Implementing UNIX with Effects Handlers

Ramsay Carslaw

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2023

Abstract

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Ramsay Carslaw*)

Acknowledgements

Any acknowledgements go here.

Table of Contents

Chapter 1

Introduction

Chapter 2

Background

2.1 Algebraic Effects and Effect Handlers

Algebraic effects [1] and their corresponding handlers [2] [3] are a programming paradigm that when paired together offers a novel way to compose programs. It starts with the definition of the effect or the *effect signature* that gives the effect a name in scope and specifies any input and the return type otherwise known as the *effect operation*. At this stage the effect operation has no implementation and is more an acknowledgement to the compiler that it should expect an implementation. For this reason any function that references these effect operations is known as an *effectful function* or a function whose definition is not complete without an effect handler. The *effect handler* provides one implementation of the given effect operation. In this way we can change the semantics of an effectful function by handling it with a different handler that provides an alternative implementation to the effect.

When programs are written that are 'black boxes', that is to say their outputs are defined entirely by their inputs and all functions are pure computation [4], it is safe to make assumptions about the inputs. Assumptions like an age will always be given as an integer or all strings will not exceed the length allocated for them. When programs interact with the real world it is no longer safe to make these assumptions. Effects allow the programmer to encapsulate these side effects and 'handle' them with control flow.

2.1.1 Example: Heads and Tails

Consider the following example of heads or tails written conventionally.

```
function choose() -> bool {
  if (rand() > 0.5) {
    return true
  }
  return false
}

function flip() -> string {
  if (choose() == true) {
    return 'Heads'
```

```
  } else {
    return 'Tails'
  }
}
```

The function choose returns true or false randomly, with equal probability and flip uses the result of chose to print heads on true and tails on false. This works fine but if we wanted multiple definitions of choose where we alter the probability or add an option for failure we would have to rewrite each function for each implementation.

We can use effects handlers as described by Pretnar et al [3] to provide a generic implementation of choose that can be handled by flip. The following is an example of choose implemented with effects.

```
effect Choose {
  choose : () -> bool
}

function choose() -> bool {
  invoke choose()
}

function flip() -> string {
  if (choose() == true) {
    return 'Heads'
  } else {
    return 'Tails'
  }
}
```

The effect Choose is defined with a single operation choose that takes no arguments and returns a boolean. The function choose is defined as a wrapper that simply invokes the choose effect. The function flip is unchanged from the previous example. We can now define a *handler* that provides the implementation details for choose. To get the same result as before we could do the following.

```
function fairToss(value, resume) {
  if (rand() > 0.5) {
    resume(true)
  }
  resume(false)
}

handle flip() with fairToss
```

Here we split the effect into the value and the remaining computation, *resume*. Depending on the rand function we continue the computation with either true or false as the value for choose. The true power of effects comes when we define multiple handlers for one effect for example we can make the coin weighted:

```
function unfairToss(value, resume) {
  if (rand() > 0.75) {
    resume(true)
  }
```

```
  resume(false)
}

handle flip() with unfairToss
```

Or even get heads and tails at once by returning the result of both true and false in a list.

```
function both(value, resume) {
  resume(true) ++ resume(false)
}

handle flip() with both
```

We get all of these without changing either of our original definitions. This is the expressive power of algebraic effect handlers.

2.2 Shallow vs. Deep Effect Handlers

There are two types of effect handler implementation, *deep handlers*, as originally defined by Plotkin and Pretnar [2] and *shallow handlers*. From Hillerström and Lindley [5] "Deep handlers are defined by folds over computation trees, whereas shallow handlers are defined as case splits." In practice deep handlers pass themselves with the computation allowing for them to be invoked again. Certain methods such as introducing a Continuation Passing Style allow for shallow handlers to implement behaviour normally reserved for deep handlers.

2.3 Affine and 'Multi-Shot' Handlers

If a continuation can be invoked only once we call it an *affine* or *one-shot* handler. If it can be invoked multiple times then it is *multi-shot*.

2.4 The State of Effect-Oriented Programming

2.4.1 Library Based Effects

- libhandler [6] is a portable c99 library that implements algebraic effect handlers for C. It implements high performance multi-shot effects using standard C functions. It is limited by the assumptions it makes about the stack such as it being contiguous and not moving. In practice this could lead to memory leaks if it copies pointers.
- libmprompt [1] is a C/C++ library that adds effect handlers. It uses virtual memory to solve the problem mentioned with libhandler. By keeping the stack in a fixed location in virtual memory it restores safety. It also provides the higher level libmpeff interface. A downside is they recommend at least 2GiB of virtual memory to allows for 16000 stacks which may be challenging on some systems.

[1] https://github.com/koka-lang/libmprompt

- cpp-effects [7] is a C++ implementation of effect handlers. It uses C++ template classes and types to create modular effects and handlers. It's performance has been shown to be comparable to C++20 coroutines. It's limitations are it only supports one-shot resumptions.

2.4.2 First-Class Effects

- Unison [2] is a function language implemented in Haskell that offers built in support for effect handlers through it's abilities system.
- Koka [8] is a statically typed functional language with effect types and handlers. It can also compile straight to C code without needing a garbage collector. Koka is developed by a small team and as such is still missing much of it's standard library.
- Frank [9] is a strict functional language that is *effectful* in that it has first class support for bi-directional effects and effct handlers.
- Links [10] is a functional programming language designed for the web. Out of the box it does not support true algebraic effects, however through an extension [11] it gains first class support for continuations.

2.5 UNIX

UNIX [12] is an operating system designed and implemented by Dennis M. Ritchie and Ken Thompson at AT&T's Bell Labs in 1974. It provides a file system (directories, file protection etc.), a shell, processes (pipe, fork etc) and a userspace. Since it's first release it has been reimplemented for a variety of systems.

2.6 Effect Handlers and UNIX

In chapter 2 of his 2022 thesis, Daniel Hillerström [13] outlines a theoretical implementation of UNIX using the effects syntax outlined by Kammar et. al. [14]. In this he provides an implementation of the original UNIX paper [12] that includes a filesystem and timesharing. Hillerström makes several assumptions about the effect system that would need to be taken into account in order to implement this with a real language. The main assumption is multi-shot handlers. For example the implementation of `fork` uses multi-shot handlers to copy the full stack on both branches. There are also some partial implementations such as `sed` [3] from which he only implements string replacement.

[2] https://github.com/unisonweb/unison

[3] It is worth mentioning `sed` has $20,000+$ lines of code

Chapter 3

Methods

Chapter 4

Results

Chapter 5

Conclusion

Bibliography

[1] Gordon Plotkin and John Power. Computational effects and operations: An overview. 2002.

[2] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, pages 80–94. Springer, 2009.

[3] Matija Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic notes in theoretical computer science*, 319:19–35, 2015.

[4] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[5] Daniel Hillerström and Sam Lindley. Shallow effect handlers. In *Programming Languages and Systems: 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2–6, 2018, Proceedings 16*, pages 415–435. Springer, 2018.

[6] Daan Leijen. Implementing algebraic effects in c. Technical Report MSR-TR-2017-23, June 2017.

[7] Dan R. Ghica, Sam Lindley, Marcos Maroñas Bravo, and Maciej Piróg. High-level effect handlers in C++. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1639–1667, 2022.

[8] Daan Leijen. Koka: Programming with row polymorphic effect types. In Paul Blain Levy and Neel Krishnaswami, editors, *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014*, volume 153 of *EPTCS*, pages 100–126, 2014.

[9] Sam Lindley, Conor McBride, and Craig McLaughlin. Do be do be do. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 500–514. ACM, 2017.

[10] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2006.

[11] Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In James Chapman and Wouter Swierstra, editors, *Proceedings of the 1st International Workshop on Type-Driven Development, TyDe@ICFP 2016, Nara, Japan, September 18, 2016*, pages 15–27. ACM, 2016.

[12] Dennis M Ritchie and Ken Thompson. The unix time-sharing system. *Bell System Technical Journal*, 57(6):1905–1929, 1978.

[13] Daniel Hillerström. Foundations for programming and implementing effect handlers. 2022.

[14] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. *ACM SIGPLAN Notices*, 48(9):145–158, 2013.