Variation Programming with the Choice Calculus*

Martin Erwig and Eric Walkingshaw

School of EECS
Oregon State University

Abstract. The choice calculus provides a language for representing and transforming variation in software and other structured documents. Variability is captured in localized choices between alternatives. The space of all variations is organized by dimensions, which provide scoping and structure to choices. The variation space can be reduced through a process of selection, which eliminates a dimension and resolves all of its associated choices by replacing each with one of their alternatives. The choice calculus also allows the definition of arbitrary functions for the flexible construction and transformation of all kinds of variation structures. In this tutorial we will first present the motivation, general ideas, and principles that underlie the choice calculus. This is followed by a closer look at the semantics. We will then present practical applications based on several small example scenarios and consider the concepts of "variation programming" and "variation querying". The practical applications involve work with a Haskell library that supports variation programming and experimentation with the choice calculus.

1 Introduction

Creating and maintaining software often requires mechanisms for representing variation. Such representations are used to solve a diverse set of problems, such as managing revisions over time, implementing optional features, or managing several software configurations. Traditionally, research in each of these areas has worked with different variation representations, obfuscating their similarities and making the sharing of results difficult. The choice calculus [12] solves this by providing a formal model for representing and reasoning about variation that can serve as an underlying foundation for all kinds of research on the topic [10].

More specifically and relevant to the central topics of this summer school, the choice calculus supports both generative and transformational techniques in the area of software engineering. The generative aspect is obvious: The representation of variation in software supports, through a process of selection, the generation of specific variants of that software.

How a variation representations can support transformations may be less obvious. To explain the relationship, we first point out that transformations can be distinguished into two kinds: (A) *simple* and *automatic* transformations, and (B) *complicated* and *(at least partially) manual* transformations. The first kind of transformation is the one we

* This work is partially supported by the Air Force Office of Scientific Research under the grant FA9550-09-1-0229 and by the National Science Foundation under the grant CCF-0917092.

R. Lämmel, J. Saraiva, and J. Visser (Eds.): GTTSE 2011, LNCS 7680, pp. 55–100, 2012.

love: We have a representation of the transformation that we can apply as often as we want to produce some desired output from all kinds of inputs in an instant.

However, the second kind of transformation is also ubiquitous in software engineering. Consider, for example, the editing of software in response to changed requirements or bug reports. Such a transformation often requires many changes in different parts of a software system and involves the creation of a network of interdependent changes. If not done carefully, inconsistencies and other errors can be introduced, which may necessitate further costly and time-consuming editing. This kind of transformation is much more arduous than the automatic kind, but is nevertheless quite common. Moreover, since it is so complicated to deal with, it is even more deserving of attention.

A structured variation representation can support complicated transformations as follows. First, we can embed variation in the software artifact at all those places where changes are required. By creating a new variant we keep the original version and so always have a consistent version to fall back on. This benefit is also provided by traditional version control systems. However, usually the representation provided by these tools are quite impoverished (line-based patches), making it difficult to view multiple independent changes in context or apply changes in different orders.

Second, a structured variation representation supports exploratory editing of software artifacts. Whenever a particular change can be applied in several different ways, we can represent several alternatives and delay a decision, which might depend on other changes not even made at this point.

Ultimately, a variation representation supports the integrated representation of a set of closely related programs, a concept we have identified as *program fields* [11]. Program fields are essentially an extensional representation of a set of programs together with a set of direct transformations between them. Under this view, applying transformations is expressed by trading decisions about which changes to apply. We will illustrate this aspect later with examples.

We will start the tutorial in Section 2 by discussing the requirements of a variation representation and then illustrating how these requirements are realized in the choice calculus, which provides a generic annotation language that can be applied to arbitrary object languages. Specifically, we will demonstrate how we can synchronize variation in different parts of an object program through the concept of *choices* that are bound by *dimensions*. We will also show how this representation supports modularity as well as dependent variation. In addition, we will discuss the need for a construct to explicitly represent the sharing of common parts in a variation representation. The behavior of the sharing construct introduced by the choice calculus poses some challenges for the transformation of variational artifacts. We will therefore ignore the sharing representation in the later parts of the tutorial that are concerned with variation programming.

The most basic operation on a variation representation is the selection of a particular variant. In Section 3 we will define the semantics of the choice calculus, which essentially defines a mapping from decisions to plain object programs. The semantics is also implemented as part of the domain-specific language that we use for variation programming and often serves a useful tool to understand variation representations.

The semantics is essentially based on a function for eliminating dimensions and associated choices. And even though choice elimination is an essential component of the choice calculus, it is only one very simple example from a set of many interesting

operations on variation structures. More sophisticated operations can be defined once we integrate the choice calculus representation into an appropriate metaprogramming environment. We will present such an integration of the choice calculus into Haskell in Section 4. We will discuss several different approaches to such an integration and choose one that is simple but powerful.

This integration provides the basis for writing programs to query, manipulate, and analyze variation structures. We call this form of writing programs that exploit variation structures *variation programming*. Variation programming embodies the transformational aspects of a static variation representation. We will introduce the basic elements of variation programming with programs on variational lists in Section 5. We will illustrate how to generalize "standard" list functions to work on variational lists and also develop functions that manipulate the variational structure of lists in a purposeful manner.

In Section 6 we consider the application of variation programming to variational programs (the maintenance of variational software). We use an extremely simplified version of Haskell for that purpose.

This tutorial is full of languages. Understanding which languages are involved, what roles they play, and how they are related to one another is important to keep a clear view of the different representations and their purpose and how variation programming works in the different scenarios. Here is a brief summary of the languages involved.

- The *choice calculus* is a generic language that can be applied to, or instantiated by, different *object languages*. Specifically, given an object language L, we write $V(L)$ for the result of L's integration with the choice calculus.
- *Object languages*, such as list data structures or Haskell, are placed under variation control by integrating their representation with the choice calculus.
- *Variational languages* are the result of the combination of an object language with the choice calculus. We write VL for the variational version of the object language L, that is, we have $VL = V(L)$. For example, we have the variational languages $\mathrm{VList} = V(\mathrm{List})$ and $\mathrm{VHaskell} = V(\mathrm{Haskell})$.
- We are using Haskell as a *metalanguage* to do variation programming, and we represent the choice calculus, all object languages, and variational languages as data types in Haskell to facilitate the writing of variation programs.

Finally, in this tutorial we assume some basic familiarity with Haskell, that is, knowledge of functions and data types and how to represent languages as data types. Knowledge of monads and type classes are useful, but not strictly required.

2 Elements of the Choice Calculus

In this section we will introduce and motivate the concepts and constructs of the choice calculus. We use a running example of varying a simple program in the object language of Haskell, but the choice calculus is generic in the sense that it can be applied to any tree-structured document.

Consider the following four implementations of a Haskell function named `twice` that returns twice the value of its argument.

```
twice x = x+x            twice y = y+y
twice x = 2*x            twice y = 2*y
```

These definitions vary in two independent *dimensions* with two possibilities each. The first dimension of variation is in the name of the function's argument: those in the left column use `x` and those in the right column use `y`. The second dimension of variation is in the arithmetic operation used to implement the function: addition in the top row and multiplication in the bottom.

We can represent all four implementations of `twice` in a single choice calculus expression, as shown below.

dim $Par\langle x, y\rangle$ **in**
dim $Impl\langle plus, times\rangle$ **in**
`twice` $Par\langle$`x,y`$\rangle$ `=` $Impl\langle Par\langle$`x,y`$\rangle$`+`$Par\langle$`x,y`$\rangle$`,2*`$Par\langle$`x,y`$\rangle\rangle$

In this example, we begin by declaring the two dimensions of variation using the choice calculus **dim** construct. For example, **dim** $Par\langle x, y\rangle$ declares a new dimension *Par* with *tags* x and y, representing the two possible parameter names. The **in** keyword denotes the scope of the declaration, which extends to the end of the expression if not explicitly indicated otherwise (for example, by parentheses).

We capture the variation between the different implementations in *choices* that are bound by the declared dimensions. For example, $Par\langle$`x,y`$\rangle$ is a choice bound by the *Par* dimension with two *alternatives*, `x` and `y`. Note that `x` and `y` are terms in the *object language* of Haskell (indicated by typewriter font), while the tags x and y are identifiers in the *metalanguage* of choice calculus (indicated by italics).

Each dimension represents an incremental decision that must be made in order to resolve a choice calculus expression into a concrete program variant. The choices bound to that dimension are synchronized with this decision. This incremental decision process is called *tag selection*. When we select a tag from a dimension, the corresponding alternative from every bound choice is also selected, and the dimension declaration itself is eliminated. For example, if we select the y tag from the *Par* dimension (*Par*.y), we would produce the following choice calculus expression in which the *Par* dimension has been eliminated and each of its choices has been replaced by its second alternative.

dim $Impl\langle plus, times\rangle$ **in**
`twice y =` $Impl\langle$`y+y,2*y`$\rangle$

If we then select *Impl*.*times*, we produce the variant of `twice` in the lower-right corner of the above grid of variants.

In the above examples, the choice calculus notation is embedded within the syntax of the object language. This embedding is not a textual embedding in the way that, for example, the C Preprocessor's `#ifdef` statements are integrated with program source code. Instead, choices and dimensions operate on an abstract-syntax tree view of the object language. This imposes constraints on the placement and structure of choices and dimensions. For example, every alternative of a choice must be of the same syntactic category. When it is necessary to do so, we represent the underlying tree structure of the object language explicitly with Y-brackets. For example, we might render the AST for `twice x = x+x` as `=-<twice,x,+-<x,x>->-`, that is, the definition is represented as a tree that has the `=` operation at the root and three children, (1) the name of the function

`twice`, (2) its parameter `x`, and (3) the RHS, which is represented by abother tree with root `+` and two children that are both given by `x`. Usually we stick to concrete syntax, however, for readability.

Returning to our choice calculus expression encoding all four variants of the function `twice`, suppose we add a third option z in the parameter name dimension. We show this extension below, where newly added tags and alternatives are underlined.

dim $Par\langle x, y, \underline{z}\rangle$ **in**
dim $Impl\langle plus, times\rangle$ **in**
`twice` $Par\langle$`x,y,`$\underline{\texttt{z}}\rangle$ `=` $Impl\langle Par\langle$`x,y,`$\underline{\texttt{z}}\rangle$`+`$Par\langle$`x,y,`$\underline{\texttt{z}}\rangle$`,2*`$Par\langle$`x,y,`$\underline{\texttt{z}}\rangle\rangle$

Exercise 1. How many variants does this choice calculus expression represent? Extend the example to declare a new independent dimension, *FunName* that is used to vary the name of the function between `twice` and `double`. Now how many variants are encoded?

As you can see, the above extension with tag z required making the same edit to several identical choices. As programs get larger and more complex, such repetitive tasks become increasingly prone to editing errors. Additionally, we often want to share a subexpression between multiple alternatives of the same choice. For example, a program that varies depending on the choice of operating system, say *Windows*, *Mac*, and *Linux*, might have many choices in which the cases for Mac and Linux are the same since they share a common heritage in Unix. It would be inconvenient, error prone, and inefficient to duplicate the common code in each of these cases.

As a solution to both of these problems, the choice calculus provides a simple sharing mechanism. Using this, we can equivalently write the above variational program as follows.

dim $Par\langle x, y, \underline{z}\rangle$ **in**
dim $Impl\langle plus, times\rangle$ **in**
share $v = Par\langle$`x,y,`$\underline{\texttt{z}}\rangle$ **in**
`twice` v `=` $Impl\langle v$`+`v`,2*`$v\rangle$

Note that now we need only extend the dimension with the new tag z and add the `z` alternative once. The choice calculus variable v stores the result of this choice and is referenced in the definition of `twice`. Because sharing is expanded only after all dimensions and choices have been resolved, the following expression encodes precisely the same variants as the above.

dim $Impl\langle plus, times\rangle$ **in**
share $v = ($**dim** $Par\langle x, y, z\rangle$ **in** $Par\langle$`x,y,z`$\rangle)$ **in**
`twice` v `=` $Impl\langle v$`+`v`,2*`$v\rangle$

This feature provides a convenient way to limit the scope of a dimension to a single choice. We call such dimensions *atomic*, a concept that will be revisited in Section 4.

Exercise 2. Extend the above choice calculus expression to include a second function `thrice` that triples the value of its input, and that varies synchronously in the same dimensions as `twice`. That is, a selection of *Impl.plus* and *Par.x* (followed by **share**-variable expansion) should produce the following expression.

```
twice x = x+x
thrice x = x+x+x
```

Exercise 3. Modify the expression developed in Exercise 2 so that the implementation methods of the two functions vary independently. (*Hint*: Since dimensions are locally scoped, you can reuse the dimension name *Impl*.) Finally, extend `thrice`'s *Impl* dimension to include an option that implements `thrice` in terms of `twice`.

Dimensions can also be *dependent* on a decision in another dimension. For example, consider the following three alternative implementations of `twice`, where those in the top row implement the function with a lambda expression, while the one in the bottom row use Haskell's operator section notation to define the function in a pointfree way (that is, without explicitly naming the variable).

```
twice = \x -> 2*x        twice = \y -> 2*y
twice = (2*)
```

Again we have two dimensions of variation. We can choose a pointfree representation or not, and we can again choose the parameter name. In this case, however, it doesn't make sense to select a parameter name if we choose the pointfree style, because there is no parameter name! In other words, the parameter name dimension is only relevant if we choose "no" in the pointfree dimension. In the choice calculus, a dependent dimensions is realized by nesting it in an alternative of another choice, as demonstrated below.

dim *Pointfree*⟨*yes*, *no*⟩ **in**
`twice v = ` *Pointfree*⟨`(2*)`, **share** *v* = (**dim** *Par*⟨*x*, *y*⟩ **in** *Par*⟨`x`, `y`⟩) **in** `\v -> 2*v`⟩

If we select *Pointfree.yes*, we get the variant `twice = (2*)`, with no more selections to make. However, if we select *Pointfree.no* we must make a subsequent selection in the *Par* dimension in order to fully resolve the choice calculus expression into a particular variant.

Throughout this discussion we have implicitly taken the "meaning" of a choice calculus expression to be the variants that it can produce. In the next section we formalize this notion by presenting a formal semantics for choice calculus expressions.

3 Syntax and Semantics of the Choice Calculus

Although much of this tutorial will focus on a domain-specific embedded language (DSEL) for variation research, one of the most important goals of the choice calculus is to serve as a *formal model* of variation that can support a broad range of theoretical

$$
\begin{array}{lll}
e ::= & a\text{-}\langle e,\ldots,e\rangle\text{-} & \textit{Object Structure} \\
\mid & \textbf{dim}\ D\langle t,\ldots,t\rangle\ \textbf{in}\ e & \textit{Dimension} \\
\mid & D\langle e,\ldots,e\rangle & \textit{Choice} \\
\mid & \textbf{share}\ v = e\ \textbf{in}\ e & \textit{Sharing} \\
\mid & v & \textit{Reference}
\end{array}
$$

Fig. 1. Choice calculus syntax

research. Before moving on, therefore, we will briefly discuss the formal syntax and semantics of choice calculus expressions. Because the DSEL is based on the choice calculus, these details will be helpful throughout the rest of this tutorial.

The syntax of choice calculus expressions follows from the discussion in the previous section and is provided explicitly in Figure 1. There are a few syntactic constraints on choice calculus expression not expressed in the grammar. First, all tags in a single dimension must be pairwise different so they can be uniquely referred to. Second, each choice $D\langle e^n\rangle$ must be within the static scope of a corresponding dimension declaration $\textbf{dim}\ D\langle t^n\rangle\ \textbf{in}\ e$. That is, the dimension D must be defined at the position of the choice, and the dimension must have exactly as many tags as the choice has alternatives. Finally, each sharing variable reference v must be within scope of a corresponding **share** expression defining v.

Exercise 4. Which of the following are syntactically valid choice calculus expressions?

(a) $\textbf{dim}\ D\langle t_1,t_2,t_3\rangle\ \textbf{in}\ (\textbf{dim}\ D\langle t_1,t_2\rangle\ \textbf{in}\ D\langle e_1,e_2,e_3\rangle)$
(b) $\textbf{share}\ v = D\langle e_1,e_2\rangle\ \textbf{in}\ (\textbf{dim}\ D\langle t_1,t_2\rangle\ \textbf{in}\ v)$
(c) $\textbf{dim}\ D\langle t_1,t_2,t_3\rangle\ \textbf{in}\ (\textbf{share}\ v = D\langle e_1,e_2,e_3\rangle\ \textbf{in}\ (\textbf{dim}\ D\langle t_1,t_2\rangle\ \textbf{in}\ v))$

The object structure construct is used to represent the artifact that is being varied, for example, the AST of a program. Therefore, a choice calculus expression that consists only of structure expressions is just a regular, unvaried artifact in the object language. We call such expressions *plain*. While the structure construct provides a generic tree representation of an object language, we could imagine expanding this construct into several constructs that more precisely capture the structure of a particular object language. This idea is central to the implementation of our DSEL, as we'll see in the next section. Also, we often omit the brackets from the leaves of structure expressions. So we write, for example, `+`$\text{-}\langle$`x,x`$\rangle$- rather than `+`$\text{-}\langle$`x`$\text{-}\langle\rangle\text{-},$`x`$\text{-}\langle\rangle\text{-}\rangle$- to represent the structure of the expression `x+x` explicitly.

In the previous section we introduced tag selection as a means to eliminate a dimension of variation. We write $\lfloor e\rfloor_{D.t}$ for the selection of tag t from dimension D in expression e. Tag selection consists of (1) finding the first declaration $\textbf{dim}\ D\langle t^n\rangle\ \textbf{in}\ e'$ in a preorder traversal of e, (2) replacing every choice bound by the dimension in e' with its ith alternative, where i is the index of t in t^n, and (3) removing the dimension declaration. Step (2) of this process is called *choice elimination*, written $\lfloor e'\rfloor_{D.i}$ (where the tag name has been replaced by the relevant index), and defined formally in Figure 2. This definition is mostly straightforward, replacing a matching choice with its ith alternative

$$
\begin{aligned}
\lfloor a\prec e_1,\ldots,e_n\succ\rfloor_{D.i} &= a\prec\lfloor e_1\rfloor_{D.i},\ldots,\lfloor e_n\rfloor_{D.i}\succ \\
\lfloor \mathbf{dim}\ D'\langle t^n\rangle\ \mathbf{in}\ e\rfloor_{D.i} &= \begin{cases} \mathbf{dim}\ D'\langle t^n\rangle\ \mathbf{in}\ e & \text{if } D=D' \\ \mathbf{dim}\ D'\langle t^n\rangle\ \mathbf{in}\ \lfloor e\rfloor_{D.i} & \text{otherwise} \end{cases} \\
\lfloor D'\langle e_1,\ldots,e_n\rangle\rfloor_{D.i} &= \begin{cases} \lfloor e_i\rfloor_{D.i} & \text{if } D=D' \\ D'\langle\lfloor e_1\rfloor_{D.i},\ldots,\lfloor e_n\rfloor_{D.i}\rangle & \text{otherwise} \end{cases} \\
\lfloor \mathbf{share}\ v=e\ \mathbf{in}\ e'\rfloor_{D.i} &= \mathbf{share}\ v=\lfloor e\rfloor_{D.i}\ \mathbf{in}\ \lfloor e'\rfloor_{D.i} \\
\lfloor v\rfloor_{D.i} &= v
\end{aligned}
$$

Fig. 2. Choice elimination

and otherwise propagating the elimination downward. Note, however, that propagation also ceases when a dimension declaration of the same name is encountered—this maintains the static scoping of dimension names.

Exercise 5. Given $e = \mathbf{dim}\ A\langle a_1,a_2\rangle\ \mathbf{in}\ A\langle A\langle 1,2\rangle,3\rangle$ what is the result of the selection $\lfloor e\rfloor_{A.a_1}$? Is it possible to select the plain expression 2?

By repeatedly selecting tags from dimensions, we will eventually produce a plain expression. We call the selection of one or several tags collectively a *decision*, and a decision that eliminates all dimensions (and choices) from an expression a *complete decision*. Conceptually, a choice calculus expression then represents a set of plain expressions, where each is uniquely identified by the complete decision that must be made in order to produce it. We therefore define the semantics domain of choice calculus expressions to be a mapping from complete decisions to plain expressions.

We write $[\![e]\!]$ to indicate the semantics of expression e. We represent the denotation of e (that is, the mapping from decisions to plain expressions) as a set of pairs, and we represent decisions as n-tuples of dimension-qualified tags. For simplicity and conciseness, we enforce in the definition of the semantics that tags are selected from dimensions in a fixed order, the order that the dimension declarations are encountered in a preorder traversal of the expression (see [12] for a discussion of this design decision). For instance, in the following example, tags are always selected from dimension A before dimension B.

$$
\begin{aligned}
[\![\mathbf{dim}\ A\langle a_1,a_2\rangle\ \mathbf{in}\ A\langle 1,\mathbf{dim}\ B\langle b_1,b_2\rangle\ \mathbf{in}\ B\langle 2,3\rangle\rangle]\!] = \\
\{(A.a_1,1),((A.a_2,B.b_1),2),((A.a_2,B.b_2),3)\}
\end{aligned}
$$

Note that dimension B does not appear at all in the decision of the first entry in this denotation since it is eliminated by the selection of the tag $A.a$.

Exercise 6. Write the semantics of the above expression if the tag ordering constraint is removed.

$$
\begin{aligned}
V_\rho(a\langle\rangle) &= \{((),a\langle\rangle)\} \\
V_\rho(a\langle e^n\rangle) &= \{(\delta^n,a\langle e'^n\rangle) \mid (\delta_1,e'_1)\in V_\rho(e_1),\ldots,(\delta_n,e'_n)\in V_\rho(e_n)\} \\
V_\rho(\textbf{dim}\ D\langle t^n\rangle\ \textbf{in}\ e) &= \{((D.t_i,\delta),e') \mid i\in\{1,\ldots,n\},\ (\delta,e')\in V_\rho(\lfloor e\rfloor_{D.i})\} \\
V_\rho(\textbf{share}\ v=e_1\ \textbf{in}\ e_2) &= \bigcup\{\{(\delta_1\delta_2,e'_2) \mid (\delta_2,e'_2)\in V_{\rho\oplus(v,e'_1)}(e_2)\} \mid (\delta_1,e'_1)\in V_\rho(e_1)\} \\
V_\rho(v) &= \{((),\rho(v))\}
\end{aligned}
$$

Fig. 3. Computing the semantics of a choice calculus expression e, $[\![e]\!] = V_\varnothing(e)$

Finally, we provide a formal definition of the semantics of choice calculus expressions in terms of a helper function V in Figure 3. The parameter to this function, ρ, is an environment, implemented as a stack, mapping **share**-variables to plain expressions. The semantics of e is then defined as an application of V with an initially empty environment, that is, $[\![e]\!] = V_\varnothing(e)$.

The definition of V relies on a somewhat dense notation, so we will briefly describe the conventions, then explain each case below. We use δ to range over decisions, concatenate decisions δ_1 and δ_2 by writing $\delta_1\delta_2$, and use δ^n to represent the concatenation of decisions $\delta_1,\ldots,\delta_n$. Similarly, lists of expressions e^n can be expanded to $e_1,\ldots,e_n$, and likewise for lists of tags t^n. We associate v with e in environment ρ with the notation $\rho\oplus(v,e)$, and lookup the most recent expression associated with v by $\rho(v)$.

For structure expressions there are two sub-cases to consider. If the expression is a leaf, then the expression is already plain, so the result is an empty decision (represented by the nullary tuple $()$) mapped to that leaf. Otherwise, we recursively compute the semantics of each subexpression and, for each combination of entries (one from each recursive result), concatenate the decisions and reconstruct the (now plain) structure expression.

On a dimension declaration, we select each tag t_i in turn, computing the semantics of $\lfloor e\rfloor_{D.i}$ and prepending $D.t_i$ to the decision of each entry in the result. Note that there is no case for choices in the definition of V. Since we assume that all choices are bound, all choices will be eliminated by selections invoked at their binding dimension declarations. In the event of an unbound choice, the semantics are undefined.

Exercise 7. Extend V to be robust with respect to unbound choices. That is, unbound choices should be preserved in the semantics, as demonstrated in the following example.

$$
\begin{aligned}
&[\![A\langle\textbf{dim}\ B\langle b_1,b_2\rangle\ \textbf{in}\ B\langle 1,C\langle 2,3\rangle,4\rangle\rangle]\!] = \\
&\quad\{(B.b_1,A\langle 1,4\rangle),(B.b_2,A\langle C\langle 2,3\rangle,4\rangle)\}
\end{aligned}
$$

The case for sharing computes the semantics of the bound expression e_1, then computes the semantics of the scope e_2 with each variant e'_1 of e_1 added to the environment ρ, in turn. Each resulting expression e'_2 is then associated with the combined decision that produces it. References to **share**-bound variables simply lookup the corresponding plain expression in ρ.

In our work with the choice calculus, we have identified a set of semantics-preserving transforming laws for choice calculus expressions and related notions of representative normal forms with desirable properties (such as minimizing redundancy) [12]. This is the theoretical groundwork for a comprehensive theory of variation that can be reused by tool developers and other researchers. In the next section we switch gears by introducing a more exploratory thrust of this reasearch—a variation programming language, based on the choice calculus, for representing and manipulating variation.

4 A Variation DSEL in Haskell

The choice calculus, as presented in the previous two sections, is an entirely static representation. It allows us to precisely specify how a program varies, but we cannot use the choice calculus itself to edit, analyze, or transform a variational program. In the previous section we supplemented the choice calculus with mathematical notation to define some such operations, for example, tag selection. In some regards, math is an ideal metalanguage since it is infinitely extensible and extremely flexible—we can define almost any operation we can imagine. However, it's difficult to test an operation defined in math or to apply it to several examples quickly to observe its effect. In other words, it's hard to play around with math. This is unfortunate, since playing around can often lead to challenged assumptions, clever insights, and a deeper understanding of the problem at hand.

In this section, we introduce a domain-specific embedded language (DSEL) in Haskell for constructing and manipulating variational data structures. This DSEL is based on the choice calculus, but is vastly more powerful since we have the full power of the metalanguage of Haskell at our disposal. Using this DSEL, we can define all sorts of new operations for querying and manipulating variation. Because the operations are defined in Haskell, certain correctness guarantees are provided by the type system, and most importantly, we can actually execute the operations and observe the outputs. Through this DSEL we can support a hands-on, exploratory approach to variation research.

In the rest of this tutorial we will be exploring the interaction of variation representations and functional programming. Combining these ideas gives rise to the notion of *variation programming*, an idea that is explored more thoroughly in Sections 5 and 6.

In the DSEL, both the variation representation and any particular object language are represented as data types. The data type for the generic variation representation is given below. As you can see, it adapts the dimension and choice constructs from the choice calculus into Haskell data constructors, `Dim` and `Chc`. The `Obj` constructor will be explained below. In this definition, the types `Dim` and `Tag` are both synonyms for the predefined Haskell type `String`.

```
data V a = Obj a
         | Dim Dim [Tag] (V a)
         | Chc Dim [V a]
```

The type constructor name `V` is intended to be read as "variational", and the type parameter `a` represents the object language to be varied. So, given a type `Haskell` representing Haskell programs, the type `V Haskell` would represent variational Haskell programs (see Section 6).

The `Obj` constructor is roughly equivalent to the object structure construct from the choice calculus. However, here we do not explicitly represent the structure as a tree, but rather simply insert an object language value directly. An important feature of the DSEL is that it is possible for the data type representing the object language to itself contain variational types (created by applying the `V` type constructor to its argument types), and operations written in the DSEL can query and manipulate these nested variational values generically. This is achieved through the use of the "scrap your boilerplate" (SYB) library [19] which imposes a few constraints on the structure of `a`. These constraints will be described in Section 5.1. In the meantime, we will only use the very simple object language of integers, `Int`, which cannot contain nested variational values.

One of the advantages of using a metalanguage like Haskell is that we can define functional shortcuts for common syntactic forms. In Haskell, these are often called "smart constructors". For example, we define the following function `atomic` for defining atomic dimensions (a dimension with a single choice as an immediate subexpression).

```
atomic :: Dim -> [Tag] -> [V a] -> V a
atomic d ts cs = Dim d ts $ Chc d cs
```

Exercise 8. Define the following smart constructors:

(a) `dimA :: V a -> V a`, which declares a dimension `A` with tags `a1` and `a2`
(b) `chcA :: [V a] -> V a`, which constructs a choice in dimension `A`

These smart constructors will be used in examples throughout this section.

Note that we have omitted the sharing-related constructs from the definition of `V`. This decision was made primarily for two reasons. First, some of the sharing benefits of the choice calculus **share** construct are provided by Haskell directly, for example, through Haskell's `let` and `where` constructs. In fact, sharing in Haskell is much more powerful than in the choice calculus since we can also share values via functions. Second, the inclusion of an explicit sharing construct greatly complicates some important results later. In particular, we will show that `V` is a monad, while it is unclear whether this is true when `V` contains explicit sharing constructs. Several other operations are also much more difficult to define with explicit sharing.

There are, however, advantages to the more restricted and explicit form of sharing provided by the choice calculus. The first is perhaps the most obvious—since sharing is handled at the metalanguage level in the DSEL, it introduces redundancy when resolved into the variation representation (the `V` data type). This puts an additional burden on users to not introduce update anomalies and makes operations on variational data structures necessarily less efficient.

A more subtle implication of the metalanguage-level sharing offered by the DSEL is that we lose the choice calculus's property of static (syntactic) choice scoping. In the choice calculus, the dimension that binds a choice can always be determined by examining the context that the choice exists in; this is not the case in the DSEL. For example, in the following choice calculus expression, the choice in A is unbound.

$$\textbf{share}\ v = A\langle 1,2\rangle\ \textbf{in dim}\ A\langle a_1,a_2\rangle\ \textbf{in}\ v$$

Meanwhile, in the corresponding DSEL expression, the choice in `A` is bound by the dimension surrounding the variable reference. This is demonstrated by evaluating the following DSEL expression (for example, in GHCi), and observing the pretty-printed output.

```
> let v = chcA [Obj 1, Obj 2] in dimA v
dim A<a1,a2> in A<1,2>
```

In effect, in the choice calculus, sharing is expanded *after* dimensions and choices are resolved, while in the DSEL sharing is expanded *before*.

Exercise 9. Compare the semantics of the following expression if we expand sharing before dimensions and choices are resolved, with the semantics if we expand sharing after dimensions and choices are resolved.

$$\textbf{share}\ v = (\textbf{dim}\ A\langle a_1, a_2\rangle\ \textbf{in}\ A\langle 1,2\rangle)\ \textbf{in}\ (v,v)$$

The result in either case is a mapping with pairs of integers such as $(2,2)$ in its range.

The lack of static choice scoping, combined with the more unrestricted form of sharing offered by Haskell functions, also opens up the possibility for *choice capture*. This is where a choice intended to be bound by one dimension ends up being bound by another. As an example, consider the following operation `insertA` that declares a dimension `A`, then inserts a choice in `A` into some expression, according to the argument function.

```
insertA :: (V Int -> V Int) -> V Int
insertA f = dimA (f (chcA [Obj 1, Obj 2]))
```

The author of this operation probably expects that the inserted choice will be bound by the dimension declared in this definition, but if the argument function also declares a dimension `A`, the choice could be captured, as demonstrated below.

```
> insertA (\v -> Dim "A" ["a3","a4"] v)
dim A<a1,a2> in dim A<a3,a4> in A<1,2>
```

Now the choice is bound by the dimension in the argument, rather than the intended dimension declared in the `insertA` function.

Despite all of these qualms, however, the additional power and simpler variation model that results from the off-loading of sharing to the metalanguage makes possible a huge variety of operations on variational expressions. Exploring these operations will form the bulk of the remainder of this tutorial. Supporting this additional functionality while maintaining the structure, safety, and efficiency of the choice calculus's sharing constructs remains an important open research problem.

An important feature of the `V` data type is that it is both a functor and a monad. Functors and monads are two of the most commonly used abstractions in Haskell. By making the variation representation an instance of Haskell's `Functor` and `Monad` type classes, we make a huge body of existing functions and knowledge instantly available from within our DSEL, greatly extending its syntax. Functors are simpler than (and indeed a subset of) monads, so we will present the `Functor` instance first, below. The

`Functor` class contain one method, `fmap`, for mapping a function over a data structure while preserving its structure.

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

For `V`, this operation consists of applying the mapped function `f` to the values stored at `Obj` nodes, and propagating the calls into the subexpressions of `Dim` and `Chc` nodes.

```
instance Functor V where
  fmap f (Obj a)      = Obj (f a)
  fmap f (Dim d ts v) = Dim d ts (fmap f v)
  fmap f (Chc d vs)   = Chc d (map (fmap f) vs)
```

Consider the following variational integer expression `ab`, where `dimB` and `chcB` are smart constructors similar to `dimA` and `chcA`.

```
> let ab = dimA $ chcA [dimB $ chcB [Obj 1, Obj 2], Obj 3]
> ab
dim A<a1,a2> in A<dim B<b1,b2> in B<1,2>,3>
```

Using `fmap`, we can, for example, increment every object value in a variational integer expression.

```
> fmap (+1) ab
dim A<a1,a2> in A<dim B<b1,b2> in B<2,3>,4>
```

Or we can map the function `odd :: Int -> Bool` over the structure, producing a variational boolean value of type `V Bool`.

```
> fmap odd ab
dim A<a1,a2> in A<dim B<b1,b2> in B<True,False>,True>
```

Exercise 10. Write an expression that maps every integer `i` in `ab` to a choice between `i` and `i+1`. What is the type of the resulting value?

The definition of the `Monad` instance for `V` is similarly straightforward. The `Monad` type class requires the implementation of two methods: `return` for injecting a value into the monadic type, and `>>=` (pronounced "bind") for sequentially composing a monadic value with a function that produces another monadic value.

```
return :: Monad m => a -> m a
(>>=)  :: Monad m => m a -> (a -> m b) -> m b
```

The monad instance definition for the variational type constructor `V` is as follows. The `return` method is trivially implemented by the `Obj` data constructor. For `>>=`, at an `Obj` node, we simply return the result of applying the function to the value stored at that node. For dimensions and choices, we must again propagate the bind downward into subexpressions.

```
instance Monad V where
  return = Obj
  Obj a     >>= f = f a
  Dim d t v >>= f = Dim d t (v >>= f)
  Chc d vs  >>= f = Chc d (map (>>= f) vs)
```

The effect of a monadic bind is essentially to replace every value in the structure with another monadic value (of a potentially different type) and then to flatten the results. The `concatMap` function on lists is a classic example of this pattern (though the order of arguments is reversed). In the context of variation representations, we can use this operation to introduce new variation into a representation. For example, consider again the expression `ab`. We can add a new dimension `S`, indicating whether or not we want to square each value (the line break in the output was inserted manually).

```
> Dim "S" ["n","y"] $ ab >>= (\i -> Chc "S" [Obj i, Obj (i*i)])
dim S<y,n> in dim A<a1,a2> in
A<dim B<b1,b2> in B<S<1,1>,S<2,4>>,S<3,9>>
```

Each value in the original expression `ab` is expanded into a choice in dimension `S`. The resulting expression remains of type `V Int`. Compare this to the result of Exercise 10.

Finally, the DSEL provides several functions for analyzing variational expressions. For example, the function `freeDims :: V a -> Set Dim` returns a list of all free (unbound) dimensions in a given variational expression (without duplicates). Several other basic static analyses are also provided. Significantly, a semantics function for variational expressions, `sem`, is provided. This is based on the semantics of the choice calculus from the previous section. Similarly, the semantics of a variational expression of type `V a` is a mapping from decisions (lists of qualified tags) to plain expressions of type `a`. More commonly, we use a function `psem` which computes the semantics of an expression and pretty prints the results. For example, the pretty printed semantics of the expression `ab` are shown below.

```
> psem ab
[A.a1,B.b1]  =>  1
[A.a1,B.b2]  =>  2
[A.a2]  =>  3
```

Each entry in the semantics is shown on a separate line, with a decision on the left of each arrow and the resulting plain expression on the right.

While this section provided a brief introduction to some of the features provided by the DSEL, the following sections on variational programming will introduce many more. In particular, Section 5.1 will describe how to make a non-trivial data type variational, Section 5.2 and Section 5.3 will present a subset of the language designed for the creation of complex editing operations on variational expressions.

5 Variational Lists

We start exploring the notion of variation programming with lists, which are a simple but expressive and pervasive data structure. The familiarity with lists will help us to

identify important patterns when we generalize traditional list functions to the case of variational lists. The focus on a simple data structure will also help us point out the added potential for variation programming. We present variation programming with lists in several steps.

First, we explain the data type definition for variational lists and present several examples together with some helper functions in Section 5.1. Second, we develop variational versions for a number of traditional list functions in Section 5.2. We can observe that, depending on the types involved, certain patterns of recursion become apparent. Specifically, we will see that dependent on the role variation plays in the types of the defined functions, variational parts have to be processed using `fmap`, effectively treating them in a functorial style, or using `>>=`, treating them as monadic values. In Section 5.3 we turn our attention to editing operations for variational lists. While the adapted traditional list functions will naturally produce variational data structures (such as lists, numbers, etc.), these are a result from already existing variations in the lists that were given as arguments and thus result more as a kind of side effect. In contrast, list editing operations introduce or change variation structure purposefully. We will present in Section 5.4 some comments and observations on the different programming styles employed in the two subsections 5.2 and 5.3.

As a motivating example we consider how to represent menu preferences using choices and dimensions. Suppose that we prefer to order meat or pasta as the main course in a restaurant and that with meat we always order french fries on the side. Also, if we order pasta, we may have cake for dessert. Using the choice calculus we can represent these menu options as follows (here ε represents an empty token that, when selected, does not appear in the list as an element but rather disappears).

dim $Main\langle meat, pasta\rangle$ **in**
$Main\langle$ `[Steak,Fries]` , `[Pasta,` **dim** $Dessert\langle yes, no\rangle$ **in** $Dessert\langle$ `Cake` $, \varepsilon\rangle$ `]` $\rangle$

Here we have used a simple list notation as an object language. This notation leaves open many questions, such as how to nest lists and how to compose variational list and a list without variations. We will look at these questions in more detail in the following.

5.1 Representing Variational Lists

Lists typically are represented using two constructors for empty lists and for adding single elements to lists. Since lists are the most important data structures in functional programming, they are predefined in Haskell and supported through special syntax. While this is nice, it prevents us from changing the representation to variational lists. Therefore, we have to define our own list representation first, which we then can extend in a variety of ways to discus the transition to variational lists.

A standard definition of lists is as follows.

```
data List a = Cons a (List a)
            | Empty
```

To create variational lists using the `V` data type, we have to apply `V` somewhere in this definition. One possibility is to apply `V` to `a` thus making the elements in a list variable.

```
data List a = Cons (V a) (List a)
            | Empty
```

While this definition is quite convenient[1] as far as varying elements is concerned, it does not allow us to vary lists themselves. For example, we *cannot* represent a list whose first element is `1` and whose tail is a either `[2]` or `[3,4]`.

This limitation results from the fact that we cannot have a choice (or any other variational constructs) in the second argument of `Cons`. This shortcoming can be addressed by throwing in another `V` type constructor.

```
data List a = Cons (V a) (V (List a))
            | Empty
```

This representation avoids the above problem and is indeed the most general representation imaginable. However, the problem with this representation is that it is *too general*. There are two major drawbacks. First, the representation makes the definitions of functions cumbersome since it requires to process two variational types for one constructor. More importantly, the way our DSEL is implemented does not allow the application of `V` to different types in the same data type, and thus cannot deal with the shown definition of `List`. This limitation is a consequence of the employed SYB library [19].[2]

A drawback of either of the two previous two approaches is that changing the type of existing constructors may break existing code. This aspect matters when variational structure is added to existing data structures. In such a situation we would like to be able to continue using existing functions without the need for any changes in existing code.

Therefore, we choose the following representation in which we simply add a new constructor, which serves as a hook for any form of variation to be introduced into lists. This definition yields what we call an *expanded list*, where "expanded" means that it can contain variational data. However, this expansion is not enough, we also need a type for *variational lists*, that is, lists that are the object of the `V` type constructor. We introduce a type abbreviation for this type. The two types `List a` and `VList a` for expanded and variational lists, respectively, depend mutually on one another and together accomplish through this recursion the lifting of the plain list data type into its fully variational version.

```
type VList a = V (List a)

data List a = Cons a (List a)
            | Empty
            | VList (VList a)
```

We are using the convention to use the same name for the additional constructor as for the variational type, in this case `VList`. This helps to keep the variational code more organized, in particular, in situations where multiple variational data types are used.

[1] Moreover, if this definition were all we needed, we could apply it directly to the predefined Haskell lists.

[2] It is possible to lift this constraint, but doing so requires rather complex generic programming techniques that would make the library much more difficult to use.

```
list :: List a -> VList a
list = Obj

single :: a -> List a
single a = Cons a Empty

many :: [a] -> List a
many = foldr Cons Empty
```

```
vempty :: VList a
vempty = list Empty

vsingle :: a -> VList a
vsingle = list . single

vcons :: a -> VList a -> VList a
vcons x = list . Cons x . VList

vlist :: [a] -> VList a
vlist = list . many
```

Fig. 4. Auxiliary functions for variational lists

With the chosen definition for the data type `List` we can represent the variational list for representing our menu choices as follows. First, we introduce a data type for representing food items.

```
data Food = Steak | Pasta | Fries | Cake
```

Note that for the above data type we also derive instances for `Eq`, `Show`, `Data` and `Typeable`. Instances of `Data` and `Typeable` are required for the SYB library to work. Every data type in this tutorial that will be used with the `V` type constructor also derives instances for these classes, although we don't show this explicitly each time.

We also introduce a few auxiliary functions that help make the writing of variational lists more concise, see Figure 4. For example, `vempty` represents an empty variational list, `vsingle` constructs a variational list containing one element, and `vcons` takes an element and adds it to the beginning of a variational list. The function `vlist` transforms a regular Haskell list into a `VList`, which lets us reuse Haskell list notation in constructing `VLists`. All three definitions are based on corresponding `List` versions and use the synonym `list` for `Obj`, which lifts an object language expression into a variational expression. The function `list` is more concrete than `Obj` in the sense that it explicitly tells us that a `List` value is lifted to the variational level. It can also be understood as indicating, within a variational expression: "look, here comes an ordinary list value". We use similar synonyms for other object languages (for example, `int` or `haskell`), and we will even use the synonym `obj` for generic values.

Exercise 11. The function `vcons` shown in Figure 4 adds a single (non-variational) element to a variational list. Define a function `vvcons` that adds a choice (that is, a variational element) to a variational list. (*Hint:* Since you have to deal with two occurrences of the `V` constructor, you might want to exploit the fact that `V` is a monad.)

Using these operations, we can give the following definition of the menu plan as a variational list.

```
type Menu = VList Food

dessert :: Menu
dessert = atomic "Dessert" ["yes","no"] [vsingle Cake,vempty]

menu :: Menu
menu = atomic "Main" ["meat","pasta"]
              [vlist [Steak,Fries],Pasta `vcons` dessert]
```

We can examine the structure we have built by evaluating `menu` (again, the line break was inserted manually).

```
> menu
dim Main<meat,pasta> in
    Main<[Steak;Fries],[Pasta;dim Dessert<yes,no> in Dessert<[Cake],[]>]>
```

Note that we have defined the pretty printing for the `List` data type to be similar to ordinary lists, except that we use ; to separate list elements. In this way we keep a notation that is well established but also provides cues to differentiate between lists and variational lists.

Since the presence of nested dimensions complicates the understanding of variational structures, we can use the semantics of `menu` to clarify the represented lists.

```
> psem menu
[Main.meat]  =>  [Steak;Fries]
[Main.pasta,Dessert.yes]  =>  [Pasta;Cake]
[Main.pasta,Dessert.no]  =>  [Pasta]
```

Exercise 12. Change the definition of `menu` so that we can choose dessert also for a meat main course. There are two ways of achieving this change: (a) by copying the `dessert` dimension expression into the other choice, or (b) by lifting the dimension declaration out of the main choice.

Before we move on to discuss variational list programs, we show a couple of operations to facilitate a more structured construction of variation lists. These operations are not very interesting from a transformational point of view, but they can be helpful in decomposing the construction of complicated variational structures into an orderly sequence of steps.

This doesn't seem to be such a big deal, but if we take a closer look at the definition of `menu` shown above, we can observe that we have employed in this simple example alone five different operations to construct lists, namely, `vsingle`, `vempty`, `vlist`, `vcons`, and `[]`. To decide which operation to use where requires experience (or extensive consultation with the Haskell type checker).

In the construction of `menu` we can identify two patterns that seem to warrant support by specialized operations. First, the definition of `dessert` is an instance of a dimension representing that something is optional. We can therefore define a function `opt` for introducing an optional feature in a straightforward way as follows.

```
opt :: Dim -> a -> VList a
opt d x = atomic d ["yes","no"] [vsingle x,vempty]
```

Second, the definition of `menu` was given by separating the tags and the lists they label. A more modular definition can be given if we define the two different menu options separately and the combine them to a menu. To do that we introduce some syntactic sugar for defining tagged variational lists.

```
type Tagged a = (Tag,V a)

infixl 2 <:

(<:) :: Tag -> V a -> Tagged a
t <: v = (t,v)
```

Next we can define an operation `alt` for combining a list of tagged alternatives into a dimension.

```
alt :: Dim -> [Tagged a] -> V a
alt d tvs = atomic d ts vs where (ts,vs) = unzip tvs
```

With the help of `opt`, `<:`, and `alt` we can thus give a the following, slightly more modular definition of `menu`.

```
dessert = opt "Dessert" Cake
meat    = "meat"  <: vlist [Steak,Fries]
pasta   = "pasta" <: Pasta `vcons` dessert
menu    = alt "Main" [meat,pasta]
```

This definition produces exactly the same (syntactic) variational list as the definition given above.

5.2 Standard Variational List Functions

Among the most common functions for lists are functions to transform lists or to aggregate them. In the following we will illustrate first how to implement some of these functions directly using pattern matching and recursion. We will later introduce more general variational list functions, such as map and fold.

Let us start by implementing the function `len` to compute the length of a variational list. The first thing to realize is that the return type of the function is not just `Int` but rather `V Int` since the variation in a list may represent lists of different lengths. The implementation can be performed by pattern matching: The length of an empty list is zero. However, we must be careful here to not just return `0` since the return type of the function requires a `V Int` value. We therefore have to lift the `0` into the `V` type using the constructor `Obj`, for which we also provide the abbreviation `int` (see the discussion of `list` above, and note that we could also use the `return` method from `Monad` for this). The length of a non-empty list is given by the length of the tail plus one. Again, because of the structured return type of `len` we cannot simply add one to the result of the recursive call. Since `len xs` can produce, in general, an arbitrarily complex variation expression overs integers, we have to make sure to add one to all variants, which can

be accomplished by the function `fmap`. Finally, to compute the length of a list whose representation is distributed within a `V` structure, we have to carry the `len` computation to all the lists in the `V` representation. One could think of doing that again using `fmap`. However, looking at the involved types tells us that this is not the right approach, because one would end up with a bunch of `V Int` values scattered all over a `V` value. What we need instead is a single `V Int` value. So we are given a `V (List a)` value `vl` and a function `len` of type `List a -> V Int` to produce a value of type `V Int`. If we abstract from the concrete types a little bit by replacing `List a` by `a`, `V` by `m`, and `Int` by `b`, we see that to combine `vl` and `len` we need a function of the following type.

```
m a -> (a -> m b) -> m b
```

As we know (or otherwise could find out quickly using Hoogle [15]), this is exactly the type of the monadic bind operation, which then tells us the implementation for the last case. Thinking about it, applying `len` to `vl` using monadic bind makes a lot of sense since our task in this case is to compute variational data in many places and then join or merge them into the existing variational structure of `vl`.

```
len :: List a -> V Int
len Empty       = int 0
len (Cons _ xs) = fmap (+1) (len xs)
len (VList vl)  = vl >>= len
```

Now if we try to apply `len` to one of the variational lists defined in Section 5, we find that the types do not match up. While `len` is a function that works for lists that *contain* variational parts, it still expects an expanded list as its input. It seems we need an additional function that can be applied to values of type `V (List a)`.

In fact, we have defined such a function already in the third case of `len`, and we could simply reuse that definition. Since it turns out that we need to perform such a lifting into a `V` type quite often, we define a general function for that purpose.

```
liftV :: (a -> V b) -> V a -> V b
liftV = flip (>>=)
```

As is apparent from the type and implementation (and also from the discussion of the third case of `len`), the `liftV` function is essentially the bind operation of the `V` monad.

With `liftV` we obtain the required additional version of the function `len`.

```
vlen :: VList a -> V Int
vlen = liftV len
```

We generally use the following naming convention for functions. Given a function `f` whose input is of type `T` we use the name `vf` for its lifted version that works on values of type `V T`.

We can now test the definition of `vlen` by applying it to the example list `menu` defined in Section 5.1.

```
> vlen menu
dim Main<meat,pasta> in Main<2,dim Dessert<yes,no> in Dessert<2,1>>
```

As expected the result is a variational expression over integers. We can obtain a more concise representation by computing the semantics of this expression.

```
> psem $ vlen menu
[Main.meat]  =>  2
[Main.pasta,Dessert.yes]  =>  2
[Main.pasta,Dessert.no]  =>  1
```

Exercise 13. Implement the function `sumL :: List Int -> V Int` using pattern matching and recursion. Then define the function `vsum :: VList Int -> V Int`.

We have explained the definition of `len` in some detail to illustrate the considerations that led to the implementation. We have tried to emphasize that the generalization of a function definition for ordinary lists to variational lists requires mostly a rigorous consideration of the types involved. In other words, making existing implementations work for variational data structures is an exercise in *type-directed programming* in which the types dictate (to a large degree) the code [39].

Before moving on to defining more general functions on variational lists, we will consider the definition of list concatenation as an example of another important list function. This will highlight an important pattern in the generalization of list functions to the variational case.

The definition for the `Empty` and `Cons` case are easy and follow the definition for ordinary lists, that is, simply return the second list or recursively append it to the tail of the first, respectively. However, the definition for a variational list is not so obvious. If the first list is given by a variation expression, say `vl`, we have to make sure that we append the second list to all lists that are represented in `vl`. In the discussion of the implementation of `len` we have seen that we have, in principle, two options to do that, namely `fmap` and `>>=`. Again, a sharp look at what happens to the involved types will tell us what the correct choice is. For the concatenation of lists we can observe that the result type stays the same, that is, it is still a value of type `List a`, which means that we can traverse `vl` and apply the function `cat` with its second argument fixed to all lists that we encounter. This can be accomplished by the function `fmap`. The situation for `len` was different because its result was a variational type, which required the flattening of the resulting cascading `V` structures through `>>=`.

```
cat :: List a -> List a -> List a
cat Empty      r = r
cat (Cons a l) r = Cons a (l `cat` r)
cat (VList vl) r = VList (fmap (`cat` r) vl)
```

As for `len`, we also need a version of `cat` that works for variational lists.[3] A simple solution is obtained by simply lifting the variational list arguments into the `List` type using the `VList` constructor, which facilitates the application of `cat`.

```
vcat :: VList a -> VList a -> VList a
vcat l r = list $ cat (VList l) (VList r)
```

[3] Remember that `List a` represents only the *expanded* list type and that `VList a` is the *variational* list type.

To show `vcat` in action, assume that we extend `Food` by another constructor `Sherry` which we use to define the following variational list representing a potential drink before the meal.

```
aperitif :: VList Food
aperitif = opt "Drink" Sherry
```

When we concatenate the two lists `aperitif` and `menu`, we obtain a variational list that contains a total of six different variants. Since the evaluation of `vcat` duplicates the dimensions in `menu`, the resuting term structure becomes quite difficult to read and understand. We therefore show only the semantics of the result.

```
psem $ vcat aperitif menu
[Drink.yes,Main.meat]  =>  [Sherry;Steak;Fries]
[Drink.yes,Main.pasta,Dessert.yes]  =>  [Sherry;Pasta;Cake]
[Drink.yes,Main.pasta,Dessert.no]  =>  [Sherry;Pasta]
[Drink.no,Main.meat]  =>  [Steak;Fries]
[Drink.no,Main.pasta,Dessert.yes]  =>  [Pasta;Cake]
[Drink.no,Main.pasta,Dessert.no]  =>  [Pasta]
```

Exercise 14. Define the function `rev` for reversing expanded lists. You may want to use the function `cat` in your definition. Also provide a definition of the function `vrev` for reversing variational lists. Before testing your implementation, try to predict what the result of the expression `vrev menu` should be.

All of the examples we have considered so far have lists as arguments. Of course, the programming with variational lists should integrate smoothly with other, non-variational types. To illustrate this, we present the definition of the functions `nth` and `vnth` to compute the nth element of a variational list (recall that we use `obj` as a synonym for `Obj`, to maintain letter-case consistency with `list` and `int`).

```
nth :: Int -> List a -> V a
nth _ Empty       = undefined
nth 1 (Cons x _)  = obj x
nth n (Cons _ xs) = nth (n-1) xs
nth n (VList vl)  = vl >>= nth n
```

We can observe that the integer parameter is passed around unaffected through the variational types. The lifting to variational lists is straightforward.

```
vnth ::  Int -> VList a -> V a
vnth n = liftV (nth n)
```

We also observe that the computation of `nth` can fail. This might be more annoying than for plain lists because in general the length of the list in a variational list expressions is not obvious. Specifically, the length can vary! Therefore, it is not obvious what argument to call `vnth` with. For example, the following computation produces the expected result that the first item in a menu list is either `Steak` or `Pasta`.

```
> vnth 1 menu
dim Main<meat,pasta> in Main<Steak,Pasta>
```

However, since there is no second item for the `Main.pasta`, `Dessert.no` list, the computation `vnth 2 menu` fails. This is a bit disappointing since for some variants a second list element does exist. A definition for `nth`/`vnth` using a `V (Maybe a)` result type seems to be more appropriate. We leave the definition of such a function as an exercise. As another exercise consider the following task.

Exercise 15. Define the function `filterL:: (a -> Bool) -> List a -> List a` and give a definition for the corresponding function `vfilter` that operates on variational lists.

The final step in generalizing list functions is the definition of a fold operation (and possibly other generic list processing operations) for variational lists. The definition for `fold` can be easily obtained by taking the definition of `len` (or `sum`/`sumList` from Exercise 13) and abstracting from the aggregating function `+`.

```
fold :: (a -> b -> b) -> b -> List a -> V b
fold _ b Empty      = obj b
fold f b (Cons a l) = fmap (f a) (fold f b l)
fold f b (VList vl) = vl >>= fold f b
```

With `fold` we should be able to give more succinct definitions for functions, such as `len`, which is indeed the case.

```
len :: List a -> V Int
len = fold (\_ s->succ s) 0
```

Finally, we could also consider recursion on multiple variational lists. We leave this as an exercise.

Exercise 16. Implement the function `zipL:: List a -> List b -> List (a,b)` and give a definition for the corresponding function `vzip` that operates on variational lists.

As an example application of `vzip`, consider the possible meals when two people dine.

```
> psem $ vzip menu menu
[Main.meat,Main.meat]  =>  [(Steak,Steak);(Fries,Fries)]
[Main.meat,Main.pasta,Dessert.yes]  =>  [(Steak,Pasta);(Fries,Cake)]
[Main.meat,Main.pasta,Dessert.no]  =>  [(Steak,Pasta)]
[Main.pasta,Main.meat,Dessert.yes]  =>  [(Pasta,Steak);(Cake,Fries)]
[Main.pasta,Main.meat,Dessert.no]  =>  [(Pasta,Steak)]
[Main.pasta,Main.pasta,Dessert.yes,Dessert.yes]  =>  [(Pasta,Pasta);
                                                      (Cake,Cake)]
[Main.pasta,Main.pasta,Dessert.yes,Dessert.no]  =>  [(Pasta,Pasta)]
[Main.pasta,Main.pasta,Dessert.no]  =>  [(Pasta,Pasta)]
```

Now, this looks a bit boring. Maybe we could consider filtering out some combinations that are considered "bad" for some reason, for example, when somebody has dessert while the other person is still having the main course. We might also consider a more relaxed definition of `vzip` in which one person can have pasta and dessert and the other person can have pasta and no dessert. Note that while we can select this possibility in the above semantics, the corresponding variant does not reflect this since when two lists of differing lengths are zipped, the additional elements of the longer list are discarded.

5.3 Edit Operations for Variational Lists

The `menu` example that we introduced in Section 5.1 was built in a rather ad hoc fashion in one big step from scratch. More realistically, variational structures develop over time, by dynamically adding and removing dimensions and choices in an expression, or by extending or shrinking choices or dimensions. More generally, the rich set of laws that exists for the choice calculus [12] suggest a number of operations to restructure variation expressions by moving around choices and dimensions. Specifically, operations for the factoring of choices or the hoisting of dimensions reflect refactoring operations (that is, they preserve the semantics of the transformed variation expression). These are useful for bringing expressions into various normal forms.

In this section we will present several operations that can be used for the purpose of evolving variation representations. Most of these operations will be generic in the sense that they can be applied to other variational structures, and we will actually reuse some of them in Section 6.

As a motivating example let us assume that we want, in our dinner decisions, to think first about the dessert and not about the main course. To obtain an alternative list representation with the `Dessert` dimension at the top we could, of course, build a new representation from scratch. However, this approach does not scale very well, and the effort becomes quickly prohibitive as the complexity of the variational structures involved grow. An alternative, more flexible approach is to take an already existing representation and transform it accordingly. In our example, we would like to split the declaration part off of a dimension definition and move it to the top level. This amounts to the repeated application of commutation rules for dimensions [12]. We can break down this operation into several steps as follows. Assume `e` is the expression to be rearranged and `d` is the name of the dimension declaration that is to be moved.

(1) Find the dimension `d` that is to be moved.
(2) If the first step is successful, cut out the found dimension expression `Dim d ts e'` and remember its position, which can be done in a functional setting through the use of a context `c`, that is, an expression with a hole that is conveniently represented by a function.
(3) Keep the scope of the found dimension declaration, `e'`, at its old location, which can be achieved by applying `c` to `e'`.
(4) Finally, move the declaration part of the dimension definition to the top level, which is achieved by wrapping it around the already changed expression obtained in the previous step; that is, we produce the expression `Dim d ts (c e')`.

To implement these steps we need to solve some technically challenging problems.

For example, finding a subexpression in an arbitrary data type expression, removing it, and replacing it with some other expression requires some advanced generic programming techniques. To this end we have employed the SYB [19] and the "scrap your zipper" [1] libraries for Haskell, which allow us to implement such generic transformation functions. Since a detailed explanation of these libraries and how the provided functions work is beyond the scope of this tutorial, we will only briefly mention what the functions will do as we encounter them. The approach is based on a type `C a`, which represents a *context* in a type `V a`. Essentially, a value of type `C a` represents a pointer to a subexpression of a value of type `V a`, which lets us extract the subexpression and also replace it. A context is typically the result of an operation to locate a subexpression with a particular property. We introduce the following type synonym for such functions.

```
type Locator a = V a -> Maybe (C a)
```

The `Maybe` type indicates that a search for a context may fail. As a generic function to locate subexpressions and return a matching context, we provide the following function `find` that locates the first occurrence of a subexpression that satisfies the given predicate. A predicate in this context means a boolean function on variational expressions.

```
type Pred a = V a -> Bool
```

The function `find` performs a preorder traversal of the expression and thus locates the topmost, leftmost subexpression that satisfies the predicate.

```
find :: Data a => Pred a -> Locator a
```

That `Data` class constraint is required for the underlying zipper machinery in the implementation of `find`. The function `find` already realizes the first step of the transformation sequence needed for refactoring the representation of the variational list `menu`. All we need is a predicate to identify a particular dimension `d`, which is quite straightforward to define.

```
dimDef :: Dim -> Pred a
dimDef d (Dim d' _ _) = d == d'
dimDef _ _            = False
```

The second step of cutting out the dimension is realized by the function `extract`, which conveniently returns as a result a pair consisting of the context and the subexpression sitting in the context. The function `extract` is an example of a class of functions that split an expression into two parts, a context plus some additional information about the expression in the hole. In the specific case of `extract` that information is simply the expression itself. This level of generality is sufficient for this tutorial, and we therefore represent this class of functions by the following type.

```
type Splitter a = V a -> Maybe (C a,V a)
```

The definition of `extract` uses `find` to locate the context and then simply extracts the subexpression stored in the context using the predefined zipper function `getHole`.

```
extract :: Data a => Pred a -> Splitter a
extract p e = do c <- find p e
                 h <- getHole c
                 return (c,h)
```

The third step of applying the context to the scope of the dimension expression requires the function `<@`, whose definition is based on elementary zipper functions that we don't show here.

```
(<@) :: Data a => C a -> V a -> V a
```

The function `<@` can also be understood as inserting the second argument into the hole of the first.

Finally, we can combine all these functions and define a function `hoist` for hoisting dimension declarations. To avoid having to deal with error handling when we call `hoist` we return as a default expression the original expression in case the process of lifting fails at some stage.

```
hoist :: Data a => Dim -> V a -> V a
hoist d e = withFallback e $ do
    (c,Dim _ ts e') <- extract (dimDef d) e
    return (Dim d ts (c <@ e'))
```

Note that the function `withFallback` is simply a synonym for `fromMaybe`. We can apply `hoist` to `menu` to obtain a different choice calculus representation, which produces the expected result (the line break was manually inserted).

```
> hoist "Dessert" menu
dim Dessert<yes,no> in dim Main<meat,pasta> in
    Main<[Steak;Fries],[Pasta;Dessert<[Cake],[]>]>
```

There are two obvious shortcomings of the current definition for `hoist`. One problem is that moving the dimension might capture free `Dessert` choices.[4] The other problem is that the `Dessert` decision might be made for nothing since it does not have an effect when the next decision in `Main` is to select `meat`.

The first problem can be easily addressed by extending the definition of `hoist` by a check for capturing unbound `Dessert` choices that returns failure (that is, `Nothing`) if `d` occurs free anywhere in `e`. This failure will be caught eventually by the `withFallback` function that will ensure that the original expression `e` is returned instead.

```
safeHoist :: Data a => Dim -> V a -> V a
safeHoist d e = withFallback e $ do
    (c,Dim _ ts e') <- extract (dimDef d) e
    if d `Set.member` freeDims e
      then Nothing
      else return (Dim d ts (c <@ e'))
```

The function `freeDims` returns a set (as Haskell's `Data.Set`) of the dimension names of unbound choices, as described in Section 4.

[4] Capturing free desserts sounds actually quite appealing from an application point of view. :)

Exercise 17. The implementation of `safeHoist` prevents the lifting of the dimension `d` if this would cause the capture of `d` choices.

(a) What other condition could cause, at least in principle, the hoisting of a dimension to be unsafe (in the sense of changing the semantics of the variational list)?
(b) Why don't we have to check for this condition in the implementation of `safeHoist`? (*Hint*: Revisit the description of how the function `find` works.)

The second problem with the definition of `hoist` can be seen if we compare the semantics of the menu with the hoisted dimension with the semantics of the original expression (shown in Section 5.1).

```
dMenu = hoist "Dessert" menu

> psem $ dMenu
[Dessert.yes,Main.meat]  =>  [Steak;Fries]
[Dessert.yes,Main.pasta]  =>  [Pasta;Cake]
[Dessert.no,Main.meat]  =>  [Steak;Fries]
[Dessert.no,Main.pasta]  =>  [Pasta]
```

It is clear that the `Dessert` decision has no effect if the `Main` decision is `meat`. The reason for this is that the `Dessert` choice appears only in the `pasta` choice of the `Main` dimension. We can fix this by moving the `Main` choice plus its dimension declaration into the `no` alternative of the `Dessert` choice.

This modification is an instance of the following slightly more general transformation schema, which applies in situations in which a choice in dimension B is available only in one of the alternatives of all choices in another dimension A. (Here we show for simplicity the special case in which A has only one choice with two alternatives.) Such an expression can be transformed so that the selection of b_1 is guaranteed to have an effect, that is, we effectively trigger the selection of a_2 by copying the alternative, because the selection of a_1 would leave the decision to pick b_1 without effect.

$$\begin{aligned} &\textbf{dim}\ B\langle b_1,b_2\rangle\ \textbf{in}\ \textbf{dim}\ A\langle a_1,a_2\rangle\ \textbf{in}\ A\langle \texttt{[a1]}, \texttt{[a2;}B\langle \texttt{b1},\texttt{b2}\rangle\texttt{]}\rangle \\ \rightsquigarrow\ &\textbf{dim}\ B\langle b_1,b_2\rangle\ \textbf{in}\ B\langle \texttt{[a2;b1]}, \textbf{dim}\ A\langle a_1,a_2\rangle\ \textbf{in}\ A\langle \texttt{[a1]}, \texttt{[a2;b2]}\rangle\rangle \end{aligned}$$

Note that the selection of b_2 does not have this effect since we can still select between a_1 and a_2 in the transformed expression. This transformation makes the most sense in the case when B represents an optional dimension, that is, $b_1 = yes$, $b_2 = no$, and $\texttt{b2} = \varepsilon$, because in this case the selection of $b_2 = no$ makes no difference, no matter whether we choose a_1 or a_2.

This transformation can be extended to the case in which A has more than two alternatives and more than one choice, which requires, however, that each A choice contains the B choice in the same alternative k.

We will next define a function that can perform the required transformation automatically. For simplicity we assume that the choice in `b` to be prioritized (corresponding to the choice in B above) is contained in the second alternative of the choice in `a` (which corresponds to A above).

```
prioritize :: Data a => Dim -> Dim -> V a -> V a
prioritize b a e = withFallback e $ do
    (dA,ae)            <- extract (dimDef a) e
    (cA,Chc _ [a1,a2]) <- extract (chcIn a) ae
    (cB,Chc _ [b1,b2]) <- extract (chcIn b) a2
    return $ dA <@ (Chc b [cB <@ b1,cA <@ (Chc a [a1,cB <@ b2])])
```

The function works as follows. Much like most transformations, it will first decompose the expression to be transformed into a collection of (nested) contexts and expressions, which are then used to build the result expression. Specifically, we first find the location of the dimension definition for `a` and remember it in the form of a context `dA`. Next, we find the context `cA` of the `a` choice. Finally, we find the choice to be prioritized in the second alternative of the `a` choice, `a2`. Both choices are found using the `extract` function with the predicate `chcFor` that finds a particular choice, similar to the `dimDef` predicate.

```
chcIn :: Dim -> Pred a
chcIn d (Chc d' _) = d == d'
chcIn _ _          = False
```

Having thus isolated all the required subexpressions, we can assemble the result by applying the contexts following the RHS of the above transformation schema.

Note that this transformation does *not* preserve the semantics; in fact, the reason for applying it is that it makes the semantics more compact. The transformation is, however, variant preserving; that is, no variants are added or removed, only the decisions to reach the variants have changed. This can be best seen by comparing the semantics of `dMenu` shown above with the semantics of `dMenu` with the `Dessert` choice prioritized over the `Main` choice.

```
> psem $ prioritize "Dessert" "Main" dMenu
[Dessert.yes]  =>  [Pasta;Cake]
[Dessert.no,Main.meat]  =>  [Steak;Fries]
[Dessert.no,Main.pasta]  =>  [Pasta]
```

The prioritization of the `Dessert` choice has removed the effectively unavailbale decision for `meat` in the case of `yes` for `Dessert`.

Exercise 18. The implementation of `prioritize` assumes that the choice to be lifted is located in the second alternative of the choice in `a`. Generalize the implementation of `prioritize` so that the choice in `b` can be lifted out of either alternative.

As a final example we illustrate how to combine the two previously defined transformations. In terms of the choice calculus, combining dimension hoisting and choice prioritization leads to a transformation that we call *dependency inversion*.

$$\begin{aligned} &\textbf{dim}\ A\langle a_1,a_2\rangle\ \textbf{in}\ A\langle \texttt{[a1]}, \texttt{[a2;}\textbf{dim}\ B\langle b_1,b_2\rangle\ \textbf{in}\ B\langle \texttt{b1,b2}\rangle\texttt{]}\rangle \\ \rightsquigarrow\ &\textbf{dim}\ B\langle b_1,b_2\rangle\ \textbf{in}\ B\langle \texttt{[a2;b1]}, \textbf{dim}\ A\langle a_1,a_2\rangle\ \textbf{in}\ A\langle \texttt{[a1]}, \texttt{[a2;b2]}\rangle\rangle \end{aligned}$$

Reusing the definitions for `hoist` and `prioritize`, the definition of inversion is rather straightforward.

```
invert :: Data a => Dim -> Dim -> V a -> V a
invert b a = prioritize b a . hoist b
```

The definition of `invert` demonstrates that we can build more complicated variation programs out of simpler components and thus illustrate the compositional nature of our variation DSEL.

5.4 Variation Programming Modes

To close this section, we share a few thoughts on the nature of variation programming. The two sections 5.2 and 5.3 have illustrated that making data structures variational leads to two different programming *modes* or *attitudes*. On the one hand, the focus can be on manipulating the data structure itself, in which case the variational parts are just maintained but not essentially changed. This is what Section 5.2 was all about. On the other hand, the focus can be on changing the variation in the data structure, in which case the existing represented objects are kept mostly intact. This is what Section 5.3 was concerned with.

The different ways of processing edits to data structures have been classified under the name of *persistence* [25]. Imperative languages typically support no persistence, that is, edits to data structures are destructive and make old versions inaccessible. In contrast, data structures in functional languages are by default fully persistent, that is, old versions are in principle always accessible as long as a reference to them is kept. (There are also the notions of partial persistence and confluent persistence that are not of interest here.) Variational data structures add a new form of persistence that we call *controlled persistence* because it gives programmers precise control over what versions of a data structure to keep and how to refer to them. In contrast to all other forms of persistence (or non-persistence), which happen rather automatically, controlled persistence requires a conscious effort on part of the programmer to create and retrieve different versions of a data structure, and it keeps information about the versions around for the programmer to see and exploit.

6 Variational Software

The motivation for the choice calculus was the representation of variation in software, and having uncovered some basic principles of variation programming in Section 5, we are finally in a position to look at how we can put the choice calculus to work, through variation programming, on variational software.

As a running example we pick up the `twice` example that was introduced earlier in Section 2. We will introduce a representation of (a vastly simplified version of) the object language Haskell in Section 6.1, together with a number of supporting functions. After that we will consider in Section 6.2 several simple example transformations for variational Haskell programs.

6.1 Representing Variational Haskell

Following the example given in Section 5.1 we will first introduce a data type definition for representing Haskell programs and then extend it to allow for variations.

Because of the limitations of our current library that are imposed by the use of the SYB library [19], we have to make a number of simplifying assumptions and compromises in our definition. One constraint is that within a data type definition the `V` type constructor can be applied only on one type. This has several implications. First, we cannot spread the definition of Haskell over several data types. We actually would have liked to do that and have, for example, different data types for representing expressions and declarations (for values, function, types, etc.). Since this is not possible, we are forced to represent function definitions using a `Fun` constructor as part of the expression data type. But this is not all. Ordinarily, we would represent parameters of a function definition by simple strings. However, since we want to consider as an example the renaming of function parameters, we would have to represent variational parameters by a type `V String` or so, which is unfortunately not possible since we have committed the `V` type constructor to the expression data type already. The solution to this problem is to represent function parameters also as expressions. Although we can ensure through the use of smart constructors that we build only function definitions that use variable names as parameters, this forced restriction on the representation is less than ideal.

Therefore, for the purpose of this tutorial, we will work with the following data type for representing Haskell expressions and programs.

```
data Haskell = App Haskell Haskell
             | Var Name
             | Val Int
             | Fun Name [Haskell] Haskell Haskell
               ...
```

As we did with lists, we can now add a constructor for introducing variational expressions.

```
type VHaskell = V Haskell

data Haskell = App Haskell Haskell
             | Var Name
             | Val Int
             | Fun Name [Haskell] Haskell Haskell
               ...
             | VHaskell VHaskell
```

Before we construct the representation of the variational `twice` function, we introduce a few more abbreviations and auxiliary functions to make the work with variational Haskell programs more convenient.

First, we introduce a function that turns a string that represents a binary function into a constructor for building expressions using that function. Consider, for example, the following simple Haskell expression.

```
2*x
```

When we try to represent this expression with the above data type, we have quite some work to do. First, we have to turn `2` and `x` into the Haskell expressions using the constructors `Val` and `Var`, respectively. Then we have to use the `App` constructor twice to form the application. In other words, we have to write the following expression.

```
haskell :: Haskell -> VHaskell
haskell = Obj

choice :: Dim -> [Haskell] -> Haskell
choice d = VHaskell.Chc d.map haskell
```

```
(.+) = op "+"
(.*) = op "*"

x,y,z :: Haskell
[x,y,z] = map Var ["x","y","z"]
```

Fig. 5. Auxiliary functions for variational Haskell programs

```
App (App (Var "*") (Val 2)) (Var x).
```

The function `op` defined below performs all the necessary wrapping for us automatically. (Less importantly, it also adds enclosing parentheses around the function name, which is exploited by the pretty printer to produce an infix representation.)

```
op :: Name -> Haskell -> Haskell -> Haskell
op f l r = App (App (Var ("(" ++ f ++ ")")) l) r
```

In Figure 5 we also define two infix operators that are defined as an abbreviation for a call to the `op` function. These are not essential but will make the `twice` example look even nicer. There we also introduce names for a few variable references. Moreover, in addition to the `haskell` synonym for the `Obj` constructor we also provide a smart constructor to build choices of Haskell expressions more directly.

Finally, we define a function `fun`, which provides an abbreviation for the `Fun` constructor.

```
fun :: Name -> [Haskell] -> Haskell -> VHaskell
fun n vs e = haskell $ Fun n vs e withoutScope

withoutScope :: Haskell
withoutScope = Var ""
```

In particular, `fun` constructs a function definition with an empty scope of the function definition since in our example we are interested only in the definition of `twice` and not its uses.

With all these preparations, we can now represent the variational definition of `twice` in our DSEL as follows.

```
twice = Dim "Par" ["x","y"]
      $ Dim "Impl" ["plus","times"]
      $ fun "twice" [v] i
      where v = choice "Par" [x,y]
            i = choice "Impl" [v .+ v, Val 2 .* v]
```

For comparison here is again the definition given in Section 2..

dim *Par*⟨*x*,*y*⟩ **in**
dim *Impl*⟨*plus*,*times*⟩ **in**
`twice` *Par*⟨x,y⟩ `=` *Impl*⟨*Par*⟨x,y⟩+*Par*⟨x,y⟩,2∗*Par*⟨x,y⟩⟩

To check that this definition mirrors the one given in Section 2, we can evaluate `twice` (the line breaks were added manually).

```
> twice
dim Par<x,y> in
dim Impl<plus,times> in
    twice Par<x,y> = Impl<Par<x,y>+Par<x,y>,2*Par<x,y>>
```

To check that this definition actually represents the desired four different implementations of `twice` we can compute its semantics.

```
> psem twice
[Par.x,Impl.plus]  =>  twice x = x+x
[Par.x,Impl.times]  =>  twice x = 2*x
[Par.y,Impl.plus]  =>  twice y = y+y
[Par.y,Impl.times]  =>  twice y = 2*y
```

Looking back at the definition of `twice`, notice how we have used Haskell's `where` clause to factor out parts of the definition. Whereas the definition of `i` is not really essential, the definition of `v` is, in fact, needed to avoid the copying of the parameter choice. In Section 2 we have seen how the **share** construct of the choice calculus facilitates the factorization of common subexpressions. We have earlier said that, for technical reasons, the current realization of the choice calculus as a Haskell DSEL does not support sharing, but we can see here that the situation is not completely dire since we can simulate the missing sharing of the choice calculus (at least to some degree) using Haskell's `let` (or `where`) bindings. Here is a slightly changed definition of the `twice` function that comes close to the example given in Section 2.

```
twice = Dim "Par" ["x","y"] $
        Dim "Impl" ["plus","times"] $
        let v = choice "Par" [x,y] in
            fun "twice" [v] (choice "Impl" [v .+ v, Val 2 .* v])
```

But recall from Section 4 that there is an important difference between Haskell's `let` and the **share** construct of the choice calculus, and that is the time when bindings are expanded. In the choice calculus shared expressions will be expanded only *after* all dimensions have been eliminated using tag selection, whereas in Haskell the expansion happens always *before* any selection.

6.2 Edit Operations for Variational Haskell

As an example for an editing operation we consider the task of turning a plain function definition into a variational one. To this end, we start with the plain variant of `twice` with parameter name `x` and implemented by `+`, and add dimensions to it.

```
xp = fun "twice" [x] (x .+ x)
```

Let us first consider the variation of the parameter name. In order to generalize the current definition `xp`, we need to do the following two things.

(1) Add a dimension declaration for `Par`.
(2) Replace references to `x` by choices between `x` and `y`.

The first step is easy and simply requires the addition of a dimension declaration (using the `Dim` constructor). The second step requires a traversal of the abstract syntax tree representing `twice` and the application of a transformation at all places where a variable `x` is encountered. This can be accomplished by employing the `everywhere` traversal function of the SYB library [19]. All we need is the definition of a transformation that identifies the occurrence of `x` variables and replaces them by choices. Such a transformation is indeed easy to define.[5]

```
addPar :: Haskell -> Haskell
addPar (Var "x") = choice "Par" [x,y]
addPar e = e
```

We can use this transformation as an argument for the `everywhere` traversal. Since `everywhere` is a generic function that must be able to traverse arbitrary data types and visit and inspect values of arbitrary types, the transformation passed to it as an argument must be a polymorphic function. The SYB library provides the function `mkT` that performs this task; that is, it generalizes the type of a function into a polymorphic one. We can therefore define the transformation to turn the fixed `x` variables in `twice` into choices between `x` and `y` as follows.

```
varyPar :: VHaskell -> VHaskell
varyPar = Dim "Par" ["x","y"] . everywhere (mkT addPar)
```

We can confirm that `varyPar` has indeed the desired effect.

```
> varyPar xp
dim Par<x,y> in twice Par<x,y> = Par<x,y>+Par<x,y>
```

A limitation of the shown transformation is that it renames *all* found variable names `x` and not just the parameter of `twice`. In this example, this works out well, but in general we have to limit the scope of the transformation to the scope of the variable declaration that is being varied. We can achieve this using the function `inRange` that we will introduce later. See also Exercise 21.

The next step in generalizing the function definition is to replace the addition-based implementation by a choice between addition and multiplication. This transformation works in exactly the same way, except that the function for transforming individual expressions has to do a more elaborate form of pattern matching on the expressions.

```
addImpl :: Haskell -> Haskell
addImpl e@(App (App (Var "(+)") l) r)
          | l == r = choice "Impl" [e, Val 2 .* r]
addImpl e = e
```

With `addImpl` we can define a transformation similar to `varyPar` that adds the variation of the implementation method as a new dimension.

[5] Here the fact that we have to represent parameters as expressions comes to our advantage since we do not have to distinguish the different occurrences of variables (definition vs. use) and can deal with both cases in one equation.

```
varyImpl :: VHaskell -> VHaskell
varyImpl = Dim "Impl" ["plus","times"] . everywhere (mkT addImpl)
```

To verify the effect of `varyImpl` we can apply it directly to `xp` or to the variational program we have already obtained through `varyPar xp`.

```
> varyImpl xp
dim Impl<plus,times> in twice x = Impl<x+x,2*x>

> varyImpl (varyPar xp)
dim Impl<plus,times> in
dim Par<x,y> in
    twice Par<x,y> = Impl<Par<x,y>+Par<x,y>,2*Par<x,y>>
```

We can see that the latter expression is not the same as `twice` since the dimensions occur in a different order. However, if we reverse the order of application for the two variation-adding transformations, we can verify that they indeed produce the same result as the hand-written definition for `twice`.

```
> varyPar (varyImpl xp) == twice
True
```

Exercise 19. One might think that even though the two expressions `twice` and `varyImpl (varyPar xp)` are not syntactically equal, their semantics might be, because, after all, they really represent the same variations. Explain why this is, in fact, *not* the case.

As a final example we consider the task of extending the parameter dimension by another option `z`, as we have illustrated in Section 2. This transformation involves the following steps.

(1) Extend the tags of the dimension declaration for `Par` by a new tag `z`.
(2) Extend all `Par` choices that are bound by the dimension declaration by a new alternative `z`.

The first step is rather straightforward and can be implemented using a similar approach to what we have done in Section 5.3, namely by extracting the definition, manipulating it, and putting it back.

However, the change to all bound choices is more complicated. This is because it is not sufficient to find one choice (or even a fixed number of choices), and we can't therefore simply reuse the `extract` function for this purpose. To deal with a variable number of choices we define a function `inRange` that applies a transformation to selective parts of a variational expression. More specifically, `inRange` takes a transformation `f` and two predicates on variational expressions, `begin` and `end` that mark regions of the expression in which `f` is to be applied; that is, `inRange` effectively applies `f` to all nodes in the expression that are "between" nodes for which `begin` is true and nodes for which `end` is not true. The function works as follows. The expression to be transformed is traversed until a node is encountered for which the `begin` predicate yields `True`. Then

the traversal continues and the transformation `f` is applied to all nodes encountered on the way until a node is found for which the predicate `end` yields `True`. In that case the traversal continues, applying `f` to other siblings of the matching end node, but does not descend beneath that node. When all descendants of a `begin`-matching node have been transformed or terminated by an `end`-matching node, the traversal continues until another node matching `begin` is found.

```
inRange :: Data a => (V a -> V a) -> (Pred a,Pred a) -> V a -> V a
```

Even though the implementation for `inRange` is quite elegant and not very complicated, we do not show it here (as we did for `find` and `<@`) because it is based on navigational functions from the underlying zipper library. The interested reader can find the definition in the accompanying source code.

With the help of the function `inRange` we can now implement the transformation for extending a dimension. This function takes four parameters, the name of the dimension, the new tag, a function to extend the bound choices, and the expression in which to perform the update. It works as follows. First, we locate the definition of the dimension `d` to be extended and remember the position in the context `c`. We then perform the extension of all choices bound by `d` by applying the function `inRange` to the scope of the found dimension, `e`. Finding all the relevant choices is accomplished by the two predicates that are passed as arguments to `inRange`. The first, `chcFor d`, finds choices in the scope of `d`, and the second `dimDef d` stops the transformation at places where another dimension definition for `d` ends the scope. In this way the shadowing of dimension definitions is respected. Finally, we construct the result by inserting a dimension declaration with the new tag `t` and the changed expression `e'` into the context `c`

```
extend :: Data a => Dim -> Tag -> (V a -> V a) -> V a -> V a
extend d t f e = withFallback e $ do
    (c, Dim _ ts e) <- extract (dimDef d) e
    let e' = f `inRange` (chcFor d,dimDef d) $ e
    return (c <@ Dim d (ts++[t]) e')
```

All we need now for the extension of `twice` by a new option for `z` is a function for extending choices by new expression alternatives. This can be easily done using the following function `addAlt`.

```
addAlt :: V a -> V a -> V a
addAlt a (Chc d as) = Chc d (as ++ [a])
```

We can extend the variational expression `twice` as planned by employing `extend` and `addAlt`.

```
twiceZ :: VHaskell
twiceZ = extend "Par" "z" (addAlt (haskell z)) twice
```

To check whether the function works as expected, we can evaluate `twiceZ`.

```
> twiceZ
dim Par<x,y,z> in
dim Impl<plus,times> in
    twice Par<x,y,z> = Impl<Par<x,y,z>+Par<x,y,z>,2*Par<x,y,z>>
```

Exercise 20. Define a function `swapOptions` that exchanges the two tags of a binary dimension and the corresponding alternatives in all bound choices.

Exercise 21. Define a function `renamePar` that adds a choice of parameter names to the definition of a specific function `f` by creating a dimension and corresponding choices that store the existing parameter name and a newly given name. Be careful to extend only those parameter names that are bound by `f`.

The function should be defined so that the expression `renamePar xp "x" "y"` produces the same result as `varyPar xp`.

The ability to programmatically edit variation representations is an important aspect of variation programming and our DSEL that we have barely scratched the surface of in this section. Identifying, characterizing, and implementing editing operations is also an important area for future research since it directly supports the development of tools for managing and manipulating variation.

7 Further Reading

In this section we provide some pointers to related work in the area of representing and transforming software variation. The purpose of this section is not to discuss the related work in depth or present a detailed comparison with the material presented in this tutorial, but rather point to several important works in the literature concerning variation representation.

In general, the field of of *software configuration management* (SCM) is concerned with managing changes in software systems and associated documents [38]. It is a subfield of the more general area of *configuration management* [21], which encompasses the theory, tools, and practices used to control the development of complex systems. Among the differnt kinds of SCM tools, *revision control systems* [26] are probably most widely used and manage changes to software and documents over time [37] and as repositories to facilitate collaboration [5]. In the context of revision control systems, the requirement to work on software in parallel with many developers leads to the problem of having to *merge* different versions of software [22]. As one interesting example for the many approaches in this field, the Darcs versioning system [8] provides a formalized [30] merge operation that can combine patches from separate branches.

The field of feature-oriented software development (FOSD) [2] takes the view that each piece of software offers a specific set of features and that these features can be modeled and implemented, at least to some degree, independently of one another. The goal is to represent features in such a way that allows software to be assembled mostly automatically from these features. Features are a specific way of expressing variation in software, and approaches to FOSD are thus relevant and an interesting source of ideas for variation representation and transformation.

On a very high level, features and their relationships are described with the help of *feature models*, which can be expressed as diagrams [16], algebras [14], propositional

formulas [3] (and more). Feature models describe the structure of software product lines (SPLs) [27, 29].

Approaches to the implementation of features can be categorized into roughly three different kinds of approaches.

First, *annotative approaches* express variation through a separate language. The most well-known annotative tool is the C Preprocessor (CPP) [13], which supports variation through #ifdef annotations, macro-expansion, etc. [35]. Even though very popular, the use of CPP often leads to code that is hard to understand [34]. A principal problem of CPP is that it cannot provide any kind of syntactic correctness guarantees for the represented variations, and consequently one can find many ill-formed variants in CPP-annotated software [20]. Other annotative approaches that, unlike CPP, respect the abstract syntax of the underlying object language and guarantee syntactic correctness of software variants include the CIDE tool [17], the TaP ("tag and prune") strategy [6], and the choice calculus on which this tutorial is based.

Second, the probably most popular approach in the area of FOSD is the *compositional approach*, in which features are implemented as separate building blocks that can be composed into programs. By selecting different sets of features, different program variants are created. This idea is often realized through extensions to object-oriented languages, such as mixins [4, 7], aspects [9, 18, 23], or both [24].

Third, in the *metaprogramming aproach*, one encodes variability using metaprogramming features [31, 32] of the object language itself. Typical examples can be found in the realm of functional programming languages, such as MetaML [36], Template Haskell [33], or Racket [28].

8 Concluding Remarks

In this tutorial we have presented both a formal model for representing variation and a DSEL that both partially implements this model, and extends it to the new domain of variation programming. We have illustrated variational programming with two extended examples of variational lists and variational Haskell programs. We would like to conclude with two final, take-home points about the motivation behind this research.

First, variation is a fact of software engineering life, but the current tools for managing this variation are often inadequate. We believe that the path to better support for variation is through a better understanding of the problems and the development of clear and reusable solutions. These things can only be achieved by establishing a simple, sound, and formal foundation on which a general theory of variation can be built. The choice calculus is a structured and flexible representation for variation that can serve as this foundation.

Second, in addition to the simple selection of variants, a structured variation representation offers many other opportunities for queries and transformations. In other words, the potential exists for variation programming. By integrating the representation offered by the choice calculus into a programming environment, this can be achieved. We have used Haskell for this purpose, but many other embeddings are conceivable.

References

1. Adams, M.D.: Scrap Your Zippers – A Generic Zipper for Heterogeneous Types. In: ACM SIGPLAN Workshop on Generic Programming, pp. 13–24 (2010)
2. Apel, S., Kästner, C.: An Overview of Feature-Oriented Software Development. Journal of Object Technology 8(5), 49–84 (2009)
3. Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
4. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. on Software Engineering 30(6), 355–371 (2004)
5. Bernstein, P.A., Dayal, U.: An Overview of Repository Technology. In: Int. Conf. on Very Large Databses, pp. 705–712 (1994)
6. Boucher, Q., Classen, A., Heymans, P., Bourdoux, A., Demonceau, L.: Tag and Prune: A Pragmatic Approach to Software Product Line Implementation. In: IEEE Int. Conf. on Automated Software Engineering, pp. 333–336 (2010)
7. Bracha, G., Cook, W.: Mixin-Based Inheritance. In: ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications, pp. 303–311 (1990)
8. Darcs, `darcs.net`
9. Elrad, T., Filman, R.E., Bader, A.: Aspect-Oriented Programming: Introduction. Communications of the ACM 44(10), 28–32 (2001)
10. Erwig, M.: A Language for Software Variation. In: ACM SIGPLAN Conf. on Generative Programming and Component Engineering, pp. 3–12 (2010)
11. Erwig, M., Walkingshaw, E.: Program Fields for Continuous Software. In: ACM SIGSOFT Workshop on the Future of Software Engineering Research, pp. 105–108 (2010)
12. Erwig, M., Walkingshaw, E.: The Choice Calculus: A Representation for Software Variation. ACM Trans. on Software Engineering and Methodology 21(1), 6:1–6:27 (2011)
13. GNU Project. The C Preprocessor. Free Software Foundation (2009), `gcc.gnu.org/onlinedocs/cpp/`
14. Höfner, P., Khedri, R., Möller, B.: Feature Algebra. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 300–315. Springer, Heidelberg (2006)
15. Hoogle, `http://haskell.org/hoogle/`
16. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (November 1990)
17. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in Software Product Lines. In: IEEE Int. Conf. on Software Engineering, pp. 311–320 (2008)
18. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: Getting Started with Aspect. J. Communications of the ACM 44(10), 59–65 (2001)
19. Lämmel, R., Peyton Jones, S.: Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In: ACM SIGPLAN Workshop on Types in Language Design and Implementation, pp. 26–37 (2003)
20. Liebig, J., Kästner, C., Apel, S.: Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In: Int. Conf. on Aspect-Oriented Software Development, pp. 191–202 (2011)
21. MacKay, S.A.: The State of the Art in Concurrent, Distributed Configuration Management. Software Configuration Management: Selected Papers SCM-4 and SCM 5, 180–194 (1995)
22. Mens, T.: A state-of-the-art survey on software merging. IEEE Trans. on Software Engineering 28(5), 449–462 (2002)
23. Mezini, M., Ostermann, K.: Conquering Aspects with Caesar. In: Int. Conf. on Aspect-Oriented Software Development, pp. 90–99 (2003)

24. Mezini, M., Ostermann, K.: Variability Management with Feature-Oriented Programming and Aspects. ACM SIGSOFT Software Engineering Notes 29(6), 127–136 (2004)
25. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, Cambridge (1998)
26. O'Sullivan, B.: Making Sense of Revision-Control Systems. Communication of the ACM 52, 56–62 (2009)
27. Parnas, D.L.: On the Design and Development of Program Families. IEEE Trans. on Software Engineering 2(1), 1–9 (1976)
28. PLT. Racket (2011), `racket-lang.org/new-name.html`
29. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
30. Roundy, D.: Darcs: Distributed Version Management in Haskell. In: ACM SIGPLAN Workshop on Haskell, pp. 1–4 (2005)
31. Sheard, T.: A Taxonomy of Meta-Programming Systems, `web.cecs.pdx.edu/~sheard/staged.html`
32. Sheard, T.: Accomplishments and Research Challenges in Meta-programming. In: Taha, W. (ed.) SAIG 2001. LNCS, vol. 2196, pp. 2–44. Springer, Heidelberg (2001)
33. Sheard, T., Peyton Jones, S.L.: Template Metaprogramming for Haskell. In: ACM SIGPLAN Workshop on Haskell, pp. 1–16 (2002)
34. Spencer, H., Collyer, G.: #ifdef Considered Harmful, or Portability Experience With C News. In: USENIX Summer Technical Conference, pp. 185–197 (1992)
35. Stallman, R.M.: The C Preprocessor. Technical report, GNU Project, Free Software Foundation (1992)
36. Taha, W., Sheard, T.: MetaML and Multi-Stage Programming with Explicit Annotations. Theoretical Computer Science 248(1-2), 211–242 (2000)
37. Tichy, W.F.: Design, Implementation, and Evaluation of a Revision Control System. In: IEEE Int. Conf. on Software Engineering, pp. 58–67 (1982)
38. Tichy, W.F.: Tools for Software Configuration Management. In: Int. Workshop on Software Version and Configuration Control, pp. 1–20 (1988)
39. Wadler, P.: Theorems for Free! In: Conf. on Functional Programming and Computer Architecture, pp. 347–359 (1989)

Appendix: Solutions to Exercises

Exercise 1

The choice calculus expression represents all of the following definitions.

```
twice x = x+x        twice y = y+y        twice z = z+z
twice x = 2*x        twice y = 2*y        twice z = 2*z
```

This gives us six total variants. Adding another dimension with two tags for the function name sproduces the following twelve variants.

```
twice  x = x+x       twice  y = y+y       twice  z = z+z
twice  x = 2*x       twice  y = 2*y       twice  z = 2*z
double x = x+x       double y = y+y       double z = z+z
double x = 2*x       double y = 2*y       double z = 2*z
```

Exercise 2

We can simply add the definition of `thrice` using an *Impl* choice for the implementation method as follows.

dim $Par\langle x,y,z\rangle$ **in**
dim $Impl\langle plus,times\rangle$ **in**
share $v = Par\langle \mathtt{x},\mathtt{y},\mathtt{z}\rangle$ **in**
`twice` v `=` $Impl\langle v\mathtt{+}v, \mathtt{2*}v\rangle$
`thrice` v `=` $Impl\langle v\mathtt{+}v\mathtt{+}v, \mathtt{3*}v\rangle$

Exercise 3

Here we create a second *Impl* dimension with three tags, and use a corresponding choice with three alternatives in the definition of `thrice`.

dim $Par\langle x,y,z\rangle$ **in**
dim $Impl\langle plus,times\rangle$ **in**
share $v = Par\langle \mathtt{x},\mathtt{y},\mathtt{z}\rangle$ **in**
`twice` v `=` $Impl\langle v\mathtt{+}v, \mathtt{2*}v\rangle$
dim $Impl\langle plus,times,twice\rangle$ **in**
`thrice` v `=` $Impl\langle v\mathtt{+}v\mathtt{+}v, \mathtt{3*}v, v\mathtt{+twice}\ v\rangle$

Exercise 4

(a) Invalid
(b) Invalid
(c) Valid

Exercise 5

The result is 1 since the selection recursively decends into the chosen alternative with the same index, which is also the reason that it is *not* possible to select 2.

Exercise 6

When the ordering constraint is removed, we obtain an additional four entries for tuples which have a B tag in their first component.

$$\begin{aligned}[\![&\textbf{dim}\ A\langle a_1,a_2\rangle\ \textbf{in}\ A\langle 1,\textbf{dim}\ B\langle b_1,b_2\rangle\ \textbf{in}\ B\langle 2,3\rangle\rangle]\!] = \\ &\{(A.a_1,1),((A.a_2,B.b_1),2),((A.a_2,B.b_2),3),\\ &\ ((B.b_1,A.a_1),1),((B.b_1,A.a_2),2),((B.b_2,A.a_1),1),((B.b_2,A.a_2),3)\}\end{aligned}$$

We can observe that the selection of either B tag has no influence on the result when $A.a_1$ is chosen as the second tag, which reflects the fact that the B dimension is dependent on the selection of $A.a_2$.

Exercise 7

The definition of V for choices is very similar to the case for trees.

$$V_\rho(D\langle e^n\rangle) = \{(\delta^n, D\langle e'^n\rangle) \mid (\delta_1, e'_1) \in V_\rho(e_1), \ldots, (\delta_n, e'_n) \in V_\rho(e_n)\}$$

Exercise 8

The definitions can be obtained directly by partial application of the `Dim` and `Chc` constructors.

```
dimA = Dim "A" ["a1","a2"]

chcA = Chc "A"
```

Exercise 9

Expanding sharing before dimensions and choices are resolved duplicates the A dimension and will thus produce two independent decisions that result in a semantics with four variants.

$$\begin{aligned}[\![\textbf{share}\ v = (\textbf{dim}\ A\langle a_1, a_2\rangle\ \textbf{in}\ A\langle 1,2\rangle)\ \textbf{in}\ (v,v)]\!] = \\ \{((A.a_1, A.a_1),(1,1)), ((A.a_1, A.a_2),(1,2)), \\ ((A.a_2, A.a_1),(2,1)), ((A.a_2, A.a_2),(2,2))\}\end{aligned}$$

Conversely, if we expand sharing after dimensions and choices are resolved, we get only one dimension, which leads to the following semantics.

$$\begin{aligned}[\![\textbf{share}\ v = (\textbf{dim}\ A\langle a_1, a_2\rangle\ \textbf{in}\ A\langle 1,2\rangle)\ \textbf{in}\ (v,v)]\!] = \\ \{(A.a_1,(1,1)), (A.a_2,(2,2))\}\end{aligned}$$

Exercise 10

The easiest solution is to employ the `fmap` function using an anonymous finction to map an integer to a choice and apply it to `ab`.

```
> fmap (\i -> Chc "A" [Obj i, Obj (i+1)]) ab
dim A<a1,a2> in A<dim B<b1,b2> in B<A<1,2>,A<2,3>>,A<3,4>>
```

The type of the result is `V (V Int)`.

Exercise 11

The monadic instance for `V` lets us combine the variational value and the variational list using a standard monadic approach. Here we employ the `do` notation in the definition.

```
vvcons :: V a -> VList a -> VList a
vvcons vx vl = do {x <- vx; vcons x vl}
```

Exercise 12

We show the code for approach (b). In out definition we can reuse the function `vvcons` defined in exercise 11.

```
fullMenu :: Menu
fullMenu = Dim "Main" ["meat","pasta"] $
           Dim "Desert" ["yes","no"] $
               chc "Main" [Obj Steak,Obj Pasta] `vvcons`
               chc "Desert" [vsingle Cake,vempty]
```

Exercise 13

We can observe that the type of `sumL` is similar to that of `len` (it is an instance), which indicates that the function definition will have the same structure.

```
sumL :: List Int -> V Int
sumL Empty       = list 0
sumL (Cons x xs) = fmap (x+) (sumL xs)
sumL (VList vl)  = vl >>= sumL
```

The definition for `vsum` is obtained through simple lifting.

```
vsum :: VList Int -> V Int
vsum = liftV sumL
```

Exercise 14

The type of `rev` indicates that it preserves the overall structure of the list values to be processed. Therefore, the last case can be defined using `fmap`.

```
rev :: List a -> List a
rev Empty       = Empty
rev (Cons x xs) = rev xs `cat` single x
rev (VList vl)  = VList (fmap rev vl)
```

The definition for `vrev` can also use the `fmap` function.

```
vrev :: VList a -> VList a
vrev = fmap rev
```

Exercise 15

The definition for `filterL` has in principle the same type structure—at least as far the transformed lists is concerned—and follows therefore the same pattern as the definition for `rev`.

```
filterL :: (a -> Bool) -> List a -> List a
filterL p Empty       = Empty
filterL p (Cons x xs) | p x       = Cons x (filterL p xs)
                      | otherwise = filterL p xs
filterL p (VList vl)  = VList (fmap (filterL p) vl)
```

The definition for `vfilter` should be obvious given the solution for `vrev`.

```
vfilter :: (a -> Bool) -> VList a -> VList a
vfilter p = fmap (filterL p)
```

Exercise 16

The interesting cases in the definition for `zipL` are the last two where a partially applied `zipL` to one list is distributed over the elements of the respective other list using `fmap`.

```
zipL :: List a -> List b -> List (a,b)
zipL Empty       ys          = Empty
zipL xs          Empty       = Empty
zipL (Cons x xs) (Cons y ys) = Cons (x,y) (zipL xs ys)
zipL (VList vl)  ys          = VList (fmap (`zipL` ys) vl)
zipL xs          (VList vl') = VList (fmap (xs `zipL`) vl')
```

The definition for `vzip` simply injects the result of applying `zipL`, which is of type `List a`, into the type `VList a`.

```
vzip :: VList a -> VList b -> VList (a,b)
vzip vl vl' = list $ zipL (VList vl) (VList vl')
```

Exercise 17

(a) Another potential problem for hoisting can the reordering of dimensions. Consider, for example, the following variational list that contains two occurrences of an `A` dimension.

```
> dimA $ chc'A [1,2] `vvcons` (dimA $ vsingle 9)
dim A<a1,a2> in A<[1;dim A<a1,a2> in [9]],[2;dim A<a1,a2> in [9]]>
```

The semantics reveals that the decision in the second, rightmost dimension does not really have any effect on the plain results, which is not surprising since the dimension binds no choice.

```
[A.a1,A.a1]  =>  [1;9]
[A.a1,A.a2]  =>  [1;9]
[A.a2,A.a1]  =>  [2;9]
[A.a2,A.a2]  =>  [2;9]
```

Now consider the following variation of the above expression in which the rightmost `A` dimension has been lifted to the top level.

```
> dimA $ dimA $ chc'A [1,2] `vvcons` (vsingle 9)
dim A<a1,a2> in dim A<a1,a2> in A<[1;9],[2;9]>
```

This expression can be the result of hoisting the rightmost occurrence of the `A` dimension. This hoisting does not capture any free choices, but it does reorder the two dimensions, which leads to a different semantics.

```
[A.a1,A.a1]  =>  [1;9]
[A.a1,A.a2]  =>  [2;9]
[A.a2,A.a1]  =>  [1;9]
[A.a2,A.a2]  =>  [2;9]
```

(b) We don't have to check for reordering since we always find the topmost, leftmost dimension definition, which, when hoisted, cannot swap positions with other dimensions of the same name since there are non on the path from the root to the topmost, leftmost dimension definition.

Exercise 18

Instead of extracting the choice in dimension `b` directly from `a2`, in `prioritize'` we attempt to extract it first from `a1`, then from `a2`. In order to make this definition more concise, we introduce several helper functions in the body of `prioritize'`. The functions `fstAlt` and `sndAlt` describe how to reassemble the alternatives if the choice in `b` is found in the first or second alternative, respectively. The `tryAlt` function takes ones of these functions as an argument, along with the corresponding alternative, and tries to find a choice in dimension `b`. If it succeeds, it will return the reassembled expression, otherwise it will return `Nothing`. Finally, in the last line of the function, we employ the standard `mplus` function from the `MonadPlus` type class to combine the results of the two applications of `tryAlt`. This will return the first of the two applications that succeeds, or `Nothing` if neither succeeds (in which case, the fallback expression `e` will be returned from `prioritize'`).

```
prioritize' :: Data a => Dim -> Dim -> V a -> V a
prioritize' b a e = withFallback e $ do
    (dA,ae)            <- extract (dimDef a) e
    (cA,Chc _ [a1,a2]) <- extract (chcFor a) ae
    let fstAlt cB b1 b2 = [cA <@ Chc a [cB <@ b1,a2],cB <@ b2]
    let sndAlt cB b1 b2 = [cB <@ b1,cA <@ Chc a [a1,cB <@ b2]]
    let tryAlt f ai = do
            (cB,Chc _ [b1,b2]) <- extract (chcFor b) ai
            return $ dA <@ Chc b (f cB b1 b2)
    tryAlt fstAlt a1 `mplus` tryAlt sndAlt a2
```

Note that this function still makes a few assumptions, such as that the involved dimensions are binary (contain two options), and that a choice in dimension `b` is contained in only one of the two alternatives. Making this function more robust is left as an exercise for the especially thorough reader.

Exercise 19

Even though the two expressions produce the same *variants*, the ordering of the tags will be different in the *decisions* (the domain of the mapping yielded by the semantics). That is, in the semantics of `xp` the tags in the `Par` dimension appear first in each decision, while in `varyImpl (varyPar xp)` the `Impl` tags appear first.

Exercise 20

This editing function can be defined in a similar way as `extend`. First we will find the relevant dimension, then swap the alternatives of all bound choices. We again reuse `extract` and `inRange` for these tasks.

```
swapOptions :: Data a => Dim -> V a -> V a
swapOptions d e = withFallback e $ do
    (c, Dim _ [t,u] e) <- extract (dimDef d) e
    let e' = swapAlts `inRange` (chcFor d,dimDef d) $ e
    return (c <@ Dim d [u,t] e')
```

The helper function `swapAlts`, passed as the transformation function to `inRange`, exchanges the two alternatives of a binary choice.

```
swapAlts :: V a -> V a
swapAlts (Chc d [a,b]) = Chc d [b,a]
```

Note that these definitions assume that the dimension to be swapped is binary, and that all bound choices have the appropriate number of tags. A pattern-matching error will occur if these assumptions do not hold, though the solution can easily be made more robust.

Exercise 21

The `renamePar` operation is similar to the `addPar` operation defined at the beginning of 6.2. The difference is that instead of replacing every variable `x` with a choice between `x` and `y`, we want to find the first function definition in the expression, and apply our changes to this definition only. The following helper function is used by `extract` to find the first function definition in a `VHaskell` expression.

```
firstFun :: Pred Haskell
firstFun (Obj (Fun _ _ _ _)) = True
firstFun _                   = False
```

A second helper function, `renameRef`, serves as a generalized version of `addPar` that takes two variable names as parameters. The first is the name of the variable to change; the second is the new variable name.

```
renameRef :: Name -> Name -> Haskell -> Haskell
renameRef x y (Var x')
    | x == x'   = choice "Par" [Var x,Var y]
    | otherwise = Var x'
renameRef _ _ e = e
```

Finally, we are able to define `renamePar` as follows. After finding the first function definition `f`, we apply the edit described by `renameRef` to the arguments and body of `f`. We do not apply the changes to the scope of `f`, thereby isolating the change to the first function definition only.

```
renamePar :: VHaskell -> Name -> Name -> VHaskell
renamePar e x y = withFallback e $ do
    (c, Obj (Fun f as b scope)) <- extract firstFun e
    let as' = everywhere (mkT (renameRef x y)) as
    let b'  = everywhere (mkT (renameRef x y)) b
    return (c <@ Dim "Par" [x,y] (Obj (Fun f as' b' scope)))
```

This function could be generalized in several ways, such as by introducing parameters for the new dimension name or for the name of the function to apply the change to (rather than just the first one found).