TU Delft
Delft University of Technology

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

Gijs van der Heide | G.vanderHeide-1@student.tudelft.nl

Introduction

Algebraic effects and handlers are a new programming technique

- **Algebraic effects**: abstractions as interfaces (stateful computation, I/O operations, exceptions, many more…)
- **Handlers**: modular implementations of these interfaces

May make it easier to write and reason about complex effectful programs

The Problem

How do algebraic effects and handlers interact with **recursive computations**?

- Little research into the interaction between effects and recursion – and specifically **value recursion** [1]
- No research done when defined using algebraic effects and handlers

Like Erkök [1], focus on **fixed-point recursion**

"How can effectful fixed-point (value) recursion be used in combination with algebraic effects and handlers in Haskell?"

Methodology

Usually no out-of-the-box support for algebraic effects and handlers in mainstream programming languages (yet)

This project: work in **Haskell** with infrastructure from Swierstra [2] and Bach Poulsen [3]

(1) Motivate the need for effectful, recursive functions
(2) Explore and explain algebraic effects and handlers under fixed-point (value) recursion with example programs
(3) Discuss and prove laws pertaining to value recursion

Background

Fixed-points

Used a lot in research – essentially just repeated application of a function *f*

$$f(f(f(f(..))))$$

So defined as:

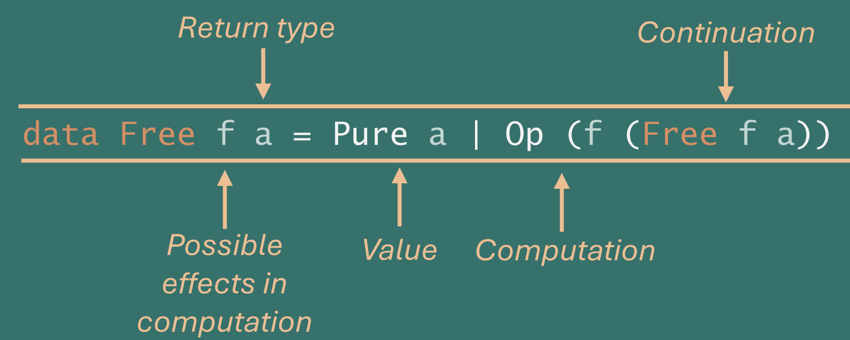
$$fix\ f = f\ (fix\ f)$$

A simple Haskell fixed-point combinator:

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

Algebraic Effects and Handlers in Haskell

Create syntax trees of computations using the **free monad**.



Ex.

```
getName :: Free (StrIn + StrOut + End) String
getName = do
            (name :: String) <- strIn
            strOut ("Hello, " ++ name)
            return name
```

Handlers handle the effects, turning a free monad into an actual value

```
un (handle handlerStrIn
     (handle handlerStrOut
          getName
     ))
```

Value Recursion

Recursion that only evaluates side-effects once!

Example (adapted from Erkök [1]):

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

Fixing with normal recursion semantics executes character reading effect multiple times

Fixing with value recursion semantics executes character reading effect only once – creates an infinite list of the input character

Implementation & Analysis

Normal Recursion

Function has to be fixed before applying handlers. But: behavior may differ depending on the handler applied later!

Functions do behave normally under normal fixed-point recursion! Two possible signatures were considered:

The *Either* signature

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

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

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

Left continues, *Right* terminates

The Regular Fix Signature

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

Can be fixed with regular *fix*!

Detailed the evaluation process to see how and to formalize why these signatures work for normal recursion

```
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))
```

Essentially, the fix function is embedded into the effect continuation by function composition

```
Op (Choose k) ---->
Op (Choose (\z -> (fold _fix Op) (k z)))
```

As soon as a handler is applied, a result is obtained which is immediately passed to the fix function again, creating recursion

Value Recursion

Much harder problem than normal recursion. How can effects be forced to only be executed once?

For example, in the *chars* function discussed earlier (desugared):

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

How to get around executing *ChIn* multiple times? The continuation on its own cannot be fixed!

No solution found! We explored some possibilities:

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

Problem: requires the continuation to be freely available, cannot work for all effects

***fixIO*-inspired Stateful Handling**
Not explored in detail. Storing the result of the function in a mutable variable may generate recursion. Unclear whether this can be adapted for the free monad, and whether it is general enough.

Formal Analysis of Value Recursion

May be possible to derive a value recursion operator from laws. Laws adapted from Erkök [1]

Especially **left shrinking** seems promising – general description of how to extract effects out of a fix:

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

Lifting out an effect ensures that it is only executed once!

Even if this can be done in general, what about effects that cannot be lifted out? For example, due to the use of recursive bindings? End up with the same problem!

Conclusion

Recursion is very important and used often in functional programming – we motivated the need for more research into its interaction with algebraic effects and handlers.

Algebraic effects behave predictably under normal fixed-point recursion. Several ways of doing this were provided and analyzed to see how and why they work.

Value recursion has been shown to be a much harder problem! Not obvious how to implement value recursion semantics, and this question unfortunately remains unanswered!

It may be possible to derive an operator with value recursion semantics from the provided laws, adapted from Erkök [1]

Value recursion with algebraic effects is an interesting open problem – we hope this lays the groundwork for future research into this topic!

References

[1] L. Erkök, "Value Recursion in Monadic Computations," AAI3063791, Ph.D. dissertation, 2002, ISBN: 0493822941.

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

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

This is a poster for the research paper "Fixed-Point (Value) Recursion with Algebraic Effects and Handlers in Haskell" (2024) by Gijs van der Heide, submitted as a BSc thesis in Computer Science & Engineering at Delft University of Technology
A digital version of this thesis is available at https://repository.tudelft.nl