AN ABSTRACT OF THE THESIS OF

Meng Meng for the degree of Master of Science in Computer Science presented on September 22, 2017.

Title: Implementation Techniques for Variational Data Structures

Abstract approved: _____

Eric T. Walkingshaw

Many applications require not only representing variability in software and data, but also computing with it. To do so efficiently requires variational data structures that make variability explicit in the underlying data and the operations used to manipulate it. Variational data structures have been developed ad hoc for many applications, but there is little general understanding of how to design them or what tradeoffs exist among them.

In this thesis, we introduce the concept of holes to represent variational data structures of different sizes and shapes. Moreover, we strive for a more systematic exploration and analysis of a variational data structure. We want to know how different design decisions affect the performance and scalability of a variational data structure, and what properties of the underlying data and operation sequences need to be considered.

Specifically, we study several alternative designs of a variational stack and analyze how these design decisions affect the performance of a variational stack with different usage profiles. We evaluate variational stacks in a real-world scenario: in the interpreter VarexJ when executing real software containing variability. Finally, we discuss different ways of representing variational priority queues and show how this affects the performance of the variational Dijkstra's algorithm.

Implementation Techniques for Variational Data Structures

by

Meng Meng

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented September 22, 2017
Commencement June 2018

Master of Science thesis of Meng Meng presented on September 22, 2017.

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.

Meng Meng, Author

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Eric Walkingshaw, for accepting me into his research group and providing me with an excellent atmosphere for doing research. Without his guidance and patience, I would never have been able to finish my thesis. Additionally, I would like to thank my committee members including Martin Erwig, Amir Nayyeri, and Kyle Niemeyer for their interest in my work.

Second, I would like to thank the various members whom I had the opportunity to work: Jens Meinicke, Chu-pan Wong, and Christian Kästner. Thanks you for sharing excellent ideas with me.

Finally, I would express a deep sense of gratitude to my parents for their constant love and giving me the opportunity to study in US. I would also like to thanks my friends and especially thank Hanzhong Xu and Hong Pei who always strengthened my morale by standing by me in all situations.

CONTRIBUTION OF AUTHORS

This thesis takes material from the paper that were co-authored with Jens Meinicke, Chu-Pan Wong, Eric Walkingshaw, and Christian Kästner [Meng et al., 2017].

Specifically, parts of chapter 1, 2, 4, 5, and 8 are from the paper.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

LIST OF TABLES (Continued)

Table Page

Chapter 1: Introduction

Variation is common in both software systems and data computation. In software systems, variability is introduced so that users can configure the software according to different use cases, for example, using command line options, plugins, or preprocessor directives. In data computation, variability arises by running a program or part of a program many times with inputs that are varied slightly, as in configuration testing, uncertainty analysis, and speculative analysis [Kästner et al., 2012, Kim et al., 2012, Nguyen et al., 2014, Sumner et al., 2011, Muslu et al., 2012]. Although variation makes software and computation flexible, it also requires more efficient techniques for analyzing and executing programs since it is usually not possible to explore each variant individually due to the combinatorial explosion of possibilities. Recent research in several domains addresses the combinatorial explosion problem with a variety of solutions that share some common ideas: analyze, manipulate, and compute with an explicit representation of variation in code and data, and exploit sharing among the variants.

By encoding variation explicitly in code and data, many approaches gain significant performance improvement without sacrificing precisions of the results. For example, uncertainty analysis proposed by Sumner et al. [Sumner et al., 2011] runs faster by operating on a vector of uncertain input values all at once, rather than on individual values sequentially, since many computations are independent and can be shared. Encoding and manipulating variability explicitly has also been extremely successful in analyzing software product lines. Such variational analyses are shown to scale to large configuration spaces but still provide sound results [Thüm et al., 2014, Meinicke et al., 2014].

Data structures for computing with explicit variation are often reinvented and optimized in an ad-hoc way with little reuse across projects. We envision that more general *variational data structures* can be essential building blocks to be reused across all of these domains, but they are not currently well understood [Walkingshaw et al., 2014]. This thesis is a step towards a more systematic exploration and analysis of the design and implementation

of variational data structures. Specifically, we focus on variational lists, stacks, and priority queues. This thesis discusses how different implementations and optimizations affect the performance of variational data structures. Specifically, we evaluate how different variational stacks perform in the variability-aware Java interpreter VarexJ, where variational stacks play a central role [Meinicke, 2014, Meinicke et al., 2016], and analyze the performance of different variational priority queues implementations when used in finding the shortest path in a variational graph.

1.1 Contributions and Outline

The main goal of this thesis is to present different ways to represent variational data structures and analyze how different implementations and optimizations affect the performance of variational data structures. The following parts describe the structure of this thesis and give the corresponding contributions that each chapter makes.

Chapter 2 introduces the related work and several basic concepts including the choice calculus and variational programming. It also defines two widely used functions that will be used throughout the thesis.

Chapter 3 introduces the concept of holes for variational data, which can generally be applied to different variational data structure designs. It also presents how variational data structures with holes are retrieved in two different ways. One way for getting the data is by calling the function with a context to indicate what variational context the operations are performed in. The other way is calling the function without the context, which means the variational data structures decide the context and values that operations perform in. For giving a intuitive understanding of the two different ways, we use a variational list as an example to show the difference between the two ways.

Chapter 4 details how to use the holes in variational stacks. It also gives a description of a small family of variational stack representations. The family consists of two alternative core stack implementations and two independently optional optimizations, leading to eight different variational stacks.

Chapter 5 provides two experiments on variational stacks and makes two main contribu-

tions:

1. It provides an exploratory analysis of variational stacks on artificially generated operation sequences (Section 5.1) that vary in the number of configuration options they reference and in the distribution of variational operations.

2. It also provides an empirical analysis of the performance of each of the eight variational stacks when used as the variational operand stack of the variational Java interpreter VarexJ (Section 5.2). Our experiments show that both optimizations are highly effective in practice, and also demonstrate that choosing the right variational data structure can have a significant impact on the overall performance of programs that compute with variability.

Chapter 6 describes an implementation of variational priority queues using a balanced binary search tree and defines two different ways of retrieving data.

Chapter 7 first provides an experiment on a variational heapsort to evaluate the performance of variational priority queues using the two different implementations. Then, we show how to solve the shortest path problems in a variational graph using a variational version of Dijkstra's algorithm. At the end of this chapter, we discuss how two different implementations influence the performance of variational priority queues on a real world example.

Chapter 8 gives a summary of this thesis and the future work.

Chapter 2: Background and Related Work

In this chapter, we provide necessary background on variational data structures and also on variability-aware execution, an analysis strategy that extensively uses variational data structures to share common execution paths across variants. We use the variability-aware interpreter VarexJ in later sections to evaluate our variational stack implementations.

2.1 The Choice Calculus

Conceptually, *variational data* represents many different concrete data values at once. However, variational data is not just a flat set of variants, but also describes which configurations each variant is associated with. Variation on atomic data values can be expressed in different forms, such as by trees of choices between alternatives or by maps from variants to the configuration context where each is relevant [Erwig and Walkingshaw, 2011, Walkingshaw et al., 2014].

The choice calculus is one way to express variational data [Erwig and Walkingshaw, 2011, Walkingshaw, 2013, Hubbard and Walkingshaw, 2016]. For example, consider the value $x = \mathit{Choice}(A, 1, 3)$, a choice that represents either the concrete value 1 if the *configuration option* A evaluates to true, or 3 otherwise. Computations on x will operate on both 1 and 3, preserving the fact that 1 is associated with the variational context A and 3 with the context $\neg A$. A configuration space is a set of all possible combinations over configuration options. A context is a set of configurations. Clearly, the size of the configuration space for variational data is exponential in the number of independent configuration options it contains.

A conditional value is an Java implementation of choice calculus, which is a mapping of concrete values to the corresponding contexts [Meinicke, 2014]. It can be implemented as a tag tree, formula tree, or formula map [Walkingshaw et al., 2014]. For simplicity, we only discuss tag tree and formula tree representations. Listing 2.1 shows an implementation

of conditional values with type `T` using a tag tree representation [Meinicke, 2014]. The Tag in Line 7 is a single configuration option. The class `One` represents concrete values and the class `Choice` maps tags to conditional values. This representation is simple but it cannot share common partitions of the configuration space [Walkingshaw et al., 2014].

```
1  interface Conditional<T> {}
2  class One<T> implements Conditional<T> {
3    T value;
4    One(T value) {...}
5  }
6  class Choice<T> implements Conditional<T> {
7    Tag t;
8    Conditional<T> yes, no;
9    Choice(Tag t, Conditional<T> yes, Conditional<T> no) {...}
10 }
```

Listing 2.1: A tag tree implementation of conditional values in Java

Compared to the tag tree representation, a formula tree is more flexible by replacing the tags by a `FeatureExpr`. A `FeatureExpr` is a formula over a set of configuration options and is used to represent variational contexts. However, we need SAT solvers to reason about valid configuration space in formula tree representations [Walkingshaw et al., 2014].

2.2 Variational Programming

In this section, we introduce two widely used functions, `map` and `flatMap`, to manipulate conditional values. The `map` function is simple and it maps function `f` with type $T \Rightarrow U$ over all variants. For example, suppose we map a `succ` function, which has type $Int \Rightarrow Int$, over $Choice(A, 1, 3)$. We will get $Choice(A, 2, 4)$. Listing 2.2 shows one implementation of the `map` function for formula trees [Meinicke, 2014]. The syntax is from Java 8 and we can see that the `map` method takes a function as input and applies it to all variants [Meinicke, 2014].

```
1  interface Conditional<T> {
2    Conditional<U> map(Function<T, U> f);
3  }
```

```
4  class One<T> implements Conditional<T> {
5     T value;
6     <U> Conditional<U> map(Function<T, U> f) {
7        return new One<>( f.apply(value) );
8        }
9  }
10 class Choice<T> implements Conditional<T> {
11    FeatureExpr ctx;
12    Conditional<T> yes, no;
13    <U> Conditional<U> map(Function<T, U> f) {
14       return new Choice<>(ctx, yes.map(f), no.map(f) );
15    }
16 }
```

Listing 2.2: Implementation of the map function for formula trees

The `map` applies a function to all variants without changing the structure of conditional values. However, the `flatMap` function can apply a function with type `T` ⇒ `Conditional<U>` to the variants. Since our variants have the type `Conditional<T>`, this operation will flatten the return value into type `Conditional<U>`.

The Listing 2.3 shows the implementation of the `flatMap` from VarexJ [Meinicke, 2014].

```
1  interface Conditional<T> {
2      <U> Conditional<U> flatmap(Function<T, Conditional<U> > f); }
3  class One<T> implements Conditional<T> {
4      T value;
5      <U> Conditional<U> flatmap(Function<T, Conditional<U> > f){
6          return f.apply(value) ; }
7  }
8  class Choice<T> implements Conditional<T> {
9      CTX ctx;
10     Conditional<T> yes, no;
11     <U> Conditional<U> flatmap(Function<T, Conditional<U> > f){
12         return new Choice<>( ctx, yes.fmap(f), no.fmap(f) ); }
13 }
```

Listing 2.3: Implementation of the flatMap for formula trees

To illustrate how `flatMap` works, we give two examples. The first one is adding two choices: $Choice(A, 1, 4)$ and $Choice(B, 10, 20)$. It can be calculated by applying the `flatMap` with the function $x \Rightarrow Choice(B, x + 10, x + 20)$ over all the variants on $Choice(A, 1, 4)$. The result is $Choice(A, Choice(B, 11, 21), Choice(B, 14, 24))$.

The second example is applying `flatMap` under a specific context, which only manipulates a part of conditional values. The `vadd` function in Listing 2.4 maps an integer n to a choice $Choice(ctx, n + v, n)$ over all variants under the context `ctx` of the conditional value `num`. For the rest of parts under $\neg$ `ctx`, the variants stay unchanged as shown in Line 3. The function `vmin` in Listing 2.5 is similar to the `vadd` function except that the `vmin` function compares the variants with a value v under `ctx` and returns the smaller one.

```
1  Conditional<Integer> vadd(Conditional<Integer> num, FeatureExpr ctx, int v) {
2     Conditional<Integer> ret = num.flatMap(FeatureExpr f, Integer n) → {
3        if(f.and(ctx).isContradiction()) {
4           return new One(n); }
5        if(f.and(ctx).isTautology()) {
6           return new One(n + v); }
7        return new Choice(ctx, n+v, n);
8     }
9     return ret;
10 }
```

Listing 2.4: Variational add function over a choice

```
1  Conditional<Integer> vmin(Conditional<Integer> num, FeatureExpr ctx, int v) {
2     Conditional<Integer> ret = num.flatMap(FeatureExpr f, Integer n) → {
3        if(f.and(ctx).isContradiction() || v > n) {
4           return new One(n); }
5        if(f.and(ctx).isTautology()) {
6           return new One(v); }
7        return new Choice(ctx, v, n);
8     }
9     return ret;
10 }
```

Listing 2.5: Variational minimum function over a choice

2.3 Variational Data Structures

Variational data structures are designed to compactly represent and compute with variational data [Walkingshaw et al., 2014]. In previous work, some designs of variational lists, maps, and sets are discussed [Walkingshaw et al., 2014]. However, it is still unclear how the design decisions described in that work affect the scalability of variational computations in practice and which design decisions have yet to be identified. Thus, a systematic evaluation and exploration of the design space is needed.

Variational data structures have diverse applications. They are commonly used for variational program analysis, such as variational ASTs for type checking [Kästner et al., 2011], variational type inference [Chen et al., 2014, Chen and Erwig, 2014], and variational executions [Erwig and Walkingshaw, 2013, Nguyen et al., 2014, Meinicke, 2014, Chen et al., 2016]. They are also proposed for variational representations of software artifacts, such as test suits, formal specification, and deductive verification [Walkingshaw et al., 2014]. Further applications from other domains already need to cope with variation, such as travel planning, uncertainty in analysis, and context-oriented programming [Walkingshaw et al., 2014].

2.4 Variability-Aware Execution

Variability-aware execution makes extensive use of variational data structures. The core idea is to combine repeated computations into a single execution, tracking differences while sharing as much data and execution as possible. This technique is useful, for example, in testing highly configurable systems [Meinicke et al., 2016, Meinicke, 2014, Kästner et al., 2012, Nguyen et al., 2014], where it enables faster testing of all configurations and scales well to large configuration spaces.

A *variability-aware interpreter* is a programming language interpreter that supports variability-aware execution [Meinicke et al., 2016, Meinicke, 2014, Kästner et al., 2012, Nguyen et al., 2014]. In such an interpreter, data values are variational and instructions are executed within variational contexts (corresponding to the selection of some or all configuration options). Since VarexJ computes with variational data, there are many

applications for variational data structures.

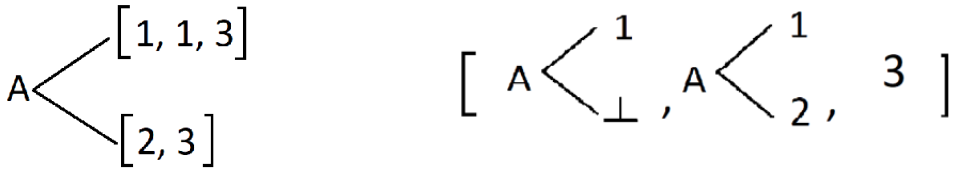
Computation in the JVM centers around operand stacks, so the operand stack is a central data structure that must handle variation efficiently. In the JVM, there are instructions for pushing constants, field values, or local variables onto the stack; arithmetic operations pop inputs from the stack and push their results; and method inputs and outputs are passed via the operand stack. There are many ways variation could be encoded in operand stacks. For example, one possibility is to keep the stack implementation as-is and but manage multiple versions dependent on the variational context (i.e. a choice of stacks). Another possibility is to make the entries in the stack variational (i.e. a stack of choices). Both implementations can represent the same data, but have different, non-obvious, and scenario-dependent effects on performance.

Since the operand stack is central to the JVM, the design of a variational stack has immediate effects on the performance of the interpreter. In chapter 4 of this paper, we explore different designs of a variational stack and investigate how the design decisions affect performance in different scenarios.

Chapter 3: Holes in Variational Data Structures

3.1 Holes in Variational Data Structures

In this section, we introduce a technique for implementing variational data structures using holes. A variational data structure can be implemented in a number of different ways. The simplest way is `Conditional<D>` where `D` stands for a plain data structure, for example, `Conditional<List<T>>` represents a variational list. Now we can represent a list $[1, 2, 3]$ under variation context A and $[2, 3]$ under $\neg A$ as $\mathit{Choice}(A, [1, 1, 3], [2, 3])$ as shown in Figure 3.1a. Each variant in `Conditional<List<T>>` is independent, which means any operations on the specific variants do not have any impact on other variants. For example, in Figure 3.1a, deleting the first element under the context A changes the left (upper) branch to list $[1, 3]$, but the right branch under the context $\neg A$ stays unchanged. This representation is straightforward. However, one main drawback is that it duplicates the shared part of the data structures.



(a) `Conditional<List<T>>` **(b)** `List<Conditional<T>>` **with holes**

Figure 3.1: Two representations of variational lists.

An alternative way of representing variational lists that supports sharing is changing `Conditional<List<T>>` to `List<Conditional<T>>`. In this way, we can represent the variational list with variant $[1, 2, 3]$ under A and $[1, 2, 4]$ under $\neg A$ as $[1, 2, \mathit{Choice}(A, 3, 4)]$ instead of $\mathit{Choice}(A, [1, 2, 3], [1, 2, 4])$. The advantage of this representation is that it supports sharing common subparts. However, this representation requires all variants

have the same length. For example, we cannot represent the variational list in Figure 3.1a. Therefore, `List<Conditional<T>>` is not expressive enough.

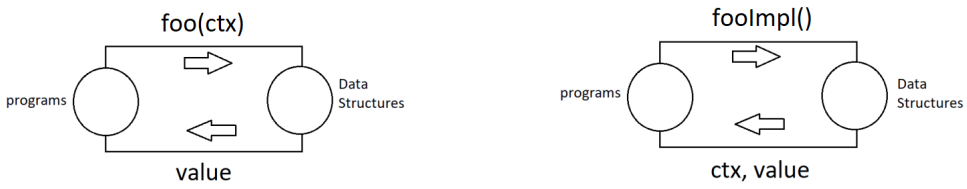
To fix this problem in the `List<Conditional<T>>` representation, we introduce holes [Smeltzer and Erwig, 2017]. A hole indicates that an element is not present in some variants, and is written $\perp$. For example, adding an element 1 under the context A into a variational list, we can simply add $Choice(A, 1, \perp)$ into the list directly. Figure 3.1b also shows how to share the common 3 value among different variants and shows how holes allow us to represent unbalanced variational lists. However, an issue regarding holes is that it may need some extra computing to assemble the return value when retrieving data from data structures. For example, when calling `popFirst(True)` in Figure 3.1b, it should return $Choice(A, 1, 2)$ instead of $Choice(A, 1, \perp)$. So, we need traverse the data stored in the list to eliminate the holes in the return value. The details will be discussed in the next section where we also show that the holes strategy is a general idea that can be applied to different variational data structures. In this thesis, we focus on variational lists, stacks, and priority queues.

In conclusion, the key advantage of the `List<Conditional<T>>` representation with holes is that the holes implementation supports more sharing compared to the naive implementation and it is more expressive than the pure `List<Conditional<T>>` implementation. However, some operations in the holes-based implementation are more expensive since they require filling holes and traversing data structures. In the next section, we give two different ways for retrieving data from variational data structures with holes and compare the tradeoffs between them.

3.2 Retrieving Data

One way of retrieving data from variational data structures is to specify the variation context ctx. The ctx in function $foo(ctx)$ is to indicate in what variational context the operations are performed. Figure 3.2a shows the general process of $foo(ctx)$. Basically, users call the function foo with context ctx and data structures return the values for all variants where the context is true. This method is flexible and can be used for all programs since users can get the values under the context they provided. However, in

most cases, $foo(ctx)$ requires extra work to fill the holes in the return value. To avoid the extra computations, we introduce another method in Section 3.2.2, called $fooImpl()$. Its general idea is shown in Figure 3.2b. When users call $fooImpl()$, the data structure itself decides the return value and ctx. Thus, $fooImpl()$ method is not used for all programs since users cannot predict the context of the data they get. We will illustrate the details and usage of these two approaches in the next two subsections.



(a) General process of foo(ctx) **(b) General process of fooImpl()**

Figure 3.2: Two ways for retrieving data from variational data structures.

3.2.1 The foo(ctx) Method

Since users get the data directly with the context ctx, the return value should be filled as much as possible under the given context. To eliminate the holes in the return value, the algorithm will traverse the whole data structure to find the part under ctx until there are no holes in the return values. During the traversal, it may change several data items in the list since we fill the holes from the rest of the data structure. For simplicity, we take the variational list in Figure 3.1b as an example.

The `popFirst(ctx)` function is used to get the first element from all variants under ctx and remove it. When calling `popFirst(True)`, it eventually returns $Choice(A, 1, 2)$ rather than $Choice(A, 1, \bot)$. Since the leftmost element has a hole under $\neg A$, the algorithm will keep traversing the items in the rest of the list until finding the part under $\neg A$ to fill the holes in the return value. In this example, it finds the 2 in the second element of the implementation list. Since we only cut the part under $\neg A$ off, the left branch under A remains unchanged and the right branch will change to a hole. After calling `popFirst(True)`, the remainder of the variational list is $[Choice(A, 1, \bot),\ One(3)]$.

Similarly, if we call `popFirst(`$\neg A$`)` on the list in Figure 3.1b, the hole under $\neg A$ is filled by the value 2 from the second item. The part of value corresponding to the A will be kept in the list. However, calling `popFirst(`A`)` on the list in Figure 3.1b will remove the leftmost item and directly return the value 1 since 1 does not have a hole. The following Table 3.1 shows the sequence of variational lists produced by two successive `popFirst(`$\neg A$`)` calls.

List	popFirst($\neg A$)
[$Choice(A, 1, \bot)$, $Choice(A, 1, 2)$, $One(3)$]	2
[$Choice(A, 1, \bot)$, $Choice(A, 1, \bot)$, $One(3)$]	3
[$Choice(A, 1, \bot)$, $Choice(A, 1, \bot)$, $Choice(A, 3, \bot)$]	$\bot$

Table 3.1: popFirst($\neg A$) states

The major advantage of this representation is that users can get any data they want for a given variation ctx, which is flexible. However, as illustrated above, this method takes extra computations to eliminate holes, assemble return values and update the rest of variational list, all of which may require traversing the whole data structure.

3.2.2 The fooImpl() Method

To avoid extra work on fillings holes, we introduce another method $fooImpl()$ for retrieving data and removing data from the data structure. Rather than allowing the user to select the variation context, we let the data structure pick a convenient context and return the corresponding value. One example of this approach for variational list is to have the popFirst operation simply return and remove the whole leftmost item. We call this implementation `popFirstImpl()`. When calling `popFirstImpl()` on the example in Figure 3.1b, it returns the $Choice(A, 1, \bot)$ as a result. The rest of the list remains unchanged. Next, we will discuss a common scenario where we can use operations implemented with either approach.

Listing 3.1 shows a plain program. This program continues to get elements from a plain data structure and processes all elements until the data structure is empty. To translate this framework to a variational version, we use `get(ctx)` to retrieve the elements from a variational data structure. Line 4 in Listing 3.2 shows that the program initializes the

context ctx as all configurations for which its variant is not empty. For example, if we have a variational list $Choice(A, [1, 2], \perp)$, the context ctx will be set as A since we cannot get elements from $\perp$ under $\neg A$. Then, we get the `data` from the data structure under ctx and the procedure in Line 7 takes data and ctx as an input.

```
1 while(!isEmpty()) {
2   T data = get();
3   procedure(data);
4 }
```

Listing 3.1: A common pattern in a plain program

```
1 while(true) {
2   FeatureExpr ctx;
3   Conditional<T> data;
4   ctx = non-empty configurations;
5   if(ctx == False) break;
6   data = get(ctx);
7   procedure(data, ctx);
8 }
```

Listing 3.2: A variational version of the program in Listing 3.1 using `get(ctx)`

After reviewing the whole program, we find that it is not necessary to get the data under all non-empty configurations in Line 4. Instead of forcing the data structure to return all non-empty data, we can let the data structure return whatever is most convenient. Listing 3.3 shows the idea of how to change the program with $foo(ctx)$ to the program with $fooImpl()$. The `isEmpty()` function in Line 1 is to check whether the underlying representation is empty or not. Line 4 shows that the data structure will decide which part of the data should be returned.

```
1 while(!isEmpty()) {
2   FeatureExpr ctx;
3   Conditional<T> data;
4   data, ctx = get();
5   procedure(data, ctx);
6 }
```

Listing 3.3: A variational version of the program in Listing 3.1 using `get()`

One more trick for Listing 3.3 is that the *ctx* returned by `get()` can be merged into a conditional value as shown in Listing 3.4 since we can represent the empty part as a hole. This framework looks like a normal program, but the data structure and procedure are variational.

```
1 while(!isEmpty()) {
2   Conditional<T> data = get();
3   procedure(data);
4 }
```

Listing 3.4: An improved version of the program in Listing 3.3

To give an intuitive understanding of the pattern we illustrated before, we give a simple concrete example. Suppose our goal is to sum up all the elements in the variational list in Figure 3.1b. Table 3.2 and Table 3.3 shows the sequence of steps using the `popFirst(ctx)` method and the `popFirstImpl()` method respectively. Table 3.2 follows the pattern in Listing 3.2 and Table 3.3 follows the pattern in Listing 3.4. We will get the same results using two ways.

List	Non-empty Context	popFirst(ctx)	Sum
[$Choice(A, 1, \perp)$, $Choice(A, 1, 2)$, $One(3)$]	`True`	$Choice(A, 1, 2)$	$Choice(A, 1, 2)$
[$Choice(A, 1, \perp)$, $One(3)$]	`True`	$Choice(A, 1, 3)$	$Choice(A, 2, 5)$
[$Choice(A, 3, \perp)$]	A	3	$One(5)$

Table 3.2: popFirst(ctx) states

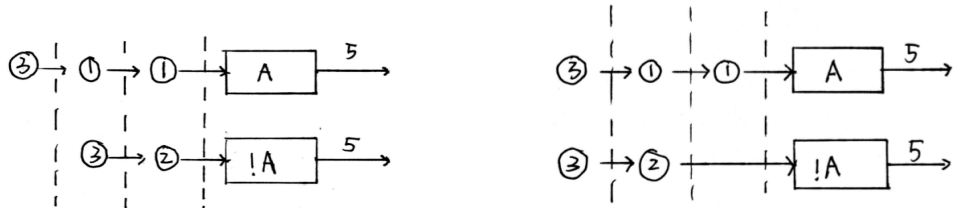
List	popFirstImpl()	Sum
[$Choice(A, 1, \perp)$, $Choice(A, 1, 2)$, $One(3)$]	$Choice(A, 1, \perp)$	$Choice(A, 1, \perp)$
[$Choice(A, 1, 2)$, $One(3)$]	$Choice(A, 1, 2)$	$One(2)$
[$One(3)$]	$One(3)$	$One(5)$

Table 3.3: popFirstImpl() states

Theorem 1. *Suppose a program* `G` *which conceptually maps a function* f *over all variants where* f *consumes all elements of the data structures in order, then the program* `G` *can be run correctly with either* $fooImpl()$ *or* $foo(ctx)$.

Proof. Conceptually, we consider all variants separately. For each variant, both $fooImpl()$ and $foo(ctx)$ will return the same elements in the same order. Then the function f will

get the same output with the same input. Thus, the program `G` can be run correctly with either $fooImpl()$ or $foo(ctx)$. □



(a) machine for popFirst(ctx). **(b) machine for popFirstImpl().**

Figure 3.3: The input order of program `G` by using `head(True)` and `headImpl()`

For example, referring back to the example shown in Table 3.2 and Table 3.3, we can conceptually consider two variants under A or $\neg A$ separately as shown in Figure 3.3. Figure 3.3a shows the `popFirst(ctx)` operations takes the data into the two variants at the same time. However, the `popFirstImpl()` method in Figure 3.3b will first feed the data to the variant under A but the input order of each machine by using the two different methods is the same. Thus, both methods will have the same outputs. In summary, the difference between the two retrieval methods is the time for programs that process all of the data. The $foo(ctx)$ function guarantees the data can be sent at the same time under ctx but $fooImpl()$ method cannot control the time that data will be sent.

In conclusion, $foo(ctx)$ is more flexible and precise since the ctx can be provided by the programmer. However, it has to compensate for the underlying representation of the variational data structure. The $fooImpl()$ approach saves some work by returning the data for the most convenient context, but it has constrains on programs. We have shown that one common programming pattern can work correctly with $fooImpl()$ in Theorem 1. There still exists an unexplored space for future work identifying other kinds of programs that can run with $fooImpl()$ correctly.

3.2.3 Comparison

In this section, we provide a method for comparing two retrieval methods by counting different operations that each implementation takes. For example, we can count how many elements need to be filled during filling holes and how many choices need to be merged using two different implementations. Table 3.4 give the count information for the sum example using two retrieval ways illustrated in Section 3.2.2. We also count the number of addition operations in this example.

	popFirst(ctx)	popFirstImpl()
additions	3	2
merges	1	1
holes-filled	2	0

Table 3.4: comparison for Figure 3.1b

	popFirst(ctx)	popFirstImpl()
additions	2	4
merges	3	1
holes-filled	3	0

Table 3.5: comparison for Figure 3.4

The following parts show how to get the Table 3.4:

1. **additions** To count the number of addition operations using the `popFirst(ctx)` method, we can easily count it by summing all the elements in the third column of Table 3.2. For the left branch under A, we execute addition operations two times to get 5, and one time to get 5 in the right branch. So, it total takes 3 addition operations.

2. **merges** After summing all elements, we get $\mathit{Choice}(A, 5, 5)$ as an intermediate result. So, it also takes one merge time to reach $\mathit{One}(5)$.

3. **holes-filled** During the whole process, we fill the holes in the return value twice.

Then, We can easily get the information for the `popFirstImpl()` method using the similar technique. Totally, `popFirstImpl()` takes one less addition operation and two less assemblies than the `popFirst(ctx)` method.

Table 3.4 shows that the `popFirstImpl()` method works better than `popFirst(ctx)`. However, the `popFirst(ctx)` method can also save operations in some cases. Consider the example in Figure 3.4, it generates the data in Table 3.5.

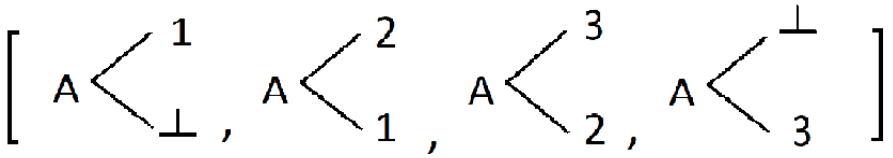


Figure 3.4: A simple example.

In Table 3.5, we can see that the `popFirst(ctx)` method saves two addition operations compared to the `popFirstImpl()` method, but the number of merges and holes to be filled are still higher than using the `popFirstImpl()` method. However, if we generalize this situation to a longer example [$Choice(A, 1, \bot)$, $Choice(A, 2, 1)$, $Choice(A, 3, 2)$, ... $Choice(A, n-1, n-2)$, $Choice(A, n, n-1)$, $Choice(A, \bot, n)$], using `popFirst(ctx)` takes $(n-1)$ times addition operations but `popFirstImpl()` takes twice times than `popFirst(ctx)`, which is $2(n-1)$ times.

In conclusion, both representations have their advantages. $foo(ctx)$ can decrease the number of operations in some cases but it usually need more merges and more holes filling than the $fooImpl()$ implementation. However, filling the holes might be the most time-consuming part since the computing on configurations is complicated. Thus, $fooImpl()$ implementation may have a significant advantage. We will discuss and evaluate both implementations for a real world example in later sections.

Chapter 4: Variational Stacks

To explore the dimensions of the design space of variational data structures, and to evaluate their impact on performance, we need some implementations. In this section, we describe a small family of variational stack data structures, illustrated by the feature diagram in Figure 4.1. It consists of two mutually exclusive core variational stack implementations and two conceptually independent, optional optimizations, leading to eight different variational stacks.

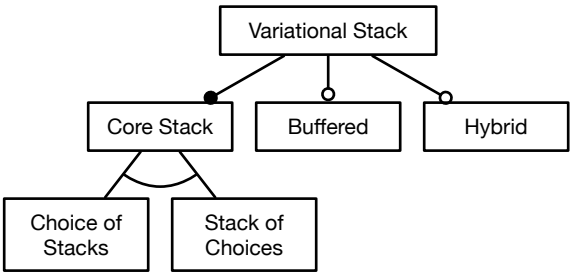


Figure 4.1: Feature diagram for variational stacks.

Throughout this section, it is important not to confuse the variation between the different variational stack implementations described in Figure 4.1, which is an implementation-time concern about which variational stack to pick, and the variation *within* a particular variational stack, which is the runtime concern that variational stacks are designed to handle efficiently.

Each variational stack implements the interface shown in Listing 4.1. For simplicity, we only discuss the operations `push` and `pop`. Note that both `push` and `pop` take an argument `ctx` to indicate in what variational context the operations are performed. The main challenge of designing an efficient variational stack is that the height of the variant stacks can differ if values are pushed and popped in different contexts. In the rest of this section, we show how unbalanced variant stacks can be managed with different tradeoffs, and how

```
interface VariationalStack {
  void push(FeatureExpr ctx, Conditional value);
  Conditional pop(FeatureExpr ctx);
}
```

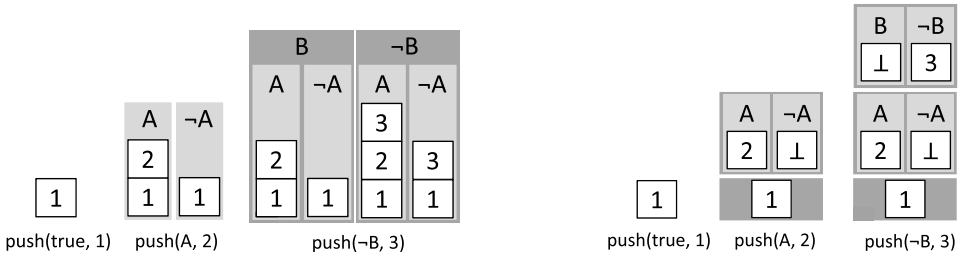
Listing 4.1: Interface for variational stacks.

these implementations can be further optimized by exploiting properties of different usage profiles. Implementation details and other stack operations, such as `peek` and `switch`, are available as open source at `http://meinicke.github.io/VarexJ/`.

4.1 Choice-of-Stacks

One way to implement a variational stack is as a choice among non-variational stacks. Since each stack may be a different size, this design represents unbalanced stacks naturally.

In Figure 4.2a, we illustrate how a choice-of-stacks stores variational data for a sequence of three conditional push operations. On the second push, the stack splits because of the non-trivial context. Since it only splits on the relevant configuration option A, these two alternative stacks will be shared among all configurations that differ in other, irrelevant options. Also note that as the stacks split, the original value 1 is redundantly stored in the first position of all variant stacks. If operations are performed in many different variational contexts, the number of variant stacks and potential redundancy grows exponentially.



(a) Choice-of-stacks. **(b) Stack-of-choices.**

Figure 4.2: Comparison of two different core variational stack implementations.

```
1  class ChoiceOfStacks implements VariationalStack {
2     Conditional<Stack> stack;
3     void push(FeatureExpr ctx, Conditional value) {
4        value.foreach((FeatureExpr c, Object v) -> {
5           push(c.and(ctx), v);
6        });
7     }
8     private void push(FeatureExpr ctx, Object value) {
9        stack = stack.flatMap((FeatureExpr f, Stack s)->{
10          if (f.and(ctx).isContradiction()) {
11             return new One(s); }
12          if (f.andNot(ctx).isContradiction()) {
13             s.push(value);
14             return new One(s); }
15          final Stack clone = s.copy();
16          clone.push(value);
17          return new Choice(ctx, clone, s);
18       });
19    }}
```

Listing 4.2: Implementation of choice-of-stacks.

Listing 4.2 shows the implementation of `push` for choice-of-stacks. The underlying data structure for the choice-of-stacks implementation is `Conditional<Stack>`, that is, a choice calculus expression with stacks at the leaves. Since the argument `value` is also variational, the `push` operation first iterates over all of the plain values in the argument (line 5).

The `push` helper method pushes a plain value to all of the variant stacks. For each variant stack, it checks the variational context of the stack, `f`, against the variational context of the push operation, `ctx`. There are three possibilities: at line 10, the push applies in no contexts that include this stack, so it is returned unchanged (the `One` constructor builds a `Conditional` value with just one variant); at line 12, the push applies in all contexts that include this stack, so the value is simply pushed; at line 15, the push applies to some contexts that include this stack, so we must clone the stack to capture the two different execution paths going forward.

The major drawback of the choice-of-stacks implementation is that if just one value of a stack differs in two contexts, all of the values common to both contexts must be duplicated, which has both space and time implications. If the two variant stacks later converge,

identifying and merging them is expensive since we must iterate over all of the variants.

4.2 Stack-of-Choices

The next implementation inverts the relationship of stacks and choices by representing a variational stack as a stack of conditional values (stack-of-choices). The idea is to avoid the cloning required by the choice-of-stacks implementation by using a single stack and using choices within that stack to encode differences among entries in different contexts.

Figure 4.2b illustrates how stack-of-choices shares the common `1` value among different contexts. To handle unbalanced stack sizes, We use the holes implementations. For example, when the number 2 is pushed under context A, we push the choice $\mathit{Choice}(A, 2, \bot)$. The rest of this subsection describes how the push and pop operations handle holes in the stack.

The push operation is straightforward. If a value is pushed for the trivial context *true* (meaning the same value is pushed in all contexts), the value can be pushed directly onto the stack. However, if a value is pushed with a non-trivial context, such as A, we must introduce a hole to represent the absence of the value under the contradictory context, $\neg A$.

The pop operation is more complicated, since it might need to eliminate holes in order to return meaningful values. For example, consider the rightmost stack of Figure 4.2b; a pop in context *true* cannot simply return $\mathit{Choice}(B, \bot, 3)$, since there are still values on the stack under context B. Thus, pop first fills the hole with $\mathit{Choice}(A, 2, 1)$ then removes and returns the value $\mathit{Choice}(B, \mathit{Choice}(A, 2, 1), 3)$.

To handle holes, the pop operation traverses the stack top-down and assembles values from different contexts until either there is no hole in the return value, or the whole stack is traversed. For example, consider again popping a value from the rightmost stack in Figure 4.2b. When calling pop under context $\neg B$, the top element can be removed and the value 3 returned directly, since 3 is a value with no holes. However, if pop is called under B, the hole is filled by traversing the rest of the stack, eventually returning $\mathit{Choice}(A, 1, 2)$. Since pop is applied under context B, the part of the top entry corresponding to context

```
1  class StackOfChoices implements VariationalStack {
2     Conditional[] stack;
3     int top = -1;
4     void push(FeatureExpr ctx, Conditional value) {
5        stack[++top] = new Choice(ctx, value, null);
6     }
7     Conditional pop(FeatureExpr ctx) {
8        Conditional pop = null;
9        for (int i = top; i >= 0; i--) {
10          Conditional current = stack[i].get(ctx);
11          pop = new Choice(ctx, current, pop);
12          stack[i] = new Choice(ctx, null, stack[i]);
13          if (i == top && stack[i].isNull()) top--;
14          ctx = ctx.and(getCtxOfNull(current));
15          if (ctx.isContradiction()) break;
16       }
17       return pop;
18    }}
```

Listing 4.3: Implementation of stack-of-choices.

$\neg B$ must be kept on the stack.

Listing 4.3 shows the implementation of stack-of-choices. The member variable `stack` is an array of conditional values, and `top` is the index of the topmost element. The method `push` adds an entry to the stack as a choice with a hole (`null`). Popping a value from the stack works as follows: The variable `pop` is used to incrementally collect values while traversing the stack. Starting from `top`, the value under the current context is retrieved (line 10) and stored in the current `pop` variable (line 11). Next, the popped value is removed from the current position in the stack. If this leaves the top entry empty, `top` can be decremented. To fill the remaining holes in `pop`, the context of the null value is determined by calling `getCtxOfNull`, and this context is used for the next iteration, until there are no holes left (i.e. `ctx` is a contradiction) or all entries are traversed. Finally, the value `pop` is returned.

Stack-of-choices supports more sharing than the choice-of-stacks implementation, but this comes at the cost of complicating stack operations (e.g. pop). In the best case, the pop operation can simply return part of the topmost entry if it does not contain a hole in the given context. In the worst case, however, the whole stack must be traversed to pop a value. We discuss and evaluate these tradeoffs in the next chapter.

4.3 Buffered Stack Decorator

Both choice-of-stacks and stack-of-choices support push and pop operations with arbitrary contexts. During the development of VarexJ, we discovered that values are usually popped from the stack under the same context as they were pushed. We turn this insight into a decorator for an underlying core variational stack that exploits this property.

The idea is to buffer pushed values in a plain stack as long as the variational context doesn't change. When several sequential pushes and pops are called in the same context, we can just push and pop from the buffer, saving the cost of manipulating variational values. When a push or pop is invoked in a different variational context, the buffered-stack decorator default to the core variational stack implementation, pushing all currently buffered values to the core stack in the context associated with the buffer.

```
1  class BufferedStack implements VariationalStack {
2     LinkedList buffer;
3     FeatureExpr bufferCTX;
4     VariationalStack coreStack;
5     void push(FeatureExpr ctx, Conditional value) {
6        if (!bufferCTX.equals(ctx)) {
7           debufferAll();
8           bufferCTX = ctx;
9        }
10       buffer.push(value);
11    }
12    Conditional pop(FeatureExpr ctx) {
13       if (bufferCTX.equals(ctx) && !buffer.isEmpty()) {
14          return buffer.pop();
15       } else {
16          debufferAll();
17       }
18       return coreStack.pop(ctx);
19    }}
```

Listing 4.4: Buffered-stack decorator.

The implementation of the buffered stack is shown in Listing 4.4. The `push` and `pop`

operations check whether they were invoked in the same context as the buffer. If not, they fall back to the core implementation using `debufferAll`. As the implementation illustrates, when pushes and pops occur in the same context, there are no map calls at all, so even very large choice values can be pushed and popped with no performance overhead.

4.4 Hybrid Stack Decorator

All of the implementations described so far assume that the stack always needs to handle variational values and operations in different variational contexts. During the development of VarexJ, it was discovered that most calls to the stack do not involve variation at all. To exploit this property, we implemented the hybrid stack decorator, shown in Listing 4.5. The hybrid stack uses a plain stack until a variational stack is necessary. The implementation of `push` checks whether the context is different from the plain stack or whether the pushed value is variational. If neither is true, it keeps using the plain stack. Otherwise, it switches to a variational stack initialized by the current contents of the plain stacks.

```
1  class HybridStack implements VariationalStack {
2     FeatureExpr stackCTX;
3     VariationalStack stack = new Stack();
4     boolean switched = false;
5     void push(FeatureExpr ctx, Conditional value) {
6        checkParameter(ctx, value);
7        stack.push(ctx, value)
8     }
9     void checkParameter(FetureExpr ctx, Conditional value) {
10       if (switched) return;
11       if (!ctx.equals(stackCTX) || !value.isOne()) {
12          // create a variational stack
13          // push all current values
14          switched = true;
15       }
16    }}
```

Listing 4.5: Hybrid-stack decorator.

The hybrid stack decorator exploits the fact that in many scenarios, stack operations are simple and do not need a variational stack. The decorator exploits these cases by working with more efficient plain values as long as possible.

Chapter 5: Experimental Analysis on Variational Stacks

Choosing the right data structure for an application often depends on both the particular data that will be stored in the data structure, and on patterns of its usage within the application. The same is true for variational data structures, except we must also take into account properties of data and usage related to variability. In this section, we identify a few of these properties and present two sets of experiments to analyze their impact and evaluate the performance of the family of variational stacks described in Chpater 4.

In the first set of experiments, described in Section 5.1, we focus on understanding how individual properties of data and usage influence the performance of variational stacks in order to help recommend a specific variational stack for a particular use case. In the second set of experiments, described in Section 5.2, we evaluate the performance of variational stacks in a real-world setting, when used as the operand stack of the VarexJ interpreter.

5.1 Analyzing Tradeoffs in Generated Data

In principle, we can view the range of applications for variational data structures as an n-dimensional space, where each dimension represents a different property of data or usage that influences which variational data structure to choose. If we fully understood this space, we could partition it into regions for each class of data structure (e.g. variational stacks) and then prescribe a particular variational data structure for a given application.

Examples of data and usage properties that impact the performance of variational data structures include: the number of configuration options and unique variants, the density and complexity of variation points within the data, and the ratio and distribution of variational operations. In this subsection, we experimentally analyze the impact of two of these properties on variational stacks: number of configuration options and one aspect of the distribution of variational operations.

5.1.1 Number of Configuration Options

The most efficient variational data structure for an application with only two variants (e.g. in delta execution [Tucek et al., 2009]) is unlikely to be the same as an application where the data varies in 10s or 100s of independent configuration options (e.g. when analyzing software product lines [Kästner et al., 2011]). Within our family of variational stacks, we expect this tradeoff to be illustrated by the choice of which core stack implementation to choose. For a small number of variants, we expect the directness of the choice-of-stacks implementation of Section 4.1 will win out over the relatively more complicated stack-of-choices implementation of Section 4.2. For a large number of variants with enough commonalities, we expect the increased sharing in the stack-of-choices implementation has a chance to pay off. This experiment attempts to identify the threshold of variants where the stack-of-choices core stack implementation pays off.

Experimental setup. For each number of configuration options from 0 to 8, we artificially generate a sequence of 500 push/pop operations. The generated operation sequences are constrained to prevent stack underflow errors, and also to approximate realistic data by preferring simple variational contexts [Liebig et al., 2010] and sequential operations in the same variational context (see Section 4.3). More specifically, each operation sequence is generated in the following way: start by generating a push operation with a random feature selected from among the 0–8 available configuration options and the trivial *true* context; for operation $n + 1$, 90% of operations will use the same variational context while the remainder will choose a new random feature; if any variant stack in the chosen context is empty, produce a push operation, otherwise randomly push or pop; for push operations, choose a random integer or, in 10% of cases, push a choice in a random configuration option between random integers.

For each operation sequence, we measure the runtime[1] and the maximum memory consumption[2] of four of the variational stacks produced by the product line described in Chapter 4. Each operation sequence is executed on each stack 10 times, choosing the fastest execution. We omit the four stack variants that include the hybrid optimization

[1] All measurements throughout the paper were performed on a machine with an Intel Core i7-5600 CPU (4 cores, 2.6 GHz), 11.6 GB of RAM, running 64-bit Ubuntu Linux.

[2] `https://github.com/meinicke/ObjectSizeMeasure`.

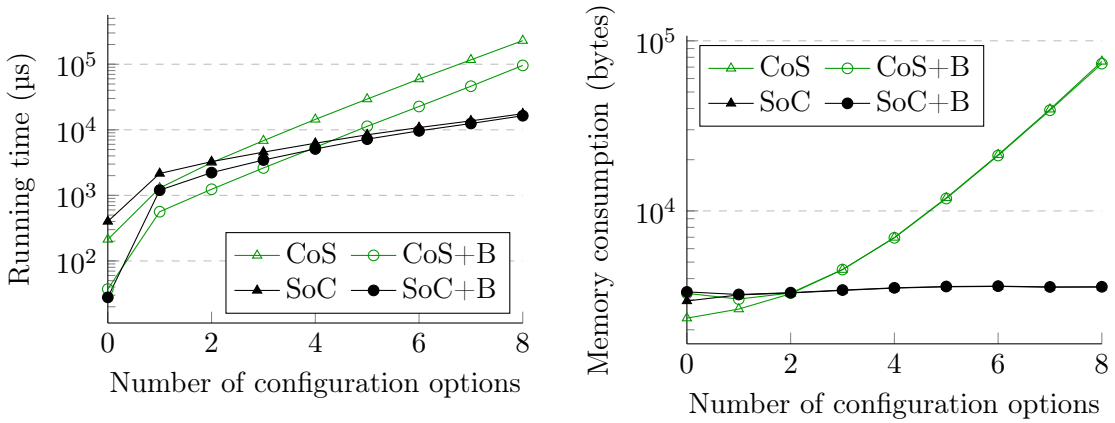


Figure 5.1: Comparing variational stacks on artificially generated operation sequences with different numbers of configuration options that may appear in the variational contexts. Y-axes are on a logarithmic scale.

since this optimization cannot provide benefits (and only causes small overhead) in such a variation-dense scenario.

Results and analysis. The results of the experiment are presented in Figure 5.1. The independent variables are: (1) the number of configuration options, plotted on the x-axis, and (2) the choice of stack, indicated by separate lines (*CoS* represents the core choice-of-stacks implementation, *SoC* is the stack-of-choices implementation, and *+B* indicates a stack decorated by the buffered optimization); the dependent variable is the runtime of the operation sequence in the left graph and amount of memory consumed in the right graph.

In general, the precise threshold where the stack-of-choices core stack implementation pays off can depend on many variables that are fixed in the experiment, such as the ratio of operations with a non-trivial context. However, what we observe in the results is that this threshold is quite low for the values of these variables that we analyzed. In our experiment, the stack-of-choices core stack implementation outperforms the choice-of-stacks implementation at three configuration options, and outperforms the buffered choice-of-stacks implementation at five configuration options.

Despite the low threshold for switching from choice-of-stacks to stack-of-choices, we expect

there are many applications where this threshold is not exceeded, either because the number of configuration options is low [Austin et al., 2013, Tucek et al., 2009] or because they do not interact much [Meinicke et al., 2016].

In the memory measurement of Figure 5.1, we observe the stack-of choices outperforms the choice-of-stacks already for three features. As expected, the choice-of-stacks implementation requires memory that is exponential in the number of configuration options. In contrast, the memory consumption of the stack-of-choices stays almost constant, independent of the number of involved configuration options. This is because adding more configuration options does not increase redundancy in the stack-of-choices, but instead just changes the context associated with each choice in the stack.

5.1.2 Distribution of Variational Operations

In Figure 5.1 we observe that the buffered implementations outperform their unbuffered counterparts. This is not surprising since in the generated sequences, 90% of operations are in the same variational context as their predecessor, which is exactly the situation the buffered optimization is intended to exploit. In the next experiment we attempt to measure the runtime and memory performance of this optimization with respect to each of our core variational stacks and to the ratio of sequential operations in the same context.

Experimental setup. For r from 0 to 100 in increments of 5, we artificially generate a sequence of 500 push/pop operations where exactly r% of sequential pairs of operations occur in the same variational context. We arbitrarily fix the variation space at six independent configuration options. As before, we constrain the operation sequences to avoid stack underflow errors. For each operation sequence, we measure the runtime of four variational stacks: the two core stacks and the two core stacks decorated by the buffered optimization. As before, each operation sequence is executed on each stack 10 times, choosing the fastest execution.

Results and analysis. The results of the experiment are presented in Figure 5.2. The independent variables are: (1) the ratio of sequential operations in the same variational context, plotted on the x-axis, and (2) the choice of stack, indicated by separate lines,

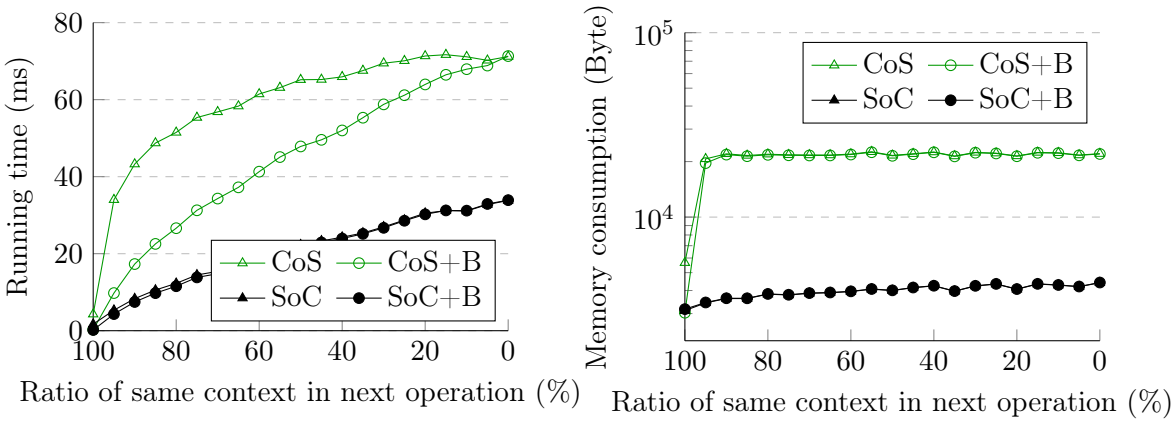


Figure 5.2: Comparing variational stacks on randomly generated operation sequences with different probability that an operation is executed in the same variational context as the previous operation.

labeled as before; the dependent variables are runtime and memory consumption of the operation sequence.

We observe that for the choice-of-stacks core implementation, the buffered optimization has a significant effect on runtime and remains profitable all the way until the ratio is nearly zero. In contrast, the buffered optimization has a very small effect for the stack-of-choices core implementation, and the effect nearly vanishes at a relatively high ratio of sequential operation in the same context. This reflects the fact that the stack-of-choices implementation already supports relatively well the scenario that the buffered optimization addresses; for example, pushing and popping a value in the same context will simply push and pop a corresponding choice from the stack. In contrast, the choice-of-stacks implementation would require splitting and copying all variant stacks on the push operation, even if the next pop operation is in the same context (rendering the copied variants irrelevant).

In the graph of memory consumption in Figure 5.2, we can see that the memory required by choice-of-stacks is significantly higher than the memory required by stack-of-choices for all ratios except for 100%. For all ratios below 100%, the memory consumption of choice-of-stacks is approximately 21 kilobytes since the stack has to represent all 2^6 variant stacks. The stack-of-choices is again more efficient since common values are shared

across variant stacks. For both implementations, the buffered-stack optimization has only minimal effects on memory consumption.

Neither of the experiments in this subsection analyzed variational stacks containing the hybrid optimization described in Section 4.4. The improved performance of hybrid stacks depends on whether variability occurs at all in each stack instance. This is something that is best measured on real-world examples, which we do in the next subsection.

5.2 Variational Stacks in VarexJ

In this subsection we consider how our variational stacks perform in practice, when used as the operand stack in the variational Java interpreter VarexJ (see Section 2.4). We use VarexJ to execute all configurations of four systems that have previously been used as benchmarks in research on configurable software: the systems Email [Hall, 2005] and Elevator [Plath and Ryan, 2001] are small academic Java programs that were designed with many interacting configuration options, ZipMe[3] is a small open-source library for accessing ZIP archives, and GPL [Lopez-Herrejon and Batory, 2001] is a small-scale configurable graph library often used for evaluations in the software product line community.

	Choice-of-stacks				Stack-of-choices			
	Core	+H	+B	+HB	Core	+H	+B	+HB
Email	556.8	492.0	499.7	517.8	508.1	502.6	506.2	501.7
Elevator	792.2	645.3	610.2	596.4	786.3	651.2	643.7	582.8
ZipMe	8 385.6	4 687.7	5 244.7	4 549.8	6 482.7	4 599.3	5 077.6	4 588.8
GPL	35 962.1	21 043.5	25 866.5	20 466.0	29 188.5	20 822.5	25 637.8	20 713.3

Table 5.1: Running time (ms) of each variational stack used as the operand stack in VarexJ while executing different variational programs. Each row measures the overall running time of VarexJ using each stack in-situ.

Experimental setup. We use VarexJ to execute each of the four systems, using each of the eight possible variational stacks described in Chapter 4 as VarexJ's variational operand stack. For each combination of system and stack, we configure VarexJ to use

[3] https://sourceforge.net/projects/zipme/

	LOC	Opt	Conf	Stacks	λH	λB	λHB
Email	644	9	40	4 938	195	21	18
Elevator	730	6	20	14 154	1 772	1 499	1 454
ZipMe	2 827	15	10	76 392	93	169	28
GPL	662	15	146	533 162	7 345	500	497

Table 5.2: Overview of four variational programs. Columns indicate: lines of code (LOC), number of configuration options (Opt), and total number of configurations (Conf) for each example; the total number of operand stacks created while executing the example in VarexJ (Stacks); and the failure rates for the hybrid optimization (λH), buffered optimization (λB), and both optimizations combined (λHB).

the corresponding stack implementation as its operand stack, then use VarexJ to execute all configurations of the system 10 times, choosing the fastest execution.

Additionally, we count how many total operand stacks are created during the execution of each system. For each stack implementation that includes either the hybrid or buffered optimization, we count how many times these optimizations *miss* during the execution of each system. For the hybrid optimization, the optimization misses when a variational operation is first performed on a particular operand stack. For the buffered optimization, the optimization misses when an operation is first performed on some operand stack in a different variational context than the previous operation.

Results and analysis. The runtime results are presented in Table 5.1, while Table 5.2 presents some basic characteristics of each system: lines of code, number of configuration options, and number of unique configurations. Table 5.2 also shows the number of operand stacks created during the execution of each system in VarexJ, the miss rates for each optimization in isolation (columns λH and λB), and the miss rate of both optimizations combined (λHB).

In the runtime results, we observe that the stack-of-choices core implementation outperforms the choice-of-stacks core implementation for all systems. More interestingly, we observe that both optimizations are highly relevant in practice—either optimization alone produces substantial speedups with the choice-of-stacks implementation, and including both optimizations renders the choice of core stack implementation moot. This suggests

that the optimizations capture an overwhelming majority of the cases in this application scenario. This observation is confirmed by the miss rates in Table 5.2. A core stack is created only when the included optimizations miss, and we observe that the miss rate for both optimizations combined (λHB) is less than 1% of the total operand stacks created in 3 out of 4 systems. The exception is the Elevator system, which was specifically designed to exhibit many interactions [Plath and Ryan, 2001], and has a miss rate for the combined optimizations of approximately 10%. This still seems low enough that the choice of core stack is insignificant. For larger applications we expect even higher success rates for the optimizations since such systems tend to have fewer interactions between configuration options and lower variation density than our test cases [Meinicke et al., 2016].

Overall, the results demonstrate that choosing the right implementation of a variational data structure can have a significant impact on the overall performance of a variational computation, even if that data structure is only a relatively small part of the tool, as is the case for the operand stack in VarexJ. For example, when executing GPL, switching from the choice-of-stacks to the stack-of-choices implementation saves 19% of the overall runtime and adding both optimizations saves overall 43% of the runtime.

Chapter 6: Variational Priority Queues

In this chapter, we introduce another useful variational data structure: variational priority queues. Like variational stacks, variational priority queues can be represented in a number of ways. Each variational priority queue implements the interface shown in Listing 6.1. The `push` function and `popMin` functions take an argument `ctx` to indicate in what variational context the operations are performed. Unlike the stacks section, we focus on two different implementations of `popMin` called `popMin(ctx)` and `popMinImpl()`. The `popMinImpl()` method itself decides in what variational context the operations are performed and users can not retrieve the variational data by providing a context. In the rest of section, we will show how the two different implementations of `popMin` affect the performance of variational priority queues.

```
1 public interface VPQ{
2    public void push(int v, FeatureExpr ctx);
3    public Conditional<Integer> popMin(FeatureExpr ctx);
4    public Conditional<Integer> popMinImpl();
5 }
```

Listing 6.1: Interface for variational priority queues

6.1 Implementation

The simplest implementation of variational priority queues is using `Conditional<PriorityQueue>`. This implementation is simple and uses the same technique as the the implementation of choice-of-stacks. We will skip the redundant explanation and implementation. Instead, we provide an idea of the design considerations for implementing variational data structures with holes. Since holes in the return value may need to be filled, the underlying implementation should be able to find what parts should be filled. For example, the underlying implementation of stack-of-choices is an array, which always stores the potential parts for

filling the holes in the next adjacent position. Then we can simply use a for loop to find the parts for filling holes from top to bottom.

Similarly, when implementing variational priority queues with holes, two aspects should be considered. First, we need to pick the right data structure, which should be able to find the parts for filling the holes in the return value. Second, the elements in the data structure need to preserve the priority information and the `ctx` information to indicate in what context that priority exists. Next, we show how to design a possible implementation of variational priorities queues.

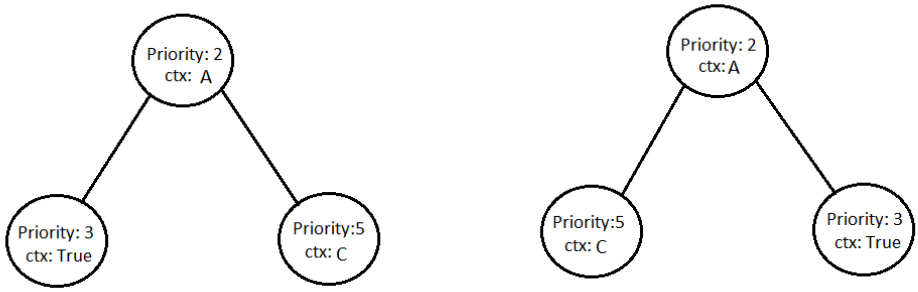


Figure 6.1: Two possible ways for representing variational priority queues using a heap

We started with heap as an underlying data structure, since a heap is a common way to implement plain priority queues. However, we found that a heap is not a good choice. For example, Figure 6.1 shows two possible ways of representing the same variational priority queue using a heap. Each node in the heap preserve the priority information and the `ctx` information. When calling `popMin(`$\neg A$`)`, it should return 3 as result. However, using the heap implementation, we cannot efficiently locate the parts for filling the holes since they can be stored in the left child or right child as shown in Figure 6.1. A list sorted by priorities can solve this problem since it can find the parts for filling holes easily as shown in Figure 6.2. But inserting a priority into this sorted list takes $O(n)$ complexity. Thus, a balanced binary search tree seems more appropriate for our needs. First, new priorities can be inserted in O(log n) complexity. Second, it maintains an order, the inorder traversal of a BST is an increasing sequence, which makes the hole-filling

algorithm easy. Next, we show the idea of how to represent variational priority queues by a balanced BST.

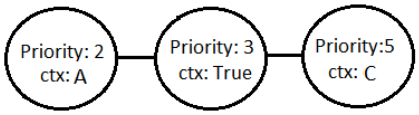


Figure 6.2: Representing variational priority queues using a list

Figure 6.3a shows the implementation of variational priority queues using a balanced BST. To support duplicate priorities under the same context, we can change the `FeatureExpr` in Figure 6.3a to a conditional value that represents the number of the same priorities under different configurations, as shown in Figure 6.3b. Now, each node in Figure 6.3b records a priority and the `count` information. For example, the conditional value $Choice(C, 2, 1)$ in priority 5 states that there exists 2 priorities 5 under the context C and one priority 5 under the context $\neg C$.

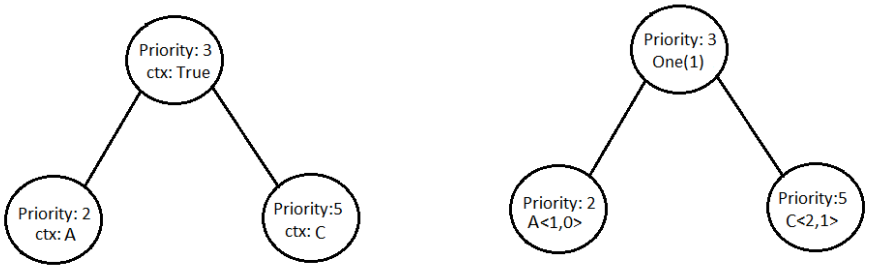


Figure 6.3: Two ways for representing variational priority queues using a BST (a) An BST implemenataion of variational priority queues (b) BST implementation with duplicate priorities under the same context

In the next section, we detail the implementation of variational priority queues using a balanced binary search tree.

6.2 Insertion and Deletion

The insertion and deletion operations cause the priorities in the tree to change. The data structure must be modified to reflect this change. As we shall see, modifying the tree to insert a new priority is relatively straightforward, but handling deletion is somewhat more intricate.

Insertion. Listing 6.2 shows the implementation of the push operation for a variational priority queue. We use the Java built-in library `TreeMap` to implement our variational priority queue. TreeMap is Red-Black tree based implementation which supports insertion and deletion in $O(logn)$ time. The key in the map stands for the priority and the value stands for the count, as shown in Line 1. Lines 2-8 shows how to push priorities into this variational priority queue. There are two cases. First, if the variational priority queue already exists the same priority, it only needs to change the counts recorded in the corresponding node by using the `vadd` function introduced in Section 2.2. Second, if the variational priority queue does not contain such priority, the algorithm creates a new node and puts it into the map.

```
1 private Map<Integer, Conditional<Integer>> map = new TreeMap();
2 public void push(int v, FeatureExpr ctx) {
3    Conditional<Integer> c = map.get(v);
4    if(c == null) {
5       c = new One<>(0);
6    }
7    c = vadd(c, ctx, 1);
8    map.put(v, c);
9 }
```

Listing 6.2: Implementation of the push function

Deletion. The `popMin(ctx)` operation is a bit more complicated than inserting a node. Before introducing the `popMin(ctx)` implementation, it is helpful to understand how

`peek(ctx)` works, as shown in Listing 6.3. The `peek(ctx)` operation is using the same technique as the `pop(ctx)` operation of stack-of-choices in Section 4.2. The only difference is that `peek(ctx)` does not remove the corresponding values in the data structure.

```
1  public Conditional<Integer> peek(FeatureExpr ctx) {
2      List<Integer> lc = new ArrayList();
3      List<FeatureExpr> lf = new ArrayList();
4
5      for(Map.Entry<Integer, Conditional<Integer>> e : map.entrySet()) {
6         Conditional<Integer>[] s = e.getValue().split(ctx);
7         if(s[0].equals(One.ZERO)) continue;
8         FeatureExpr fe = s[0].getFeatureExpr(0);
9         lc.add(e.getKey());
10        lf.add(fe.not());
11        ctx = ctx.and(s[0].getFeatureExpr(0));
12        if(ctx.isContradiction()) {
13           break;
14        }
15     }
16     Conditional<Integer> ret = (One<Integer>)One.NULL;
17     for(int i = lc.size() - 1; i >= 0; --i) {
18        ret = ChoiceFactory.create(lf.get(i), One.valueOf(lc.get(i)), ret);
19     }
20     min = ret;
21     return min;
22 }
```

Listing 6.3: Implementation of the peek(ctx) function

Then the following steps show the procedure for `popMin(ctx)`, which is based on the `peek(ctx)` method:

1. First call the `peek(ctx)` function to get the result `ret`. The variable `ret` records the priorities that need to be popped.

2. After retrieving `ret`, the algorithm needs to decrease the corresponding counts. Listing 6.4 shows how to decrease counts by using a `flatMap`. If the count is changed to `One(0)` in Line 7, it means this priority does not exist in any context. Then, the

node in the map will be removed. Otherwise, it calls the `vadd` function to decrease `1` for each variant that popped.

Since the `peek(ctx)` needs to fill the holes until there are no holes in the return value, the worst case complexity for `popMin(ctx)` is $O(n)$.

```
1  public Conditional<Integer> popMin(FeatureExpr ctx) {
2     Conditional<Integer> ret = peek(ctx);
3     ret.flatMap((FeatureExpr f, Integer v)->{
4           if(v != null && !f.isContradiction()) {
5              Conditional<Integer> c = map.get(v);
6              c = vadd(c, f, -1);
7              if(c.equals(One.ZERO)) {
8                 map.remove(v);
9              } else {
10                map.put(v, c);
11             }
12          }
13          return (One<Integer>)One.NULL;
14    });
15    return ret;
16 }
```

Listing 6.4: Implementation of the popMin(ctx) function

Compared to the `popMin(ctx)` operation, the implementation of `popMinImpl()` for variational priority queues is simple. It does not need to traverse the data structure. One implementation we provide here is to return one single piece of the value with the highest priority as shown in Listing 7.3. First, on Line 5, we check the count in the leftmost node to get the context the priorities exist, then simply return the priorities under that context.

```
1  public Conditional<Integer> popMinImple() {
2      Map.Entry<Integer, Conditional<Integer>> e = map.firstEntry()
3      if(e == null) return (One<Integer>)One.NULL;
4     Integer t = e.getKey();
5     FeatureExpr fe = e.getValue().getFeatureExpr(0).not();
6     Conditional<Integer> nv = vadd(e.getValue(), fe, -1);
7     map.put(t, nv);
```

```
8     return ChoiceFactory.create(fe, One.valueOf(e.getKey()),
          (One<Integer>)One.NULL);
9 }
```

Listing 6.5: Implementation of popMinImpl()

Chapter 7: Variational Priority Queues Evaluation

In this chapter, we present two sets of experiments to analyze the performance of the two different ways implementing priority queues. In Section 7.1, we analyze how individual properties of data (numbers of features) influence the performance of variational heapsort using the two approaches.

In the second set of experiments, described in Section 7.2, we evaluate the performance of variational priority queues in a real-world airlines example.

7.1 Heapsort

Since our underlying implementation is a balanced BST, the data is sorted when pushing elements into the variational priority queues. Thus, this experiment is actually directly comparing the efficiency between two approaches to add and retrieve data as we defined in Section 6.2.

Experimental setup. For each number of configuration options from 1 to 15, we artificially generate a variational list with 50 elements and push the variational list to our variational priority queues using two different implementation. For a balanced BST implementation, we provide two approaches, `popMin(ctx)` and `popMinImpl()`, to get the results. For `Conditional<PriorityQueue>` implementation, we provide the `popMin(ctx)` method. For each variational list we generated, we measure the runtime of those three approaches by computing the fastest over 10 executions of heapsort.

Results and analysis. The results of the experiment are presented in Figure 7.1. The independent variables are: (1) the number of configuration options, plotted on the x-axis, and (2) the three implementations: `popMin(ctx)` and `popMinImpl()` using a balanced BST implementation and `popMin(ctx)` using `Conditional<PriorityQueue>` implementation, indicated by separate lines. The dependent variable is the runtime of variational heapsort

on a generated variational list.

For a balanced BST implementation, the runtime of the `popMin(ctx)` method increases with the number of configuration options since the time of context computations and the number of features are positively correlated. Comparatively, the `popMinImpl()` is more stable since it avoids most context computations. However, `popMinImpl()` can only be used under the conditions described in Section 3.2.2.

Compared to `Conditional<PriorityQueue>` implementation, the `popMinImpl()` approach in a balanced BST implementation outperforms the `Conditional<PriorityQueue>` implementation at two features and the `popMin(ctx)` approach in a balanced BST implementation outperforms `Conditional<PriorityQueue>` implementation at 7 features.

7.2 Shortest path problems

In this section, we solve the shortest path problem on a variational graph using a Variational Dijkstra's algorithm. To do so, we first define a variational graph and introduce how to change our general implementation of variational priority queues to key-value pair priority queues. Then, evaluate how the two ways, `popMin(ctx)` and `popMinImpl()`, affect the performance of the variational priority queues when used in Variational Dijkstra's algorithm.

The original shortest path problem is to find the minimum distance between two vertices in a graph. There are several algorithms for solving this problem. Dijkstra's algorithm is one of the most famous algorithms and its classic implementation uses priority queues. Before discussing how a Variational Dijkstra's algorithm works, it is important to recall how Dijkstra's algorithm works. We shall break the procedure of Dijkstra's algorithm into three major steps. First, the algorithm assigns all vertices in a graph as infinity distance except the source as 0 and marks all vertices as unvisited. Second, we shall get an unvisited vertex v with the minimum distance by using a priority queue. Third, if v is the same as the destination, the algorithm stops. Otherwise, it marks v visited and traverses all its neighbors and calculates the new distances. Each time the algorithm compares the new calculated results with the current distance for neighbors. If the new value is smaller than the current value, it updates the current value to the new value.

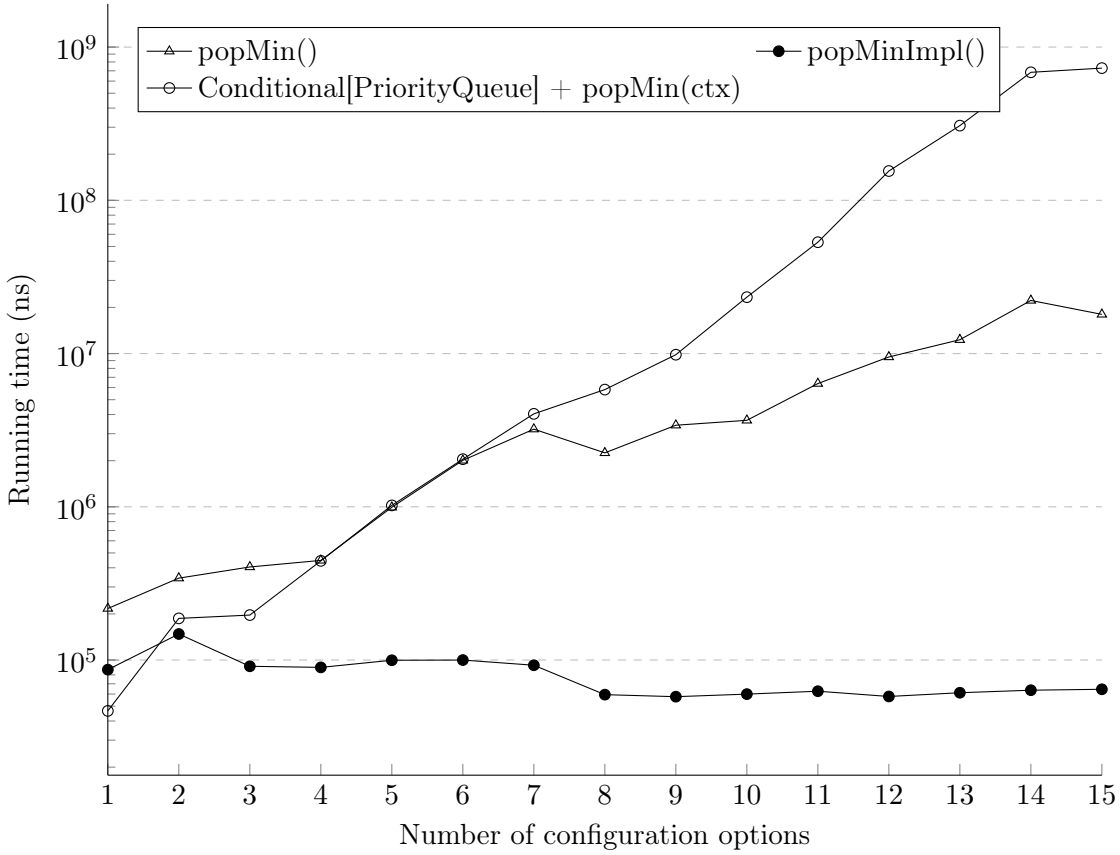


Figure 7.1: Comparing variational heapsort on artificially generated variational lists with different numbers of configuration options that may appear in the variational contexts.

Then the algorithm will jump to step 2 until getting the destination.

Next, we define a variational graph and a key-value variational priority queue for Variational Dijkstra's algorithm.

7.2.1 Variational Graph

Analogous to variational stacks, variational graphs do not only represent one graph but represents several different plain graphs [Erwig et al., 2013]. In a variational graph, the variation can exist in vertices or edges. In this chapter, we only consider the variational graphs with variations on edges. In a plain graph, an edge is represented as $e = (v, u, w)$, which means an edge from v to u with weight w. For building a variational graph, we can simply add the context to each edge as $e' = (v, u, w, ctx)$, which means the edge exists under `ctx`. Consider the example in Figure 7.2. In this graph, if we select the context B, there exist one edge with weight 60 from vertex 1 to vertex 3. Consider the shortest path from vertex 1 to vertex 5, selecting $A \wedge B$ makes the shortest distance 100, while the result will be 460 under $A \wedge \neg B$. There is no path found from node 1 to node 5 under the context $\neg A \wedge \neg B$ or $\neg A \wedge B$.

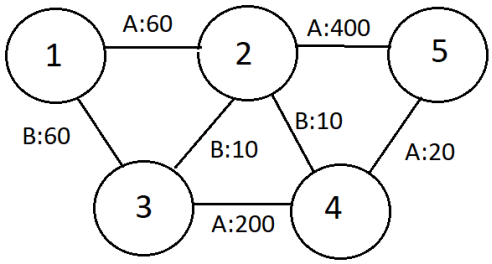


Figure 7.2: One simple eaxample of variational graphs with all variations on edges.

7.2.2 Key-Value Variational Priority Queue

Since the priority queue in Dijkstra's algorithm needs to preserve the vertex information, we also need change our general variational priority queue to a key-value priority for supporting storing more information such as name or id in the variational priority queue. Listing 7.1 shows the interface for key-value variational priority queues. `update` in Line 2 is to update the smaller shortest distance under certain `ctx` to the source and `popMin(ctx)` is for getting nodes with the shortest distance from source under `ctx`. We can also provide the `popMinImpl()` method since Dijkstra's algorithm is supposed to run all variants independently and consume all the data until it reaches the destination. The return value of `popMin(ctx)` and `popMinImpl()` is a list of triples since several vertices can have the same least distance from the source. The triple is a class which can store three information. Here we use it to store the context `ctx`, priority, and the node label which has type of FeatureExpr, Integer, and T receptively.

```
1 public interface IVPriorityKey<T> {
2   public void update(FeatureExpr ctx, final T k, final Integer p);
3   public List<Triple<FeatureExpr, Integer, T>> popMinImpl();
4   public List<Triple<FeatureExpr, Integer, T>> popMin(FeatureExpr f);
5 }
```

Listing 7.1: Interface of key-value variational priority queue

Suppose the variational graph does not contain duplicate vertices, we only need to replace the count by a variational set to record all the vertices that have the same distance from the source. Listing 7.2 shows the underlying representation of variational priority queues. Now, `priorityTable` maintains a priority and a variational set. For example, after visiting the vertex 1 in the priority queue in Figure 7.2, the leftmost node in the priority queue will be a $(60, \{2 \rightarrow A, 3 \rightarrow B\})$, which means the distance from the source to vertex 2 under A and the distance to vertex 3 under B are the same, namely, 60. `keyTable` is an index table which can quickly locate the corresponding node's priority (distance) to enable updates. For example, when updating the nodes in priority queues, the old distance in the variational priority queue should be deleted. We can quickly find the node in the variational priority queue with the old distance using the index table.

```
1 public class VPriorityKey<T> implements IVPriorityKey<T> {
```

```
2    Map<T, Conditional<Integer>> keyTable = new HashMap();
3    TreeMap<Integer, VHashSet<T>> priorityTable = new TreeMap();
4    ...
5  }
```

Listing 7.2: Implementation of variational priority queue

```
1  private Map.Entry<Integer, VHashTable<T>> popMinImpl() {
2    Map.Entry<Integer, VHashTable<T>> e = priorityTable.firstEntry();
3    if(e == null) return null;
4    removeKeyTable(e.getValue());
5    priorityTable.remove(e.getKey());
6    return e;
7  }
```

Listing 7.3: Implementation of the popMinImpl() function

The implementation of `popMin(ctx)` for a the key-value priority queue is almost the same as the `popMin(ctx)` method we defined in Listing 6.4 except it needs to modify the index table at the same time. We will not show the redundant code here. For the `popMinImpl()` implementation, we can also use the same technique as before, which only returns the piece of data in the leftmost node. However, this implementation can be optimized after observing a property of priority queues in Dijkstra's algorithm. Because the minimum distance from the source is continuously increasing, the elements popped from the priority queue are monotonically increasing. Based on this observation, we can pop all elements with the minimum priority, which optimizes the `popMinImpl()` implementation from popping one element in the leftmost node to popping all elements in the leftmost node since the elements have the same shortest distance from the source vertex. So, popping everything in the leftmost node in such a situation is safe. Listing 7.3 shows the implementation of `popMinImpl()`. When we remove the leftmost node, we also need to make the corresponding changes in the index table as shown in line 4.

Listing 7.4 shows an implementation of a variational update function. Basically, given an object, the update function first gets the shortest path information from the `keyTable` and compares the priority with the given priority. If the priority of the give priority is higher than the current priority, it will update the priority and update the corresponding entry in the `keyTable`.

```
1  public void update(FeatureExpr ctx, final T k, final Integer p) {
2       if(ctx.isContradiction()) return;
3       Conditional<Integer> cp = keyTable.get(k);
4       cp = cp.flatmap((FeatureExpr f, Integer v)->{
5           if(f.and(ctx).isContradiction() || v != null && p.compareTo(v) >= 0) {
6               if(v == null) return One.NULL;
7              return One.valueOf(y);
8           }
9          if(v != null && priorityTable.get(v) != null) {
10            priorityTable.get(v).remove(ctx, k);
11            if(priorityTable.get(v).isEmpty()) {
12               priorityTable.remove(v);}
13         }
14         if(priorityTable.get(p) == null) priorityTable.put(p, new
               VHashTable());
15          priorityTable.get(p).put(ctx, k);
16          if(v == null) {
17            return ChoiceFactory.create(ctx, One.valueOf(p),
                  One.NULL).simplify();}
18         return ChoiceFactory.create(ctx, One.valueOf(p),
              One.valueOf(y)).simplify();});
19      keyTable.put(k, cp);
20 }
```

Listing 7.4: Implementation of the update functions

7.2.3 Variational Dijkstra's algorithm

After introducing the implementation of key-value priority queues and variational graphs, we can translate Dijkstra's algorithm to Variational Dijkstra's algorithm. Variational Dijkstra's algorithm is similar to Dijkstra's algorithm except the algorithm will run under different contexts and maintains the solutions for all variants.

Listing 7.5 shows the implementation details of Variational Dijkstra's algorithm. It includes a variational graph and ids for the source (s) and destination (t). `running` in Line

4 has type `FeatureExpr` which stands for the environment the user specifies. For example, setting running as $\neg A$ means the algorithm will not compute the shortest path under $\neg A$. `vpq` in Line 5 stands for a variational priority queue, which has type `IVPriorityKey` and the key in map `allDist` is for recording all visited nodes. The value in the map is the shortest distant under different contexts.

```
1 public class VDijkstra {
2   public VGraph graph;
3   public int s, t;
4   public FeatureExpr running;
5   public IVPriorityKey<Vertex> vpq;
6   public Map<Vertex, Conditional<Integer>> allDist;
7 }
```

Listing 7.5: The underlying implementation of VDijkstra

Listing 7.6 shows the implementation of Variational Dijkstra's algorithm using two different implementations. The code for each version is the same except for calling the different functions `popMin(ctx)` or `popMinImpl()` in line 4. We use a slash to indicate the two different implementations at line 4.

First, we will get a list of vertices which have the same shortest distance from the source in Line 4. For convenience, we use an iterator for the list returned from the variational priority queue. Then, we can get the `ctx`, distance, and node information from the triple as shown from line 9 to Line 11. If the conjunction of `ctx` and `running` is a contradiction, which means that this edge is not to be considered, we will discard the computations on that vertex. For example, setting the `running` context as $\neg A$ means to not consider the vertices which can only be reached under the context A . If we meet an edge which is labeled A, the algorithm will discard that vertex since $A \wedge \neg A$ contradicts.

Line 14 to 17 show the case that it finds the destination. For a plain graph, the algorithm stops. However, for a variational graph, we have only finished the part under the context `ctx`. Thus, we need to set the `running` environment as the conjunction of `running` and $\neg$`ctx`. This means we already found the shortest path under the `ctx` and do not need to consider the case under `ctx` in later steps.

If we do not reach the destination, we get the node's neighbors from the graph in Line 19

and traverse all the neighbors to update the shortest path from Line 20 to Line 28. For each neighbor node, `curCtx` in Line 21 records the context environment which is initialized by the conjunction of `running` and the condition on the edge to the neighbor.

In Line 23 and 24, we use the `vmin` function to compare the new distance and the recorded distance of the neighbor node which is retrieved from the map `allDist`. Meanwhile, the algorithm collects the context that its variants changed during the `vmin` function as `trackCtx`. Thus, `trackCtx` records the configurations in which a shorter path was found. If `trackCtx` is not null, we will update the priority queue. Last, we also need to update the `allDist` map to update the shortest distance we already found.

Note that this algorithm only calculate the shortest distance from the source. Once we found the destination, we can either print the results or use another data structure to store the results. It does not record the whole path information. However, we can easily use a hashmap, which has type of `Map<Vertex, Conditional<Vertex>>`, to record the previous nodes and record the path. The implementation is available at https://github.com/lambda-land/VPriorityQueue

```
1  public void run(FeatureExpr running) {
2     while(!running.isContradiction()) {
3        Iterator<Triple<FeatureExpr, Integer, Vertex>> e ;
4         e = vpq.popMin(running)/vpq.popMinImpl();
5        if(e == null) break; //No path found
6
7        while(e.hasNext()) {
8           Triple<FeatureExpr, Integer, Vertex> triple = e.next();
9           FeatureExpr ctx = triple.t1;
10          int curDist = triple.t2;
11          Vertex vertex = triple.t3;
12          if(running.and(ctx).isContradiction())  continue; // discard
13
14          if(vertex.id == t) {
15             System.out.println(running.and(ctx) + " " + currDist);
16             running = running.andNot(ctx);
17             continue;}
18
19          Map<Integer, Vertex> vertice = graph.nodesMap(vertex.id); // get
                neighbors
20          for(Edge edge : vertex.edge) {
21             FeatureExpr curCtx = ctx.and(edge.fe).and(running);
22             if(!curCtx.isContradiction() & !allDist.containsKey(al.u)) {
23                Conditional<Integer> tmp = vmin(curCtx, allDist.get(edge.u),
                      curDist + al.weight);
24                FeatureExpr trackCtx = disjunction of the features that its value
                      changed in tmp during vmin function.
25                if(trackCtx != null) vpq.update(trackCtx.and(curCtx), al.u, y);
26                allDist.put(al.u, tmp);}
27          }
28       }
29    }
30 }
```

Listing 7.6: Variational Dijkstra's algorithm

7.3 Case Study

In this subsection we consider a real world airlines example and compare the performance of different versions of Variational Dijkstra's algorithms. Table 7.1 shows the data format of the airline information. It has the unique carrier and flight number information listed in the second and third column. Also, it provides the departure city id, departure time, the destination city id, and the arrival time. There are 14000 rows for one day. For simplicity, we suppose the flight information is the same every day. Given a departure time and city id, our goal is to find the earliest flight to the destination by specifying different carriers.

DATE	CARRIER	FLNUM	ORIGIN_AIRPROT_ID	DEST_ID	DEP_TIME	ARR_TIME	CRS_ELAPSED_TIME
2017-01-01	AA	1766	13930	11298	710	948	158
2017-01-01	AA	1767	12889	11298	1235	1718	163
2017-01-01	AA	1768	11278	13303	1020	1313	173
2017-01-01	AA	1769	15304	13303	730	833	63
...	...	...	...	...	...	...	...
2017-01-01	UA	260	11292	14747	755	958	183
2017-01-01	UA	261	12892	13204	1037	821	284

Table 7.1: The original data from airlines

DATE	CARRIER	FLNUM	ORIGIN_AIRPROT_ID	DEST_ID	DEP_TIME	ARR_TIME	CRS_ELAPSED_TIME
1/1/17	AA	1766	1	2	100	150	60
1/1/17	AA	1767	1	3	100	350	180
1/1/17	UA	1768	2	3	200	250	60
1/1/17	AA	1769	2	4	200	650	300
1/1/17	AA	1770	3	4	300	350	60
1/1/17	AA	1773	3	4	400	450	60

Table 7.2: A small example

This problem is a shortest path problem in a variational graph. To solve this problem, we first build a variational graph illustrated in Section 7.2.1 from the data. For each city c_i, let $T_c = \{t_{i1}, \ldots, t_{ik}\}$ be all possible arrival times at c_i. We create k vertices $\{c_{i1}, \ldots, c_{ik}\}$ for all arrival time at c_i. For each flight (c_i, d, c_j, a), d and a stands for the departure time and arriving time separately, if $t_{ix} < d$ and $t_{jy} = a$, then we create an edge between c_{ix} and c_{jy}, the weight of the edge is $t_{jy} - t_{ix}$, and the context of the edge is the carrier. For example, suppose travelers plan to depart from vertex 1 at 0:50 to destination 4, we can build the data in Table 7.2 to a variational graph as shown in Figure 7.3.

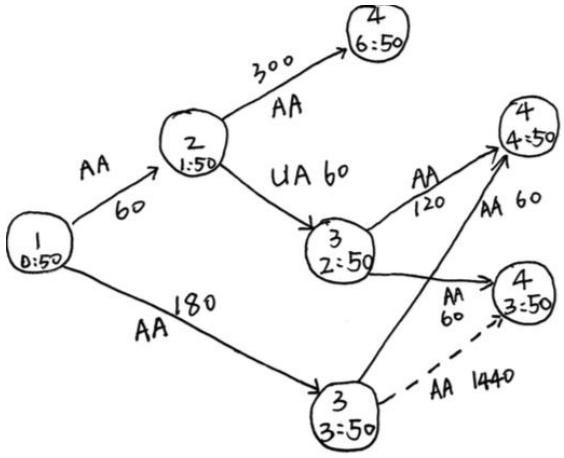


Figure 7.3: A variational graph for representing the example in Table 7.2

Each edge records the distance and the carrier encoded as a feature. For example, the vertex 2 can be reached by taking AA airline at 1:50. So, we create an edge with feature AA and distance 60 between node (1, 0:50) and node (2, 1:50). Using the same technique, we can build a graph with 28531 vertices and 24376211 edges from Table 7.1.

7.3.1 Comparison

In this section, we present two sets of experiments to compare the performance of variational Dijkstra's algorithm using the `Coniditional<PriorityQueue>` implementation and the balanced BST implementations. Moreover, we analyze how the two ways of retrieving data impact the performance of the balanced BST implementation. In the first set of experiments, we fix the number of configuration options, and evaluate the algorithm by specifying the `running` context. In the second set of experiments, we fix several `running` contexts and evaluate how the number of configuration options influences the performance of Variational Dijkstra's algorithm.

7.3.2 Different Running Context

Experimental setup. We first create a variational graph from the airlines data. When creating the graph, we can control the number of features on the graph by simply introducing a *Other* feature to represent the airlines we do not consider. For this experiment, we arbitrarily fix the variation space at six independent configuration options and randomly choose the start city id and destination city id. The features we considered are: United Airlines(UA), American Airlines(AA), Delta Airlines(DL), SkyWest Airlines(OO), Hawaiian Airlines(HA) and *Others*. The departure city id and destination city id are Dallas, TX(11298) and Philadelphia, PA(14100). Suppose we depart at 23:00 pm, we are trying to get the earliest flight by running the Variational Dijkstra algorithm on it with customer's preference (specifying the different `running` context).

Results and analysis.

The results of the experiment are presented in Figure 7.4. The independent variables are: (1) the different `running` context, plotted on the x-axis, and (2) the three implementations, indicated by separate bars. The dependent variable is the runtime of the Variational Dijkstra's algorithm on the graph.

Figure 7.4 demonstrates that the `popMinImpl()` method can improve the performance especially when users consider all possibilities. For example, the `popMinImpl()` uses less than half of the time that `popMin(ctx)` and the naive implementation use in Figure 7.4. The performance of `popMinImpl()` is comparatively stable and is hardly influenced by the `running` context. However, both the naive implementation and `popMin(ctx)` do well when the `running` context is more specific. For example, when we only consider the DL and HA carriers, the `popMin(ctx)` only takes 3 seconds to calculate the results, which is 2 times faster than the `popMinImpl()` method. Another observation is that the performance of the `popMin(ctx)` implementation is unstable and more influenced by a specific context that users specify. For example, the third column and the fourth column in Figure 7.4 both consist of two configuration options but the runtime is dramatically different. The naive implementation is more influenced by the number of features since the number of priority queues will exponentially increase as the features increase. For example, if we fix eight features and set `running` as `True`. The naive implementation will takes 839 seconds

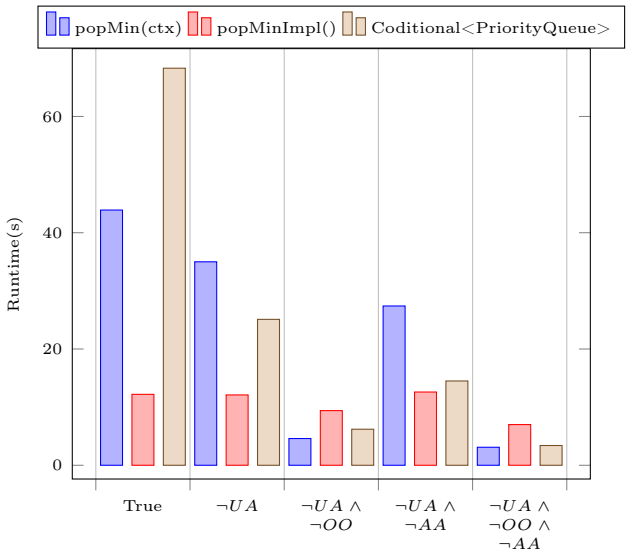


Figure 7.4: A performance comparison between three different implementations

which is 8 times slower than the `popMin(ctx)` and 40 times slower than the `popMinImpl()` method.

7.3.3 Number of Configuration Options

Experimental setup. Since we can control the number of configuration options on the graph, we run Variational Dijkstra's algorithm on different numbers of configuration options, 6, 8 and 10. We fix several different `running` contexts. Like the first experiment, we take the depart city id and destination city id as 11298 and 14100 and run Variational Dijkstra's algorithm on it.

Results and analysis. The results of the experiment are presented in the following tables. For each table, we compare the performance of three different implementations in different number of configuration options under a specific `running` context. The four figures correspond to four different `running` contexts: $True$, $\neg UA$, $\neg UA \wedge \neg OO$, and $\neg UA \wedge \neg OO \wedge \neg AA$.

Number of features	Conditional<PirorityQueue>	popMin(ctx)	popMinImpl()
6	104.3	42.0	14.4
8	839.9	135.4	20.3
10	17137.4	1030.4	26.0

Table 7.3: Comparing variational Dijkstra's algorithm with different configuration options under $True$ context

Number of features	Conditional<PirorityQueue>	popMin(ctx)	popMinImpl()
6	25.3	29.0	11.7
8	237.4	88.2	17.4
10	3669.1	434.5	21.1

Table 7.4: Comparing variational Dijkstra's algorithm with different configuration options under $\neg UA$ context

From those 4 tables, we can see that all implementations take longer time as the number of configuration options increase. However, the `popMin(ctx)` method and the naive implementation are more influenced by the increasing number of configuration options since the time of context computations and the number of configuration options are positively correlated. For 10 options and $True$ context, the naive implementation takes around 600 times slower than the `popMinImpl()` version since the naive implementation does not support sharing and the context computation becomes more complicated as features increase. Comparatively, `popMinImpl()` is more stable since it avoids most context computations. However, `popMinImpl()` does not work for all programs.

Overall, we analyze how different implementations affect the performance of programs. The `popMinImpl()` implementation is weakly influenced by the changes to the number of configuration options and the `running` context. It is more stable compared to the

Number of features	Conditional<PirorityQueue>	popMin(ctx)	popMinImpl()
6	6.4	4.3	5.0
8	46.8	17.2	11.6
10	653.0	75.4	14.8

Table 7.5: Comparing variational Dijkstra's algorithm with different configuration options under $\neg UA \wedge \neg OO$ context

Number of features	Conditional<PirorityQueue>	popMin(ctx)	popMinImpl()
6	4.3	3.5	4.7
8	23.8	16.5	12.0
10	222.6	55.8	15.6

Table 7.6: Comparing variational Dijkstra's algorithm with different configuration options under $\neg UA \wedge \neg OO \wedge \neg AA$ context

`popMin(ctx)` method but with some constraints on the program design. The `popMin(ctx)` method can work with all programs but its performance is more affected by the `running` context and the number of configuration options in graphs, it only works well when users have a very specific choice on airlines for example only considering one to three carriers.

Chapter 8: Conclusion

Variational data structures are needed to efficiently compute with variability in data and code. Toward a systematic understanding of the design space for variational data structures:

- We have presented a family of variational stack implementations. We evaluated the performance of these variational stacks when used as the operand stack in the variational interpreter VarexJ.
- We provided an optimization for retrieving data from variational data structures with holes and evaluated the performance of variational priority queues when used in variational Dijkstra's algorithm.

Both results demonstrate that the choice of variational data structure can have a significant impact on the performance of a program that computes with variability.

For variational stacks, variational priority queues and any other data structures with holes, we provide a common patten that can run with the two different ways as illustrated in Chapter 3. As future work we should explore some other situations which can run correctly with the two ways of retrieving data.

A distinguishing feature of the application domain targeted by VarexJ is that it involves the creation of many short-lived stacks, where relatively few contain variation in multiple different variational contexts (see Section 5.2). As future work we should evaluate the family of variational stacks in application scenarios that involve longer-lived variational stacks with different variability profiles.

Bibliography

[Austin et al., 2013] Austin, T. H., Yang, J., Flanagan, C., and Solar-Lezama, A. (2013). Faceted Execution of Policy-Agnostic Programs. In *Proc. Workshop Programming Languages and Analysis for Security (PLAS)*, pages 15–26. ACM.

[Chen and Erwig, 2014] Chen, S. and Erwig, M. (2014). Type-Based Parametric Analysis of Program Families. In *Proc. Int'l Conf. Functional Programming (ICFP)*, pages 39–51, New York, NY, USA. ACM.

[Chen et al., 2014] Chen, S., Erwig, M., and Walkingshaw, E. (2014). Extending Type Inference to Variational Programs. *ACM Trans. Programming Languages and Systems (TOPLAS)*, 36(1):1:1–1:54.

[Chen et al., 2016] Chen, S., Erwig, M., and Walkingshaw, E. (2016). A Calculus for Variational Programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*.

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

[Erwig and Walkingshaw, 2013] Erwig, M. and Walkingshaw, E. (2013). Variation Programming with the Choice Calculus. In *Proc. Generative and Transformational Techniques in Software Engineering*, pages 55–100, Berlin, Heidelberg. Springer.

[Erwig et al., 2013] Erwig, M., Walkingshaw, E., and Chen, S. (2013). An abstract representation of variational graphs. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*, pages 25–32. ACM.

[Hall, 2005] Hall, R. J. (2005). Fundamental Nonmodularity in Electronic Mail. *Automated Software Engineering (ASE)*, 12(1):41–79.

[Hubbard and Walkingshaw, 2016] Hubbard, S. and Walkingshaw, E. (2016). Formula Choice Calculus. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 49–57.

[Kästner et al., 2011] Kästner, C., Giarrusso, P. G., Rendel, T., Erdweg, S., Ostermann, K., and Berger, T. (2011). Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824, New York, NY, USA. ACM.

[Kästner et al., 2012] Kästner, C., von Rhein, A., Erdweg, S., Pusch, J., Apel, S., Rendel, T., and Ostermann, K. (2012). Toward Variability-Aware Testing. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 1–8, New York, NY, USA. ACM.

[Kim et al., 2012] Kim, C. H. P., Khurshid, S., and Batory, D. (2012). Shared Execution for Efficiently Testing Product Lines. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*, pages 221–230, Washington, DC, USA. IEEE.

[Liebig et al., 2010] Liebig, J., Apel, S., Lengauer, C., Kästner, C., and Schulze, M. (2010). An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 105–114, Washington, DC, USA. IEEE.

[Lopez-Herrejon and Batory, 2001] Lopez-Herrejon, R. E. and Batory, D. (2001). A Standard Problem for Evaluating Product-Line Methodologies. In *Proc. Int'l Symposium Generative and Component-Based Software Engineering (GCSE)*, pages 10–24, London, UK. Springer.

[Meinicke, 2014] Meinicke, J. (2014). VarexJ: A Variability-Aware Interpreter for Java Applications. Master's thesis, University of Magdeburg.

[Meinicke et al., 2014] Meinicke, J., Thüm, T., Schröter, R., Benduhn, F., and Saake, G. (2014). An Overview on Analysis Tools for Software Product Lines. In *Proc. Workshop Software Product Line Analysis Tools (SPLat)*, pages 94–101, New York, NY, USA. ACM.

[Meinicke et al., 2016] Meinicke, J., Wong, C. P., Kästner, C., Thüm, T., and Saake, G. (2016). On Essential Configuration Complexity: Measuring Interactions in Highly-Configurable Systems. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 483–494. ACM.

[Meng et al., 2017] Meng, M., Meinicke, J., Wong, C.-P., Walkingshaw, E., and Kästner, C. (2017). A choice of variational stacks: Exploring variational data structures. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*.

[Muslu et al., 2012] Muslu, K., Brun, Y., Holmes, R., Ernst, M. D., and Notkin, D. (2012). Speculative Analysis of Integrated Development Environment Recommendations. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 669–682. ACM.

[Nguyen et al., 2014] Nguyen, H. V., Kästner, C., and Nguyen, T. N. (2014). Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 907–918, New York, NY, USA. ACM.

[Plath and Ryan, 2001] Plath, M. and Ryan, M. (2001). Feature Integration Using a Feature Construct. *Science of Computer Programming (SCP)*, 41(1):53–84.

[Smeltzer and Erwig, 2017] Smeltzer, K. and Erwig, M. (2017). Comparisons and design guidelines. In *ACM International Workshop on Feature-Oriented Software Development*. To appear.

[Sumner et al., 2011] Sumner, W. N., Bao, T., Zhang, X., and Prabhakar, S. (2011). Coalescing Executions for Fast Uncertainty Analysis. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 581–590. ACM.

[Thüm et al., 2014] Thüm, T., Apel, S., Kästner, C., Schaefer, I., and Saake, G. (2014). A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 47(1):6:1–6:45.

[Tucek et al., 2009] Tucek, J., Xiong, W., and Zhou, Y. (2009). Efficient Online Validation with Delta Execution. *ACM SIGARCH Computer Architecture News*, 37(1):193–204.

[Walkingshaw, 2013] Walkingshaw, E. (2013). *The Choice Calculus: A Formal Language of Variation*. PhD thesis, Oregon State University.

[Walkingshaw et al., 2014] Walkingshaw, E., Kästner, C., Erwig, M., Apel, S., and Bodden, E. (2014). Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In *Proc. Int'l Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, pages 213–226. ACM.