Handling Effects in Variational Programs

Ghadeer Al Kubaish and Eric Walkingshaw
Oregon State University, Corvallis, Oregon, USA
(*e-mail:* {alkubaig,walkiner}@oregonstate.edu)

Abstract

In order to test software in different configurations the traditional approach is to run every possible configuration one after the other. This process is tedious, inefficient and time consuming. Variational programming solves this problem by allowing all configurations to be run at once. Software testing is not the only application of variational programming; it has been used in other domains that involve variation such as enforcing information flow security and variability-aware execution. Even though this has been effectively used to avoid repeated executions in variational contexts, we still face the problem of dealing with variational side effects, such as state updates and file I/O. Variational programs suggest the need for special infrastructure to capture variational side effects but this would be too complex to build and in-feasible in some domains. For example, the file system and database would need to have a variational environment built into them. In this paper we provide an approach to work around this problem by providing a library that allows variational side effects using algebraic effects.

1 Introduction

Variational programming Erwig & Walkingshaw (2013); Chen *et al.* (2016) 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* Austin & Flanagan (2012); Yang *et al.* (2012); Austin *et al.* (2013); Schmitz *et al.* (2016); Austin & Flanagan (2014); Pretnar (2015) for enforcing dynamic information-flow security, and *variability-aware execution* Nguyen *et al.* (2014, 2015); Meinicke *et al.* (2016) for testing software product lines, and has a range of other applications across computer science.

A core insight of variational programming is that the execution of sets of related programs, and/or of 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 following two related arithmetic expressions, e_1 and e_2, and the corresponding *variational* arithmetic expression e_v.

$$e_1 = 2 \times 3 + 4 \qquad e_2 = 2 \times 3 + 5 \qquad e_v = 2 \times 3 + A\langle 4, 5\rangle \tag{0}$$

The subexpression $A\langle 4, 5\rangle$ represents a *choice* between the alternatives 4 and 5. The *condition* of the choice is an *option* A, 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. The notation and semantics of choices is from the second

author's previous work on the *choice calculus* Erwig & Walkingshaw (2011); Walkingshaw (2013); Hubbard & Walkingshaw (2016).

We can extend the evaluation semantics of Haskell to include choices by simply mapping over their alternatives. This allows us to evaluate e_v with the following sequence of reduction steps.

$$\underline{2\times 3}+A\langle 4,5\rangle \;\mapsto\; \underline{6+A\langle 4,5\rangle} \;\mapsto\; A\langle \underline{6+4},6+5\rangle \;\mapsto\; A\langle 10,\underline{6+5}\rangle \;\mapsto\; A\langle 10,11\rangle$$

The choice $A\langle 10,11\rangle$ represents the result of evaluating both variants encoded by e_v, but observe that we only evaluated the subexpression 2×3 once rather than twice if we had evaluated e_1 and e_2 separately. Although the savings here is small, it adds up as the size of shared subexpressions and the number of independent options increase. 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.

2 Background

As a background, we provide details about algebraic effects, what they mean and how they are used. The programming language used in this work is called *Eff*[1] which is inspired by the programming language OCaml and has similar syntax and features. Eff has a new feature that did not exist in Ocaml called *algebraic effects* which is the main tool used in our project. We will be using Eff syntax throughout this paper in our coding examples.

Eff is based on *algebraic effects* handlers. Algebraic effects expand on the traditional exception handling mechanism by allowing computational effects to be thrown and caught by handlers that perform computations. Handlers in Eff are able to redirect outputs and modify state. Multiple handlers can be written for the same effect which increases their power and flexibility. Effects are first-class citizens in Eff which means they can be combined easily. Eff allows programmers to defined their own effects by declaring them. We use the more intuitive word *operation* very often in this paper to refer to an effect.

To get a sense of how algebraic effects are used and defined, refer to the following example of the integer state handler from Pretnar (2015). We will use Eff's new syntax to write this example. Two operations are declared for writing a state in algebraic effects: *set* and *get*. *Get* takes a unit value and returns the corresponding state value. *Set* takes a new state value and returns a unit value. The handler is defined in a case pattern where each case is denoted by an effect. In the case of *get*, the handler returns a function which takes the current state and passes it back unchanged. In the case of *set*, the handler returns a function which takes the a unit value and updates the state to the new value passed to *set*. The state is initialized to 0 in the case finally. In this handler, we use *k* to denote continuation, which allows handling other effects within the code block. The details of the state handler are in Figure 1.

The state effect can then be used easily as in Figure 2. We initiate the handler bound using the following.

with handler_name handle

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

```
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
;;
```

Fig. 1. Integer state effect using algebraic effects

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

Fig. 2. Using the state handler

Any effect operation that occurs within that code block will be within the bound of this state handler. This code sets the state to 5, then increments it by 1, and then evaluates the state again with the #Get() operation which gives the result of 6.

3 Variational execution

The choice calculus is the library we use to formally represent variational values in our programs. It allows us to build large configurable systems and compute variational results. Variational execution is the concept of running a variational program once and getting a variational result. The main property to be maintained in variational execution is variation preservation. This property states that ....

The work done in Austin & Flanagan (2014) shows the use of faceted execution to enforce dynamic information-flow security. The inspiration behind their work is the prevention of 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 variational values or choices is faceted values. They use faceted values to dynamically simulate multiple executions of the the same program under different private policies.

The following is an example that shows how to enforce a security level on certain values by marking them as private. Only people with the correct password key can access the private value. We declare a user record with a name, data of birth, security question, and social security number. We declare a variable for each element of the record data which will then be passed to an instance of the user record type after checking the key. The variable $name'$ has no security level enforced while all other variables do. The variables: $securityQuestion'$, $dataOfBirth'$, $socialSecurity'$ are set to undefined when the password is wrong. As a result, the variable visitor, which is an instance of the user type, would only have the name set while other fields are undefined. If the password is correct, all fields are set to the correct corresponding values.

```
    type User = {name: string; dataOfBirth: date;
```

```
                securityQuestion: string; socialSecurity: int}

    let name' = "John Smith" in
    let dataOfBirth' =  private ("password") "9/9/1999" in
    let securityQuestion' =  private ("password") "Mother's maiden name?" in
    let socialSecurity' =  private ("password") "000 00 0000" in

    let visitor = {name = name'; dataOfBirth = dataOfBirth';
                securityQuestion = securityQuestion';
                socialSecurity = socialSecurity'}
    in visitor

  ;;
```

Next, we show how we can write variational values using algebraic effects. This is done by defining a choice that can be thrown and then caught by a handler. This handler captures multiple results of different configurations. As we mentioned in section 2, effects can be combined together, which allows us do the following basic computation.

```
with choose_all handle
    A⟨10,A⟨20,30⟩⟩ + B⟨1,A⟨2,3⟩⟩
;;
```

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

```
with choose_all handle
    chc "A" 10 (chc "A" 20 30) + chc "B" 1 (chc "A" 2 3)
;;
```

This basic computation adds two variational values together where each value has nested configurations. The handler *choose_all* evaluates the final variational result by visiting every branch of the choice and finding its corresponding value. The handler is triggered every time a choice is encountered in its scope.

In the above computation, there are 4 possible results based on the two configurations A and B. If A and B are true, the result is 11. If A is true and B is false the result is 12. If A is false and B is true, the result is 31. If A is false and B is true, the result is 33. This result is captured by the following choice notation.

$$A\langle B\langle 11,12\rangle, B\langle 31,33\rangle\rangle$$

Another basic application is constructing a variational list. The basic list notation is extended by the choice notation to allow capturing different lists under different configurations. The notation _ represents nothing or an empty element. For example, the third element in the list is 10 when the choice A is false. Otherwise, it is empty or does not exist. Writing a variational list using the choice operation is as follows.

```
with choose_all handle
    [5,6,A⟨_,10⟩,B⟨_,30⟩,B⟨_,A⟨_,3⟩⟩]
    ;;
```

The result should capture all possible combinations of configurations and provide the list corresponding to each configuration value. The following choice represents the 4 different lists resulting from evaluating the example above.

$A\langle B\langle[5,6],[5,6,30]\rangle,\rangle B\langle[5,6,10],[5,6,10,30,3]\rangle$

4 Variational Programming and Effects

Variational programming does not support side effects which are common to real world programs. Working with side effects is challenging in a variational setting as we would need to implement a variational environment for every variational effect we need to work with. This process is tedious and in-feasible for certain types of effects. For example, for variational file I/O, we need a variational file system to keep track of every variational context in which the read and write happens. For variational database queries, we need variational databases and so on. Moreover, some dimensions of variation correspond to differences in platforms which can not be simulated together. Because of these issues, we want to give the programmer the tools to deal with the interaction of variation and effects in an expressive and systematic way. Our approach provides the programmer with the libraries needed to do variational side effects using algebraic effects.

In the rest of this section, we show examples of variational effects and how they occur in various applications. The effect operations we show are mainly file I/O operations which are a significant portion of our work. We show variational read and write operations implemented using algebraic effects.

The following example is again inspired by the work of enforcing dynamic information-flow security in Austin & Flanagan (2014). In the previous example in section 2, there are no side effects within the program. This example involves side effects and enforce security levels on them. The secret value is assigned to the correct password if the key is true. Otherwise, it is undefined. Then, if the secret value is set, the state is incremented by one. Otherwise, it prints an error message. Therefore, different side effects are performed in different levels of security.

```
let secret = private ("password") "MyPassword"  in

with state handle
    (#Set(0))
    if secret
        then
            (#Set(#Get() + 1))
        else
            print("wrong password")
;;
```

4.1 Variaitonal File I/O

A major contribution of our work in variational effects is in file I/O. We introduce the format we use for writing variational files which is commonly used in many applications. We show how the use of varational file operations, specifically, read and write. Finally, we show applications that emphasize the importance of this work.

4.1.1 Variaitonal File Format

The format we use for writing variational text files is the same format of C-Preprocessors [2] or what's called CPP. C-Preprocessor is a text substitution tool that must to be processed before compile time. We choose this format because it is a real world widely used format which makes our work useful for existing applications. Our work in compared to C-Preprocessors can be processed dynamically at run-time and not at pre-compile time. Variational programming is based on the idea of running a variational program once and getting the variational results of all variational contexts which is not feasible in C-Preprocessors runs one set of configurations at a time. C-Preprocessors has many commands but the ones relevant to our work are #if, #else and #endif. Later in our programs we use ifdef instead of if to represent an if effect operation because if is a reserved word. However, we can use the word if in variational text files.

Figure 3 shows a possible variational file which uses the format mentioned above. Note that text files are extended by this format and not forced to have it. We want our library to work as an additional feature that works flexibly with programs and files that do not involve variation. Only when variation exists a special syntax is needed. The file shown in figure 3 is a real example from Massart college for Art and Design showing their housing prices of 2018-2019 [3]. This file is an explanation of costs in different scenarios for their students.

The file starts with one introductory line which is not variational. The next is a nested variational line. When the configuration "MEAL" is true, we have the variational line which when the configuration "Partial" is true, gives this line "$1,902" otherwise, gives this line, "$3372". Note that it is not necessarily to have an else when there is an if, but there must have an endif whenever there is an if to ensure balance. Also, note that the third variational line has in its nested variaitonal line an OR expression; "Single" | "Efficiency" because configurations can be represented using boolean formulas; more about that will be discussed in section 5.

4.1.2 Variaitonal File Write

In order to write variational lines to files we need a method to specify the context in which the write happens. As we mentioned above, we use the format of the c-Preprocessors. In the example file in figure 3 we show a text file with variational lines while this example below shows actual effect operations that write to files. We are using two sets of operations in this example, first, ifdef(config), else and endif which are responsible for updating the

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

```
Below is a breakdown of the preliminary housing costs for
the 2018-2019 academic year.
#if "MEAL"
    #if "Partial"
        $1,902
    #else
        $3372
    #endif
#endif
#if "ROOM"
    #if "Activity Fee"
        $20
    #endif
    #if "Technology Fee"
        $650
    #endif
#endif
if "ARTISTS' RESIDENCE"
    Partial (default) or optional regular meal plan.
    Total Including Activity/Technology Fee:
    #if "Single" | "Efficiency"
        $13,360
    #else
        $12,216
    #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"
            $11,234
        #else
            $10,530
        #endif
    #else
        #if "Single"
            $13,360
        #else
            $11,002
        #endif
    #endif
#endif
```

Fig. 3. File example

context, second, writefile "text" which does the actual write. Note that this implementation allows writing to files with no variation and a special syntax is needed only when writing variational lines.

In the example below, we start by opening the file and then we use the handler "write_handler" to deal with the write operations which takes the file that we opened as input. Using the handler we write the first few lines of the example file in figure 3. A typical close file operation has to be done after finishing.

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

with write_handler w_file handle
    writefile "Below is a breakdown ...."
    writefile "These charges are subject to change."
    ifdef ("MEAL");
        ifdef ("Partial");
            writefile "$1,902";
        else;
            writefile "$3372";
        endif;
    endif;
;;

 #Close_out w_file;;
```

4.1.3 Variaitonal File Read

The same principle of file writing applies to file reading. We define read operations and other operations: ifdef(config), else, and endif to update the context in case a variational read is needed. The difference is that in reading files, we need to maintain a variational stack that helps keeping track of different varational contexts. To clarify this refer to figure 4 which visualizes the first two lines of the example file in in figure 3. Rectangles represent lines of text in the file, rounded rectangles represent holes or nothing, off page pointers represent choices (the left side of the choice represents the true alternative and the right side represents the false alternative), and horizontal arrows represent file pointers labeled with a context.

In the beginning, the file pointer is at the first line and the context is true. After performing a read, it returns back the first line as in figure 4. After performing a second read in context (meal & partial), the file pointer is at the true branch of the choice (partial) in line 2, and so it returns "$1,902" as in figure 5.

After performing a third read but in context (meal & ! partial), we end up with two file pointers at different locations. One with context (meal & ! partial) pointing to the right branch of partial in line 2 and the other with context (meal & partial) pointing to the third line, and so he third read would return "$3372" as in figure 6.

These figures show why we need to maintain a variaitonal stack and why variaitonal reading is very complicated. Our approach in implementing the variational stack is different from the file pointer approach and will be discussed in section 5. For now, we use this variational stack as a black box that is needed to perform read operations in various contexts.

In the example below, we open the file and read the whole file into a variational stack; using read_v file. After that, the file could be closed because the stack would be used instead to represent the file. The handler read_handler takes the stack as its input and performs

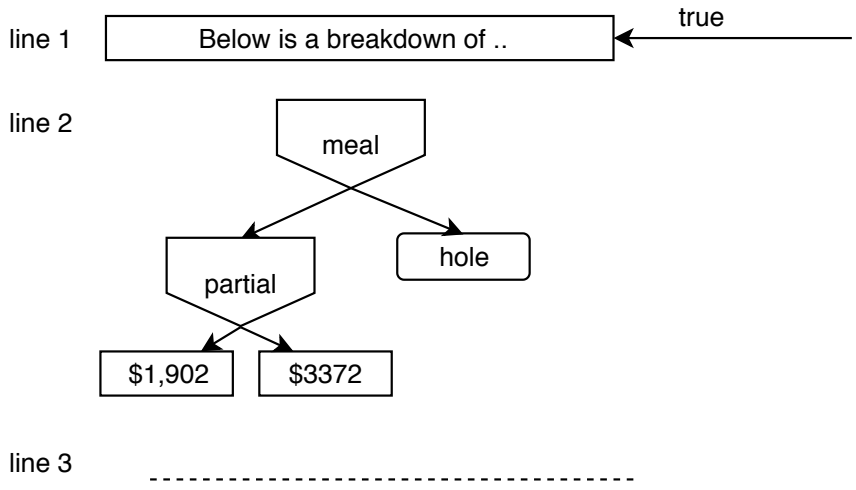

Fig. 4. file representation

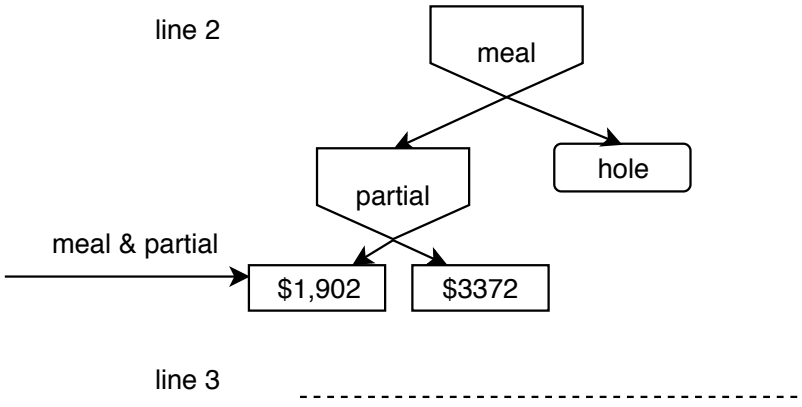

Fig. 5. Variational file read

all the file reading by catching read operations. This code shows the same scenario of the figures explaining file pointers.

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

with read_handler s handle
```

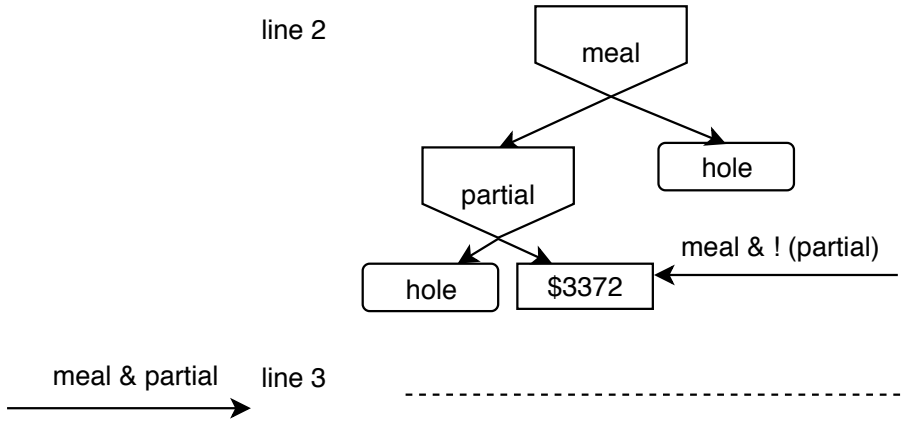

Fig. 6. Variational file read

```
    readvline;
    ifdef ("MEAL");
        ifdef ("Partial");
            readvline;
        else;
            readvline;
        endif;
    endif;
;;
```

After understating basic read operations, we look at more interesting applications for reading variational lines from files. We may perform something useful on our reading results such as filtering, or concatenation. Note that the result of a read operation might be a choice itself and not necessarily a line of text. In this case we can combine our work in reading and choice operations. The read operation returns a choice operation and then the choice handler catches the choice and computes the variational result. The two handlers corresponding to the two operations are nested in order to accomplish that.

In the following example, we concat the results of multiple read operations into one string. Because the result of any read operation might be a choices, the final concatenated string would be variaitonal. Consider the following sequence of read operations on the example file in figure 3.

```
with choose_all s handle
    with read_handler handle
        let (a1,s') = readvline s in
        ifdef ( "MEAL");
            let (a2,s') =  readvline s' in
        endif ();
        ifdef ("Room" & "Technology Fee");
            let (a3,s') =  readvline s' in
```

```
        endif;
    endif;
        a1  ^ " : " ^ a2  ^ " : " ^ a3
```

The results a1 and a3 are lines of text while the result a2 is a choice as shown below.

```
a1 =   "Below is a breakdown of the preliminary... "
a2 = Partial⟨"$1,902","$3372 "⟩
a3 =   "$20"
```

Therefore, the result of concatenating the results is the following variational value.

```
Partial⟨"Below is... : $1,902 : $20","Below is... : $3372 : $20"⟩
```

This implementation enables us to evaluate all alternatives without having to numerate through each of them. Eventually, we get the variational value corresponding to the different alternatives.

The next example is another useful case which shows how variational programming on side effects can prevent from repetitive work. Being able to read all alternatives of the file at once, allows performing a filter operation on each alternative dynamically at once and getting the variational result resulted from all alternatives. The following program counts the number of times the word "partial" occurs in any alternative of the variational file. Note that s is the variaitonal stack representing the file.

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

Running this program on the example file in figure 3 will result 2 when the two configurations "ARTISTS' RESIDENCE" and "SMITH" are true, 1 if one of the two configurations is true and the other is false, and 0 when both are false.

5 Variational Effects in Eff

In this section we will be discussing the details of our entire library in Eff. We will start by discussing our algebraic effect definition of the choice which is the base for working with variational values, and then our file I/O library which involves our implementation for read and write as variaitonal operations.

5.1 The Choice Library

The choice library is an algebraic definition which consists of an effect and a handler. The choice effect is declared to take a variational context (ctx) and returns a boolean value. The variational context is represented by a boolean formula of ORs, Ands and Nots. The handler choose_all catches the choice by invoking the continuation with either true or false, and then concatenating the results it gets back. The handler uses select to avoid duplication

```
effect Choice : ctx -> bool;;
let chc d l r = if #Choice d then l else r;;

let choose_all = 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)]
;;
```

Fig. 7. The choice library

by removing the context that already exists in the result, which is part of the reduction process we discussed in the introduction. Prepend adds the context and its result to the list. We apply Prepend twice: once with the given context and once with the negation of the given context. That's why we are able to simulate all possible configurations by running the program once. There are 2^k possible results with k distinct configurations. The result of invoking the continuation with either true or false is in the form (ctx, a) and all results are concatenated in a list as you see in the val case. Note that the value x can be of any type, and so this definition is general to work in different domains. The choice library is shown in figure 7.

5.2 File I/O Library

In this section we introduce our File I/O library. We mainly discuss how to define our C-preprocessor format as algebraic effects which we use to wrap our read and write operations. For file I/O read, we will be discussing our stack definition which is needed to maintain the veriaitonal context in which the read operation is performed as discussed in section 4.1.3.

5.2.1 Variational Context Update

When it comes to working with side effects, it is useful to maintain a *variational context*; will use context for short. If we run a side effect with no variations, our context is true. If we run it in with variation, our context should be updated accordingly. As already discussed in section 4.1.1, we use the c-preprocessor format to define the three main operations we need *ifdef*, *else*, and *endif*. We will be showing the details of how these operations work.

In order to maintain a variational context, we need to use the state handler which is another algebraic effect definition discussed in the background 2. For our current purpose, our state is defined as a tuple. The first argument in the tuple is the current context and second argument is a stack of contexts. Our first operation *ifdef* takes a boolean expression of configurations. It updates the state by pushing the new configuration into the stack of contexts and then updating the current context to be the conjunction of all contexts in the stack. Our second operation *else* takes no arguments. It updates the state by popping the top element of the stack, pushing the negation of it into the stack, and then updating the current context to be the conjunction of all elements in the stack. Our third operation *endif*

```
let ifdef d =
    let (s, l) = #Get() in
    let l' = (d:: l) in
    #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''))
;;

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

Fig. 8. C-preprocessor format for updating the context

takes no arguments. It updates the state by popping the top element of the stack, and then updating the current context to the conjunction of all elements left in the stack. The entire implementation is shown in figure 8.

5.2.2 File Write implementation

In order to write to a file, all we need is a basic Write_file operation which works as a side effect in a normal Eff program. We define a complete Algebraic effect definition for writing to files because we want to use the continuation feature which allows us to perform multiple writes in the same code block. Therefore, we declared an effect WriteFile to take string and returns a string. Then a handler write_handler which performs the write into the file using Write_file which is a built-in function.

Now, to write variaitonal lines, we need to write another algebraic effect definition with the following set of effects: Ifdef, Else, and Endif and the handler ifdef_handler. Note that the operations we defined in 4.1.1 are overwritten here to invoke the appropriate effect operation after doing their pre-discussed work in updating the context. Therefore, all we need to do in the handler is take the context from the state and write it into the file. Because of that, we declare the effects to take a unit and return a unit. The details of the handler are straightforward; all it does is write the keyword relevant to its case plus the context in the ifdef case. The effect #WriteFile is wrapped in a function writefile for easier use. The entire library is in figure 9.

```
effect WriteFile :string -> string ;;
effect Ifdef :unit -> unit ;;
effect Else :unit -> unit ;;
effect Endif :unit -> unit ;;

let ifdef d =
    ...
    #If ()
;;
let else () =
    ...
    #Else ()
;;
let endif () =
   ...
   #Endif ()
;;

let write_handler w_file = handler
    | #WriteFile t k ->
         #Write_file (w_file, t ^ "\n");
     k t
;;

let ifdef_handler w_file = handler
    | #Ifdef () k ->
         #Write_file (w_file, "#if ");
         let (s,l) = #Get() in
         k (#Write_file (w_file, printCtx (s )^ "\n"))
    | #Else () k ->
        k (#Write_file (w_file, "#else\n"))
    | #Endif () k ->
        k (#Write_file (w_file, "#endif\n"))
;;

let writefile t = let s = #WriteFile t in s;;
```

Fig. 9. The write library

5.2.3 File Read implementation

Similarly to file write, all we need to do perform a read operation is use a built-in operation as a side effect in any program, but we need a complete library to allow this operation to happen in a variational context. Moreover, instead of working directly on the file, we will be working on a stack (variational stack) which will simulate where we are in the file in a given variational context. The inspiration for this was discussed in section 4.1.3. Therefore, after reading the entire file into the stack, our read operations will be stack pop operations. In this section we will use the stack as a black box and in the next section, we will discuss its implementation details.

```
effect ReadVLine : vstack -> (string * vstack) ;;

let read_handler  = handler
    | #ReadVLine l k ->
            let (s',ls) = #Get() in
            let (a,l') = read_line l (s') in
            k (makeChc a, l')
;;

let readvline () = #ReadVLine () ;;
```

Fig. 10. Read line library

The algebraic effect definition for reading a line consists of an effect, ReadVLine and a handler read_handler. ReadVLine effect operation takes the variational stack as its first argument and returns a tuple consisting of the line read and the updated stack. The handler extracts the context from the state, perform the read on that context and then return the line read with the updated stack. The variational context is set to true by default and so only when we need the operation to happen in a variaitonal context, we use the operations in 4.1.1. Note that we return the result by invoking the choice effect because our line might be a variational line and not a plain text, so we return the result as a choice using the function makeChc, which would then be handled by the choice handler. We wrap the effect in the function readvline for easier use. The entire library is shown in figure 10.

5.2.4 Variational Stack implementation

The variational stack is mainly implemented to allow us to perform variational read line operations as discussed in section 4.1.3. Therefore, we introduce the Opt format which is a tuple of a value and a context. Below is the Eff type definition of the opt stack.

```
type 'a opt = ('a * ctx) ;;
type 'a stack = ('a opt ) list;;
```

With this format, every line in the file is inserted into the stack with its context. For example, if we write the first two lines of our example file in figure 3 into our opt stack, this is the resulted stack.

```
[("Below is a...", true); ("$1,902" , Meal&Partial); ("$3372", Meal&!Partial)]
```

This format allows us to skip elements that we already read in a given context, but at the same time introduces holes which are a bit hard to work with. For example, if our example file starts with the second line and we read in context (Meal&Partial), we pop the line $\ddot{\$}$1,902; but introduce a hole in context (Meal&Partial). If we then came to read in a different context, for example, in the context (Room), we get the variational line $Meal\langle Partial\langle$"$1,902","$3372"$\rangle,_\rangle$ because the line $\ddot{\$}$1,902$\ddot{}$has not been read in the context (Room) even through it been read in the context (Meal&Partial). Working with holes and updating values' contexts is all done in the pop function. The resulted line from

pop has the a v type which is either a hole, one value or a choice (variational value). The definition of pop is shown below.

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

let rec popHelper cIn s v =
    match s with
    [] -> (v, [])
    | ((a,cElem)::os) ->
    if unsatCtx cIn then (v, s)
    else
        let cPop = And (cIn, cElem) in
        let cRem = And (Not cIn, cElem) in
        let cToDo = And ( cIn, Not cElem) in
        let  (v', s') = popHelper cToDo os (Chc (cPop, (One a), v) ) in
        (v', (a,cRem)::s')
;;

let pop c s = popHelper c s Hole ;;
```

When we pop an element, we have to think of three different contexts: the context of the element we pop (cPop), the context of the element that remains in the stack (cRem) and the context of the hole we need to fill (cToDo). The context (cPop) is the conjunction of the two contexts: the context we perform the pop at (cIn) and the context of the top element in the stack (cElem). The context (cRem), is the the conjunction of the two contexts: the negation of the context we perform the pop at (cIn) and the context of the top element in the stack (cElem). The context (cToDo) is the conjunction of the two contexts: the context we perform the pop at (cIn) and the negation of the context of the top element in the stack (cElem). The helper function popHelper is recursively called on the remaining of the stack (os) with the context of the hole to be filled (cToDo) and at the same time, it builds the choice of the value to be returned (Chc (cPop, (One a), v)). The stack is updated with every call so that every element we pop gets the updated context (cRem). when the context (cIn) is not satisfiable, then we now we should stop looking. We also run a lot of simplification functions to clean up the stack and the contexts with every pop.

6 Efficiency

The problem main problem that arises with the current implementation is efficiency. Even though we are able to avoid repeated executions from the user end, we are still visiting still repeating the execution of the code with every possible set of configurations. The size of the computation grows exponentially with respect to the number of configurations; $O(2^n)$. To illustrate how the computation gets repeated with respect the number of choices, consider the following program. This program uses the state handler to count the number of time an element is visited in the list when the list involves variaitonal elements. Every time an element is visited, the state is incremented as a side effect. Note that we have 4 possible

lists in this program because we have 2 configurations A and B and each list has 5 possible elements.

```
with state handle
    (#Set(0));
        let rec listLoop l = handle
                match l with
                 [] -> #Set(#Get()+ 1); 0
                 | x::xs -> #Set(#Get()+ 1); 1 + listLoop xs
        with
            | #Count () k -> #Get(); k()
        in
        with choose_all handle
            let l = [5; A⟨6,3⟩ ; B⟨6,_⟩ ; 9; 7]
        in
        listLoop l;
        #Get()
;;
```

The result of the program above is the following. When $A \ \& \ B$ are true the state count is 5, which means every element is accessed once. Then When $\neg A \ \& \ B$ the count is 10 which means every element in the list has been accessed again, and so one.

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

In the ideal case we do not want every element to be accessed once and only the choice has to be accessed twice; once for every alternative. For example, 5, 9 and 7 are shared among all lists and the list varies only in second and third element. Ideally, only the second and third elements should be visited twice and all other elements should be visited once, however, we can not make this optimal solution happen. Therefore, we argue that having built-in variational features in the language can improve efficiency in contrast to the current solution which enforces an external library into the language to work with.

7 Related Work

[Following paragraph cut-and-pasted from ECOOP paper. Adapt and revise.]

Work on algebraic effects and effect handlers has frequently used a choice as an example of a non-deterministic effect **????**. 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 choice that clarifies the relationship between a sequence of selections and the variant it yields, as in .... A notable exception is the "selection functional" effect presented in **?**, which essentially implements dimensioned choices in the *Eff* programming language. Both choice effects and selection functionals are more restrictive than our choices since they do not

permit the alternatives of the choice, or the produced variants, to be values of different types. Additionally, 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 Chen *et al.* (2014), or to join converged execution paths early. Finally, encoding choices as effects loses the ability to explicitly reflect on the structure of a variational expression and perform transformations. This ability is often needed for variational analyses, for example, the ability to commute selections with computations is a critical transformation for analyzing SPLs **?**.

8 Conclusion

Bibliography

Austin, Thomas H., Knowles Kenneth, & Flanagan, Cormac. (2014). Typed faceted values for secure information flow in haskell. *Technical report ucsc-soe-14-07*. To appear.

Austin, Thomas H., & Flanagan, Cormac. (2012). Multiple facets for dynamic information flow. *Pages 165–178 of: Proc. Symp. Principles of Programming Languages (POPL)*. New York: ACM Press.

Austin, Thomas H., Yang, Jean, Flanagan, Cormac, & Solar-Lezama, Armando. (2013). Faceted execution of policy-agnostic programs. *Pages 15–26 of: Proc. ACM SIGPLAN Work. on Programming Languages and Analysis for Security*. ACM.

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

Chen, Sheng, Erwig, Martin, & Walkingshaw, Eric. (2016). A Calculus for Variational Programming. *Pages 6:1–6:26 of: European Conf. on Object-Oriented Programming (ECOOP)*. LIPIcs, vol. 56.

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

Erwig, Martin, & Walkingshaw, Eric. (2013). Variation Programming with the Choice Calculus. *Pages 55–99 of: Generative and Transformational Techniques in Software Engineering IV (GTTSE 2011), Revised and Extended Papers*. LNCS, vol. 7680.

Hubbard, Spencer, & Walkingshaw, Eric. (2016). Formula Choice Calculus. *Pages 49–57 of: Int. Work. on Feature-Oriented Software Development (FOSD)*. ACM.

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

Nguyen, Hung Viet, Kästner, Christian, & Nguyen, Tien N. (2014). Exploring variability-aware execution for testing plugin-based web applications. *Pages 907–918 of: Proc. Int'l Conf. Software Engineering (ICSE)*. New York: ACM Press.

Nguyen, Hung Viet, Nguyen, My Huu, Dang, Son Cuu, Kästner, Christian, & Nguyen, Tien N. (2015). Detecting semantic merge conflicts with variability-aware execution. *Pages 926–929 of: Proc. of the Joint Meeting on Foundations of Software Engineering*. ACM.

Pretnar, Matija. (2015). An introduction to algebraic effects and handlers. *Mfps 2015*. To appear.

Schmitz, Thomas, Rhodes, Dustin, Austin, Thomas H., Knowles, Kenneth, & Flanagan, Cormac. (2016). Faceted dynamic information flow via control and data monads. *Pages 3–23 of:* Piessens, Frank, & Viganò, Luca (eds), *Principles of Security and Trust*. Springer.

Walkingshaw, Eric. (2013). *The Choice Calculus: A Formal Language of Variation*. Ph.D. thesis, Oregon State University. `http://hdl.handle.net/1957/40652`.

Yang, Jean, Yessenov, Kuat, & Solar-Lezama, Armando. (2012). A language for automatically enforcing privacy policies. *Pages 85–96 of: ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. ACM.