AN ABSTRACT OF THE THESIS OF

Ghadeer Alkubaish for the degree of Master of Science in Computer Science presented on February 24, 2020.

Title: Integrating Side Effects in Variational Programs Using Algebraic Effects

Abstract approved: ______________________________

Eric Walkingshaw

Variational programming supports efficiently executing many related programs at once by encoding all of the programs in one "variational program" that captures the differences among them statically and explicitly. An open problem in variational programming is how to handle side effects—if two program variants perform different side effects, we cannot separate the effect of one variant from the other since the outside world is not variational. A potential solution is to create variation-aware execution environments for variational programs, for example, a variational file system that keeps track of file variants corresponding to program variants. However, it is infeasible to do this for all kinds of effects. Also, there are different ways to handle the interaction of effects and variation that are incompatible with each other, preventing a one-size-fits-all solution.

In this thesis, we argue that *algebraic effects* can be used to resolve the problem of combining variation and effects by enabling programmers to flexibly and incrementally extend a variational programming environment to handle new kinds of effects. We present a proof-of-concept prototype in the *Eff* programming language that demonstrates how a variational programming environment can be extended to support file input/output. Crucially, such extensions are done at the library level, which enables handling new kinds of effects and handling existing effects in multiple ways, both of which are essential in variational programming applications.

Integrating Side Effects in Variational Programs Using Algebraic Effects

by

Ghadeer Alkubaish

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented February 24, 2020
Commencement June 2020

ACKNOWLEDGEMENTS

First and foremost, I thank my God for all the blessings in my life.

Then, I would like to thank my wonderful loving husband Ali who has always been for me in my rough and good times. Ali became my family and best friend in my life away from home. He believed in me and did all he could to let me achieve my goals. I thank him for all support, understanding and encouragements. He is the one secret behind every success I make or challenge I overcome.

I thank my two adorable sons Hassan and Rida for being in my life - making it joyful, exciting and a bit more busy!

I thank my loving family: Mom, Dad, Sisters and Brothers. I appreciate that they are always standing up for me during all of my milestones, pushing me hard, trusting me, and cheering for every success I make. My special thanks to my great super Mom. She is the one person who believes in me the most. I thank her for her prayers, love and constant communication regardless of the time difference and long distance. I do what I do to make her proud.

I thank my great advisor Professor Eric Walkingshaw. I thank his high confidence in me, his inspiring words throughout my journey and his powerful encouragement which kept me going. I honor his time, lessons, feedback and enthusiasm. I appreciate his understanding and accommodations for my special needs and restricted schedule. I appreciate his respect to my values. I give him all the credit for me continuing my studies and accomplishing this work.

I thank Professor Martin Erwig who has been a main source of knowledge in the PL group. I appreciate everything I learned from him through classes or the reading group.

My very special thanks to my mentor Jed Irvine for the great sessions in improving my technical writing. He has effectively improved my skills, reviewed my paper, and gave me the time and effort I needed with great generosity.

I thank my PL research mates for the reading group meetings where we exchanged many great ideas. I specially thank my group mates Jeff, Parisa, Fariba and Qiaoran. I appreciate all of their feedback during the writing workshops.

I thank the contributors of the language *Eff*—Ziga Luksic, Matija Pretnar, and Andrej Bauer who helped me in editing the compiler for adding missing features of the language needed in my work.

I thank the great faculty members of OSU who I learned from or worked with. I thank Professor Prasad Tadepalli who I enjoyed working with on the GTA assignment project. I thank Professor Jennifer Parham-Mocello for her love and support. She is one of the first people who inspired me to proceed in this field. I thank the adviser Calvin Hughes for his unlimited advising and support. I thank Professor Glencora Borradaile for her support and empathy. She has always helped me through my difficult times in personal and graduate life and inspired me by her excitement in teaching and work. I will absolutely miss my many stops by her office.

I thank my very special friend Mahsa for her exceptional support, especially after moving out from Corvallis and starting to commute. I thank her for always checking on me, hosting me in her apartment during lunch time and cooking for me. In my hardest days during pregnancy and commuting, Mahsa has been there for me.

I thank my friends in Corvallis and Portland. I appreciate all of the great times we spend together. I thank my very special extended family who always provide me with love and support.

I thank OSU for the GTA position which provided me with teaching skills and secured me financially. I also thank SACM for the scholarship which provided financial stability plus child care support.

I thank my committee members—Eric Walkingshaw, Martin Erwig, Arash Termehchy, and Vergara Hector for their time, willingness to evaluate my work, and feedback.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

Chapter 1: Introduction

Variational programming [10, 7] is an emerging paradigm for representing and computing with explicit variation in code and data. It is a generalization of the ideas underlying *faceted execution* [2, 24, 4, 20, 3, 19] for enforcing dynamic information-flow security and *variability-aware execution* [16, 17, 14] for testing software product lines, and has a range of other applications across computer science such as model checking [8]. However, integrating side effects (e.g. state updates, exceptions, file I/O, standard input/output, and database queries) in variational programs is challenging because the execution environment of these programs is not variational. Therefore, we cannot separate the effect of one variant from the other which causes effects to be performed in wrong contexts (the words effect and side effect are interchangeably used in this thesis).

In this thesis, we argue that *algebraic effects* can be used to resolve the problem of combining variation and effects by enabling programmers to extend variational programming environments to handle new kinds of effects or handle existing effects in different ways. We present a proof-of-concept prototype in the *Eff* programming language that demonstrates how the variational programming environment can be extended to support file input/output.

In this chapter, we discuss the insight, formalization, and execution of variational programming. We show the challenge of integrating variational programming with side effects. We explore various approaches in overcoming these challenges and discuss their limitations. We discuss our solution and its advantages and drawbacks. Finally, we outline the contributions of this work.

A core insight of variational programming is that the execution of sets of related programs, and/or execution over sets of related data, can be made faster by computing over a *single variational artifact* that shares common parts while capturing differences through localized, explicit *variation points*. To illustrate this insight, consider the three related programs which calculate the flight cost for different types of tickets in a booking system and the corresponding

variational flight cost program in Figure 1.1. We assume tickets come in three categories: **Economy**, **Business** or **First**. Also, there are $55 flight fees and a $45 booking fee added to the cost. In each case, the price is calculated by summing up the price per ticket with the total fees.

Economy
```
let ecoPr = 649;;
let cost =
   55 + 45 + ecoPr;;
```

Business
```
let busPr = 3955;;
let cost =
   55 + 45 + busPr;;
```

First
```
let fstPr = 5812;;
let cost =
   55 + 45 + fstPr;;
```

Variational flight cost
```
let vTicketPr =
   Economy⟨649,Business⟨3955,5812⟩⟩
let vCost =
   55 + 45 + vTicketPr;;
```

Figure 1.1: Variational flight cost.

The value `Business⟨3955,5812⟩` represents a *choice* between the alternatives `3955` and `5812` (this choice is nested in another choice). The *condition* of the choice is an *option* `Business`, which may be either enabled (true), disabled (false), or open. A choice is similar to a conditional expression (if-then-else), except that if its condition is open, it represents *both* alternatives at the same time. Multiple choices with the same condition will always be synchronized while choices with conditions based on different options may vary independently.

Enabling or disabling a choice is what we call *selection.* To select the left alternative of the choice `vTicketPr`, we say (`selectV Economy vTicketPr`). This returns `649`, which is a plain value (no variation). To select the right alternative of it, we say (`selectV (!Economy) vTicketPr`). This returns `Business⟨3955,5812⟩`. An open selection (`selectV true vTicketPr`) selects both alternatives, so this returns `vTicketPr`. The notation and semantics of choices is a previous work on the *choice calculus* [9, 22, 11].

We can extend the evaluation semantics of our programming language to include choices by simply mapping over their alternatives. This allows us to evaluate `vCost` to the following choice with the following sequence of reduction steps.

$$\begin{aligned}
&\underline{\texttt{55+45+Economy}\langle\texttt{649},\texttt{Business}\langle\texttt{3955},\texttt{5812}\rangle\rangle}\\
&\qquad\mapsto\ \underline{\texttt{55+45}}\texttt{ + Economy}\langle\texttt{649},\texttt{Business}\langle\texttt{3955},\texttt{5812}\rangle\rangle\\
&\qquad\mapsto\ \underline{\texttt{100 + Economy}\langle\texttt{649},\texttt{Business}\langle\texttt{3955},\texttt{5812}\rangle\rangle}\\
&\qquad\mapsto\ \texttt{Economy}\langle\underline{\texttt{649+100}},\texttt{Business}\langle\texttt{3955+100},\texttt{5812+100}\rangle\rangle\\
&\qquad\mapsto\ \texttt{Economy}\langle\texttt{749},\underline{\texttt{Business}\langle\texttt{3955+100},\texttt{5812+100}\rangle}\rangle\\
&\qquad\mapsto\ \texttt{Economy}\langle\texttt{749},\texttt{Business}\langle\underline{\texttt{3955+100}},\texttt{5812+100}\rangle\rangle\\
&\qquad\mapsto\ \texttt{Economy}\langle\texttt{749},\texttt{Business}\langle\texttt{4055},\underline{\texttt{5812+100}}\rangle\rangle\\
&\qquad\mapsto\ \texttt{Economy}\langle\texttt{749},\texttt{Business}\langle\texttt{4055},\texttt{5912}\rangle\rangle
\end{aligned}$$

This reduction process represents *variational execution* which dynamically evaluates all variants of a program, but observe that we only visited the expression `55 + 45` once rather than three times if we had evaluated the three programs separately. Although the saving here is small, it adds up as the size of shared programs and the number of independent options increases. In fact, many applications of variational programming involve efficiently computing or exploring an exponential number of variants by exploiting the ideas of capturing variation locally (e.g. in choices) and sharing common parts of data and computations.

When mapping analyses or computations over variational terms, the crucial property that is maintained is *variation preservation* [7]. This is illustrated by the commuting diagram at right, where `f` is a plain function that transforms a plain expression `e` into a plain result `r`, and $\texttt{f}_\texttt{v}$ is the corresponding *variational function* that transforms the variational expression $\texttt{e}_\texttt{v}$ into a variational result $\texttt{r}_\texttt{v}$. The operation $[\![.]\!]_\texttt{c}$ refers to the select operation mentioned above; for example, $[\![\texttt{e}_\texttt{v}]\!]_\texttt{c}$ is equivalent to $\texttt{select c e}_\texttt{v}\texttt{ = e}$ The variation-preservation property states that $\texttt{f}_\texttt{v}$ is exactly equivalent to running `f` on every plain variant of $\texttt{e}_\texttt{v}$. That is, we obtain the same plain result `r` by either $\texttt{f}([\![\texttt{e}_\texttt{v}]\!]_\texttt{c})$ or $[\![\texttt{f}_\texttt{v}(\texttt{e}_\texttt{v})]\!]_\texttt{c}$, for any option `c`.

$$\begin{tikzcd}
e \arrow[r, "f"] & r \\
e_v \arrow[u, dashed, "{[\![\cdot]\!]_c}"] \arrow[r, "f_v"] & r_v \arrow[u, dashed, "{[\![\cdot]\!]_c}"']
\end{tikzcd}$$

Now, consider adding side effects to a variational program. For example, someone might like to write the variational flight cost calculated in Figure 1.1 into a file as shown below.

```
#WriteFile "flight.txt" ("The total flight cost:");;
#WriteFile "flight.txt" ("$" +  Economy⟨"749",Business⟨"4055","5912"⟩⟩);;
```

With the current implementation of variational execution, every choice alternative would be visited. Hence, side effects that occur in choices would be performed during the process even in the alternative context. To see how this is an issue, look at the following output of the program above.

Flight.txt

```
The total flight cost:
$749
4055
5912
```

This program writes the text of each variant while the desired behavior is to only write the text of the corresponding variant. Variational programming uses variational data-structures to manage different variants and separate their results, but the current environments of the effects we work with are not variational, so we can not separate their variants which causes this problem.

There are several solutions to allow working with side effects in variation. One possible solution is to make a variation-aware interface for every effect type we work with, that is a variational file system, database [1] and so on. Thus, if the file system was variational, there would be conceptually a different file instance for every new option introduced by a choice as shown below.

Economy.txt

```
The total flight cost:
$749
```

NotEconomyBusiness.txt

```
The total flight cost:
$4055
```

NotEconomyNotBusiness.txt

```
The total flight cost:
$5912
```

As we introduce more options, more instances should be created. However, making a special variational infrastructure for every effect type is very complex, infeasible or incompatible in some domains. Another solution is to avoid variational execution from running side effects when they occur in variation, but this would cut their useful applications.

In this thesis, we argue that algebraic effects can be used to resolve the problem of combining variation and effects by enabling programmers to flexibly and incrementally extend variational programming environments to handle new kinds of effects or handle existing effect differently. We present a proof-of-concept prototype in the *Eff* programming language. We implement variation as an algebraic effect and then demonstrate how the variational programming environment can be extended to support file input/output. Given that there is not a comprehensive solution for all effects, this approach is the best given the constraints. It allows programmers to work with effects on a case-by-case basis by building new libraries to handle them. Unfortunately, implementing variation as an effect is not efficient as the number of options increases. Therefore, we argue that having built-in features in the programming language itself for working with variation would be more effective.

This work includes the following contributions.

- We implement variation as an effect and show examples of how it works (Chapter 3). We show the limitations of this implementation when we start to integrate side effects in variational programs (Chapter 4). Our approach in addressing these limitations is by writing new handlers for the variation effect to work with interaction of variation and effects on a case-by-case basis using algebraic effects. As a proof-of-concept, we write new handlers to support file writing (Chapter 5) and reading (Chapter 7) which are hard to work with otherwise. As part of this, we also extend text files with a variation encoding (Chapter 5.1).
- We introduce an abstract variational queue that can be used in different variational programming applications (Chapter 6). We use the variational queue to represent variational files as part of our proof-of-concept.
- We discuss the limitations of implementing variation as an effect (Chapter 8).

Our prototype implementation is publicly available at the GitHub repository VEffect[1].

[1]https://github.com/lambda-land/VEffect

Chapter 2: Background: Algebraic Effects

```
effect Get: unit -> int;;
effect Set: int -> unit;;

let state = handler
  | val y -> (fun _ -> y)
  | #Get () k -> (fun s -> k s s)
  | #Set s' k -> (fun _ -> k () s')
  | finally g -> g 0
;;
```

Figure 2.1: The integer state effect and handler.

As background, we introduce *algebraic effects* - what they mean and how to use them - by showing a code example. The programming language used in this work is called *Eff*[1] which is based on the programming language *OCaml* and has similar syntax and features [19]. *Eff* has a flexible extensible effect system that does not exist in *OCaml* called *algebraic effects* which is the main tool used in our work (the words effects and algebraic effects are interchangeably used in this thesis). We use *Eff* syntax throughout this thesis in our coding examples.

Eff is based on algebraic effects and handlers. An effect is defined by a set of primitive operations (e.g. *set* and *get* for a *state* effect) and implemented by one or more handlers. The fact that multiple handlers can be written for the same effect increases the power and flexibility of this system. Algebraic effects expand on the traditional exception handling mechanism by allowing computational effects to be thrown and caught by a handler that performs certain computation. Thrown effects have a continuation parameter that could be invoked with an argument of the thrown effect's return type. The continuation resumes the execution from the point where the effect operation was thrown.

To get a sense of how algebraic effects are used and defined, refer to the integer state effect

[1]https://www.eff-lang.org/

[19] in Figure 2.1. This state effect has two primitive operations `#Set` and `#Get` declared with the keyword `effect`. `#Get` takes a unit value and returns the corresponding state value. `#Set` takes a new state value and returns a unit value.

The handler has a separate case for each effect operation plus the return and action cases. The continuation parameter `k` in `#Set` and `#Get` is invoked with values of the return type; in `#Get` the return type is `int` and in `#Set` the return type is `unit`. The result of each continuation invocation (`k ()` or `k s`) is a function that takes a parameter of type `int` corresponding to the current state value, which in the case of `finally` gets its initial value 0. In the case of `#Get`, the handler returns a function that will invoke the continuation with the current state value and pass it unchanged to the result of the continuation. In the case of `#Set`, the handler returns a function that will invoke the continuation with a unit value and pass the new state value (the parameter of `#Set`) to the result of the continuation. The `finally` case starts the computation with the initial value `0`. The `val` case is the return case.

Note that we establish the handler scope for a given expression using the following syntax.

```
with handler_name handle expression
```

Any effect operation that occurs within that code block will be within the bound of the `state` handler instance.

```
with state handle
    #Set 5; #Set (#Get () + 1); #Get ()
;;
```

This code sets the state to `5`, then increments it by `1`. Then, evaluates the state again with the effect operation `#Get`, which gives the result of `6`.

Chapter 3: Variation as an Algebraic Effect

```
effect Choice : ctx -> bool;;

let v_handler = handler
    | #Choice s k ->
        let l = select s (k true) in
        let r = select (Not s) (k false) in
            prepend s l @ prepend (Not s) r
    | val x -> [(Lit true,x)]
;;

let chc d l r = if #Choice d then l else r;;
```

Figure 3.1: The variation effect and handler.

We use the algebraic effects system introduced in Chapter 2 to implement variation as an effect. We use it as part of our proof-of-concept for the integration of variation and side effects. However, implementing variation as an effect is not efficient when the number of options increases. We discuss the limitation of this implementation in Chapter 8.

In this chapter, we show the algebraic effect definition of variation and its handler. Then, we show how variational values of lists and arithmetic expressions are encoded with the variation effect.

To implement variation as an effect, we use the choice as its primitive operation. We handle choices by performing variational execution which dynamically evaluates the alternatives of the choices. The choice effect operation `#Choice` shown in Figure 3.1 is declared to take a variation context (or context for short) `ctx` and return a boolean value. The variation context is represented by a boolean formula over options. We show syntactic sugar of boolean formulas in our examples in this thesis.

The handler `v_handler` catches the choice and handles it by invoking the continuation with either `true` or `false`. The result of invoking the continuation is of type `(ctx * 'a) list` which is a list of tuples: each with a context and a corresponding return value of any type. The results from both invocations are concatenated in one list representing the variational value to be returned from the handler. When invoking the continuation with `true`, the corresponding return value is the left alternative of the choice. Otherwise, it is the right alternative of the choice, as shown in the definition of `chc`. In the return case `val`, we take the underlying return value as parameter and return a list of one tuple that contains the context `true` and the underlying return value.

To build the context corresponded with each choice's return value, we use the functions `select` and `prepend`. The function `prepend` updates the context of every element in the list produced by the continuation (each element is a tuple of a context and value) to be the conjunction of two contexts: the element's context and underlying choice context. The function `select` removes elements that would be unsatisfiable after the conjunction of the underlying choice context from the list. We apply `prepend` twice: once with the underlying choice context on the list from the left alternative and once with the negation of the underlying choice context on the list from the right alternative. Hence, we are able to simulate all possible variants by running the program once (there are 2^k possible results with k distinct options).

After we define the variation effect, we can encode variational values in multiple forms using the choice operation. In this example, we write an arithmetic expression which varies around the two options `Economy` and `Business` assuming fees also vary around the ticket type. Therefore, we sum up two expressions: the first represents fees and the second represents ticket costs.

```
with v_handler handle
    let fee = Economy⟨50,Business⟨100,150⟩⟩ in
    let ticket = Economy⟨600,Business⟨4000,6000⟩⟩ in
    fee + ticket
;;
```

Note that the notation used here is syntactic sugar to make the choice operation look cleaner and will be used throughout this thesis. Below is the version of the expression with no syntactic sugar.

```
with v_handler handle
    let fee = chc "Economy" 50 (chc "Business" 100 150) in
    let ticket = chc "Economy" 600 (chc "Business" 4000 6000) in
    fee + ticket
;;
```

In the above computation, there are 3 possible results based on the two options `Economy` and `Business`. The program above should output the choice below as a result.

```
Economy⟨650,Business⟨4100,6150⟩⟩
```

The truth table below illustrates the result for each possible selection of `Economy` and `Business`.

And		Economy	
		T	F
Business	T	650	4100
	F		6150

Another useful application is constructing a variational list as a fundamental data structure form used to incorporate variational values [21]. It allows mapping options to different lists which can then be used in further purposes such as searching over different lists corresponding to different contexts at once. For example, we can collect information about different flights in a list of tuples: each has a corresponding flight id and price.

```
with v_handler handle
    [(1,4890);
    (2,Economy⟨790,Business⟨_,4500⟩⟩);
    (3,2876);
    (4,Economy⟨580,Business⟨3200,_⟩⟩);
    (5,Economy⟨_,Business⟨3780,5939⟩⟩)]
;;
```

The basic list notation is extended by the choice notation to allow capturing variational lists. The notation _ represents `Nothing` or an empty element. For example, for flight id 2, the element is empty or does not exist when the option `Economy` is false and the option `Business` is true. Note, the notation used above is syntactic sugar for a list of variational `Maybe` values. Below is the list version with no syntactic sugar.

```
with v_handler handle
    [(1,Just 4890);
    (2,Economy⟨Just 790,Business⟨Nothing,Just 4500⟩⟩);
    (3,Just 2876);
    (4,Economy⟨Just 580,Business⟨Just 3200,Nothing⟩⟩);
    (5,Economy⟨Nothing, Business⟨Just 3780, Just 5939⟩⟩)]
;;
```

The result should capture all possible list variants. The truth table below illustrates the list corresponded with each possible selection of `Economy` and `Business`. Note that the `Nothing` cases are omitted.

And		Economy	
		T	F
Business	T	[(1,4890);(2,790);(3,2876);(4,580)]	[(1,4890);(3,2876);(4,3200);(5,3780)]
	F		[(1,4890);(2,4500);(3,2876);(5,5939)]

Chapter 4: Variational Programming and Effects

Integrating side effects with variation is challenging because the execution environment of variational programs is not variational, so we can not separate the effect of one variant from the other. The variation effect introduced in the previous chapter composes well with some types of effects, but still fails with others. We discuss an example in faceted execution illustrating this point. Then, we discuss our solution in resolving this problem.

The work done in [3] shows the use of faceted execution to enforce dynamic information-flow security. The inspiration behind this work is preventing untrusted sources from running sensitive programs with full privileges. The authors argue that their faceted execution techniques can protect against malicious attacks and enforce integrity and confidentiality. The term they use to indicate a variational value or choice is *faceted value*. They use faceted values to simulate the public and private versions of the same program under a privacy policy.

```
with state handle
    with v_handler handle
        #Set 5;
        let ok = check secret in
        if ok then
            #Set (#Get () * 2)
        else
            ()
        ;
        #Get ()
;;
```

The example above has a privacy-protected variable `secret` represented as a choice. It can only be enabled to access the sensitive information when the privacy policy indicates it. Using this feature, we declare the state to be 5 and update it based on the value of `secret`, which we check. If the private facet has the correct password, we update the state by doubling its value in the private facet and keep it as it is in the public facet as follows.

$$\texttt{secret = P}\langle\texttt{"mypassword"}, \perp\rangle \quad \texttt{state = P}\langle 10, 5\rangle$$

If the private facet has the wrong password, the state is unchanged and has the same value for public and private facets as follows.

$$\texttt{secret = P}\langle\texttt{"wrong"}, \bot\rangle \quad \texttt{state = 5}$$

Therefore, this choice mechanism allowed two facets of the information presented. Both private and public information (for authorized and unauthorized users respectively) are supported, which is useful in making secure applications and it works well with the variation effect so far.

Sometimes however, variation and effects do not compose well. Consider adding other side effects to the program above such as a print statement.

```
with state handle
    with v_handler handle
        #Set 5;
        let ok = check secret in
        if ok then
            #Set (#Get () * 2)
        else
            #Print("wrong password")
        ;
        #Get ()
;;
```

The error message would be printed with wrong and correct password values in `secret`, while we hoped it would only be printed with wrong password values. This explains how we fail to separate the effects of one variant form the other in the current implementation.

Someone might consider building a variational environment for every effect type to solve the problem above, but this is tedious and infeasible in some domains. Moreover, some dimensions of variation correspond to differences in platforms which cannot be simulated together. Therefore, we want to give the programmer the tools to deal with the interaction of variation and effects in an expressive and systematic way using algebraic effects. This is done on a case-by-case basis because every effect type might need a special treatment. The use of algebraic effects enable us to flexibly and incrementally extend the execution environment to handle new kinds of effects or handling existing effects in multiple ways.

Chapter 5: Variational File I/O: Writing

To illustrate how algebraic effects enable programmers to deal with the interaction of variation and effects, we extend the execution environment of variational programs to support file I/O, which is a major contribution of this work. Specifically, we write new handlers for the variation effect to support file reading and writing. In this chapter, we focus on writing, and in Chapter 7 we focus on reading. In addition, we extend text files with a variation encoding.

5.1 Variational File Encoding

We extend plain text files with a variation encoding known as *C-Preprocessor*[1] (CPP). CPP is a text substitution tool that must be executed before compile time. We choose CPP because it is a real-world widely used convention, which makes our work useful for existing applications. CPP has many directives, but the ones relevant to our work are `#if`, `#else` and `#endif`.

The file shown in Figure 5.1 is an example of a variational file that uses the encoding mentioned above. It is inspired by a real dataset from Massart College for Art and Design showing their housing prices of 2018-2019[2]. This file represents an explanation of costs in different scenarios for their students. Note that text files are extended by this encoding and not required to have it; we can still work with plain files flexibly and the special syntax is needed only when variation exists.

To understand the concept of variational lines, we look at the first 3 lines of the file. The file starts with a plain line (not variational). The next is a nested variational line. When the option `meal` is enabled, it yields the variational line `partial`⟨`"$1902"`, `"$3372"`⟩, which when the option `partial` is enabled, it yields the plain line `"$1902"`. Otherwise, it yields the plain line `"$3372"`. Note that it is not necessary to have an `#else` when there is an `#if`, but there must be

[1]https://www.tutorialspoint.com/cprogramming/c_preprocessors.html
[2]https://massart.edu/cost-housing

```
Below is a breakdown of the preliminary housing costs for 2018—2019.
#if meal
    #if partial
        $1902
    #else
        $3372
    #endif
#endif
#if artists’ residence
    Partial (default) or optional regular meal plan.
    Total Including Activity/Technology Fee:
    #if single | efficiency
        $13360
    #else
        $12216
    #endif
#endif
#if smith
    Regular meal plan required, except for students living in kitchen suites.
    Kitchen suite students receive the partial (default)
    or optional regular meal plan.
    Total Including Activity/Technology Fee:
    #if kitchen
        #if single
            $11234
        #else
            $10530
        #endif
    #else
        #if single
            $13360
        #else
            $11002
        #endif
    #endif
#endif
```

Figure 5.1: Variational text file showing housing prices.

an `#endif` whenever there is an `#if` to ensure balance and correct parsing. Also, note that the third variational line has in its nested variational line an `OR` expression `single | efficiency` because the context of the choice is a boolean formula over options.

```
let v_write_handler file = handler
    | #Choice s k ->
        wr_if s file;
        (k true);
        wr_else file;
        (k false);
        wr_end file
    | #Write_file _ k ->  k ()
    | val x ->
        if (is_empty x) then () else #Write_file(file, x ^ "\n")
;;

let writeFile t = let _ = #Write_file(dfile, t) in t;;

let write_h file v =
    with v_write_handler file handle
        v ()
;;

let opt d t = chc d t "";;
```

Figure 5.2: File writing library.

5.2 Variational Writing

In this section, we write a new handler for the variation effect to support writing and show an example of its use. Our goal is to be able to use a program like the example below to write a variational line. The following variational line has two alternatives `"749"` and `"4055"` based on the option `Economy`.

```
WriteFile "flight.txt" Economy⟨"749","4055"⟩
```

Equivalently, we could write the program below for the same result.

```
Economy⟨#WriteFile "flight.txt" "749",#WriteFile "flight.txt" "4055"⟩
```

The handler `write_v_handler` in Figure 5.2 catches and handles the variation effect for the purpose of writing. It has a similar behavior to the handler in Figure 3.1 for handling the choice operation. However, for handling the choice effect operation here, we do not construct a variational value, we just run the alternatives of the choice and write the corresponding keywords `#if`, `#else`, or `#endif` into the text file via the functions `wr_if`, `wr_else`, and `wr_end` respectively.

For handling the built-in effect operation `#Write_file`, we simply run the continuation with a unit value. The helper function `writeFile` enforces the text parameter of `#Write_file` to be the return value. Note that we can neglect passing the file channel in every call to `writeFile` and use a default file channel instead because the corresponding channel is already passed to the handler. In the return case `val`, we write the underlying return value into the text file.

For more convenience, we use two other helper functions. We use the helper function `opt` for cases where we only care about one alternative of the choice and not the other, so we ignore the else case. The function `opt` is a choice with an empty string in the right alternative. We use the helper function `write_h` to wrap the handler syntax.

Using this mechanism, we could write variational lines and/or plain text to files.

```
let w_file = #Open_out "file.eff" ;;

let line1 () = writefile "Below is a breakdown ....";;
let line2 () = meal⟨partial⟨writefile "$1,902",writefile "$3372"⟩⟩;;

write_h w_file line1 ; write_h w_file line2;;

#Close_out w_file;;
```

The example above shows the program used to write the first two lines of the example file in Figure 5.1. Note that this implementation does not prevent us from writing plain (with no variation) text to files. We only use choices when intending to write variational lines. Also note that we use syntactic sugar for the opt notation (as shown in `line2` in the example above) similar to the syntactic sugar for the choice notation. The code below shows the version without the syntactic sugar of opt.

```
opt "meal" (partial⟨writefile "$1,902",writefile "$3372"⟩);;
```

Chapter 6: Variational Queue

Similar to other kinds of variational values described in this thesis, a variational queue conceptually represents many different plain queues. Variational queues, and other variational data structures [23], are useful in a variety of variational programming applications. Our variational queue implementation is inspired by the variational stack implementation in [15]. The work in [15] also explores different implementations of variational stacks and their trade-offs in terms of efficiency and space complexity. We particularly use variational queues in this work to represent variational files. Variational queues allow us to simulate multiple plain files in one data structure and would be useful for supporting file reading (Chapter 7).

In this chapter, we explain the details of the variational queue implementation we use. We discuss the two main operations: `enqueue` and `dequeue`. Because the implementation of `dequeue` is particularly complicated, we provide detailed examples illustrating the behavior we expect.

6.1 Variational Queue Implementation

```
type 'a v = Hole
    | One of 'a
    | Chc of ctx  * 'a v * 'a v
;;
type 'a opt = 'a * ctx;;

type 'a queue = ('a opt) list;;
val dequeue: ctx -> 'a queue -> 'a v * 'a queue;;
val enqueue: ctx -> 'a queue -> 'a v  -> 'a queue;;
```

Figure 6.1: Variational queue signature.

In this section, we show the implementation of the variational queue and its abstract signature. In the next sections we show the details of its operations.

```
let toOpts =
    let rec go c v =
        (match v with
            Hole   -> []
            | (One a) -> [(a,c)]
            | (Chc (c', l,r) ) -> go (And (c, c')) l ++ go (And (c, (Not c'))) r)
      in go (Lit true)
;;

let enqueue c v q = (q ++ toOpts (Chc (c, v, Hole)));;
```

Figure 6.2: Variational `enqueue` implementation.

We represent variational queues as lists of optional values. An optional value is represented by the `opt` type, which is a tuple of a value and context as shown in Figure 6.1.

With this implementation, we could keep track of the variation context corresponding to every value in the queue. Because the queue is simulating different contexts, `dequeue` and `enqueue` each takes a variation contexts as parameter, which differentiates them from plain queue operations. Moreover, the return value of `dequeue` is of type `v` because the return value is either a plain value (`One`), a variational value (`Chc`), or nothing (`Hole`) as shown in Figure 6.1. Similarly, `enqueue` appends values of type `v` into the queue.

6.2 Variational Enqueue

The operation `enqueue` simply takes a value of type `v` and appends it into the queue with the corresponding context passed as parameter (shown in Figure 6.2). The operation `enqueue` invokes `toOpts` on the `v` value to convert it into an `opt` value. The operation `toOpts` also recurs on each choice alternative to convert it into an `opt` value with the conjunction of the context parameter and underlying context of the choice.

```
Flight cost:
#if Business
    4055
#else
    749
#end
```

Consider the example above of a file with two tickets' costs. This file could be inserted into the variational queue by the following sequence of `enqueue` operations.

```
$ enqueue true (One "Flight cost:") q
$ enqueue Business (One "4055") q
$ enqueue !Business (One "749") q
```

Or by following sequence of `enqueue` operations (we use the same choice notation here as syntactic sugar).

```
$ enqueue true (One "Flight cost:") q
$ enqueue true Business⟨"4055","749"⟩ q
```

As a result, we end-up with the following variational queue representing the file above.

```
[("Flight cost:",true);("4055", Business);("749",!Business)]
```

6.3 Variational Dequeue in Action

To understand the behavior expected when dequeuing values form the variational queue, consider the following example operations performed on the queue above. These examples correspond to scenarios that are more likely to occur in `dequeue`.

Example 1: Suppose we perform `dequeue` with context `Business` on the queue above as follows (we use `B` as a shortcut for `Business`).

```
dequeue B [("Flight cost:",true);("4055", B);("749",!B)] =
    ("Flight cost:", [("Flight cost:",!B);("4055", B);("749",!B)])
```

The context of the first value on the queue is `true`, meaning that it's present in all variants. If we dequeue it only in context `Business` then that means it has not yet been dequeued in context `!Business`. Therefore, we return the value `"Flight cost:"`, but still keep it in the queue with context `!Business`.

Example 2: Suppose we perform `dequeue` with context `Business` again on the queue that resulted from Example 1 as follows.

```
dequeue B [("Flight cost:",!B);("4055", B);("749",!B)] =
    ("4055", [("Flight cost:",!B);("749",!B)])
```

The context of the first value on the queue is `!Business` (we already dequeued it in context `Business`), so we should skip it because it conflicts with the context we dequeue with. The context of the second value on the queue is `Business` which is the same context we are dequeuing with, so we return the value and remove it completely from the queue.

Example 3: Suppose we perform `dequeue` with context `!Business` on the queue that resulted from Example 1 as follows.

```
dequeue !B [("Flight cost:",!B);("4055", B);("749",!B)] =
    ("Flight cost:", [("4055", B);("749",!B)])
```

The context of the first value on the queue is `!Business` which is the same context we are dequeuing with, so we return the value and remove it completely from the queue. This value has been dequeued in both variants after this (in Example 1 and in this Example).

Example 4: Suppose we perform `dequeue` with context `true` on the queue that resulted from Example 3 as follows.

```
dequeue true [("4055", B);("749",!B)] =
    (B⟨"4055","749"⟩, [])
```

Dequeuing with `true` means to perform `dequeue` on all variants of the queue. Therefore, the result of this operation would be the value of every variant in the queue. The first value in the queue has context `Business` and the second value has context !Business, so both are returned constructing the choice `Business⟨"4055","749"⟩`. In addition, they are removed entirely from the queue.

```
let rec dequeue_ cIn q v =
    if unsatCtx cIn then
        (v, q)
    else
        match q with
            [] -> (v, [])
            | ((a,cElem)::oq) ->
                let cDeq = And (cIn, cElem) in
                let cRem = And (Not cIn, cElem) in
                let cToDo = And (cIn, Not cElem) in
                let  (v', q') = dequeue_ cToDo oq (Chc (cDeq,(One a),v)) in
                (v',(a,cRem)::q')
;;

let dequeue c q = dequeue_ c q Hole;;
```

Figure 6.3: Variational `dequeue` implementation.

6.4 Variational Dequeue Implementation

In this section we show the details of the `dequeue` operation shown in Figure 6.3. When we dequeue an element, we have to think of three different contexts: the context of the element we dequeue `cDeq`, the context of the element that remains in the queue `cRem` and the context of the hole we need to fill `cToDo`. We could also think of the hole as the element we still need to find and dequeue. These three contexts are built from two contexts we start with: the context we perform dequeue with `cIn` and the context of the first element on the queue `cElem`.

Intuitively and with the examples we showed in the last section, consider the following cases:

1. The element is dequeued if the context we dequeue with is satisfiable with its context. Therefore, `cDeq` is (`cIn & cElem`). If `cDeq` is satisfiable, the element is returned. Otherwise, it is skipped.

2. After we dequeue an element, there is a chance we need to dequeue it in other contexts, so we keep it in the queue and avoid dequeuing it again in the same context. Therefore, `cRem` is (`!cIn & cElem`). If `cRem` is unsatisfiable, the element is removed from the queue (we do not need to dequeue it again).

3. After we dequeue or skip an element in one context, we might still need to dequeue or find

Ex	Inv	Queue in	Contexts in		Contexts out		(value * Queue out)
1	1	`[("Flight cost:", true);`	cIn	`B`	cEnq	`B&true`	
		`("4055", B);`	cElem	`true`	cRem	`!B&true`	
		`("749",!B)]`			cToDo	`B&false`	
	2	`[("4055", B);`	cIn	`false`	cPop		`("Flight cost:",`
		`("749",!B)]`	cElem		cRem		`[("Flight cost:", !B);`
					cToDo		`("4055", B);("749",!B)]`
2	1	`[("Flight cost:", !B);`	cIn	`B`	cDeq	`B&!B`	
		`("4055", B);`	cElem	`!B`	cRem	`!B&!B`	
		`("749",!B)]`			cToDo	`B&B`	
	2	`[("4055", B);`	cIn	`B`	cDeq	`B&B`	
		`("749",!B)]`	cElem	`B`	cRem	`!B&B`	
					cToDo	`B&!B`	
	3	`[("749",!B)]`	cIn	`false`	cDeq		`("4055",`
			cElem		cRem		`[("Flight cost:", !B);`
					cToDo		`("749",!B)])`
4	1	`[("4055", B);`	cIn	`true`	cDeq	`true&B`	
		`("749",!B)]`	cElem	`B`	cRem	`false&B`	
					cToDo	`true&!B`	
	2	`[("749",!B)]`	cIn	`!B`	cDeq	`!B&!B`	
			cElem	`!B`	cRem	`B&!B`	
					cToDo	`!B&B`	
	3	`[]`	cIn	`false`	cDeq		`(B⟨"4055","749"⟩,`
			cElem		cRem		`[])`
					cToDo		

Figure 6.4: A step-by-step illustration of contexts in `dequeue`.

other elements of other contexts as in Example 4. Therefore, `cToDo` is (`cIn & !cElem`). When `cToDo` is unsatisfiable, we stop the search and return.

The helper function `dequeue_` is recursively invoked on the queue with two base cases: reaching the end of the queue or if `cIn` is unsatisfiable. We mentioned above that we stop the search when `cToDo` is unsatisfiable. This is because we recursively invoke `dequeue_` with `cToDo` so `cToDo` of one invocation is `cIn` of the next. In every invocation of `dequeue_` we update the context of the corresponding first element to be `cRem`, construct a choice of the return value with context `cDeq`, and simplify all contexts.

The table in Figure 6.4 shows the value of each context and the content of the queue in each invocations. The last column has the resulting value and updated queue after the process is done.

The sections of the table represent Examples 1, 2, and 4, respectively.

Example 1: In invocation 1, `cEnq` is satisfiable, so the value is returned. `cRem` is satisfiable, so the value remains in the queue. `cToDo` is unsatisfiable, so the next invocation terminates.

Example 2: In invocation 1, `cEnq` is unsatisfiable, so the value is skipped. `cRem` is satisfiable, so the value remains in the queue. `cToDo` is satisfiable, so we keep looking for the next value to dequeue.

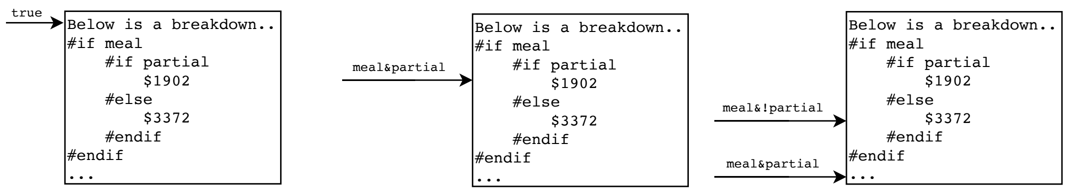
In invocation 2, `cEnq` is satisfiable, so the value is returned. `cRem` is unsatisfiable, so the value is removed from the queue. `cToDo` is unsatisfiable, so the next invocation terminates.

Example 4: In invocation 1, `cEnq` is satisfiable, so the value is returned. `cRem` is unsatisfiable, so the value is removed from the queue. `cToDo` is satisfiable, so we keep looking for the next value to dequeue.

In invocation 2, `cEnq` is satisfiable, so the value is returned. `cRem` is unsatisfiable, so the value is removed from the queue. `cToDo` is unsatisfiable, so the next invocation terminates.

Chapter 7: Variational File I/O: Reading

Continuing in our goal of illustrating how algebraic effects enable programmers to deal with the interaction of variation and side effects, we show the rest of the file I/O library, specifically, we show how we implement reading in variational programs. Reading is more complex than writing because we need to manage different variation contexts of the same file (to simulate multiple plain files at once). To illustrate how reading should work, follow the sequence of operations in Figure 7.1.



(a) Read-line in context `true`. (b) Read-line in `meal&partial`. (c) Read-line in `meal&!partial`.

Figure 7.1: File representation in a sequence of read-line operations.

Plain read-line calls have context `true` by default. When we start to read from the file, the file pointer is at the first line of the file with context `true` as shown in Figure 7.1a. We perform a read-line operation as shown below.

```
readLine file = "Below is a breakdown.."
```

Reading in context `true` corresponds to reading from all file variants. Since the first line is plain, it is available in all variants, so we return it.

We then perform a second read-line operation in the context `meal&partial` as shown below.

```
(meal&partial)⟨readLine file⟩ = "$1902"
```

The file pointer now moves to the true alternative of the choice `partial` in line 2 as shown in Figure 7.1b, and we return the line `"$1902"` corresponding to this context.

We then perform a third read-line operation but in the context `meal&!partial` as shown below.

```
(meal&!partial)⟨readLine file⟩ = "$3372"
```

We end up with two file pointers at two different locations. One with the context `meal&!partial` pointing to the false alternative of the choice `partial` in line 2 and the other with context `meal&partial` pointing to the third line as shown in Figure 7.1c. We return the line `"$3372"` corresponding to the context `meal&!partial`.

In the rest of this section, we show how we could manage these variational read-line calls and how we extend the environment of variational programs to support file reading using algebraic effects.

7.1 Variational Queues for Representing Files

We need to use a special data structure that could simulate the different plain files corresponding to different variation contexts in a variational file. Therefore, we use the variational queue implementation in Chapter 6 to represent files. This way we could load variational lines from the file into the queue using enqueue operations and read-lines from the file through dequeue operations. However, we still need to parse lines from the variational text file to be able to enqueue them.

In this section we show how to parse variational files into variational queues. The parser details are shown in Figure 7.2. The parser takes a variational text file as input and returns a variational queue as output. The queue is initialized to empty at the beginning. Parsing mainly occurs in the helper function `read`. The function `read` takes a file channel, queue, text and context as parameters. The text corresponds to the last line of text read from the file. The queue is updated with enqueue operations after parsing each line (plain or variational). The context corresponds to the context of the parent variational line if it exists; it is `true` by default.

In the function `read`, we check if the text corresponds to a variational line by checking if it

```
let rec read file q text ctx =

    if is_choice text then
        let ctx' = get_ctx text in
        read_vline file q ctx ctx'
    else
        enqueue' q text (Lit true)

and read_insert file q line ctx ctx'=

    if is_choice line then
        read file q line ctx'
    else
        enqueue' q line (And (ctx',ctx))

and read_vline file q ctx ctx' =

    let line = get_next_line file in

    if is_endif line then
        q
    else if is_else line then
        read_vline file q ctx (Not ctx')
    else
        let q' = read_insert file q line ctx ctx' in
        read_vline file q' ctx ctx'
;;
```

Figure 7.2: Variational file parser.

starts with the keyword `#if`. If the text corresponds to a plain line, we directly enqueue it with context `true`. Otherwise, we parse the variational line in the helper function `read_vline`. We pass two contexts as parameters: the context of the underlying choice (`ctx'`) and the context of the parent line (`ctx`).

In the helper function `read_vline`, we read a line from the file, then check in which variant of the variational line we are. If the line has the keyword `#end`, we return the queue unchanged. If the line has the keyword `#else`, we recursively invoke `read_vline` with the negation of the context corresponding to the underlying choice (`ctx'`). Otherwise, we invoke the function `read_insert`, which would insert the text of the underlying variant into the file. Then, we recursively invoke `read_vline` again with the same context `ctx'` to parse the rest of the line.

In the helper function `read_insert`, we check if the variant has a nested variational line (choice). If there is a nested variational line, we recursively invoke `read` on it. Otherwise, we directly enqueue the line with its corresponding enqueue context. The corresponding enqueue context is the conjunction of two contexts: the context of the parent variational line and the context of the underlying variational line.

Besides parsing variational and plain lines, we also parse the contexts of variational lines. We show this parser code in the Appendix (Section A).

7.2 Variational Read-line

In this section, we write a new handler for the variation effect to support file reading and show an example of its use. We also show some useful applications of this work.

The handler `read_v_handler` in Figure 7.4 allows us to perform read-line operations in variation. It also has a similar behavior to the variation handler in Figure 3.1 for handling the choice operations. However, in this handler we run the alternatives of the choice and return their corresponding results. In addition, we mange the variation context (or context for short), which helps separate the effects of the choice alternatives. With this implementation, we are able to determine in which alternative of the choice we are and thus figure out its corresponding context. The context is by default `true` with plain read-line operations. Otherwise, we use the operations `if_`, `else_`, and `end_` to mange it.

The variation context is implemented as a mutable state (the algebraic effect and handler definitions for state are discussed in Section 2) that holds a stack of contexts. To make things easier, we also hold the current context in the state making a tuple of two arguments: the current context and the stack of contexts. The entire implementation is shown in Figure 7.3. The first operation `if_` takes a boolean formula over options and pushes it into the stack of contexts. The second operation `else_` pops the top element (a context) of the stack and pushes the negation of it into the stack. The third operation `end_` pops the top element (a context) of the stack. In every case after updating the stack of contexts, we update the current context to be the conjunction of

```
let if_ d =
    let (s,l) = #Get() in
    let l' = (d :: l) in
    #Set((conjList l',l'))
;;

 let end_ () =
    let (s,l) = #Get() in
    match l with
        [] -> #Set((conjList l, l))
        | (x::l') -> #Set((conjList l', l'))
;;
```

```
let else_ () =
    let (s,l) = #Get() in
    match l with
        [] -> #Set((conjList l,l))
        | (s':: l') ->
            let s'' = (Not s') in
            let l'' = (s''::l') in
            #Set((conjList l'', l''))
;;
```

Figure 7.3: Operations for managing the variation context.

```
let read_line f ctx = dequeue ctx f;;

let read_v_handler = handler
    | #Choice s k ->
        if_ s;
        let l = (k true) in
        else_ ();
        let r = (k false) in
        end_ ();
        (Chc (s,l,r))
    | #Read_line _ k -> k "c"
    | val x ->
        if (is_empty x) then
            Hole
        else
            let q = #GetQueue() in
            let (c,_) = #Get() in
            let (r,q') = read_line q c in
            #SetQueue(q');
            r
;;

let readLine () = #Read_line dfile;;

let read_line_h f =
    with read_var_line_handler handle
        f ()
;;

let opt d t = chc d t "";;
```

Figure 7.4: File reading library.

all contexts in the stack.

Besides the variation context, we also manage another state in this handler that holds the variational queue representing the file. We choose to hold it in the state to avoid passing it between read-line operations.

For catching the built-in effect operation `#Read_line`, which has a return type of string, we simply keep the continuation running with a default string. Note that we can neglect passing the file in every call to `#Read_line` and use a default file instead because the corresponding file representation is already stored in the state.

In the return case, we retrieve from the states both the variation context and the variational queue (representing the file). Then, we perform `read_line` on the file with the given variation context. The operation `read_line` is basically a dequeue operation on the variational queue, which takes the variation context as parameter. The result of performing dequeue is a tuple of two arguments: the resulting line and the updated variational queue. The state holding the queue is set to the new queue `q'` and the result `r` is returned. Note that the return value of `read_line` has type `v` because the result is either a plain line of text (`One`), a variational line (`Chc`), or nothing (`Hole`).

Using this mechanism, we could read lines from variational or plain files, as shown below.

```
let r_file = #Open_in "file.eff";;
let queue = read_v r_file;;
#Close_in r_file;;

let line1 () = readLine ();;
let line2 () = (meal&partial)⟨readLine ()⟩;;
let line3 () = (meal&!partial)⟨readLine ()⟩;;

let f () =
        read_line_h line1 ; read_line_h line2 ; read_line_h line3
;;

read_lines_h queue f;;
```

We open the file and then load it in a variational queue using the operation `read_v` which is a parser explained in Section 7.1. After that, the file could be closed because the queue would be used instead to represent the file. The helper function `read_line_h` is used to wrap syntax of the variational read handler. The operation `read_lines_h` is used to wrap the syntax of states' handlers (variational queue state and variation context state). The example above shows the same sequence of read-line operations in Figure 7.1.

Since the results from read-line could be variational we combine our work in the variational file reading handler and the variation handler to concatenate lines.

```
let line1 () = readLine ();;
let line2 () = meal⟨readLine ()⟩;;

let g () =
    let res1 = read_line_h line1 in
    let res1 = read_line_h line2 in
    let r () =
        let a1 = reflect_v res1 in
        let a2 = reflect_v res1 in
        a1 ^ "  " ^ a2
    in reify_v r
;;

read_lines queue g;;
```

We use the *reify* and *reflect* technique, which are the inverse of each other, to move from variational values to choice operations and from choice operations to variational values. In reflect, we convert the `v` type result into a choice effect operation and in reify, we calculate the corresponding variational result using the variation handler in Figure 3.1. The example above concatenates the results of two read-line operations. The first is a plain read-line operation and the second is a variational read-line operation with the context `meal`. The result `a1` is a plain line of text while the result `a2` is a choice as shown below.

```
a1 = "Below is a breakdown of the preliminary housing costs for 2018-2019."
a2 = partial⟨"$1902","$3372"⟩
```

Therefore, the final result of concatenating the lines above is the following variational value.

```
partial⟨"Below is a breakdown of ..  $1902","Below is a breakdown of ..  $3372"⟩
```

This implementation enables us to see the results corresponding to different variants in one run of the program.

The next example is another useful application that shows how the integration of variation and side effects can prevent from repetitive work. Being able to read all alternatives of the file, allows performing a filter operation on all alternatives at one run. The following program counts the number of times the text "\$13360" occurs in any alternative of the variational file.

```
let checkDocs w q =
  let words = concatMap (fun str -> split str x) q in
  let matches = filter (fun a -> a = w) words in
  length matches
;;

let q' () = map (fun v -> (reflect_v v)) q in
let f ()  = checkDocs "$13360" (q' ()) in
reify_v f;;
```

In a similar manner, we run a map operation converting every line to a choice operation (reflect). Then, we calculate the variational result using the variation handler (reify). In each alternative, we run the filter operation `checkDocs`. Running this program on the example file in Figure 5.1 will result in `2` when the two expressions (`artists' residence & (single | efficiency)`) and (`smith & (!kitchen & single)`) are true, in `1` when only one of them is true, and in `0` when both are false. The following choice represents the result (we use artists instead of artists' residence for short).

$\texttt{(artists\&(single|efficiency))}\langle\texttt{(smith\&!kitchen\&single)}\langle 2,1\rangle,\texttt{(smith\&!kitchen\&single)}\langle 1,0\rangle\rangle$

Chapter 8: Limitations of Variation as an Effect

Our choice effect is closely related to similar choice effects described in previous work on algebraic effects [5, 19, 12, 18]. The main difference is that our choices also include a formula that can be used to coordinate choices in different parts of the program. However, both our work and previous work share a common limitation: choices implemented as effects are very inefficient compared to built-in language support for variation.

One of the core ideas underlying variational execution is that parts of the program shared among two or more variants are executed only once. However, this is unfortunately not the case for variation implemented via choice effects.

Consider the program below executing a variational list of numbers.

```
with v_handler handle
    [5;9;7;B⟨5,6⟩;A⟨11,10⟩]
;;
```

Ideally, we would like the raw elements to be visited once: elements 5, 9 and 7. Now consider doing an operational computation on this list such as summing up the list. To figure out how many times every element in the list gets executed in calculating the sum, we use a state counter. This counter is incremented every time an element is visited as follows in `sum`.

```
let rec sum l =
    match l with
        [] -> 0
        |(x::xs) -> #Set (#Get () + 1); x + sum xs
;;
```

We return the sum of the list along with the value of the state counter as follows.

```
with state handle
    with v_handler handle
        let l = [5;9;7;B⟨5,6⟩;A⟨11,10⟩] in
        (sum l, #Get ())
;;
```

Because this lists has 4 variants, every element in the list has been visited 4 times. The program above produces the following result.

```
[(A & B, (37,5));
(¬A & B, (36,10));
(A & ¬B, (38,15));
(¬A & ¬B, (37,20))]
```

This means that the size of the computation increases exponentially with respect to the number of options in the program, i.e., the complexity is $O(2^n)$. Unfortunately, we are not able to avoid this with the current approach, while it could have been more efficient to visit the list once until reaching the first variational element. The work in [15] deals with this issue by designing multiple variational stack implementations that avoid repeating shared parts. However, doing this for every kind of variation we deal with is overwhelming.

In this work, we chose to implement variational execution via choice effects for expediency. Our goal is to demonstrate how algebraic effects can be used to solve the problem of managing side effects during variational execution. However, a proper language for variational execution with side effects should include both algebraic effects and built-in support for variation. That is, built-in constructs for choices and optional values, and corresponding language run-time support for variational execution.

The language *Eff* we use to implement this work is a language under development, which makes this work more challenging. Its lack of features forces us to implement a basic incremental SAT solver to solve boolean formulas in our choices' contexts. If the language was more advanced with features, we could use a more efficient solver. Moreover, the language is lacking operations for file I/O and string manipulation features needed in parsing. This also forces us to alter the type system to introduce new types such as `channel` and `char`, import various string operations from *OCaml* and build a new compiler instance to work with.

Chapter 9: Related Work

In this section we discuss various related work. We briefly discuss a work that addresses the problem of integrating variation with side effects but does not solve it. We review in details a work that approaches the same problem we address and handles file I/O differently. Eventually, we compare our variation effect with the non-determinism effect, which is an effect with a similar behavior that has been implemented in previous work on algebraic effects.

9.1 VarexJ: A Variability-Aware Interpreter for Java Applications

This work builds a virtual machine (VarexJ) for testing software product lines in variational settings allowing the testing of all configurations at one run [13]. This machine has been successful in various aspects, but still suffers from the limitations of integrating side effects such as file I/O.

In this work, the authors could not deal with files for testing in variation, because the different configurations affect each other. For example, running the program to test with some configuration would read and write something to the file that would be changed or deleted when testing with another configuration. This could be managed with individual runs by resetting the file to the starting state, but with multi-execution software this is infeasible as the result of one configuration might overwrite the other. Solutions suggested, but not implemented, to the problem include new file models, separating files for different variation contexts, or variability-aware file content encoding.

9.2 Typed Faceted Values for Secure Information Flow in Haskell

A major concern in programs that carry sensitive data is enforcing a policy that keeps the data safe and prevent attacks or unauthorized accesses to it [3, 2]. A particular challenge is that the sensitive information in data can be leaked not only directly but also indirectly by the results of computations that use that data. Information-flow security is about preventing such leaks by allowing programmers to mark data that is sensitive and then tracking the flow of that data throughout the execution of the program. One approach to enforcing information flow security is secure multi-execution, which effectively executes two version of the program, "high" and "low", where the high execution corresponds to higher security and can use private data, while the low execution uses only public data.

The work in [2] introduced faceted values to simulate both versions of the program in one process. The faceted value is equivalent to the choice in our work where one alternative holds the private facet and the other holds the public facet. However, these programs can still leak sensitive information through implicit flow when side effects are invoked. Therefore, the work in [3] extends on faceted values to make facet-aware I/O operations such as mutable references cells (mutable states) and socket communication through channels (file I/O). We discuss the work on file I/O in the rest of this section.

Similar to our approach, the authors of [3] build a Haskell library that can be integrated into languages. They implement faceted values as monads separately from the side effects. Then, they extend them to work with side effects. Programs with side effects could be encoded as follows where `secret` is a faceted value.

```
do v <— secret
    do if v == 42
            then writeIORef x 1
            else return ()
        readIORef x
```

To figure out the path of the program flow, there is a program counter, which a list of selections (public or private) corresponding to the labels of faceted values used throughout the program.

This implementation allows simulating both: the private view and the public view of the program, which is the same goal we accomplish with the variation effect. However, we do not use a program counter to figure out the selections, we use the continuation which we invoke with true and false simulating all the variants of a program (variational execution). The return value of the continuation is the corresponding selection of the underlying choice label (option).

For file I/O and network sockets, the authors of [3] address the same issue we address that the file system or the channel have external environments that are not facet-aware. To make file I/O facet-aware, the authors introduce new operations that work with faceted values explicitly as shown below.

```
openFileF :: View -> FilePath -> IOMode -> FIO FHandle
hGetCharF :: FHandle -> FIO (Faceted Char)
hPutCharF :: FHandle -> Faceted Char -> FIO ()
```

In our work however, we introduce new handlers and avoid introducing new effects. Our handlers catch the built-in effects and enforce new behaviors on these effects. This makes our work flexible and extensible by allowing us to handle the same effects in multiple ways.

To avoid changing the external environment (the file system), the authors of [3] require raw data (either the public or private alternative) to be sent to the channel, or no data at all if the operation requires access to data that should not be accessed. Thus, the trick they use to work with file I/O on the library level is by attaching a view, which is a list of labels, to the file handle in `openFileF` operation indicating the privacy policy.

The view attached to the handle is checked against the program counter; if the program counter contains a label that is not in the view, the underlying operations are disabled. Consider the following program, which has a view attached to `openFileF` that contains the labels `"z"` and `"w"`. This means, only labels `"z"` and `"w"` allow a write to the channel.

```
do h <- openFileF ["z","w"] "file" WriteMode
    hPutCharF h (makeFacets "z" 'a' 'b')
```

Now, consider if the program counter was `["z","x"]`. Since the program counter contains `"x"`, which is not in the view attached to `openFileF`, no write should happen to avoid leaking information to "x". If the program counter was ["z"], character 'a' would be written. If the program counter was ["w"], character 'b' would be written. Similarly, the read operation below is protected by the view holding "k" and "l".

```
do h <— openFileF ["k","l"] "file" ReadMode
    hGetCharF h
```

In comparison to our work, we extend text files with the CPP format which lets us control every line of the file and flexibly read and write variational lines from the file. On the other hand, in the work above all labels are required to be mentioned upfront when the file is opened.

The main property we maintain is the preservation property which ensures symmetry; one alternative of the choice does not control the other alternative. In contrast, faceted values in [3] are asymmetric because the private alternative can control the public alternative, but the private alternative can not control the public alternative. Consider the example below.

```
hPutCharh k⟨'a','b'⟩
hGetChar h
```

Since only raw data could be sent to the file, either 'a' or 'b' could be written. If the user had access to the private alternative of k, the program would write 'a' into the file. Then, the second line of code returns k⟨'a', _⟩ to avoid information leak. While if the user had access to the public alternative of k, the program would write 'b' into the file. Then, the second line of code returns 'b' and not k⟨_, 'b'⟩ which breaks the symmetry property. The authors claim that this approach is more natural and avoids the challenge of keeping track of all labels.

9.3 Algebraic Effects for a Non-deterministic Effect

Work on algebraic effects and effect handlers has frequently used a choice as an example of a non-deterministic effect [5, 19, 12, 18]. These choice effects differ from our notion of choices in two key ways: (1) Each choice effect is independent, there is no concept analogous to dimensions to synchronize selections across choices. (2) The evaluation of an expression with choice effects yields an unstructured set of variants, rather than a structured variational value that clarifies the relationship between a sequence of selections and the variant it yields, as in our variation effect. A notable exception is the *selection functional* effect presented in [5], which essentially implements dimensioned choices in the *Eff* programming language. However, the control flow enforced by these encodings of choices rules out several ways to optimize variation-preserving computations. Since computations for different alternatives are performed in different continuations, we lose the opportunity to perform choice reduction to proactively eliminate unreachable alternatives [6], or to join converged execution paths early.

Finally, encoding choices as effects loses the ability to explicitly pattern match on them, which is often needed for variational analyses.

Chapter 10: Conclusion and Future Work

Integrating variation with side effects does not have a one-size-fits-all solution because each effect requires a special treatment and the solution for one effect might be incompatible with other effects. The best solution given the constraints is to solve this problem at the library level - giving the programmer the tools to deal with the interaction of variation and effects. As a proof-of-concept, we use algebraic effects to encode variation as an effect. Then, we extend the execution environment of variational programs to support file I/O. Algebraic effects allow programmers to handle new types of effects or handle existing effects in multiple ways. This makes them highly complementary with variational programs. As part of our contribution, we implement an abstract variational queue, which is a variational data structure that could be used in many applications of variational programming to incorporate variational data collections.

For future work, we could extend the execution environment of variational programs to handle new types of effects such as mutable states, standard input/output, and exceptions. We could also handle file I/O differently. For example, we could create a new file instance for every context initiated in the variational program. The file name of every file instance corresponds to its context. We already showed a simulation of this in Chapter 1. Our potential goal of this work is to include both algebraic effects and built-in support for variation in languages, that is, built-in constructs for choices and optional values, and corresponding language run-time support for variational execution. This would properly perform variational execution with side effects.

Bibliography

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

[2] Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 165–178, New York, 2012. ACM Press.

[3] Thomas H. Austin, Kenneth Knowles, and Cormac Flanagan. Typed faceted values for secure information flow in haskell. *Technical Report UCSC-SOE-14-07*, 2014.

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

[5] Andrej Bauer and Matija Pretnar. Programming with Algebraic Effects and Handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015.

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

[7] Sheng Chen, Martin Erwig, and Eric Walkingshaw. A Calculus for Variational Programming. In *European Conf. on Object-Oriented Programming (ECOOP)*, volume 56 of *LIPIcs*, pages 6:1–6:26, 2016.

[8] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *ACM/IEEE Int. Conf. on Software Engineering*, pages 335–344, 2010.

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

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

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

[12] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. *ICFP*, pages 145–158, 2013.

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

[14] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. On Essential Configuration Complexity: Measuring Interactions In Highly-Configurable Systems. In *IEEE Int. Conf. on Automated Software Engineering*, 2016.

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

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

[17] Hung Viet Nguyen, My Huu Nguyen, Son Cuu Dang, Christian Kästner, and Tien N Nguyen. Detecting semantic merge conflicts with variability-aware execution. In *Proc. of the Joint Meeting on Foundations of Software Engineering*, pages 926–929. ACM, 2015.

[18] Gordon Plotkin and John Power. Adequacy for algebraic effects. *FoSSaCS*, pages 1–24, 2001.

[19] Matija Pretnar. An introduction to algebraic effects and handlers. *MFPS 2015*, 2015.

[20] Thomas Schmitz, Dustin Rhodes, Thomas H. Austin, Kenneth Knowles, and Cormac Flanagan. Faceted dynamic information flow via control and data monads. In Frank Piessens and Luca Viganò, editors, *Principles of Security and Trust*, pages 3–23. Springer, 2016.

[21] K. Smeltzer and M. Erwig. Variational Lists: Comparisons and Design Guidelines. In *ACM SIGPLAN Int. Workshop on Feature-Oriented Software Development*, pages 31–40, 2017.

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

[23] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. Variational data structures: Exploring tradeoffs in computing with variability. In *Proc. Symp. New Ideas in Programming and Reflections on Software at SPLASH (Onward!)*, New York, 2014. ACM Press.

[24] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 85–96. ACM, 2012.

APPENDICES

Appendix A: Parsing the Boolean Formula of a Choice

```
let rec getChoice_ split_string left =

    let rec go xss left' =
        match xss with
            [] -> ((Lit false),[])
            | (x::xs) ->
                if (x =  ")") then
                    (left',xs)
                else
                    let (c',es') = (getChoice_ xss left') in
                    go es' c'
    in
    match split_string with
        [] -> (left,[])
        | (e::es) ->
            if (e =  "(") then
                go es left
            else if (e = "!") then
                let (l',es') = getChoice_ es left in
                (Not l',es')
            else if (e = "&") then
                let (r',es') =  (getChoice_ es left) in
                (And (left, r'), arr')
            else if (e = "|") then
                let (r',es') =  (getChoice_ es left) in
                (Or (left, r'),arr')
            else
                ((Ref e),es)
;;

let rec get_choice split_string left =
    let (c,l) =  getCh split_string left in
    match l with
        [] -> c
        | _ -> get_choice l c
;;

let parse_choice split_string =
    match split_string with
        [] -> (Lit false)
        | (x::xs)-> get_choice xs (Lit true);;
```