AN ABSTRACT OF THE THESIS OF

Alex Grasley for the degree of Master of Science in Computer Science presented on August 27, 2018.

Title: Imperative Programming with Variational Effects

Abstract approved: _____

Eric T. Walkingshaw

Variation is a commonly encountered element of modern software systems. Recent research into variation has led to increasing interest in the development of variational programming languages and corresponding variational models of execution. Variational imperative languages pose a particular challenge due to the complex interactions between side effects and variation. The development of interpreters for variational imperative languages has produced a number of techniques to address these interactions. This thesis builds upon and formalizes these techniques by defining a formal operational semantics for a simple imperative language that supports both variation and common side effects.

We also provide an example of the successful implementation of these techniques in the form of the Resource DSL. One area in which variation is frequently encountered is in defining configurations and resource requirements for the deployment of modern software systems. To this end, we have developed the Resource DSL, a language that aids in the specification of resource requirements for highly configurable software systems.

© Copyright by Alex Grasley
August 27, 2018
All Rights Reserved

Imperative Programming with Variational Effects

by

Alex Grasley

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented August 27, 2018
Commencement June 2019

Master of Science thesis of Alex Grasley
presented on August 27, 2018.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

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

Alex Grasley, Author

ACKNOWLEDGEMENTS

First and foremost I want to thank my wife, Sara, without whom this certainly would not have been possible. Thank you for supporting me while I finished school, and especially for taking such good care of Henry and me. It's been a long road to get here and I'm glad you were there with me. I love you and Henry and I'm happy that with the conclusion of my studies we'll have more time to spend together.

Special thanks to my advisor, Eric Walkingshaw, who took a chance on me coming out of my undergraduate studies without much formal knowledge of computer science. Your kindness and support have been invaluable during this whole endeavor. I'm grateful for the opportunity that I've had to learn from and collaborate with you.

A special recognition goes to Jeff Young, a great classmate and colleague, but most importantly a true friend. I already miss our many interesting and fruitful discussions about both computer science and everything else. Having someone that I could bounce ideas off of and who brings an extreme level of passion and dedication to the field has been an enormous benefit during my time here. Similarly, I'd like to thank and recognize Mike McGirr, as a frequent participant in these same discussions.

Thanks to all the members of the Lambda group here at Oregon State. You have been wonderful colleagues and I've learned so much from all of you.

Thank you to the members of my program committee: Martin Erwig, Arash Termehchy, and Adam Branscum. In particular I've had the pleasure of being in many of Martin's excellent classes and interacting with him during our meetings as a research group. Thank you for your excellent feedback and comments on much of my work.

I've been grateful for the tireless efforts of the membership of the Coalition of Graduate Employees, who have successfully fought for many of the benefits that made tackling graduate school as a young father much more achievable.

Thank you to my new employer, Marketo, for allowing me to complete and defend my thesis and for hiring me as a fresh graduate.

Thank you to my parents, who have been constantly supportive, especially as we moved back to Oregon. I'm glad we're staying close by, especially because Henry loves seeing his grandparents so much. I'd also like to recognize my siblings and members of my extended family, in particular Claudia Webb for caring for me so much during my undergraduate studies. I love you all and am glad I have such an amazing family.

This work was supported by AFRL Contract FA8750-16-C-0044 (via Raytheon BBN Technologies) under the DARPA BRASS program.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

Chapter 1 – Introduction

Variation is an extremely common element in modern software. Variation in software can be thought of as broadly belonging to several interrelated categories: variation that is resolved statically, variation that is preserved during analysis, and variation that is present during execution. Variation that is resolved staticially is commonly found in preprocessor directives, command line options, or aspect- or feature-oriented programming [1, 13, 23]. These all share the common behavior of taking some set of features as input and producing a program or executable uniquely determined by those features as output. Variational analysis seeks to preserve this variation during program analysis. For example, the Typechef tool preserves the variation present in C preprocessor annotations in order to typecheck massively variational codebases such as the Linux kernel [12]. Variation that is preserved at runtime, or variational execution, occurs whenever a single program is run multiple times on related inputs, but may also be viewed as running a program with static variation without eliminating that variation first. Serial execution of the same program over different inputs can be viewed in the abstract as equivalent to a single execution across a value representing all inputs and producing a similar value representing all outputs.

There is considerable overlap between these kinds of variation. Tooling or static analysis that deals with statically resolved variation often encounters the same problems and employs the same techniques to solve them that are found in variational execution. Chief among these issues is that brute-force techniques for dealing with variation quickly become infeasible as the number of features grows. For a variational software system with n features, the number of unique variants produced is 2^n. Similarly, the naive approach of running a program multiple times over varying inputs requires 2^n executions for inputs with n features. This exponential growth poses what appears to be an insurmountable obstacle to efficiently handling variation in the face of a large number of distinct features.

Of course, the situation is not quite as hopeless as it might first appear. In all these types of variation there are significant elements that are shared in common between variants. In statically resolved variation this might take the form of pieces of code such as functions or classes that are shared between multiple variants. When analyzing such programs, significant portions of the analysis will be shared across many variants. In the

case of variational execution often the same computations are performed across several variants. By exploiting these commonalities what previously seemed impossible often becomes quite efficient. Software systems analysis that takes advantage of variation in this way—commonly termed *variationally-aware* analysis—has been developed across a number of areas, including typechecking, type inference, and testing [12, 6, 25]. In turn, and often in support of such variationally-aware analysis tools, systems of variational execution have been developed for a number of languages, including PHP, Java, and Javascript [20, 17, 4].

The work in this thesis was primarily motivated by our work in defining a DSL for describing resource requirements in complex modern software systems. The Resource DSL allows users to write programs that assert certain conditions of a successful resource configuration for a project. Resource requirement specifications are an ideal setting for the use of variational execution. For many modern software systems there are a whole host of configuration options available. Using a plain, non-variational model of execution a user must individually check each configuration that they wish to test. As discussed above, if a user wishes to check all configurations it quickly becomes infeasible due to the exponential explosion of the number of times they must rerun the checker. By contrast, the Resource DSL due to its fully variational model of execution is capable of exploiting the common computations shared among many different possible configurations, leading to efficiency gains. In addition to the potential efficiency gains, directly modeling variation in our language allows for easier reasoning about large configuration spaces. For instance, users can easily preconfigure a variational program to only run for certain features that they are interested in. Users can also more easily query results across options, making it potentially easier to identify patterns for particular sets of features.

Of course, the design and implementation of the Resource DSL was not without its particular challenges. One particularly interesting and difficult aspect of developing the variational interpreter for the Resource DSL was accounting for side effects in a variational setting. The Resource DSL itself is primarily interested in side effects: programs are made up primarily of either operations that somehow modify a resource environment or assertions against the current state of that environment. For this reason, the Resource DSL is implemented as an imperative language, with side-effectful statements forming the basis of programs in the language. The interaction of side effects like mutable state and throwing exceptions with variation proved to be difficult to get right. Previous work on making imperative languages variational has developed a number of techniques to handle

the interactions between side effects and variation. This thesis seeks to formalize these techniques using a big-step operational semantics for a simple imperative programming language. With the formal underpinnings thus established, we then turn to a discussion of how they are utilized in more full-featured language like our Resource DSL.

1.1 Contributions and Outline

This thesis provides two primary contributions: (1) a formal semantics for variational imperative languages and (2) a description of our design of the Resource DSL. These are of course intrinsically linked, where the work on formalizing the semantics of variational imperative languages serves as the foundation for our work on the Resource DSL.

Chapter 2 (*Background*) provides an overview of the choice calculus, which provides a formal model for reasoning about variation. It describes one way of implementing variational values in the choice calculus via formula trees. Finally, several desirable properties that are enabled by or key to the choice calculus are discussed.

Chapter 3 (*Motivation*) establishes the motivation for our work on formalizing the semantics of variational imperative languages. We first establish a baseline by describing the typical process of incorporating variation into a language without side effects. Then we show why this approach is inadequate for dealing with imperative languages with side effects. We examine how both mutable state and exceptions can pose problems in a variational setting.

Chapter 4 (*IMP*) describes the imperative language IMP [29, 21, 22]. We use IMP because its simplicity allows for the definition of the key interaction between variation and side effects without unnecessary linguistic clutter. We set out the syntax of IMP, which we have extended to include catchable exceptions, as well as a big-step operational semantics for IMP. This allows us to have a useful point of comparison for our later work on variationalizing IMP.

Chapter 5 (*Variational IMP*) describes the process of adding explicit support for variation to the IMP language. We extend the syntax with choices and then develop the necessary elements in order to define a big-step operational semantics. These elements include maintaining a variational context, a variational variable store, an error context, and special interactions with control structures. With these established we define the formal semantics of Variational IMP.

Chapter 6 (*The Resource DSL*) describes our work on developing a variational language for checking resource configurations. We describe the syntax and general function of programs in the Resource DSL. Then, we describe the implementation of the variational interpreter for the Resource DSL based on the common pattern of a monad transformer stack. This provides a real-world example of an interpreter that implements the techniques formalized in Chapter 5.

Chapter 7 (*Related Work*) provides an overview of existing approaches to variational imperative programming and compares our contributions with the prior state of the art.

Finally, Chapter 8 (*Conclusion and Future Work*) discusses possible applications of our research on variational imperative programming and the Resource DSL. We discuss possible future areas of research that build on the contributions presented here.

Chapter 2 – Background

2.1 The choice calculus

In order to formally represent choices in our programs we employ the choice calculus [26, 7]. The fundamental unit of the choice calculus is the *choice*, which is a set of values called *alternatives*. There are many data structures that can be used to model variational values, but for our purposes we utilize a data structure we refer to as a formula tree [27, 28]. Formula trees represent the set of alternatives as a tree of choices, with concrete values at the leaves of the tree. Each node of the tree is tagged with a formula from a boolean algebra consisting of boolean literals, variables, negation, conjunction, and disjunction. We call individual boolean variables the *dimensions* of the choices, while the boolean expressions used as tags we call *conditions*. Notationally we represent these choice trees via angle brackets. For example, a choice in dimension A between the values 1 and 2 is written $A\langle 1,2\rangle$.

Each dimension can be viewed as coding for the presence or absence of a particular feature that we wish to vary. To continue the above example, $A\langle 1,2\rangle$ represents a variational value that is 1 when feature A is present and 2 otherwise. Conditions are therefore analogous to conditional expressions built out of these features. For example, the choice $(A \wedge \neg B)\langle 1,2\rangle$ is a value that represents 1 when feature A is present and B is absent, and 2 otherwise.

The *selection* operation on a choice can be used to eliminate conditions. We define a function *sel* that takes a *selector* and a variational value and eliminates conditions based off of the selector. For formula trees selectors are also booleans formulas, and can be seen as specifying a condition that is assumed to hold true. Therefore, for a selector and a choice with a given condition, we eliminate the choice and replace it with the recursively selected left alternative if the selector logically implies the truth of the condition. Similarly, if the selector logically implies the falsehood of the condition, we eliminate the choice and replace it with the recursively selected right alternative. If neither implication is valid, the choice remains untouched with both alternatives left intact.

It is helpful to see this selection operation in action. The operation $sel(A, A\langle 1,2\rangle)$, can be seen as encoding the presence of feature A, which will therefore produce the value 1

as a result. Similarly, $sel(\neg A, A\langle 1,2\rangle)$ can be seen as encoding the absence of feature A and will produce the value 2. $sel(B, A\langle 1,2\rangle)$ will result in an unchanged variational value $A\langle 1,2\rangle$ because the selector B does not logically imply either the truth or falsehood of dimension A.

Selection is *synchronized* for a given condition, meaning that two choices in the same value or expression with logically equivalent conditions must always select the same alternative. For example, in the expression $A\langle 1,2\rangle + A\langle 3,4\rangle$, the only possible selections are $1+3$ and $2+4$, while $1+4$ and $2+3$ can never occur due to synchronization.

We say that a variational value is *configured* when all choices have been eliminated, yielding a plain value without choices. We call these resulting plain values *variants*. We define a function *conf* which takes a *configuration* and a variational value and produces a variant. Under the view of dimensions as representing distinct features, a configuration for a formula tree gives a boolean assignment for each dimension of the variational value being configured. A simple way to do this is to define a configuration as a set of the features that are present, with all other features assumed to be absent. In other words, configurations are a set of variables that should evaluate to true, with all other variables evaluating to false. Configuring a formula tree is then simply evaluating each condition given the variable assignment. If the condition evaluates to true, then the left variant is chosen, otherwise the right variant is chosen.

Using boolean formulas as our tags provides several advantages. For example, we can simplify trees like $B\langle 1, A\langle 1,2\rangle\rangle$ to the more compact form $(B \vee A)\langle 1,2\rangle$. This is just one of a number of optimizations that are made possible by representing tags as boolean formulas [28, 9]. Formula choices are also more convenient for maintaining a variational context during execution of a program, a fact that will become more relevant when we present our work on variational imperative programming later in Chapter 5. Despite these advantages, formulas incur some cost by introducing boolean satisfiability problems into many common operations, which is an NP-complete problem. However, thanks to improvements in the efficiency of modern SAT solvers, often the worst-case NP-complete performance of these satisfiability problems can be avoided. Several studies have shown that the types of boolean formulas typically encountered in variational software are efficiently solved by modern SAT solvers [16, 18].

2.2 Implementation of Formula Trees

When integrating formula trees into an existing non-variational language we can choose to either embed choices directly into the abstract syntax of the host language or utilize a type-generic implementation. In this work we employ both strategies, opting to embed choices directly when dealing with programs, while using the type-generic implementation for the values produced by these programs. Here we present the type-generic implementation, although the basic principles explored here apply equally to embedded representations.

Generic formula trees can be represented by the following Haskell datatypes:

```
type Dim = String
data Cond =
   Lit Bool
  | Ref Dim
  | Not Cond
  | Cond :&&: Cond
  | Cond :||: Cond

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

The datatype `Cond` represents expressions in our boolean algebra, with dimension names as strings. The constructor `One` creates the leaves of a formula tree, while `Chc` creates the nodes.

This type-generic representation allows us to easily define some useful algebraic properties of formula trees via typeclasses. Specifically, formula trees readily support implementation of the Functor, Applicative, and Monad typeclasses:

```
instance Functor V where
  fmap f (One a) = One (f a)
  fmap f (Chc d l r) = Chc d (fmap f l) (fmap f r)

instance Applicative V where
  pure = One
  (One f) <*> v = fmap f v
  (Chc d l r) <*> v = Chc d (l <*> v) (r <*> v)

instance Monad V where
```

```
return = One
(One v) >>= f = f v
(Chc d l r) >>= f = Chc d (l >>= f) (r >>= f)
```

Astute readers will note that because formula trees are merely a special class of binary tree with labeled nodes they share identical instances. The usefulness of defining these instances will be made apparent further on in this work during the discussion of pure, side-effect free variational languages.

2.3 Properties of choices

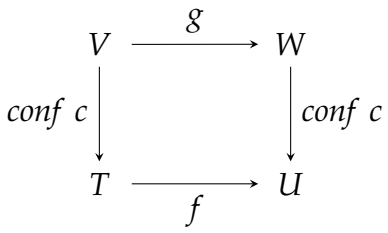
The choice calculus enables several desirable properties for variational programs. The first property that is enabled through the choice calculus is *sharing*. A naive approach to executing a variational program is simply to configure the variational program in order to produce each variant and run them all sequentially. For a program with n dimensions this results in 2^n variants that must be executed in the worst case. Crucially, the naive approach must recompute any common elements shared between variants. Embedding choices in our program, combined with a variability-aware model of execution allows us to exploit this inherent sharing by only computing shared components once, greatly improving the efficiency of executing a variational program.

This sharing can be accomplished by pushing down choices as deeply as possible in variational values. For example, consider the variational expression $A\langle 3*2+4, 3*2+8\rangle$. If we push the choice down as far as possible, we get $3*2+A\langle 4,8\rangle$. This identifies the subexpression $3*2$ as a shared element between variants A and $\neg A$. A variational interpreter of this language could then safely only compute that subexpression once.

This sharing is enabled but not required by the choice calculus. It is up to the implementor to ensure that the maximum degree of sharing is achieved. Nevertheless, there are a number of strategies for systematically simplifying variational values [26]. The use of formula trees permits additional simplification operations [9, 28].

Another property of choices and variation that we are interested in is *variation-preservation*. In order to take advantage of the benefits of sharing, we must develop a variability-aware model of execution that produces variational values as its result. In order to determine whether our variational results are correct we need a way to relate them back to the individual variants they represent. Variation-preservation assures us that each variant is faithfully modeled by the overall variational computation.

Given a function $f : T \to U$ and its variational counterpart $g : V \to W$, we say that the function g is variation-preserving if for some configuration c the following equality holds: $conf\ c \circ g = f \circ conf\ c$ [9]. Put simply, configuring the result of passing a variational value to a variational function should be the same as pre-configuring the input and the function itself. This can be visualized as the following commuting diagram:



Chapter 3 – Motivation

In order to take advantage of the benefits of sharing provided by choices, we must develop a variational model of execution for a given language. We begin by showing that defining variational execution in a pure, side-effect free evaluation context is a fairly straightforward exercise before turning to the much more difficult problem of variational execution with side effects.

3.1 Pure variational execution

If our target language has no side effects, then our task is comparatively easy. We begin by considering a simple arithmetic expression language defined by the following Haskell datatype:

```
data Arith =
   N Int
 | Add Arith Arith
 | Mul Arith Arith
```

In the non-variational setting we can easily define an interpreter for this language which produces values of type `Int`:

```
eval :: Arith → Int
eval (N i) = i
eval (Add l r) = eval l + eval r
eval (Mul l r) = eval l * eval r
```

The process of "variationalizing" this small example is quite straightforward. First, we extend the syntax by adding a constructor to support choices in our language:

```
data Arith =
    N Int
  | Add Arith Arith
  | Mul Arith Arith
  | AChc Cond Arith Arith
```

Next we modify our interpreter. Instead of returning plain values of type `Int`, we now return *variational* values of type `V Int`. We then take the cases from the non-variational interpreter and make them variational by situating them within the applicative functor instance for `V` we defined in Section 2.2. Finally, we add a case that handles our new `AChc` constructor and we have completed the conversion of our interpreter:

```
eval :: Arith → V Int
eval (N i) = pure i
eval (Add l r) = (+) <$> eval l <*> eval r
eval (Mul l r) = (*) <$> eval l <*> eval r
eval (AChc d l r) = Chc d (eval l) (eval r)
```

This basic pattern of variationalizing a language by re-situating the plain interpreter within the applicative functor for choices works well for any language that is pure and side-effect free. The only area of concern is associated with the use of data structures in the interpretation of the language, which often benefit from being replaced with custom variational data structures that provide greater sharing and efficiency. Recent work has explored the benefits of custom variational data structures for maps [27], linked lists [24], and stacks [19]. As such, exercising proper care in the selection and integration of variational data structures into a variational interpreter resolves this issue.

Nevertheless, even with the support of more efficient variational data structures, our basic pattern of converting a plain interpreter to a variational one encounters significant difficulties in the presence of a language with side effects. Specifically, we demonstrate that the presence of side-effectful statements, control flow structures, mutable state, and exceptions commonly encountered in imperative programming languages requires more than the approach based on applicative functors as we have outlined here.

3.2 Mutable State

To demonstrate why imperative programming with side effects requires us to rethink how we variationalize a language and its execution, we turn to a small example in a

simple imperative language that supports basic control structures and mutable variable references:

$x := 0$
$y :=$ getSecret()
if y is true **then**
  $x := 1$
end if

In a non-variational setting, we can reasonably expect the final state of the variables in this example to be either $\{x \Rightarrow 0,\ y \Rightarrow \textbf{false}\}$ or $\{x \Rightarrow 1,\ y \Rightarrow \textbf{true}\}$ depending on the value that getSecret() returns. Now we consider the execution of the above example in a variational setting. Suppose that the call to `getSecret` produces the variational value $A\langle\textbf{true},\textbf{false}\rangle$. We can view this as simply combining the two possible non-variational execution paths described above into a single execution varying in dimension A. As such, we would expect the final variable state in this example to be $\{x \Rightarrow A\langle 1,0\rangle,\ y \Rightarrow A\langle\textbf{true},\textbf{false}\rangle\}$ by combining the final states above into choices over dimension A. We can verify this to be the proper variational result using the variation-preserving property specified in Section 2.3.

Of course, to arrive at the above result, we simply worked backwards from the non-variational results. This naive approach cannot scale to domains with a high number of dimensions because of the exponential explosion of variants. In order to take advantage of the benefits provided by sharing, we must instead develop a model of variational execution that supports mutable state. As we will see, the applicative functor instance for formula trees is no longer sufficient by itself, necessitating a different approach to how we variationalize programs that use mutable state.

The central problem becomes obvious when we compare the required semantic domains for evaluating statements in the plain and variational versions of the example. The semantic domain of plain statements is a function from state to state, where state is a mapping from variable names to values. As such, we might then assume that the corresponding semantic domain for variational statements is also from state to state, only in this case we use a variational map. But when we attempt to evaluate our example under this semantic domain we soon reach an issue that causes us to rethink our definition.

The problem comes in line 4, where we assign the variable x in a conditional block. In the previous line, we check if the value of y is **true**. When y is a variational value, allowing it to be both **true** or **false** in different variants, we only assign x in those variants

in which y is **true**. But if the state in our semantic domain is only a map, we have no way of passing the necessary information to the assignment that it must only occur in particular variants and not in others. Our only choice is to assign the value of x for all variants, which would clearly be incorrect, as the assignment should only be carried out in the variant where y is **true**. We must therefore conclude that we need to carry this variational context information around in our state during execution in order to handle cases where assignment occurs in a branch that is conditionally executed. Similar logic can be used to see that this context is also necessary to handle variational lookup of variable values.

It is now clear why our previous approach of relying solely on the applicative functor instance for choices is inadequate. The method provides no way of maintaining and manipulating variational context during execution. This nicely demonstrates the problems we encounter when trying to introduce choices to languages with side effects.

3.3 Exceptions

Exceptions are another common type of effect that proves challenging in a variational setting. Consider the following program:

$y :=$ getSecret()
if y is true **then**
  throw e
end if
$x :=$ expensiveFn()

In a non-variational context, the behavior of this program should be clear. If the variable y evaluates to **true**, then an error is thrown with value e. At this point evaluation should stop, meaning that the variable x is never assigned in the final statement, avoiding the costly computation of expensiveFn. This effect of halting execution and returning an error value is the essence of exceptions in non-variational settings.

Now we consider the behavior of the same program in a variational context. Suppose that the variable y evaluates to the variational value $A\langle \mathbf{true}, \mathbf{false} \rangle$. This means that in the left alternative of A we would evaluate the body of the **if** statement, while ignoring it otherwise. Therefore, in the left alternative of A we throw an exception, but otherwise we continue on to the final statement.

Clearly, maintaining the same behavior from the non-variational setting in the variational setting violates variation-preservation. If we halt execution whenever an exception is thrown in any variant, then we also stop the evaluation of variants that never encountered an error, as in our example for the case $\neg A$. Variation-preservation dictates that variants that never encountered an exception should complete their execution uninhibited.

We also don't want to simply continue evaluation in every variant regardless of whether or not we have encountered an error. Suppose that calling `expensiveFn` normally would evaluate to the variational value $A\langle 1,2\rangle$, but at considerable computational cost in both alternatives. Because we know that the left alternative of A will ultimately evaluate to the thrown exception e, we would like to avoid the cost involved in computing the value 1 that we will just throw away later, mirroring how throwing an exception short-circuits evaluation in the non-variational setting.

Another problem concerns how to keep track of which variants are in error states and what the error values are. If we want to avoid the cost of pointless evaluation in variants that are in an error state, we must have some efficient way of determining when evaluation is about to enter such a variant. In the non-variational context we have no need to store and remember error values during evaluation because we simply return the error value immediately when it is thrown. In the variational context, we must now store these values while we continue to evaluate variants that are not in an error state.

3.4 Towards a formal semantics

At this point, it should be apparent that introducing choices into a side-effectful imperative language is anything but straightforward. These issues are not new, and have been encountered in previous work on variationalizing imperative languages [11, 20, 17]. Along the way a number of techniques have been developed to address these issues. Our work synthesizes these techniques into a single big-step operational semantics for a simple imperative language. Along the way we also develop novel techniques of our own, particularly when it comes to catchable exceptions, which to our knowledge have not been adequately explored in a variational setting.

It is our hope that a formal semantics for common side effects encountered in variational imperative programming will serve as the base for future work in this area. By collecting and formalizing the techniques shared among variational imperative programming languages, future researchers should be able to (1) more easily define and

create variational languages by extending our semantics to more complex languages and (2) more easily formalize desriable properties in these languages. In Chapter 6 we demonstrate one successful application of these techniques in the form of the Resource DSL.

Chapter 4 – IMP

The IMP language is a simple but Turing-complete imperative programming language [29, 21, 22]. In our case, IMP's relative simplicity allows for straightforward extension to support exceptions as well as variational execution. IMP also lacks elements such as variable scoping, procedure calls or declarations, or the need for a type system. This permits us to focus more completely on the work of formalizing the semantics of variational imperative programming without unnecessary complication. Additionally, a less complex semantics should allow our work to serve as a base that can be readily extended to more full-featured languages.

We begin by establishing the syntax and semantics of plain IMP before turning to variational IMP. The syntax of plain IMP is defined in Figure 4.1. At its base IMP defines simple arithmetic expressions made up of integer literals, variable references, and addition. IMP also supports boolean expressions made up of literals, negation, conjunction, and less-than comparison of arithmetic expressions. Statements include variable assignment, sequencing, and control structures. Only arithmetic expressions are allowed in variable assignments, and only boolean expressions are allowed in the conditions of control structures, thus obviating the need for a type system. Finally, we have extended the definition of the language to include basic support for catchable exceptions as one of the effects that we wish to formally define for variational execution. **throw** statements take an arithmetic expression and produce an error with the value the expression evaluates to. **try ... catch** statements allow for the handling of any thrown error values in their main block via a handler block.

We define a formal big-step semantics for this non-variational version of IMP, which we can then use as a point of reference for our variational semantics. Figure 4.2 defines the big-step semantics of arithmetic and boolean expressions. Expressions are evaluated in the context of the current variable store and are otherwise straightforward in their definition. Figure 4.3 defines the big-step semantics of IMP statements. Statements have two possible side effects: updating the variable store or modifying the current error context. In practice the variable store can be simply implemented as a map data structure, while the error context is an optional integer value indicating the current error code.

(Arithmetic expressions)

a	::=	n	*Integer literal*
	|	x	*Variable reference*
	|	$a + a$	*Addition*

(Boolean literals)

b	::=	**true**	
	|	**false**	

(Boolean expressions)

e	::=	b	*Boolean literal*
	|	**not** e	*Negation*
	|	e **and** e	*Conjunction*
	|	$a < a$	*Less*

(Statements)

s	::=	**skip**	*Noop*
	|	$x := a$	*Assignment*
	|	s ; s	*Sequencing*
	|	**if** e **then** s **else** s	*Conditional*
	|	**while** e **do** s	*Looping*
	|	**throw** a	*Throw error*
	|	**try** s **catch** x **with** s	*Catch error*

Figure 4.1: Syntax of IMP with exceptions

(Store)

S	::=	*empty*	*Empty store*
	|	$ins((x,n),S)$	*Insert value n as variable x in S*

LOOKUP-HERE
$$\frac{}{lookup(ins((x,n),S),x,n)}$$

LOOKUP-THERE
$$\frac{lookup(S,x,n)}{lookup(ins((x',n'),S),x,n)}$$

A-NUM
$$\frac{}{(S,n) \Downarrow_A n}$$

A-REF
$$\frac{lookup(S,x,n)}{(S,x) \Downarrow_A n}$$

A-ADD
$$\frac{(S,a) \Downarrow_A n \qquad (S,a') \Downarrow_A n'}{(S,a+a') \Downarrow_A n+n'}$$

B-BOOL
$$\frac{}{(S,b) \Downarrow_B b}$$

B-NOT
$$\frac{(S,e) \Downarrow_B b}{(S,\textbf{not}\ e) \Downarrow_B \neg b}$$

B-AND
$$\frac{(S,e) \Downarrow_B b \qquad (S,e') \Downarrow_B b'}{(S,e\ \textbf{and}\ e') \Downarrow_B b \wedge b'}$$

B-LESS
$$\frac{(S,a) \Downarrow_A n \qquad (S,a') \Downarrow_A n'}{(S,a<a') \Downarrow_B n<n'}$$

Figure 4.2: Big-step semantics of IMP expressions

(Store)

S	::=	*empty*	*Empty store*
	|	$ins((x,n),S)$	*Insert value n as variable x in S*

(Error context)

E	::=	$\bullet$	*No error*
	|	n	*Error with value n*

S-Err
$$\frac{}{(n,S,s) \Downarrow_S (n,S)}$$

S-Skip
$$\frac{}{(E,S,\textbf{skip}) \Downarrow_S (E,S)}$$

S-Assn
$$\frac{(S,a) \Downarrow_A n}{(\bullet,S,x := a) \Downarrow_S (\bullet,ins((x,n),S))}$$

S-Seq
$$\frac{(E,S,s) \Downarrow_S (E',S') \qquad (E',S',s') \Downarrow_S (E'',S'')}{(E,S,s\ ;\ s') \Downarrow_S (E'',S'')}$$

S-IfT
$$\frac{(S,e) \Downarrow_B \textbf{true} \qquad (\bullet,S,s) \Downarrow_S (E,S')}{(\bullet,S,\textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s') \Downarrow_S (E,S')}$$

S-IfF
$$\frac{(S,e) \Downarrow_B \textbf{false} \qquad (\bullet,S,s') \Downarrow_S (E,S')}{(\bullet,S,\textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s') \Downarrow_S (E,S')}$$

S-WhileT
$$\frac{(S,e) \Downarrow_B \textbf{true} \qquad (\bullet,S,s) \Downarrow_S (E,S') \qquad (E,S',\textbf{while}\ e\ \textbf{do}\ s) \Downarrow_S (E',S'')}{(\bullet,S,\textbf{while}\ e\ \textbf{do}\ s) \Downarrow_S (E',S'')}$$

S-WhileF
$$\frac{(S,e) \Downarrow_B \textbf{false}}{(E,S,\textbf{while}\ e\ \textbf{do}\ s) \Downarrow_S (E,S)}$$

S-Throw
$$\frac{(S,a) \Downarrow_A n}{(\bullet,S,\textbf{throw}\ a) \Downarrow_S (n,S)}$$

S-Try
$$\frac{(\bullet,S,s) \Downarrow_S (\bullet,S')}{(\bullet,S,\textbf{try}\ s\ \textbf{catch}\ x\ \textbf{with}\ s') \Downarrow_S (\bullet,S')}$$

S-Catch
$$\frac{(\bullet,S,s) \Downarrow_S (n,S') \qquad (\bullet,ins((x,n),S'),s') \Downarrow_S (E,S'')}{(\bullet,S,\textbf{try}\ s\ \textbf{catch}\ x\ \textbf{with}\ s') \Downarrow_S (E,S'')}$$

Figure 4.3: Big-step semantics of IMP statements

Chapter 5 – Variational IMP

Having established the syntax and semantics of plain IMP, we now turn to the definition of Variational IMP (or VIMP for short). Figure 5.1 defines the syntax of VIMP. The syntax remains largely the same, with the only difference being the addition of choices into the expressions and statements of the language. We opted for embedding choices directly into all of IMP's language constructs rather than using a generic formula tree implementation. While the end result should not depend on whether a generic or embedded implementation of choices is used, we felt that directly embedding choices led to cleaner semantic definitions.

We also present the big-step operational semantics for VIMP expressions and statements. In the following sections we will discuss specific elements of these definitions. We begin by defining variational operations, such as configuration and selection as explained in Chapter 2. We then examine the need for maintaining a variational context during execution, especially in its relation to the variable store. After, we turn to a discussion of catchable exceptions. Next we discuss interacting with control structures. Finally, we present the formal big-step operational semantics for VIMP expressions and statements.

5.1 Variational Operations

In order to successfully relate variational IMP back to its non-variational counterpart we must define configuration and selection for VIMP. We assume solving for satisfiability as a primitive operation; in practice this can be implemented by offloading these calls to a SAT solver.

Figure 5.2 shows the implementation of the selection and configuration options for generic formula trees as defined in Section 2.2, using Haskell as a metalanguage. We show the implementation for generic formula trees here, with the understanding that it is trivially extended to VIMP's use of choices directly embedded into the syntax of the language. We begin by defining some useful utility functions when dealing with satisfiability, particularly the ability to express logical implication.

The selection function *sel* takes a condition and a variational value. If the variational value is a choice, *sel* checks to see if the supplied condition implies that the condition of

(Boolean literals)			
b	$::=$	**true**	
	$\mid$	**false**	
(Conditions)			
C	$::=$	b	*Boolean literal*
	$\mid$	D	*Dimension name*
	$\mid$	$\neg C$	*Negation*
	$\mid$	$C \vee C$	*Disjunction*
	$\mid$	$C \wedge C$	*Conjunction*
(Arithmetic expressions)			
a	$::=$	n	*Integer literal*
	$\mid$	x	*Variable reference*
	$\mid$	$a + a$	*Addition*
	$\mid$	$C\langle a, a\rangle$	*Choice*
(Boolean expressions)			
e	$::=$	b	*Boolean literal*
	$\mid$	**not** e	*Negation*
	$\mid$	e **and** e	*Conjunction*
	$\mid$	$a < a$	*Less*
	$\mid$	$C\langle e, e\rangle$	*Choice*
(Statements)			
s	$::=$	**skip**	*Noop*
	$\mid$	$x := a$	*Assignment*
	$\mid$	s ; s	*Sequencing*
	$\mid$	**if** e **then** s **else** s	*Conditional*
	$\mid$	**while** e **do** s	*Looping*
	$\mid$	**throw** a	*Throw error*
	$\mid$	**try** s **catch** x **with** s	*Catch error*
	$\mid$	$C\langle s, s\rangle$	*Choice*

Figure 5.1: Syntax of Variational IMP

```
unsat = not ∘ sat
taut = unsat ∘ Not
x ⇒ y = taut ((Not x) :||: y)

sel :: Cond → V a → V a
sel _ (One x) = One x
sel c (Chc c′ l r)
  | c ⇒ c′ = sel c l
  | c ⇒ (Not c′) = sel c r
  | otherwise = Chc c′ (sel c l) (sel c r)

evalCond :: [Dim] → Cond → Bool
evalCond ds (Lit b) = b
evalCond ds (Ref x) = x `elem` ds
evalCond ds (Not c) = not (evalCond c)
evalCond ds (And c c′) = evalCond c && evalCond c′
evalCond ds (Or c c′) = evalCond c || evalCond c′

conf :: [Dim] → V a → a
conf _ (One x) = x
conf ds (Chc c l r) = if evalCond ds c then conf ds l else conf ds r
```

Figure 5.2: Variational operations for VIMP

the choice is either true or false, returning the recursively selected left or right alternative, respectively. If the provided condition does not necessarily imply either the truth or falsehood of the choice's condition, then both alternatives are selected on and maintained as a choice with the original condition.

The configuration function *conf* takes a list representing a set of dimensions and a variational value as its arguments. These dimensions can be viewed as features that are "turned on" and should therefore evaluate to `True`, with all other dimensions evaluating to `False`. Put another way, the configuration function requires a variable assignment for all of the dimensions in a configuration space in order to evaluate all conditions. With this assignment, configuration is a straightforward task of evaluating each choice's condition and selecting either the left or right alternative depending on if it evaluates to `True` or `False`, respectively.

We utilize the selection function when performing variable lookup using a variational store, as explained in the following section. The configuration function is used to relate the result of variational execution back to individual variants, a process which we detail in Section 5.5.

5.2 Variational Context and Variational Store

When describing the issues involved in the interaction between mutable state and variation in Section 3.2, we noted the need for (1) maintaining a variational context (a boolean formula or condition) during execution and (2) using this variational context when interacting with the variable store. This is a solution first discovered by [11] in their work on a variational model of execution for the WHILE language.

Maintaining the variational context is fairly straightforward. During execution of a variational program, whenever a choice is encountered, the execution splits into two separate paths corresponding to each alternative. First, the interpreter must check whether a particular alternative is even possible given the current variational context. For example, if the current context is the condition $\neg A$, then for a given choice $A\langle l, r\rangle$, there is no need to evaluate the left alternative, as the current variational context implies that configuration is impossible and can be automatically ruled out. Therefore, for a given variational context C and a choice with condition C', the interpreter checks whether $C \wedge C'$ and $C \wedge \neg C'$ are satisfiable, only then proceeding to execute the left and right alternatives, respectively.

In the case that a context and a choice's condition are satisfiable, the execution then proceeds by extending the current variational context for each alternative. For variational context C and a choice with condition C', the left alternative should be evaluated with the context $C \wedge C'$ and the right alternative with $C \wedge \neg C'$.

With the variational context established, we can move on to the definition of the variational variable store. A non-variational store is a data structure that supports a lookup operation of type `lookup :: Var → Store → Value`, and update of type `update :: Var → Value → Store → Store`. A variational store is a data structure that supports a lookup operation of type `vlookup :: Cond → Var → VStore → V Value` and update of type `vupdate :: Cond → Var → V Value → VStore → VStore`. In essence, a variational store takes the current variational context as an additional argument to all of its operations, which now operate on variational values rather than plain values. Taking the current variational context as an argument to all operations is necessary in order to avoid situations like those described in Section 3.2, where a block is conditionally evaluated only in some variants but not others. This information can now be safely carried by the variational context and passed to the variational store when required.

In Figure 5.3 we provide a sample implementation of a variational store based on the variational map data structure [11, 27]. Variational maps are an efficient map data structure for storing variational values and are easily built using an existing non-variational map. The basic idea is that for a non-variational map of values of type `v`, a variational map uses the same underlying data structure but with values of type `V (Maybe v)`. The variational values in the map must be of type `Maybe v` in order to account for cases where a value is assigned in one variant but not in others. Our sample variational store is built on top of Haskell's `Data.Map` library, but in theory any map data structure should suffice.

In `vlookup` we throw a runtime exception when a variable is undefined. Our semantics does not define the behavior of trying to access an undefined variable, but such cases are easily detected and prevented by a type system. Variational type systems have been studied extensively, and readers seeking static guarantees that undefined variable access is impossible are referred to those works [12, 5, 6].

The implementation of `vupdate` that we present here, while adequate, has the potential for additional optimization. Currently, `vupdate` simply creates what amounts to a labeled list of variational values, where the variational context in which a variable was assigned is used as the label. This means that in order to lookup a variational value in a context stored at the end of such a list of length n, we must first perform n satisfiability checks.

```
type VOpt v = V (Maybe v)
type VStore v = Map Var (VOpt v)

assertExists :: VOpt v → V v
assertExists (One Nothing) = error "trying to access an undefined variable"
assertExists (One (Just v)) = One v
assertExists (Chc c l r) = Chc c (assertExists l) (assertExists r)

vlookup :: Cond → Var → VStore v → V v
vlookup c k vst = assertExists $ sel c (lookup' k vst)
  where
    lookup' k m = case lookup k m of
      Just v → v
      Nothing → error "trying to access an undefined variable"

vupdate :: Cond → Var → V v → VStore v → VStore v
vupdate c k v vst = case lookup k vst of
   Nothing → insert k (Chc c (fmap Just v) (One Nothing))
   Just old → insert k (Chc c (fmap Just v) old)
```

Figure 5.3: Variational map operations

Because satisfiability checks have the potential to be expensive operations, we would like to reduce the need for them as much as possible. BDD solvers are built to solve these sorts of problems, but integrating them with variational values is a non-trivial task and is left as a subject for future research. Short of integrating a BDD solver, there are other optimization or simplification operations that could be performed [26, 9], but which to perform and how frequently is also a subject beyond the scope of the current work.

In VIMP, each evaluation step requires the current variational context. In VIMP's expressions, the variational context is passed to the lookup operation for any variable reference. Similarly, in VIMP's statements the variable assignment operation takes the current variational context as an argument.

5.3 Variational Exceptions

Catchable exceptions are the other type of side effect that we want VIMP to support. In Section 3.3 we outlined the primary requirement for exceptions in a variational setting: when an exception is thrown in a particular variant, all execution in that variant should stop until the error is (optionally) caught, but other variants should continue their execution unaffected. In order to achieve this, we utilize the existing framework we have built for tracking the current variational context but with some modifications to account for the presence of exceptions.

If we wish to stop execution in a variant when an exception is thrown, then the **throw** operation needs to be capable of returning its current variational context along with the specific error code of the exception. We have already incorporated a variational context into the execution of VIMP, so it is a trivial extension to have **throw** return this value. The variational context framework also easily supports preventing the continued execution of variants that are in an error state. We simply need to incorporate an error condition into our satisfiability checks such that the variational context is unsatisfiable for variants in an error state. Because an exception may have been thrown in the current variant, we extend our satisfiability checks to occur before executing each statement. This means that before beginning execution on a new statemtent we now check for (1) whether the configuration for the current variant is even possible and (2) whether the variant is currently in error and should not be executed. The error condition can be easily derived by taking the disjunction of the variational contexts returned by all **throw** operations. To make entering an error context impossible we take the conjunction of the current variational context

```
type Err = (Cond, V Int)
type Stack a = [a]

ctx :: Cond → Stack [Err] → Cond
ctx c st
   | [] ← concat st = c
   | errs ← concat st = c :&&: f errs
  where
   f [(c,_)] = c
   f ((c,_):es) = c :||: f es

catchCtx :: Cond → [Err] → Cond
catchCtx c errs = c :&&: foldl f (Lit False) errs
  where
   f c (c′,_) = c :||: c′

extract :: [Err] → V Int
extract [] = error "no values to extract"
extract [(_,v)] = v
extract ((c,v):es) = Chc c v (extract es)
```

Figure 5.4: Variational error helper functions

and the negation of the error context and check it for satisfiability before entering a new variant.

Supporting catchable exceptions gives us several additional challenges. We need to be able to "roll back" the error context to a previous state. This suggests that during execution the error context and corresponding values should be kept in a stack. When we enter a new **try** block we push an empty error frame onto the stack. All error values and their contexts thrown in the execution of the **try** block get added to the top of the stack. We pop the top frame of the stack when we reach the corresponding **catch** block. If we reach the **catch** handler with a non-empty top stack frame, we then need to (1) extract the variational value of all of the error codes thrown during the try block and assign it to the provided variable name and (2) execute the catch block from within the combined variational context that those exceptions were thrown in.

With these requirements in mind, we define the error context carried during execution to be a stack of lists of pairs of error conditions and error codes. We also define the helper functions *ctx*, *catchCtx* and *extract*. *ctx* retrieves the condition for the satisfiability check

before entering a new variant by combining the error context and the variational context. *catchCtx* determines the correct variational context for a **catch** block based on the top frame of the error stack. *extract* creates a variational value from all of the error codes thrown in a try block, again based on the top stack frame. Haskell implementations of these functions are presented in Figure 5.4.

While we do not include a mechanism for it in our formal semantics, it is possible to implement an additional optimization for variational languages with exceptions. Satisfiability checks are expensive, and so performing a satisfiability check before each statement is executed is undesirable. In order to avoid this, variational interpreters can add a flag to signal when a block is "dead" and can safely stop execution. A call to **throw** sets the flag to indicate that further execution can be safely halted. The flag is removed when it occurs within a **try** block once execution reaches the corresponding **catch** block. For all other instances the flag indicates that the execution of a statement can safely be skipped.

This flag value has an interesting interaction with choices. We only wish for a choice statement to set its flag when all variants can safely be ignored. If there is a single variant that is still active, then a choice statement should not set its flag, in this way suppressing the flags of its children. When the flag is set from inside of a choice, there is a simple procedure to check if the choice should propagate or suppress the flag. If both alternatives are either unsatisfiable or have the flag set, then the parent choice should set its flag, indicating that execution is halted for all possible variants. In the Resource DSL (Section 6.2.3), we implement this system via an error monad, with errors serving as the flags and inserting catches for choices in order to inspect whether the parent should propagate the errors or suppress them.

5.4 Control Structures

Control structures are another aspect of VIMP where the interaction with variation plays a crucial role. IMP's control structures—**if . . . then . . . else** and **while . . . do**—conditionally execute statements based on the evaluation of a boolean expression. In VIMP, this presents two issues. First, any variation in the condition of a control structure necessarily affects the conditionally executed block. Second, a formula tree representing a variational boolean value may contain multiple distinct **true** or **false** values. As with most other operations in VIMP, such as addition or comparison, we simply lift them into the applicative functor

```
whenTrue :: V Bool → Cond
whenTrue (One True) = Lit True
whenTrue (One False) = Lit False
whenTrue (Chc c l r) = (c :&&: whenTrue l) :||: (Not c :&&: whenTrue r)

whenFalse = Not ∘ whenTrue
```

Figure 5.5: Control structure helper functions

for formula trees. However, when interacting with control structures this approach causes significant inefficiencies. The applicative functor simply unwraps each leaf value in the tree and continues execution. This might lead us to reevaluate the same conditional block repeatedly, only with different variational contexts. Any sharing between executions of this conditional block would be lost.

The first issue is easily resolved by adding the context that caused the boolean value to be **true** (or in the case of **else** blocks, **false**) to the variational context for the conditionally executed statements. For example, in the statement **if** $A\langle\textbf{true},\textbf{false}\rangle$ **then** s **else** s', the statement s would be evaluated with variational context A, while s' would be evaluated under $\neg A$.

For the second issue, we need a way to extract a condition that encompasses all variants where a variational boolean value is **true** or, in the case of **else** blocks, **false**. With these conditions in hand, we can simply evaluate a conditional block under the current variational context extended by the proper condition. For this, we define two functions, *whenTrue* and *whenFalse* that return these conditions for a given variational boolean. This technique was also originally discovered in [11]. Figure 5.5 presents sample implementations of these functions.

5.5 Big-Step Semantics of VIMP

Finally we are prepared to present the big-step operational semantics for VIMP. Figures 5.6 and 5.8 define the big-step semantics for VIMP expressions and statements, respectively.

The judgments for variational expressions require a triple of the current variational context (which has been merged with the error context at the statement level), the variable store, and the expression, producing a variational integer or boolean. The variational context is used as an argument to perform a lookup in the variable store whenever a

variable reference is countered. It is also used to check when entering a new variant via a choice that the configuration is satisfiable. Because exceptions cannot be thrown at the expression level we do not need to incorporate an error context or perform satisfiability checks before each expression. We utilize the standard applicative functor lifting functions, *liftA* and *liftA2* to support embedding operations such as addition or comparison within a variational context.

(Variational optional integers)

$$
\begin{array}{rcll}
m & ::= & \bullet & \textit{Nothing} \\
 & | & n & \textit{Integer literal} \\
 & | & C\langle m, m\rangle & \textit{Choice}
\end{array}
$$

(Store)

$$
\begin{array}{rcll}
S & ::= & \textit{empty} & \textit{Empty store} \\
 & | & \textit{ins}((x,m),S) & \textit{Insert value m as variable x in S}
\end{array}
$$

(Variational integers)

$$
\begin{array}{rcll}
\bar{n} & ::= & n & \textit{Integer literal} \\
 & | & C\langle \bar{n}, \bar{n}\rangle & \textit{Choice}
\end{array}
$$

(Variational booleans)

$$
\begin{array}{rcll}
\bar{b} & ::= & b & \textit{Boolean literal} \\
 & | & C\langle \bar{b}, \bar{b}\rangle & \textit{Choice}
\end{array}
$$

VA-Num

$$\frac{}{(C,S,n) \Downarrow_{VA} n}$$

VA-Ref

$$\frac{}{(C,S,x) \Downarrow_{VA} \textit{vlookup}(C,x,S)}$$

VA-Add

$$\frac{(C,S,e) \Downarrow_{VA} \bar{n} \qquad (C,S,e') \Downarrow_{VA} \bar{n}'}{(C,S,e+e') \Downarrow_{VA} \textit{liftA2}(+,\bar{n},\bar{n}')}$$

VA-Chc1

$$\frac{\textit{sat}(C \wedge C') \qquad \textit{sat}(C \wedge \neg C') \qquad (C \wedge C',S,a) \Downarrow_{VA} \bar{n} \qquad (C \wedge \neg C',S,a') \Downarrow_{VA} \bar{n}}{(C,S,C'\langle a,a'\rangle) \Downarrow_{VA} C'\langle \bar{n},\bar{n}'\rangle}$$

VA-Chc2

$$\frac{\textit{unsat}(C \wedge C') \qquad \textit{sat}(C \wedge \neg C') \qquad (C \wedge \neg C',S,a') \Downarrow_{VA} \bar{n}}{(C,S,C'\langle a,a'\rangle) \Downarrow_{VA} \bar{n}}$$

VA-Chc3

$$\frac{\textit{sat}(C \wedge C') \qquad \textit{unsat}(C \wedge \neg C') \qquad (C \wedge C',S,a) \Downarrow_{VA} \bar{n}}{(C,S,C'\langle a,a'\rangle) \Downarrow_{VA} \bar{n}}$$

Figure 5.6: Big-step semantics of Variational IMP expressions

$$\text{VB-Bool}\ \frac{}{(C,S,b) \Downarrow_{VB} b} \qquad \text{VB-Not}\ \frac{(C,S,e) \Downarrow_{VB} \bar{b}}{(C,S,\textbf{not}\ e) \Downarrow_{VB} liftA(\neg,\bar{b})} \qquad \text{VB-And}\ \frac{(C,S,e) \Downarrow_{VB} \bar{b} \qquad (C,S,e') \Downarrow_{VB} \bar{b}'}{(C,S,e\ \textbf{and}\ e') \Downarrow_{VB} liftA2(\wedge,\bar{b},\bar{b}')}$$

$$\text{VB-Less}\ \frac{(C,S,a) \Downarrow_{VA} \bar{n} \qquad (C,S,a') \Downarrow_{VA} \bar{n}'}{(C,S,a < a') \Downarrow_{VB} liftA2(<,\bar{n},\bar{n}')}$$

$$\text{VB-Chc1}\ \frac{sat(C \wedge C') \qquad sat(C \wedge \neg C') \qquad (C \wedge C',S,e) \Downarrow_{VB} \bar{b} \qquad (C \wedge \neg C',S,e') \Downarrow_{VB} \bar{b}}{(C,S,C'\langle e,e'\rangle) \Downarrow_{VB} C'\langle \bar{b},\bar{b}'\rangle}$$

$$\text{VB-Chc2}\ \frac{unsat(C \wedge C') \qquad sat(C \wedge \neg C') \qquad (C \wedge \neg C',S,e') \Downarrow_{VB} \bar{b}}{(C,S,C'\langle e,e'\rangle) \Downarrow_{VB} \bar{b}}$$

$$\text{VB-Chc3}\ \frac{sat(C \wedge C') \qquad unsat(C \wedge \neg C') \qquad (C \wedge C',S,e) \Downarrow_{VB} \bar{b}}{(C,S,C'\langle e,e'\rangle) \Downarrow_{VB} \bar{b}}$$

Figure 5.6: Big-step semantics of Variational IMP expressions (continued)

The rules VA-Num, VA-Add, VB-Bool, VB-Not, VB-And, and VB-Less are fairly straightforward extensions of their non-variational counterparts. We simply lift values and operations into the applicative functor for formula trees as in Section 3.1. VA-Ref is the only rule that interacts with the variable store, performing variational lookup using the variational context as described in Section 5.2. Finally, we have rules for handling choices. VA-Chc1 and VB-Chc1 deal with the case in which both alternatives are satisfiable given the current variational context, executing both alternatives and placing the result values back into a choice with the same condition as the expression. VA-Chc2 and VB-Chc2 handle the case where the left alternative is unsatisfiable given the current variational context. This means we can safely eliminate the choice and replace it with the result of evaluating the right alternative. VA-Chc3 and VB-Chc3 handle the same case but for when the right alternative is unsatisfiable. There is no need for a rule for when both variants are unsatisfiable, as our semantics prevents fully unsatisfiable contexts at the statement level before evaluating any expressions.

The semantics of VIMP statements is slightly more complex when compared with expressions. First, we develop two separate judgment forms $\Downarrow_V$ and $\Downarrow_{VS}$. $\Downarrow_V$ judgments check whether the current context is satisfiable before executing the statement. $\Downarrow_{VS}$ judgments evaluate a statement given that it is in a satisfiable context. In this sense, $\Downarrow_V$ can be seen as a wrapper ensuring satisfiability of contexts before proceeding with evaluation. Both judgments take a 4-tuple of the current variational context, the variable store, the current error context, and the statement to evaluate, and produce an updated state and error context.

V-Unsat expresses the notion that if the current variational context combined with the error context is unsatisfiable, then the statement should not be executed and the execution state should not change. V-Sat confirms that a the current variational context combined with the error context is satisfiable and then evaluates the statement with the appropriate $\Downarrow_{VS}$ rule.

VS-Skip is self-explanatory. VS-Assn assigns a variable to the result of evaluating the given variational arithmetic expression. The store is updated according to the variational update operation defined in Section 5.2, taking the current variational context as an additional argument. VS-Seq simply sequences the execution of two statements.

For control structures, first we have the rule VS-If. First we evaluate the conditional expression e to a variational boolean value $\bar{b}$. We then use the functions *whenTrue* and *whenFalse* (Section 5.4) to extract the variational contexts under which $\bar{b}$ is **true** or **false**, respectively. In the case that $\bar{b}$ is **false** in all variants, *whenTrue* returns the context value **false** (the same holds for **true** and *whenFalse*). We evaluate the **then** statement under the current variational context combined with the value returned by *whenTrue* and the **else** statement under the current variational context combined with the value returned by *whenFalse*. Because the *whenTrue* and *whenFalse* functions return **false** when there are no variants that satisfy their condition, the combined variational context is unsatisfiable and will be caught by the rule V-Unsat, resulting in an unchanged program state. Otherwise, these branches are evaluated under the variational context that represents all variants where they are reachable.

VS-While operates on similar principles to VS-If, using *whenTrue* to extract the context under which the value returned by evaluating the condition is **true** and then evaluating the **do** statement. We then continue execution by reevaluating the **while** statement with the current variational context combined with the context value returned by *whenTrue* and with the updated variable store and error context produced by the **do** statement. For

terminating loops, the context returned by *whenTrue* should grow successively smaller until no variants are **true**. At this point, the variational context will be unsatisfiable and execution on the loop will be halted by the V-Unsat rule. It should be noted that one consequence of allowing possibly infinite loops in a variational language is that if one variant has an infinite loop then execution on any other variant cannot proceed.

VS-Throw evaluates its variational arithmetic expression to a variational integer error code and adds that value and the variational context in which the error was produced to the top frame of the error stack. VS-Try handles the case where the **try** block does not throw any errors. It first pushes an empty frame onto the top of the error stack, evaluates the **try** block statement, and finally checks to see if the top frame of the stack remains empty, signaling that no exceptions were thrown in the **try** block. In this case we can proceed without evaluating the **catch** block.

VS-Catch handles the case where a **try** block does throw an exception. Like VS-Try, it pushes an empty frame onto the top of the error stack, but in this case, the error frame is nonempty after evaluation of the **try** block, meaning that at least one variant has thrown an exception. It extracts a context value and variational integer representing all the error codes thrown via the *catchCtx* and *extract* functions, respectively (see Section 5.3). We then update the variable store at the supplied variable name, using the values returned by *catchCtx* and *extract* as the variational context and variational value arguments to the variational update operation. In this way, we extend the variable environment with the variational error code value but only for the variants that threw an exception within the **try** block. With the error code information now placed in the variable store, we execute the **catch** statement in the variational context returned by *catchCtx*.

Finally, VS-Chc handles choices at the statement level. The left alternative is evaluated under the current variational context combined with the condition of the choice, and the right alternative is evaluated under the current variational context combined with the negation of the condition. We can safely thread the resulting state of evaluating the left alternative to the right alternative without any concerns, because the variation-preservation property does not permit execution in one condition to affect its negation.

The initial evaluation step should begin with the variational context **true**, an empty variable store, and the error context $[[]]$. Derivations result in a variational variable store and an error context. In order to relate these results back to the non-variational IMP, we must define proper configuration operations for these results. Sample configuration functions are given in Figure 5.7. Configuring a variational map is a matter of configuring all

```
type Store v = Map Var v

confStore :: [Dim] → VStore Int → Store Int
confStore ds = map fromJust (filter isNothing (map (conf ds)))

confErr′ :: [Dim] → [Err] → Maybe Int
confErr′ ds [] = Nothing
confErr′ ds ((c,i):es)
  | evalCond ds c = Just i
  | otherwise = confErr′ ds es

confErr :: [Dim] → Stack [Err] → Maybe Int
confErr ds st = confErr′ ds (concat st)
```

Figure 5.7: Configuration functions for variable store and error contexts

the values in the map, filtering out those that configure to `Nothing`, and then unwrapping the remainder from `Just`. Configuring the error context just requires us to search all of the values until we find one that has a condition that evaluates to `True` under the given configuration.

(Variational optional integers)			
m	$::=$	$\bullet$	*Nothing*
	$\mid$	n	*Integer literal*
	$\mid$	$C\langle m, m\rangle$	*Choice*
(Store)			
S	$::=$	*empty*	*Empty store*
	$\mid$	$ins((x, m), S)$	*Insert value m as variable x in S*
(Variational integers)			
$\bar{n}$	$::=$	n	*Integer literal*
	$\mid$	$C\langle \bar{n}, \bar{n}\rangle$	*Choice*
(Variational booleans)			
$\bar{b}$	$::=$	b	*Boolean literal*
	$\mid$	$C\langle \bar{b}, \bar{b}\rangle$	*Choice*(Error lists)
l	$::=$	$[]$	*Nil*
	$\mid$	$(C, \bar{n}) : l$	*Cons*
(Error context)			
E	$::=$	$[]$	*Nil*
	$\mid$	$l : E$	*Cons*

Figure 5.8: Big-step semantics of Variational Imp statements

$$\textsc{V-Sat}\ \frac{sat(ctx(C,E)) \qquad (C,S,E,s) \Downarrow_{VS} (S',E')}{(C,S,E,s) \Downarrow_{V} (S',E')} \qquad \textsc{V-Unsat}\ \frac{unsat(ctx(C,E)))}{(C,S,E,s) \Downarrow_{V} (S,E)}$$

$$\textsc{VS-Skip}\ \frac{}{(C,S,E,\textbf{skip}) \Downarrow_{VS} (S,E)} \qquad \textsc{VS-Assn}\ \frac{(ctx(C,E),S,a) \Downarrow_{VA} \bar{n}}{(C,S,E,x := a) \Downarrow_{VS} (vupdate(ctx(C,E),x,\bar{n},S),E)}$$

$$\textsc{VS-Seq}\ \frac{(C,S,E,s) \Downarrow_{V} (S',E') \qquad (C,S',E',s') \Downarrow_{V} (S'',E'')}{(C,S,E,s\ ;\ s') \Downarrow_{VS} (S'',E'')}$$

$$\textsc{VS-If}\ \frac{\begin{array}{c}(ctx(C,E),S,e) \Downarrow_{VB} \bar{b} \\ (C \wedge whenTrue(\bar{b}),S,E,s) \Downarrow_{V} (S',E') \qquad (C \wedge whenFalse(\bar{b}),S',E',S'') \Downarrow_{V} (E'',)\end{array}}{(C,S,E,\textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s') \Downarrow_{VS} (S'',E'')}$$

$$\textsc{VS-While}\ \frac{\begin{array}{c}(ctx(C,E),S,e) \Downarrow_{VB} \bar{b} \qquad (C \wedge whenTrue(\bar{b}),S,E,s) \Downarrow_{V} (S',E') \\ (C \wedge whenTrue(\bar{b}),S',E',\textbf{while}\ e\ \textbf{do}\ s) \Downarrow_{V} (S'',E'')\end{array}}{(C,S,E,\textbf{while}\ e\ \textbf{do}\ s) \Downarrow_{VS} (S'',E'')}$$

$$\textsc{VS-Throw}\ \frac{(ctx(C,l:E),S,a) \Downarrow_{VA} \bar{n}}{(C,S,l:E,\textbf{throw}\ a) \Downarrow_{VS} (S,((C,\bar{n}):l):E)} \qquad \textsc{VS-Try}\ \frac{(C,S,[]:E,s) \Downarrow_{V} (S',[]:E)}{(C,S,E,\textbf{try}\ s\ \textbf{catch}\ x\ \textbf{with}\ s') \Downarrow_{VS} (S',E)}$$

$$\textsc{VS-Catch}\ \frac{\begin{array}{c}(C,S,[]:E,s) \Downarrow_{V} (S',l:E) \\ (catchCtx(C,l),vupdate(ctx(catchCtx(C,l),E),x,extract(l),S'),E,s') \Downarrow_{V} (S'',E')\end{array}}{(C,S,E,\textbf{try}\ s\ \textbf{catch}\ x\ \textbf{with}\ s') \Downarrow_{VS} (S'',E')}\ nonempty\ l$$

$$\textsc{VS-Chc}\ \frac{(C \wedge C',S,E,s) \Downarrow_{V} (S',E') \qquad (C \wedge \neg C',S',E',s') \Downarrow_{V} (S'',E'')}{(C,S,E,C'\langle s,s'\rangle) \Downarrow_{VS} (S'',E'')}$$

Figure 5.8: Big-step semantics of Variational Imp statements (continued)

Chapter 6 – The Resource DSL

The work on variational imperative programming we have described up to this point was motivated by our experience developing the Resource DSL. The Resource DSL is designed with the intent of allowing easy specification of resource requirements in highly configurable systems. Modern software systems often have complex interactions in terms of resource requirements between different subsystems, and managing these interactions and determining configurations that fulfill a certain set of requirements is often a nontrivial task. The Resource DSL makes the specification and management of these interactions much more feasible and maintainable. Because of the highly configurable and constantly evolving nature of modern software systems the Resource DSL utilizes the principles of variational programming discussed above in order to support the simultaneous verification of resource requirements for systems with large feature sets. In this section we describe the design and implementation of the Resource DSL including several case studies of its use. In particular we pay special attention to the practical concerns brought on by integrating choices into an imperative language.

6.1 Syntax

6.1.1 Resources

In our language a resource is represented by a distinct primitive value. These primitive values can be either booleans, integers, floating point numbers, strings, or the unit value, as defined by the following Haskell datatype:

```
data PVal
    = Unit
    | B Bool
    | I Int
    | D Double
    | S Symbol
```

All resources are maintained in a global resource environment during execution of a program in the Resource DSL. This environment is a mapping from paths to resource

values. Paths are analogous to the Unix filesystem paths that most programmers are already familiar with, allowing for simple organization of resources in treelike structures. For example, a system may place its server bandwidth resource at `/Server/Bandwidth`, while placing its client bandwidth at `/Client/Bandwidth`. Again, by analogy to filesystem paths, resource paths can be either absolute or relative. Relative paths are evaluated with respect to the execution environment's current path. For example, a reference to the resource at the relative path `../GPU/Speed` would evaluate to `/Servers/Hardware/GPU/Speed` given an execution environment with `/Servers/Hardware/CPU` as its current path.

The global resource environment is quite similar to VIMP's variable store. Both are used to maintain a global, mutable store of values. As such, we define the resource environment to be a variational map from paths to values of type `V (Maybe PVal)`.

6.1.2 Expressions

Of course, primitive resource values are of little use without ways of manipulating them. To this end the Resource DSL supports a number of common operations over primitive values in the form of a simple expression language. Expressions are formed by immutable variable references, references to resources by their specific path, literal primitive values, and a number of common unary, binary, and ternary operations, such as addition, negation, equality, ternary conditions, etc.

```
data Expr
    = Ref Var
    | Res Path
    | Lit (V PVal)
    | P1 Op1 (V Expr)
    | P2 Op2 (V Expr) (V Expr)
    | P3 Op3 (V Expr) (V Expr) (V Expr)
```

In order to support variational execution of programs in the resource DSL, we embed choices directly into the syntax of the language, which we first encounter here in the definition of expressions. In contrast to VIMP's approach, we opt to use the generic implementation of formula trees at all points in the language. This permitted us to write a single monadic handler for dealing with variational values at all syntactic levels of the language. We present this handler in Section 6.2.2.

The Resource DSL also supports unary functions over primitive values. The body of the function is simply a choice of expressions as defined above. The function takes a single parameter, which is a combination of the name of the parameter it will be bound to in the body of the function and the type of the parameter. Because the type of the parameter may depend on the variant it is located in, we define types to be a choice of primitive types. Typechecking is currently handled dynamically at runtime, although in the future we would like to potentially implement static typechecking for the language.

```
data PType = TUnit | TBool | TInt | TFloat | TSymbol

data Param = Param Var (V PType)

data Fun = Fun Param (V Expr)
```

In contrast with VIMP, all variables, such as those created here as the parameters to unary functions, are immutable. This helps to simplify the semantics of the language, as well as preventing hard-to-detect user error when interacting with mutable variables.

6.1.3 Effects on Resources

Having established the representation of resources in our DSL as well as their manipulation via expressions and unary functions, we now turn to defining effects on the global resource environment. The Resource DSL supports four basic effects on resources: creation, deletion, modification, and checking/verification. As with VIMP, all effects on a mutable variational map like the resource environment must be performed with a given variational context. Deletion is the simplest, removing the resource at a particular path and replacing it with `Nothing`. Creation takes a path and an expression that produces a variational primitive value and inserts that value into the resource environment. Modification takes a unary function over the value of the resource and replaces it with the result of the function. Checks take a unary function over the value of the resource that returns a boolean, with `True` indicating success and `False` indicating failure. When a check fails, an error is thrown, with an error message indicating what check failed. In this way, checks allow for a user to specify conditions or requirements that must hold for a particular resource.

```
data Effect
    = Create (V Expr)
    | Check Fun
    | Modify Fun
    | Delete
```

6.1.4 Statements

Statements are used in order to sequence and combine the above effects into programs in the Resource DSL. The most basic statement is the `Do` command, which takes a path to a resource and applies an effect to the resource located there. The other statements are a collection of control structures that allow for the sequencing and construction of programs out of these `Do` commands. `If` and `For` are straightforward control structures that allow for conditional execution and looping, respectively. `In` executes a block in the context of a particular path, where all relative paths in an `In` block are evaluated with reference to the supplied path value. This allows users to easily describe work that should be done in particular resource sub-environments. `Let` allows for the extension of the current variable environment at the statement level. Finally, `Load` permits the calling of named subroutines with a list of arguments.

One particular difficulty arises in determining how to represent a variational block of statements. Typically we would use a simple list of statements, but in the variational setting we must ask what kind of variational list we should use. Fortunately, recent work [24] has determined *segment lists* to be a particularly efficient implementation of variational linked lists. A segment list is a list of segments, where segments are either plain lists or a choice between segment lists. Therefore, we define our blocks to be segment lists of statements.

```
data Segment a = Elems [a] | Split Cond (SegList a) (SegList a)
type SegList a = [Segment a]
type Block = SegList Stmt

data Stmt
    = Do Path Effect
    | If (V Expr) Block Block
    | In Path Block
    | For Var (V Expr) Block
    | Let Var (V Expr) Block
    | Load (V Expr) [V Expr]
```

6.1.5 Models and Profiles

Programs in the resource DSL are organized into *models* and *profiles*. Both take a list of initial parameters as input, which are converted into variables when evaluating the body of the program. Models are more straightforward to define, as their body is simply a statement block. By contrast, a profile body is a direct mapping from resource paths to variational lists of effects. Profiles therefore do not support any of the control structures available to models. Viewed another way, profiles are models that are restricted to `In` and `Do` statements. In fact, the Resource DSL provides a utility function `toProfile` that automatically converts a model to a profile provided that the model's body only contains `In` and `Do` statements.

```
data Profile = Profile [Param] (Map Path (SegList Effect))
data Model = Model [Param] Block
```

One might ask why include profiles at all if they are strictly a subset of models? The answer is that, where possible, profiles are to be preferred over models because they are declarative definitions of resource specifications. Declarative Profiles are much easier to compose and reason about in comparison to imperative Models. Certain parts of the Resource DSL even go so far as to require profiles, such as the specification of mission requirements, which should only ever be made up of `Check` effects. Models should only be used where necessary, which typically means resource configurations that are sensitive

to the order in which effects are carried out or that require runtime decision making enabled through control structures.

The execution environment maintains a dictionary of profiles and models. This is used to look up the corresponding profile or model when a call to the `Load` command is made. In this way users can organize their programs via subroutines that are called from a parent model.

6.2 Execution

The links between VIMP and the Resource DSL should now hopefully be apparent. Both are variational imperative languages that require support for mutable state and errors/exceptions. We now demonstrate how the Resource DSL implements some of the techniques detailed in VIMP in a real-world interpreter written in Haskell. In Haskell, side effects are typically implemented monadically, with multiple side effects being implemented by a monad transformer stack [15]. The principal evaluation mechanism in the Resource DSL is also monadic. Specifically, we utilize a reader monad instance for handling the current resource path prefix (set by `In` statements), the immutable variable environment, and the dictionary of Models and Profiles; a state monad instance for handling the resource environment; and an exception monad instance for throwing runtime exceptions. We encountered some challenges in integrating variation into a monad transformer stack. This section details our conclusions with regards to how to successfully integrate variation into a monadic approach to effect handling.

6.2.1 Uncooperative Monads

Our first thought was simply to integrate the monad for formula trees into an existing reader-state-exception transformer stack by creating a monad transformer for formula trees. After all, we already have an existing monad instance for formula trees. This would provide the benefit of easily integrating choices in the highly composable pattern that is the hallmark of monad transformers. As our work on VIMP shows, this would have never succeeded. Adding the monad instance for formula trees to the stack would not provide a variational context, or any other of the elements necessary to support side effects in a variational setting.

This result is also unsurprising considering existing research into monad transformers. The list monad transformer has been proved to only be valid when composed with commutative monads, which includes our use of the state and exception monads [10]. The list monad is used to model nondeterminism. The monad instance for formula trees can be seen as modeling a kind of labeled nondeterminism. This can be further verified by comparing the behavior of both monads, which unwrap a collection of values, perform some computation, and place the results back into the collection. Given the proof that the list transformer monad only holds for commutative monads, it is no surprise that the similar variational monad also does not work by itself as a monad transformer.

This result is somewhat unfortunate, as it means that the variational monad is not easily composable with existing languages that utilize monad transformer stacks for effect handling. The work on VIMP further supports this conclusion, as it suggests that integrating variation into an effectful language requires explicit support for a variational context that governs the execution of the language. However, we found that, while easily composing variation into an existing monad transformer stack is not possible, it is possible to keep the existing monad transformers as long as the proper care is taken with their usage. The next section describes our custom monadic handler for dealing with variation generically in the Resource DSL.

6.2.2 The Evaluation Monad

Rather than extending our monad transformer stack to handle variation, we simply retain the same monad transformer stack and add a special handler for dealing with variational values. We add the variational context to the reader context and the error context to the state context. Unlike VIMP, we do not have a method for catching exceptions in the Resource DSL, so we have no need for a stack to roll back errors. Instead, the error context is made up of a condition that indicates which variants are in error and a single variational error value that contains all of the errors encountered. We provide the full definition of the monad transformer stack and their respective contexts in Figure 6.1.

From here, we can begin to define a monad instance for handling variational values. Because it makes generically handling errors easier, we wrap all values in a `Maybe`, so that variational values in the Resource DSL all have the form `V (Maybe a)`. This allows us to return a default value from alternatives that are in an error state while introducing minimal computational overhead.

```
data Error = ...

data StateCtx = SCtx {
  renv :: Map Path (V (Maybe PVal)),
  errCtx :: Cond,
  errVal :: V (Maybe Error)
}

data ReaderCtx = RCtx {
   prefix :: Path,
   environment :: Map Var (V (Maybe PVal)),
   dictionary :: Dictionary,
   vCtx :: Cond
}

type MonadEval m =
  (MonadError (V (Maybe Error)) m, MonadReader RCtx m, MonadState StateCtx m)

-- | A specific monad for running MonadEval computations
type EvalM a = ExceptT Mask (StateT StateCtx (Reader Context)) a
```

Figure 6.1: Monad transformer stack definition

We define a function `vBind` that is analogous to the monadic bind operation, but for the type `MonadEval m => m (V (Maybe a))` (Figure 6.2). As Haskell does not allow us to write monad instances for this type, we define it as a standalone function. Of note is that `vBind` demonstrates the flag-based exception handling optimization described in Section 5.3.

`vBind` employs several helper functions. `vMove` takes the condition of a choice and an action in the evaluation monad representing the computation to be carried out in evaluating a single alternative of a choice. It retrieves the current variational context and error context, combines them with the condition argument, and checks if the combined context is unsatisfiable. If it is unsatisfiable, there is no need to evaluate the action, and instead an error is thrown for the current variant, which will be caught by the original choice. Otherwise, it evaluates the action in the context of the variational context extended by the condition argument.

`vHandle` takes the condition of the choice and monadic actions for both alternatives. It calls `vMove` for each alternative. It then catches any exceptions thrown in the alternatives, either by `vMove` before execution of the action or during execution of the action itself. If both alternatives throw errors, the error is propagated by the choice, as in the optimization detailed in Section 5.3. Otherwise, it suppresses the exceptions thrown by single alternatives and replaces them with a placeholder `Nothing` value.

`vBind` handles the variational value itself. First, it extracts the variational value from within the evaluation monad. Then, it handles the value based on its case. If the value is `One Nothing`, it simply returns `One Nothing` in a manner analogous to the behavior of the plain `Maybe` monad. In the case of a variational value that is not a choice or `Nothing`, we simply apply the function passed to `vBind` to the underlying value. Finally, in the case of a choice, we wrap each alternative in the provided action and pass execution to the `vHandle` helper function.

To resolve the problem of Haskell not permitting a monad definition for the type of `vBind`, we wrap it in a newtype `VM m a` and define the monad instance for this type, with the constraint that `m` implements `MonadEval`:

```
vMove :: MonadEval m ⇒ Cond → m a → m a
vMove c m = do
  vctx ← getVCtx
  e ← getErrCtx
  let c′ = vctx :&&: c
  if (unsat (c′ :&&: Not e)) then
    throwError (One Nothing)
  else
    local (λ(Ctx p m dict _) → Ctx p m dict c′) m

vHandle :: MonadEval m ⇒
    Cond →
    m (V (Maybe a)) →
    m (V (Maybe a)) →
    m (V (Maybe a))
vHandle c l r = do
  let r′ = vMove (Not c) r
  let l′ = vMove c l
  l′′ ← l′ `catchError` (λe → do
    r′ `catchError` (λe′ → throwError (Chc c e e′))
    return (One Nothing))
  r′′ ← r′ `catchError` (λ_ → return (One Nothing))
  return (Chc c l′′ r′′)

vBind :: MonadEval m ⇒ m (V (Maybe a)) → (a → m (V (Maybe b))) →
m (V (Maybe b))
vBind v f = do
  v′ ← v
  case v′ of
    (One Nothing) → return (One Nothing)
    (One (Just a)) → f a
    (Chc c l r) → let
      f′ v = vBind (return v) f
    in vHandle c (f l) (f r)
```

Figure 6.2: Definition of vBind

```
newtype VM m a = VM { unVM :: (m (VOpt a)) }

instance (MonadEval m) ⇒ Functor (VM m) where
  fmap = liftM

instance (MonadEval m) ⇒ Applicative (VM m) where
  pure = return
  (<*>) = ap

instance (MonadEval m) ⇒ Monad (VM m) where
  return = VM ∘ return ∘ One ∘ Just
  v >>= f = VM $ vBind (unVM v) (unVM ∘ f)
```

6.2.3 Throwing Exceptions

Throwing exceptions is handled by the function `vError`, which retrieves the current variational context from the reader context, updates the error context in the state to be the disjunction of the current error context and the variational context, updates the variational error value in the state with the `Error` value just thrown, and finally throws an error in the exception monad to halt execution for the current variant.

```
vError :: MonadEval m ⇒ Error → m a
vError e = do
  vctx ← getVCtx
  updateErrCtx (λc → c :||: vctx)
  updateErrVal (λm → Chc vctx (One (Just e)) m)
  throwError (One (Just e))
```

6.2.4 Interacting with the Resource Environment

Just like with VIMP, all interaction with the resource environment occurs within a particular variational context, retrieved from the current reader monad context. Additionally, the Resource DSL places requirements that attempting to access or modify a resource that does not exist, or create a resource at a path in which one already exists, results in a thrown exception. For brevity, we only present the variational lookup operation in Figure

```
vLookup′ :: (MonadEval m, Ord k) ⇒
   k →
   Map k (V (Maybe v)) →
   m (V (Maybe v))
vLookup′ k env
   | Just v ← lookup k env = do
     c ← getVCtx
     return $ select c v
   | otherwise = return (One Nothing)

checkExists :: MonadEval m ⇒ Error → V (Maybe v) → m ()
checkExists e (One Nothing) = vError e
checkExists _ (One (Just _)) = return ()
checkExists e (Chc c l r) = vHandleUnit c (checkExists e) l r

vLookup :: (MonadEval m, Ord k) ⇒
   Error →
   k →
   Map k (V (Maybe v)) →
   m (V (Maybe v))
vLookup e k env = do
  v ← vLookup′ k env
  checkExists e v
  return v
```

Figure 6.3: Implemenation of variational lookup

6.3 for the resource environment, omitting the implementations for modification, deletion, and creation. The helper function `vLookup'` performs lookup on the variational map by selecting on the value at the given key using the current variational context. The resulting value is then passed to the `checkExists` helper function, which throws a variational error for variants where the requested value does not exist. Note that `vHandleUnit`, here omitted, is similar to `vHandle` except for the type `m ()` rather than `m (V (Maybe a))`.

6.3 Evaluation and Use Cases

We implemented a full variational interpreter for the Resource DSL. Users provide a main application model, an initial resource environment, mission requirements in the form of a profile of `Check` commands, a dictionary of available profiles and models, and initial

parameters to the application model. The interpreter runs the main application model and then checks it against the mission requirements. As outputs it produces the final variational resource environment, the error context, and the error value. All successful configurations should result in a variant that is not present in the error context. For unsuccessful configurations, selecting or configuring the variational error value produces an appropriate error message to help the user in determining the cause of the failure, which could be either a runtime error like a failed typecheck, or a failed call to the `Check` command.

Our implementation also comes with a convenient command-line interface for interacting with variational programs. Users are given the option of performing selection or configuration prior to execution. This allows users to eliminate configurations that they are not interested in, increasing efficiency in cases where certain configurations can be ruled out ahead of time. Keeping track of the error context allows us to use this value to support queries where a user supplies a boolean formula representing a configuration that they are interested in and the query returns whether that condition has any successful configurations, including a sample list of possible configurations generated by the built-in SAT solver.

Our DSL has been tested against three example problems as part of its evaluation. The first test scenario concerned determining correct specification of resource environments that depend on location-based data, such as with GPS systems. The second test scenario concerned common networking resource requirements, such as making certain that there is enough available bandwidth in a network to handle transmissions of a particular size. The final test scenario concerned maintaining agreement between servers and clients in their use of cryptographic protocols provided by different available libraries.

All of these test scenarios demonstrated the utility of a fully variational model of execution, but the final test scenario in particular provided an interesting test case. We were able to encode each cryptographic cipher, mode, keysize, and padding as a distinct dimension, along with the various libraries that provide this functionality. This allowed for our variational interpreter to potentially check all possible cryptographic configurations of server and client at once. This also meant that if a user were only interested in a particular subset of features, such as only AES encryption with CBC mode, they could simply preconfigure the program according to those specifications, avoiding unnecessary computation.

Chapter 7 – Related Work

As discussed at various points in this thesis, the formal semantics we have developed for variational imperative languages is in part a synthesis of existing techniques developed for existing variational interpreters. The present work builds upon this by bringing all of these techniques together and providing a formal semantics for reasoning about them. It also demonstrates how to support catchable exceptions in a variational environment, which to our knowledge is absent from previous work. In this chapter we compare our work to preceding work on variational execution, highlighting the development and implementation of several techniques utilized here.

The first to develop a notion of variational execution were Erwig and Walkingshaw in their 2011 GTTSE tutorial [8]. While it does not address the interaction of side effects with variation, it set the stage for future research by developing the choice calculus, as well as the functor, applicative, and monad instances that we have made use of in this work.

Kästner et al. [11] developed a proof-of-concept variational interpreter for the WHILE language. This works shares a significant amount in common with our work on IMP, as both utilize toy imperative languages to develop a theory of variational execution with side effects. This work originated the concepts of maintaining a variational context during execution, defining a variational store as a variational map with lookup and update operations that take the variational context as an argument, and the use of the *whenTrue*() helper function in conjunction with control structures. However, the variational WHILE language only supports a single type of side effect—mutable state—and therefore is incapable of analyzing the interactions between multiple different types of effects and variation. Additionally, the evaluation of the language is only presented as Scala functions rather than a formal big-step semantics.

Around the same time, Kim, Khurshid, and Batory developed *shared execution* for testing software product lines [14]. Shared execution primarily focuses on low-level support for variation at the level of a language VM. Unlike the variational WHILE language, shared execution is not built on the formal foundation of the choice calculus, and thus its support for variation is handled on a more ad hoc case-by-case basis, in particular in its merging of shared contexts. Nevertheless, shared execution does contain an analogue of

the variational context with variational store pattern for dealing with variational mutable state.

Another seemingly independent development of variational execution was Austin and Flanagan's *faceted evaluation* for Javascript [4]. In this case, choices, or *facets* as they are termed by Austin and Flanagan, represent desirable security properties of Javascript, where private dataflows correspond to the right alternative and public dataflows correspond to the left alternative. Again, this work contains an analogue of the variational context with variational store pattern, in this case referring to the variational context as the *program counter*. Faceted evaluation only allows dimensions as tags for its choices, rather than arbitrary boolean expressions, which decreases the expressiveness of the language but obviates the need for satisfiability checking. Faceted evaluation additionally provides a formal semantics for a language they call λ^{facet}. It is also the only approach that attempts to provide a mechanism for variational IO side effects such as interacting with the file system. However, in comparison with the other approaches to variational execution presented here, faceted evaluation is primarily concerned with dataflow security and not with developing general models of variational programming.

Building on the above work with the WHILE language, Nguyen, Kästner, and Nguyen developed the Varex interpreter for a subset of PHP [20]. Varex successfully demonstrated the ability to test plugin interactions in the highly configurable WordPress application. Varex adds support for throwing exceptions, but they are not catchable. Varex's exceptions are also reported directly when encountered as a side effect of execution, rather than producing a variational error value as the result of execution. Varex especially demonstrates the utility of developing a variational interpreter for a subset of an existing language.

Meinicke's VarexJ is an implementation of a variational Java VM [17]. VarexJ handles both throwing and catching exceptions, but it does so at the level of Java bytecode instructions. This low-level approach is therefore difficult to generalize to arbitrary variational imperative languages, and is of more use when variationalizing a VM.

Variational interpreters are not the only places where side effects and variation interact. For example, Chen, Erwig, and Walkingshaw's work on typing variational lambda calculus makes use of *masks* that are similar to VIMP's error contexts [5]. The central notion here is that when typechecking a variational program, it is desirable to continue typechecking variants that are not in an error state when an error is encountered in a separate variant.

Chapter 8 – Conclusion and Future Work

This thesis has provided a formal operational semantics for the variational imperative language VIMP. The techniques formalized in VIMP were then demonstrated in the implementation of the Resource DSL. Our intention is for this work to serve as the basis for future development of variational imperative languages. By providing a formal semantics as well as a detailed description of the various techniques involved in the implementation of variational imperative languages, we hope that future language developers will be able to more fully focus on language-specific features, rather than being bogged down in the common details of the interaction of variation with side effects.

It is our hope that the formal semantics that we have developed here could serve as the basis for a formal proof that VIMP satisfies the correctness property outlined in Section 2.3. Such a proof would give the highest level of confidence that the techniques outlined herein are sound and can safely be applied to more complex languages.

This work has focused primarily on mutable state and exceptions. It is an area of future research to explore the interactions of other types of effects with variation. In particular, it is unclear how variational programs that must somehow interact with the outside world through IO operations should be handled. It is likely that a variational language that supports IO would also need to be paired with variational constructs in the operating system itself, such as variational files and filesystems, variational databases, etc. There has already been some progress made in this area on the database front [2, 3], but more work is necessary with regards to how to integrate variational execution with such operating system entities.

For the Resource DSL, we are interested in the possibilities that built-in support for SAT solving has for optimization. In our implementation of the test scenarios for the Resource DSL we noted that many computations that currently occur at runtime in our language could potentially be offloaded to a SAT or SMT solver. We are also interested in continuing to explore the domain of resource specifications for the potential to support fully declarative specifications. In the current iteration of the Resource DSL we reluctantly included models as a way to encode fully imperative programs with support for sequencing and control structures. If possible, we would like to move further in the direction of our declarative profiles. This however would require ways of addressing

resource configurations in which the order in which effects are performed and sequenced is important in a declarative way, which ultimately may not be possible.

Bibliography

[1] Sven Apel, Christian Kästner, and Christian Lengauer. Feature featherweight java: A calculus for feature-oriented programming and stepwise refinement. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 101–112. ACM, 2008.

[2] Parisa Ataei, Arash Termehchy, and Eric Walkingshaw. Variational Databases. In *Int. Symp. on Database Programming Languages (DBPL)*, pages 11:1–11:4. ACM, 2017.

[3] Parisa Ataei, Arash Termehchy, and Eric Walkingshaw. Managing Structurally Heterogeneous Databases in Software Product Lines. In *VLDB Work. on Polystores and Other Systems for Heterogeneous Data*, 2018. To appear.

[4] Thomas H Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL)*, volume 47, pages 165–178. ACM, 2012.

[5] Sheng Chen, Martin Erwig, and Eric Walkingshaw. An Error-Tolerant Type System for Variational Lambda Calculus. In *ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*, pages 29–40, 2012.

[6] Sheng Chen, Martin Erwig, and Eric Walkingshaw. Extending Type Inference to Variational Programs. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 36(1):1:1–1:54, 2014.

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

[8] Martin Erwig and Eric Walkingshaw. Variation Programming with the Choice Calculus. In *Generative and Transformational Techniques in Software Engineering IV (GTTSE 2011), Revised and Extended Papers*, volume 7680 of *LNCS*, pages 55–99, 2013.

[9] Spencer Hubbard and Eric Walkingshaw. Formula Choice Calculus. In *Int. Work. on Feature-Oriented Software Development (FOSD)*, pages 49–57. ACM, 2016.

[10] Mark P Jones and Luc Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Department of Computer Science. Yale University, 1993.

[11] Christian Kästner, Alexander Von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward variability-aware testing. In *Int. Work. on Feature-Oriented Software Development (FOSD)*, pages 1–8. ACM, 2012.

[12] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. TypeChef: toward type checking #ifdef variability in C. In *Int. Work. on Feature-Oriented Software Development (FOSD)*, pages 25–32. ACM, 2010.

[13] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242. Springer, 1997.

[14] Chang Hwan Peter Kim, Sarfraz Khurshid, and Don Batory. Shared execution for efficiently testing product lines. In *Proc. International Symp. Software Reliability Engineering (ISSRE)*, pages 221–230. IEEE, 2012.

[15] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *ACM SIGPLAN Symp. on Principles of Programming Languages (POPL)*, pages 333–343. ACM, 1995.

[16] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable analysis of variable software. In *ACM SIGSOFT Int. Symp. on the Foundations of Software Engineering*, pages 81–91. ACM, 2013.

[17] Jens Meinicke. VarexJ: A variability-aware interpreter for java applications. Master's thesis, University of Magdeburg, 2014.

[18] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. Sat-based analysis of feature models is easy. In *Int. Software Product Line Conference (SPLC)*, SPLC '09, pages 231–240, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[19] Meng Meng, Jens Meinicke, Chu-Pan Wong, Eric Walkingshaw, and Christian Kästner. A Choice of Variational Stacks: Exploring Variational Data Structures. In *Int. Work. on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 28–35. ACM, 2017.

[20] Hung Viet Nguyen, Christian Kästner, and Tien N Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *Proc. International Conf. Software Engineering (ICSE)*, pages 907–918. ACM, 2014.

[21] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal aspects of computing*, 10(2):171–186, 1998.

[22] Tobias Nipkow and Gerwin Klein. *Concrete Semantics*. Springer, 2014.

[23] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 419–443. Springer, 1997.

[24] Karl Smeltzer and Martin Erwig. Variational lists: Comparisons and design guidelines. In *Int. Work. on Feature-Oriented Software Development (FOSD)*, FOSD 2017, pages 31–40, New York, NY, USA, 2017. ACM.

[25] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys (CSUR)*, 47(1):6, 2014.

[26] Eric Walkingshaw. *The Choice Calculus: A Formal Language of Variation*. PhD thesis, Oregon State University, 2013. `http://hdl.handle.net/1957/40652`.

[27] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. Variational Data Structures: Exploring Trade-Offs in Computing with Variability. In *ACM SIGPLAN Symp. on New Ideas in Programming and Reflections on Software (Onward!)*, pages 213–226, 2014.

[28] Eric Walkingshaw and Klaus Ostermann. Projectional Editing of Variational Software. In *ACM SIGPLAN Int. Conf. on Generative Programming: Concepts and Experiences (GPCE)*, pages 29–38, 2014.

[29] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, 1993.