds a -> SList ds b -> SList ds c
vZipWith f (SL fr1 (B b1)) (SL fr2 (B b2)) =
  case vZipWwithF f fr1 fr2 of
    (fr, Left rest)  => (SL fr (vZipWithB f (B (relaxL rest ++ b1)) (B b2)))
    (fr, Right rest) => (SL fr (vZipWB f (B b1) (B (relaxL rest ++ b2))))

split : SList ds a -> SList ds a
split (SL f (B [])) = SL f (B [])
split sl@(SL (F xs) bs) = let (hd,tl) = unconsB bs
  in case unM hd of
        Just v => split (SL (F (xs ++ [norm v])) (B tl))
        Nothing => sl
```

```
split' : LList ds a -> LList ds a
split' (Tidy f (B [])) = Tidy f (B [])
split' ll@(Tidy (F xs) bs) =
  let (hd,tl) = unconsB bs in
    case unP hd of
      Just v => split' (Tidy (F (xs ++ [norm v])) (B tl))
      Nothing => ll
split' (Messy (B xs) (Tidy (F fs) (B zs))) =
      split' (Tidy (F []) (B (ls ++ relaxL fs ++ bs)))
split' (Messy (B xs) (Messy (B ys) rest)) =
      split' (Messy (B $ xs ++ ys) rest)
```

Appendix C: Variational Visualization Purescript Code

```
pieDet :: VVis
pieDet = vPie [ Chc "Region 1" (One [3.0]) (One [1.9, 0.7, 0.4])
              , Chc "Region 2" (One [2.0]) (One [1.0, 0.4, 0.6])
              , Chc "Region 3" (One [0.8]) (One [0.2, 0.3, 0.3])]


-- The unsafe code is to allow users to pass in arrays for [] syntax.  Passing
-- in an empty array will cause a runtime crash.
vPie :: Array (V (Array Number)) -> VVis
vPie xs =
  let mna = map (map (unsafeMaybe <<< minimum)) (map plainVals xs)
      mn  = unsafeMaybe (minimum (map minimum mna))
      mxa = map (map (unsafeMaybe <<< maximum a)) (map plainVals xs)
      mx  = unsafeMaybe (maximum (map maximum mna))
      fh = Frame { frameMin: 0.0, frameMax: 1.0 }
      fw = Frame { frameMin: min 0.0 mn, frameMax: max 0.0 mx}
      fs = map (fillsWA fh fw) xs
  in Polar $ NextTo $ unsafeMaybe (fromFoldable fs)
```

```
fillsWA :: Frame -> Frame -> V (Array Number) -> VVis
fillsWA fh fw (Chc d l r) = V d (fillsWA fh fw l) (fillsWA fh fw r)
fillsWA fh fw (One x) =
  let farr = map (mkMark fh fw) x
  in NextTo $ unsafeMaybe (fromFoldable farr) where
    mkMark fh fw v = let theVPs = VPs { height: 1.0, width: v
                                      , color: green, visible: true
                                      , orientation: Horizontal }
                     in Mark { vps: theVPs
                             , frameH: fh, frameW: fw
                             , label: Nothing }
```