Fixed-Point (Value) Recursion with Algebraic Effects and Handlers in Haskell

Gijs van der Heide[1]
Supervisor(s): Casper Bach Poulsen[1]**, Jaro S. Reinders**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

Abstract

Algebraic effects and handlers are a new programming technique that allows for the definition of abstractions as interfaces, with handlers providing modular, concrete implementations of these interfaces. In this paper, we consider algebraic effects and handlers implemented in Haskell, and explore how they behave under fixed-point (value) recursion. We give different possible implementations of fixed-point combinators for effectful functions, and work out their evaluation processes. We find that these functions behave very predictably under normal fixed-point recursion, while value recursion seems to be a much harder problem. We discuss the difficulties of implementing value recursion, and several possibile solutions are explored, but the question of whether a fixed-point combinator with value recursion semantics can exist at all in the presence of algebraic effects remains unanswered.

1 Introduction

Algebraic effects and handlers are a new programming technique. Effects allow for the definition of abstractions as interfaces, while handlers modularly provide concrete implementations of these interfaces. Possible effects include, but are not limited to, stateful computation, I/O operations, and exceptions. Using these abstractions may make it easier to write complex programs with effects, and to verify that they behave as intended. The main benefits come from this modularity and the separation of interface and implementation, as well as the making explicit of the effects used in a certain program.

Because algebraic effects and effect handlers are still mostly a topic of research, there is usually no out-of-the-box support in mainstream programming languages, and it is usually not immediately clear what would be the best way to define the infrastructure needed to work with algebraic effects. In this paper, we will work with algebraic effects and handlers in Haskell using the approach described by Swierstra [1] and a series of blog posts by Bach Poulsen [2].

Because recursively defined functions are very common in functional programming, one of the questions that naturally arises when using these algebraic effects and handlers is how they interact with recursive computations. Erkök [3] notes that there is very little research into the interaction between effects and recursion, and specifically value recursion - recursion over the result values of a computation while only executing side-effects once. They analyze effects encoded with monads and use fixed-point combinators as a theoretical framework for recursion. We will attempt to apply similar principles, and also focus on fixed-point combinators, but consider effects encoded as algebraic effects with handlers instead. We aim to explore and explain how these algebraic effects and handlers behave under fixed-point (value) recursion, and to also prove certain laws pertaining to this recursion.

To this end, the main research question of this paper is "How can effectful fixed-point (value) recursion be used in combination with algebraic effects and handlers in Haskell?". To answer this question, we propose the following subquestions:

- What motivates the need for effectful fixed-point (value) recursion with algebraic effects and handlers in Haskell?
- What is the intended behavior of effectful fixed-point (value) recursion in Haskell and how can this behavior be implemented?

- What are the mathematical laws pertaining to effectful fixed-point (value) recursion, and can the provided implementation in Haskell be proven to respect these laws?

The contributions made in this paper consist of a description and (formal) analysis of a way of working with recursion with algebraic effects and handlers in Haskell. We also describe the difficulties and possible solutions for working with value recursion using algebraic effects and handlers. Unfortunately, none of the discussed solutions were able to adequately implement value recursion semantics. We hope that the discussion in this paper can serve as the basis for future research into the problem of value recursion with algebraic effects and handlers.

In section 2, we will discuss background information on the way we will work with algebraic effects and handlers in this paper, what fixed-point recursion is exactly, as well as what value recursion is. In section 3, we will discuss the solutions found to working with (value) recursion and algebraic effects and handlers. Then, in section 4, we focus on normal recursion. Some example programs that use effects and recursion will be provided to motivate the need for the implementations discussed in section 3, as well as to show how and why these implementations work. In section 5 we focus more on value recursion. Because no working implementation was found, we will instead discuss laws that should hold for a possible value recursion operator, and we discuss the possiblity and challenges of deriving a value recursion operator from these laws. Then, in section 6, we provide some insight into the responsible research concepts relevant to this paper. Finally, we provide a short discussion of the results, a conclusion and recommendations for future work in section 7.

We also provide an introduction to working with algebraic effects and handlers in Appendix A, and discuss how to access the full code for this paper in Appendix B.

2 Background

In this section, we will quickly discuss our method of working with algebraic effects and handlers in Haskell, and provide explanations of fixed-point recursion and value recursion.

2.1 Algebraic Effects and Handlers in Haskell

As noted in section 1, the approach taken to working with algebraic effects and handlers in Haskell in this paper is based on the one described by Bach Poulsen [2]. It is highly recommended to read the relevant blog posts. Alternatively, a short explanation discussing only those details relevant to understanding this paper has been included in Appendix A.

2.2 Fixed-Point Recursion

Like Erkök [3], we will use fixed-point recursion as a theoretical framework for recursion in this paper. Functions written in this way are not inherently recursive, but rely on a fixed-point combinator to make them recursive. This fixed-point operator is defined as:

$$fix f = f (fix f)$$

A simple fixed-point combinator can then be implemented in Haskell like so:

```
fix :: (a -> a) -> a
fix f = let x = f x in x
```

Note how this combinator takes a function f and creates a repeated application of this function - *f(f(f(...)))*. Due to Haskell's laziness, *fix f* is not immediately expanded to an infinite sequence of applications of *f*, and it is in fact possible to write terminating recursive functions with *fix*. This does require f to indicate in some way when to stop the recursion. We will demonstrate what this could look like by implementing a function that calculates the factorial of a number n - given by $(n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 1)$.

```
factorial :: (Int -> Int) -> Int -> Int
factorial f n
    | n <= 1 = 1
    | otherwise = n * (f (n - 1))

fact :: Int -> Int
fact = fix factorial
```

The *factorial* function certainly looks very similar to the normal recursive version, but note that it is not recursive as it never calls itself. It only calls a function *f*, which we can provide with *fix* to make the function recursive.

To make clear why this works, the evaluation process of *fact 2* is detailed below.

```
fact 2
(fix factorial) 2  -- expand 'fact'
(let x = factorial x in x) 2  -- expand and apply 'fix'
let x = factorial x in (x 2)  -- apply 'x'
let x = factorial x in (factorial x 2)  -- evaluate let-statement
let x = factorial x in (if 2 <= 1 then 1 else 2 * (x (2 - 1)))  -- apply 'factorial'
let x = factorial x in (2 * (x 1))  -- execute if-statement
let x = factorial x in (2 * (factorial x 1))  -- evaluate let-statement
let x = factorial x in (2 * (if 1 <= 1 then 1 else 1 * (x (1 - 1))))  -- apply 'factorial'
let x = factorial x in (2 * 1)  -- execute if-statement
let x = factorial x in 2  -- multiply
2  -- evaluate let-statement
```

Here, it becomes clear how Haskell's laziness helps - when evaluating the let-statement, *factorial x* is substituted for x only once, which allows for the application of *factorial* only as many times as is necessary.

2.3 Value Recursion

As mentioned, we will not just explore normal recursion, but also value recursion with algebraic effects and handlers in this paper. Moggi *et al.* [4] explain that value recursion differs from normal recursion only in the presence of effects. They describe how, in normal recursion, the repeated application of the function is unfolded in such a way that all side-effects are duplicated, and may thus be executed multiple times. In value recursion, however, side-effects are evaluated only once (the first time the function is executed), such that the recursion happens only over the values of the function. It is not obvious at all how this behavior can be implemented, and while the problem has thus been solved for effects encoded with monads, there exists no clear analogue for a value recursion operator when working with algebraic effects and handlers. This problem and possible solutions will be discussed in more detail in section 3.

3 Implementation

In this section, the solutions found to working with (value) recursion using algebraic effects and handlers will be presented.

3.1 Normal Fixed-Point Recursion

Analyzing the way algebraic effects interact with the normal fixed-point combinator to perform regular recursion was, initially, the main purpose of this paper. Whether this should work is not at all immediately obvious, because the function would have to be fixed before applying any handlers (as once the handlers have been applied, what remains is just a pure value). But, the behavior of the function may differ significantly depending on the handler that will be applied later. Consider, for example, a choice effect that determines branching, executed with either a handler that always picks *True* or a handler that picks either *True* or *False* randomly.

Several different function signatures and their relevant fixed-point operators were considered, and it became clear that all seem to work, aided by Haskell's laziness. Indeed, it seems algebraic effects behave normally under fixed-point recursion. In this subsection, we will discuss the different function signatures and combinators. In section 4, we will walk through a sample evaluation process showing how and why they work..

3.1.1 The *Either* Signature

The *Either* signature is the signature initially recommended as the focus for this paper. It makes explicit whether or not the computation should recurse again by use of the *Either* data type.

```
data Either a b = Left a | Right b
```

Following the convention of the *Right* case of the *Either* data type usually holding a result value, we chose to represent a value that should be used in a recursive step again (continuing the computation) with the *Left* case, whereas a value that represents the computation should finish is represented with the *Right* case.

With *Either*, we can write fixable functions with algebraic effects with the following signature:

```
fun :: a -> Free f (Either a b)
```

So, one constructs a syntax tree using the free monad which may include any effects from *f*, and results in a final value of type *Either a b*. In the case of *Left a*, the combinator should ensure that all the steps of the syntax tree are repeated again on this new value, and in the case of *Right b*, it should ensure that this is the final returned value. The following is a possible implementation of such a combinator:

```
efixEither :: (a -> Free f (Either a b)) -> a -> Free f b
efixEither f x = do
```

```
    y <- f x
    case y of
        Left  l -> efixEither  f  l
        Right  r -> Pure r
```

An interesting note here is that functions constructed using this signature must necessarily be tail-recursive, as they can only return a *Right x* or a *Left x*, which will be used directly as the argument to a possible recursive call.

3.1.2 The Regular Fix Signature

The second signature is very similar to the one used for regular fixed-point operations without the free monad. Fixable functions are implemented with the following signature:

```
fun :: (a -> Free f a) -> a -> Free f a
```

In fact, functions with this signature are even accepted by the regular *fix* combinator. Indeed, *fix* takes a function with type *a -> a* - substituting *a -> Free f a* for *a* gives exactly the signature of *fun*.

So, it seems that when constructed in the right way, it is possible to use normal fixed-point combinators to make functions with algebraic effects recursive. This is further evidence to support the idea that effectful functions behave normally under (fixed-point) recursion. As mentioned, this will be supported and further formalized in the following section.

3.2 Value Recursion

With the findings of the investigation into the interaction between algebraic effects and normal recursion in mind, it was decided that it would also be worthwhile to explore value recursion with algebraic effects. It can be concluded from Erkök [3] and Moggi *et al.* [4] that value recursion is a much harder problem. In the context of effects encoded with monads, Erkök [3] concludes that no implementation can be given that can work for all monads, and much time is spent on a formal analysis of value recursion. Moggi *et al.* [4] use a different semantics that also invalidates one of the axioms found by Erkök [3], but under which a generic implementation *can* be given.

While a fair amount has been written about value recursion with monads, there seems to be no research into value recursion with algebraic effects and the free monad at all. The analysis performed in this paper is, unfortunately, inconclusive. However, we hope to lay the groundwork for future research into value recursion with algebraic effects with it. Two possible solutions that have been explored will be presented in this subsection, but the question of whether any of them can provide a working implementation of value recursion remains unanswered. In fact, it is not clear whether such an implementation can exist at all.

3.2.1 The Problem

To see why an implementation of a value recursion operator for the free monad is not trivial, we will look at its intended behavior with the help of an example. We consider a function that reads in a character from the standard input and appends this character to a list of characters received as an argument to the function (with *ChIn* the effect to read a character). This function is adapted from Erkök [3].

```
chars :: [Char] -> Free (ChIn + End) [Char]
chars cs = do
                c <- chin
                return (c:cs)
```

When fixed with a normal recursion operator, we would expect the function to continuously ask for a new input character, and then construct a list of all the provided input characters. Under value recursion, however, the effect to read in a character is only performed once, and an infinite list of only this character is constructed. The desugared version of the function above looks like this[1]:

```
chars = \cs -> Op (ChIn (\c -> Pure (c:cs)))
```

From this, it becomes clear that if the entire function is fixed, there is seemingly no way to get around executing the *ChIn* effect multiple times. We also cannot only fix the continuation of the effect, as the return value is a list of characters, but it only takes a single character as input - so there is no way to construct the infinite list that we want. Ideally, then, the entire function should be fixed, but after the first execution all effects should be ignored.

We should also note again that we discuss the intended behavior of a value recursion operator more formally in section 5, where we also discuss the possibility of deriving an operator from these laws. In particular, the left shrinking law may allow for only the continuation to be fixed in this example, but there are some further challenges to it that we discuss in more detail in section 5.

First, we will now look at two other possible solutions that were considered for this problem, but again, it is not clear for any of them whether they can work at all.

3.2.2 Dummy Handlers

Dummy handlers are handlers that will be used instead of the normal handlers after the first evaluation of the function. They attempt to simply execute the continuation without executing the effect again. Then, something similar to the following (pseudo-)code could be used to implement value recursion.

```
(handleAll normalHandlers f x) >>= (\res -> handleAll dummyHandlers (fix f) res)
```

This method initially seemed promising, and it does actually work for some effects. For example, for the *StrOut* effect, the continuation k is freely available:

```
data StrOut k = Out String k
    deriving Functor
```

This means a dummy handler for *StrOut* can easily be provided:

```
hStrOutDummy :: Functor f' => Handler StrOut a f' a
hStrOutDummy = Handler {
    ret = pure,
```

[1]We ignore the L and R that normally surround the effect to select it from the effect row, as they are not relevant here

```
    hdlr = \x -> case x of Out _ k -> k
  }
```

However, this approach unfortunately cannot work for effects that don't have a freely accessible continuation. Consider for example the *State* effect introduced in section 2, which has a case *Get (s -> k)*. This means an *s* has to somehow be provided in order to access the continuation. However, using this approach, there is no way to store and provide such an *s*, or any other inputs necessary for other effects in general.

A second problem is that this approach seems to be semantically incorrect, as the effects are, in fact, executed multiple times, just with a different handler. In real value recursion, we would expect the effects to really only be executed once.

3.2.3 *fixIO*-inspired Stateful Handling

The *fixIO* function is the value recursion operator for the *IO* monad discussed by Erkök [3]. An implementation and an explanation of how it works are given by Fancher [5]. In essence, the argument passed to the function to be fixed is wrapped in a mutable variable, and the result returned by the function is stored in this variable again. This interaction is able to create value recursion for the IO monad, and we may expect there to exist an analogue of this process for working with the free monad. Unfortunately, this has not been explored further due to time constraints. While it does seem promising, it should be noted that mutable variables are not necessary to implement value recursion for most monads, so whether a value recursion operator for the free monad using this concept will be general enough is not clear.

4 Applications and Analysis of Effectful Normal Recursion

In this section, we will focus on an analysis of normal recursion with algebraic effects and handlers. We discuss a motivating example, explaining the need for effectful, recursive functions. We will also detail the evaluation process for the combinators for normal recursion given in section 3 with the help of another example program. This detailed working out already makes explicit how these combinators work, and can easily be generalized to other effects if necessary. Combined with the fact that no specific laws were found to be relevant for normal recursion, it was decided a formal analysis of the normal recursion operators would not have any additional value.

4.1 Motivating Example: A Circuit Simulator

Erkök [3] discusses how the modelling of electronic circuits can benefit from the use of effects, and how these models frequently makes use of (value) recursion. In this subsection, we will discuss an example with algebraic effects and handlers largely based on the one presented by Erkök [3].

Indeed, as Erkök [3] mentions, if we are able to provide an abstract description of a circuit, describing only its components and wires, this would allow us to modularly choose exactly what we want to do with our circuit. For example, one might want to simply simulate it, or render it graphically, or perhaps translate it to a different hardware design language, etc. Effects allow us to do exactly this.

We first define a signal, representing the values (high/low) at certain points in the circuit at different discrete time steps, encoded as a list of boolean values. We could then define the components of a circuit as an effect called *CircuitComponent* like so:

```
type Signal = [Bool]

data CircuitComponent k =
    CircAnd Signal Signal (Signal -> k)
    | CircXor Signal Signal (Signal -> k)
    | CircInv Signal (Signal -> k)
    | CircDelay Bool Signal (Signal -> k)
```

CircAnd, for example, represents a logical-and gate. It takes two input signals, and then provides an output signal to the continuation (the result of the and operation). Exactly what this result is depends on the handler we choose to apply later, of course.

This is a possible implementation of a handler that just simulates the circuit:

```
hCircuitSimulate :: Functor f' => Handler CircuitComponent a f' a
hCircuitSimulate = Handler
    {
        ret = pure,
        hdlr = \x -> case x of
                        CircAnd x y k -> k (zipWith (&&) x y)
                        CircXor x y k -> k (zipWith (/=) x y)
                        CircInv x k -> k (map not x)
                        CircDelay v x k -> k (v : x)
    }
```

Then, this is how one could describe a simple half-adder with this system:

```
halfAdder :: Signal -> Signal -> Free (CircuitComponent + End) (Signal, Signal)
halfAdder x y = do
    res <- circ_xor x y
    carry <- circ_and x y
    return (res, carry)

halfAdderExample :: (Signal, Signal)
halfAdderExample = un (handle hCircuitSimulate (halfAdder [True, True] [False, True]))
-- ([True, False], [False, True])
```

To see how recursion is used in circuit simulation, we will look at a toggle gate, the output of which is defined as the inverse of its input, which clearly results in a gate that will toggle between the high and low signals forever. To implement it, we will need the invert gate, and we will also need to make use of the delay component, which takes a single boolean value as well as a signal. It is defined as being this single boolean value for one time step, and becomes the input signal after that.

If the function is only executed once, we will not get the infinite list we want. To achieve this, we will also have to fix the function. For example, using the *Either* signature. This results in the following definition for the toggle gate:

```
toggleFixable :: (Signal, Signal) -> Free (CircuitComponent + End) (Either (Signal, Signal) a)
toggleFixable (inp, out) = do
    inp <- circ_inv out
    out <- circ_delay False inp
    return (Left (inp, out))

toggle :: Signal
toggle = un ((handle hCircuitSimulate ( efixEither  toggleFixable  ([],  []))) >>=
                 (\(inp, out) -> return out)
             )
-- [False, True, False, True, ...
```

It should be noted that, while normal recursion does work in this case for simulating the circuit, things get more complicated if we wanted to, for example, create a graphical representation of the circuit using this same description. Of course, our intention is not to draw an infinite amount of invert gates. So, indeed, we would then only want the effects of this function to be executed once, meaning the function would have to be fixed with a value recursion operator.

We have now implemented most of the behavior also implemented by Erkök [3] with algebraic effects and handlers. The approaches very similar, but we have all the benefits of working with algebraic effects: easily switchable modular handlers, explicit effects, and the ability to easily incorporate multiple different effects into the function, if desired.

4.2 The Evaluation Process

To explore why the combinators and signatures for normal recursion found in section 3 work, and how they are able to work around the problem described in the beginning of section 3 (the fact that functions need to be fixed before the application of any handlers), we will look at an example program. This program is implemented using the *Either* signature and uses a random choice effect to branch. The evaluation process of the regular fix signature is very similar, and will not be discussed separately.

The following is our example program. This program uses a choice effect to generate a random boolean. If it is *True*, the computation terminates with the input value. If it is *False*, the computation continues with the input value plus one.

```
data Choice k = Choose (Bool -> k)
    deriving Functor

choices :: Int -> Free (Choice + End) (Either Int Int)
choices k = do
                           b <- choice
                           if b then do
                               return (Right k)
                           else do
                               return (Left (k + 1))
```

We will now detail the evaluation process of *efixEither choices 0* and see how recursion can be achieved without knowledge of the implementation of the handler for the *Choice* effect. For this, we will use implementations of ≫=, *fold*, and *handle* provided by Bach Poulsen [2]. These are also given in the full code (see Appendix B).

```
un (handle ... ( efixEither  choices  0))

-- The expression 'efixEither choices 0' will be evaluated first , so:

un (handle ... (do  y <- choices 0;
                    case y of {Left  l  -> efixEither  choices  l ;  Right  r  -> Pure r}
    ))  -- apply 'efixEither'
un (choices  0 >>=
        (\y -> case y of {Left l -> efixEither  choices  l ;  Right  r  -> Pure r})
    )   -- desugar 'do'

-- We substitute 'lby' for '(\y -> case y of {Left l -> efixEither choices l; Right r -> Pure r})'

un (handle ... (choices  0 >>= lby))
un (handle ... (fold  lby  Op (choices  0)))    -- apply '>>='
un (handle ... (fold  lby  Op
                 (Op (L (Choose (\b -> if b then do return (Right 0) else do return (Left (0 + 1))))))
                 )
    ) -- evaluate 'choices 0'

-- We substitute lbb0 for '(\b -> if b then do return (Right 0) else do return (Left (0 + 1)))'

un (handle ... (fold  lby  Op (Op (L (Choose lbb0)))))
un (handle ... (Op (fmap (fold  lby  Op)
                        (L (Choose lbb0))
                    )
                 )
    ) -- apply 'fold'
un (handle ... (Op (L
                    (fmap (fold  lby  Op)
                        (Choose lbb0)
                    )
                 ))
    ) -- apply 'fmap'
un (handle ... (Op (L (Choose
                    (\z -> (fold lby  Op) (lbb0 z))
                 )))
    ) -- apply 'fmap'
```

At this point, the expression cannot be evaluated any further without applying handlers. Note, however, how the *lby* from the *efixEither* function has been brought inside the *Choose* operation by function composition with the original function (*lbb0*) that was inside the *Choose* operation. When the *Choice* effect is handled now, the boolean value will be provided to this composed function. We get either a *Pure (Left n)* or a *Pure (Right n)* from evaluating *lbb0* with this value. Then, *fold lby Op* is applied to this new value *Pure v*.

We know from the definition of *fold* that, when applied to a *Pure* value, it will simply evaluate to (in this case) *lby v*. And indeed, from the definition of *lby*, we see that depending on whether *v* was a *Right* or a *Left* case, the computation ends here or we end up with a recursive step - applying all the same steps again. To see exactly how this recursive step is computed, we will apply a handler for the *Choice* effect that supplies a random boolean value. Exactly how this random value is generated is not relevant here, but can be found in the full code (see Appendix B).

```
hChoice :: Functor f' => Handler Choice a f' a
hChoice = Handler
    {
        ret  = pure,
        hdlr = \x -> case x of Choose k -> k bool
        -- with 'bool' some random boolean value
    }
```

We now continue the evaluation with this handler.

```
un (handle hChoice (Op (L (Choose
                        (\z -> (fold lby Op) (lbb0 z))
                    )))
    )
un (fold (ret hChoice) (\x -> case x of
                            L y -> hdlr h y
                            R y -> Op y
                        )
                        (Op (L (Choose
                                (\z -> (fold lby Op) (lbb0 z))
                        )))
    ) -- apply 'handle'

-- We substitute lbx for '(\x -> case x of {L y -> hdlr h y; R y -> Op y})'
-- And substitute lbz0 for '(\z -> (fold lby Op) (lbb0 z))'

un (fold (ret hChoice) lbx (Op (L (Choose lbz0))))
un (lbx (fmap (fold (ret hChoice) lbx) (L (Choose lbz0)))) -- apply 'fold'
un (lbx (L (fmap (fold (ret hChoice) lbx) (Choose lbz0)))) -- apply 'fmap'
un (hdlr hChoice (fmap (fold (ret hChoice) lbx) (Choose lbz0))) -- apply 'lbx'
un (hdlr hChoice    (Choose
                        (\m -> (fold (ret hChoice) lbx) (lbz0 m))
                    )
    ) -- apply 'fmap'
```

Note how at this point we end up with another function composition inside the *Choose* operation. Continuing the evaluation by executing the *hChoice* handler gives:

```
un ((fold (ret hChoice) lbx) (lbz0 b1))
-- with 'b1' some random boolean value
```

Then, as discussed previously, we know that evaluating *lbz0* will give either the value *Pure 0*, or a recursive call *efixEither choices 1* depending on whether *b1* was *True* or *False*, respectively. We will work out both cases below.

```
-- In case 'b1' was True
un ((fold (ret hChoice) lbx) (Pure 0))
un (ret hChoice 0) -- apply 'fold'
un (Pure 0) -- apply 'ret'
0 -- apply 'un'


-- In case 'b1' was False
un ((fold (ret hChoice) lbx) ( efixEither choices 1))
```

```
-- Similarly to the above steps to evaluate ‘efixEither choices 0’
-- And similarly with ‘lbz1’ equal to ‘(\z -> (fold lby Op) (lbb1 z))’
-- In which ‘lbb1’ is equal to ‘(\b -> if b then do return (Right 1) else do return (Left (1 + 1)))’
un ((fold (ret hChoice) lbx) (Op (L (Choose lbz1))))

-- Then, similar to the above steps to evaluate the handlers, we get
un (hdlr hChoice    (Choose
                        (\m -> (fold (ret hChoice) lbx) (lbz1 m))
                    )
   )
un ((fold (ret hChoice) lbx) (lbz1 b2)) -- evaluate ‘hdlr’
-- with ‘b2’ some random boolean value
```

Here, similarly, we will end up with either a value *Pure 1* or a recursive call *efixEither choices 2*, depending on the value of *b2*, which completes all the recursive evaluation steps.

As mentioned in the introduction to this section, this evaluation process can easily be generalized for other effects. The function *lbb0* can be any continuation function in general, and the same steps of composing functions lazily apply to create a recursive computation.

5 Formal Analysis of Effectful Value Recursion

In this section, we will focus on an analysis of value recursion with algebraic effects and handlers. We discuss the laws that will have to hold for a value recursion operator for algebraic effects and handlers. These laws are given for monads by Erkök [3], and have been translated to be relevant for algebraic effects and handlers.

Erkök [3] gives multiple laws, but explains that strictness, purity, and left shrinking are enough to capture all the necessary requirements for a value recursion operator. It *may* thus be possible to derive a value recursion operator for algebraic effects from these laws. Especially the left shrinking law seems to be very interesting.

5.1 Strictness

The strictness law captures the idea that fixing a strict function should result in ⊥ - that is, a non-terminating expression. Essentially, because fixing a function is repeated self-application of some function *f*, if *f* is strict in its argument this will result in the forced evaluation of an infinite chain of self-applications, which can of course never terminate. Our law is very similar to the one given by Erkök [3]. We assume that *vfix* is some value recursion operator for algebraic effects.

$$f :: \alpha \rightarrow Free\ \phi\ \beta$$

$$f \bot \equiv \bot \ \Rightarrow \ vfix\ f \equiv \bot$$

5.2 Purity

The purity law states that a value recursion operator should behave exactly like the *fix* operator for normal recursion for pure functions (that is, functions that do not use any effects). We encode the law as follows, where *vfix* is some value recursion operator for algebraic effects, and *efix* is a normal recursion operator for algebraic effects.

$$f :: \alpha \rightarrow Free\ End\ \beta$$
$$efix\ h \equiv vfix\ h$$

5.3 Left shrinking

The left shrinking law states that if a computation is fixed, parts that do not use any recursive bindings may be lifted out of the fix, starting from the left. This happens because value recursion only executes effects once, so lifting an effect out whenever possible should not change the behavior of the fixed function. We consider a function f defined as follows - with E some effect[2]:

```
h :: a -> b -> Free h' a

f :: a -> Free (E + h' + End) a
f x =  do
          y <- e
          h x y

-- Desugaring do-notation gives
f = \x -> Op (E (\y -> h x y))
```

Then, we consider $g = vfix\ f$ with *vfix* some value recursion operator. If E does not use any recursive bindings (that is, x or anything defined in the continuation), we should be able to lift it out of the fix - which gives the following:

```
g :: Free (E + h' + End) a
g = do
       y <- e
       vfix (\x -> h x y)

-- Desugaring do-notation gives
g = Op (E (\y -> vfix (\x -> h x y)))
```

Formally, we get the following law:

$$h :: \alpha \rightarrow \beta \rightarrow Free\ \phi\ \alpha$$
$$vfix\ (\lambda x.\ Op\ (E\ (\lambda y.\ h\ x\ y))) \equiv Op\ (E\ (\lambda y.\ (vfix\ (\lambda x.\ h\ x\ y))))$$

5.4 Deriving an Operator from the Laws

In section 3 and in the introduction to this section, we briefly mentioned that it may be possible to derive an operator for value recursion from these laws. Especially the left shrinking law seems to capture a lot of the desired behavior of such an operator, as lifting out an effect ensures that it is only executed once. Still, it is not obvious how to derive the operator. Even if there is a way to perform the transformation described by the left shrinking law in general, the law does not consider effects that *do* use recursive bindings - as they cannot

[2]We should make use of higher-order effects to properly write down the type of f, but those are not relevant here - see Bach Poulsen [2] for more details

be lifted out of the fix. If effects cannot be lifted out, we end up with the same problem described in section 3: it is simply not clear how to force effects to only be executed once, if this can even be done at all.

6 Responsible Research

In this section, we discuss responsible research considerations taken into account in the writing of this paper.

6.1 Reproducibility

Care has been taken to ensure that it is possible to reproduce all results found in this paper. To this end, an extensive background section was included, and full code listings are available (see Appendix B). We clearly mention all details skipped for brevity in the main text, and always refer to where a more detailed explanation can be found.

6.2 Ethics

No specific ethical issues are relevant to this paper, as it focusses on the discussion of abstract concepts such as (value) recursion and algebraic effects and handlers in general.

7 Discussion, Conclusion, and Future Work

In this paper, we have explored fixed-point (value) recursion with algebraic effects and handlers in Haskell.

Our first subquestion was: "What motivates the need for effectful fixed-point (value) recursion with algebraic effects and handlers in Haskell?". We discussed the benefits of working with algebraic effects and handlers, and a motivating example of a circuit simulator was discussed in section 4 that makes clear why recursive, effectful functions might be used.

Then, we asked: "What is the intended behavior of effectful fixed-point (value) recursion in Haskell and how can this behavior be implemented?". This was explored in section 2 and section 3. We found that effectful functions behave very predictably under normal recursion, and we have seen that the provided fixed-point combinators are able to implement normal recursion in the presence of effects. Value recursion, on the other hand, has been shown to be a much harder problem. It is not obvious how to implement the idea of executing effects only once and recursing only over the values of a function for any effect in general. Several possibilities were explored, but none were able to provide a conclusive answer as to whether value recursion with the free monad is even possible at all.

Our final subquestion was: "What are the mathematical laws pertaining to effectful fixed-point (value) recursion, and can the provided implementation in Haskell be proven to respect these laws?". For normal recursion, we found that the functions behave very predictably, and the how and why behind the operation of the provided combinators was already clearly explained with the help of a detailed working out of an example program in section 4. Furthermore, no specific laws were found to be relevant for normal recursion in this context. For value recursion, we discussed the laws that should hold for a possible value recursion operator in section 5. We also discussed that it may be possible to derive a value recursion operator from these laws, but that there are still some challenges to doing so.

In this process, we have (partially) answered the main research question: "How can effectful fixed-point (value) recursion be used in combination with algebraic effects and handlers in Haskell?". Partially, because while it is now clear how effectful functions behave under and can be used with normal recursion, it is unfortunately not clear how to use value recursion with the free monad.

Value recursion with the free monad thus presents an interesting open problem for future research. If it can be solved, it would not only be useful, but also provide valuable further insight into the relation between effects encoded with monads and effects encoded with algebraic effects. We thus hope that this paper can serve as the foundation for future research into this topic.

A Appendix: Algebraic Effects and Handlers in Haskell

Unless otherwise specified, code examples and explanations in this appendix may be assumed to be adapted from Bach Poulsen [2].

In this appendix, we will provide an introduction to working with algebraic effects and handlers in Haskell using the approach described by Bach Poulsen [2]. It should be noted that the introduction given to algebraic effects and handlers here is relatively short, providing the information crucial to understanding the paper, but in some places omitting implementation details. We will refer to Bach Poulsen [2] for an explanation of these details.

A.1 Algebraic Effects

Programs using algebraic effects are written using the free monad. This monad allows for the construction of syntax trees with computations and values.

```
data Free f a = Pure a | Op (f (Free f a))
```

In this free monad, f represents a list of possible effects that can be used in the syntax tree, while a is the type of the final result of the program. The *Pure* case represents a value of type a, while the *Op* case represents a computation, applying some effect from f, and then continuing execution with its continuation, itself another syntax tree of type *Free f a*.

As an example, we consider the *State* effect. It has *Put* and *Get* operations, allowing for the storing and retrieval of some value, respectively. It looks like this:

```
data State s k = Put s k | Get (s -> k)
    deriving Functor
```

Note how the effect is parameterized over s, the type of the value being stored in the state. It also has a parameter k, which allows us to encode a continuation. The *Put* operation, in this case, allows us to take in some value of type s, and a continuation to execute after that. The *Get* operation contains a function that requires a value of type s to be provided in order to be able to execute the continuation. The concrete implementation of these effects will be provided later by handlers - for now, it only matters that the interface is correct.

A.2 Writing Programs with Algebraic Effects

An example of a program that could be written using the free monad and the *State* effect is given below. It first retrieves an integer value from the current state, increments it by one, and subsequently stores this new value in the state. It also returns this incremented value at the end of the computation. Note the signature of this function - it is a free monad (syntax tree representing a program), with as a possible effect only *State Int*, and a final return value of type *Int*.

```
increment :: Free (State Int) Int
increment = Op (Get (\s ->
                  Op (Put (s + 1) (Pure (s + 1)))
            ))
```

From this, it should hopefully be clear how programs using effects are constructed. One might note, however, that it could quickly become quite cumbersome to write programs in this manner, with nested *Op*'s. To overcome this, infrastructure is borrowed from Swierstra [1], allowing us to write the above program much more elegantly in combination with Haskell's do-notation:

```
increment' :: Free (State Int + End) Int
increment' = do
                (s :: Int) <- get
                put (s + 1)
                return (s + 1)
```

The *End* effect seen here is an implementation detail - essentially, it encodes the idea of "no effect".

A.3 Handlers

Finally, we briefly discuss effect handlers. Handlers allow us to provide concrete implementations for the effects used in programs constructed using the free monad. When applied to a free monad, they handle and remove effects, until eventually only a value remains.

Handlers are defined using the following signature:

```
data Handler f a f' b =
    Handler {     ret  :: a -> Free f' b
                , hdlr :: f (Free f' b) -> Free f' b
            }
```

Here, *f* is the effect this handler handles, *a* the return type of the program before applying the handler, *f'* the effects used in the program being handled, and *b* the return type of the program after applying the handler. Handlers must then provide implementations for the two functions *ret* (to take raw values of type *a* to type *b* while wrapping them in the free monad) and *hdlr* (actually handling the *f* effect).

We define a function to apply handlers like so (omitting its implementation for brevity):

```
handle :: (Functor f, Functor f') => Handler f a f' b -> Free (f + f') a -> Free f' b
```

We also need a function to take a free monad for which all effects have been handled to a normal value (unwrapping it from the free monad):

```
un :: Free End a -> a
```

With this infrastructure, we can finally execute programs we write using the free monad. For example, to execute the increment program defined earlier, we could do this:

```
un (
    (handle handlerState
        increment'
    )
)
```

A reader familiar with effects and handlers may note that this definition makes it quite difficult to properly handle the state effect - for example, no initial state can be provided. A version of *handle* that takes an initial state as input and allows for the passing along of states during the handling does also exist, and this version should actually be used for the *State* effect. However, we will not see any more effects like *State* in this paper, so it was decided to omit the explanation of the alternative *handle*. As mentioned, we refer to Bach Poulsen [2] for more details.

B Appendix: Code

The full code used in the process of writing this paper can be found in the following GitHub repository: `https://github.com/gijsh21/RP-PWEA-FixedPointRecursion`. For convenience, we will also list the implementations of some important functions referenced in the text here.

Handle

This is the implementation of the *handle* function, as given by Bach Poulsen [2]:

```haskell
handle :: (Functor f, Functor f')
    => Handler f a f' b -> Free (f + f') a -> Free f' b
handle h = fold
    (ret h)
    (\x -> case x of
        L y -> hdlr h y
        R y -> Op y
    )
```

Fold

This is the implementation of the *fold* function, as given by Bach Poulsen [2]:

```haskell
fold :: Functor f => (a -> b) -> (f b -> b) -> Free f a -> b
fold gen _ (Pure x) = gen x
fold gen alg (Op f) = alg (fmap (fold gen alg) f)
```

»=

This is the implementation of the *»=* function for the monad instance of *Free*, as given by Bach Poulsen [2]:

```haskell
instance Functor f => Monad (Free f) where
    m >>= k = fold k Op m
```

References

[1] W. Swierstra, "Data types à la carte," *Journal of Functional Programming*, vol. 18, no. 4, pp. 423–436, 2008. DOI: `10.1017/S0956796808006758`.

[2] C. Bach Poulsen. "Algebraic Effects and Handlers in Haskell." (2023), [Online]. Available: `http://casperbp.net/posts/2023-07-algebraic-effects/`.

[3] L. Erkök, "Value recursion in monadic computations," AAI3063791, Ph.D. dissertation, 2002, ISBN: 0493822941.

[4] E. Moggi and A. Sabry, "An abstract monadic semantics for value recursion," *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications*, vol. 38, no. 4, pp. 375–400, 2004. DOI: `10.1051/ita:2004018`.

[5] W. Fancher. "MonadFix is Time Travel." (2017), [Online]. Available: `https://elvishjerricco.github.io/2017/08/22/monadfix-is-time-travel.html/`.