Implementing UNIX with Effects Handlers

Ramsay Carslaw

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

2024

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

1 Introduction **1**

2 Background **2**
- 2.1 Algebraic Effects and Effect Handlers 2
  - 2.1.1 Example in Unison 3
- 2.2 Affine and 'Multi-Shot' Handlers 4
- 2.3 The State of Effect-Oriented Programming 4
  - 2.3.1 Library Based Effects 4
  - 2.3.2 First-Class Effects 4
- 2.4 Shallow vs. Deep Effect Handlers 5
- 2.5 UNIX 5
  - 2.5.1 The UNIX Philosophy 5
- 2.6 Effect Based File Systems 5
- 2.7 Effect Handlers and UNIX 5

3 Base Implementation **7**
- 3.1 Effect Oriented Programming in Unison 7
- 3.2 Program Status 7
  - 3.2.1 Unique vs. Structural Types 7
  - 3.2.2 The Handler 7
  - 3.2.3 The Request Type 8
- 3.3 Basic I/O 8
  - 3.3.1 IO and Exception abilities 8
  - 3.3.2 Defining Multiple Handlers 9
- 3.4 Users and Environment 10
  - 3.4.1 The Apostrophe in Unison 10
  - 3.4.2 Environment as a handler 10
- 3.5 Nondeterminism 11
  - 3.5.1 Joining Lists in Unison 12
- 3.6 Scheduling 12
- 3.7 Serial File System 13
- 3.8 Pipes 13
- 3.9 Unix Fork 13

4 Extensions **14**
- 4.1 Environment Variables 14

4.2 Generic Users 15

5 Evaluation **18**

6 Conclusion **19**

Bibliography **20**

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*. For example, we might define the effect signature *State* that stores state for some type `a`. In order to make use of our *State* effect we can define the effect operations *put* and *get* where put will update the value of type `a` stored in state and get will return the current value. 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. In the put and get example, any function that uses put and get to store values would be an effectful function. The *effect handler* provides one implementation of the given effect operation. We could define a simple handler for state that simply updates a variable of the given type or we could define a more complex one that uses hash maps. 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. Crucially, we can have multiple handlers defined in the same program for one effect allowing for much more modular programming or *effect-oreinted programming*.

When are programs rely on input from the real world like connecting to a server on the internet or getting input from a user, it is no longer safe to assume this input will be passed as we expect. For example, the server could time out or not be at the address the program is expecting it to be at or the user could enter a string that is too long for the input. These real world uncertainties are known as *Side Effects*. Effect handlers can be used to provide alternate implementations of functions that may have side effects and allow for control flow with these effects.

2.1.1 Example in Unison

Unison [1] is a functional language implemented in Haskell that offers built in support for effect handlers through it's abilities system.

Unison provides the *ability* keyword which allows users to define their own effects. It also provides the *handle ... with ...* pattern to attach handlers to effectful functions.

```
structural ability Store a where
  put: a -> {Store a} ()
  get: {Store a} a
```

Listing 2.1: The *put* and *store* example in Unison. Note that the *structural* keyword refers to the fact that Unison stores type definitions as a hash. Even if we changed all the variable names it would still view it as the same type. To avoid that behaviour you can swap the *structural* keyword for *unique*

This defines the two effect operations `put` and `get` that have the effect signature `Store a`. Put takes a value of type `a` and returns the unit type `()`. The prefix of {`Store a`} to the `()`, refers to the fact that in order to allow for `put` to return, it must be run from an effectful function that is handled with an appropriate handler for `Store a`. Similarly, `put` takes an argument of type `a` and must be handled.

```
addStore : a -> {Store a} ()
addStore x =
  y = get
  put (x + y)
```

Listing 2.2: An example of an effectful function that uses the `Store` effect

The code in listing 2.1.1 is an example of how you would use the effects in Unison. It takes an argument of type `a` and 'adds' it to the current value by using `get`. Note that in order for this to work the infix operation '+' must be implemented for type `a`. Now we only need to define the handler.

```
    storeHandler : a -> Request (Store a) a -> a
    storeHandler value = cases
        {Store.get -> resume} -> handle resume value with
   storeHandler value
        {Store.put v -> resume} handle resume () with storeHandler v
        {result} -> result
```

Listing 2.3: The handler for the `Store` effect

The handlers in Unison use tail recursion to reduce to the case where just the value is left `result -> result`. For both store and put we use the resumption and the handler to reach the final value. The special type `Request` allows us to perform pattern matching on the possible types of the computation.

```
handle (addStore 10) with storeHandler 10
```

Finally, we can put it all together by calling the function `addStore` with the handler `storeHandler`.

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

2.2 Affine and 'Multi-Shot' Handlers

If remaining computation or continuation of an effect can be resumed once from a handler then the effect system implements *one-shot* or *affine* effect handlers. If it is able to resume the computation multiple times then it is a *multi-shot* handler.

2.3 The State of Effect-Oriented Programming

2.3.1 Library Based Effects

- libhandler [4] 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 [2] 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 allow for 16000 stacks which may be challenging on some systems.
- cpp-effects [5] 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.
- There are several Haskell libraries that implement effect handlers [6, 7, 8]. Some are discussed in more detail below.
  - EvEff uses lambda calculus based evidence translation to implement it's effects system. It provides deep effects.
  - fused-effects [3] fuses the effect handlers it provides with computation by applying *fusion laws* that avoid intermediate representation. The handlers in fused-effects are one-shot however.

2.3.2 First-Class Effects

- Unison is shown in more detail in section 2.1.1
- Koka [9] 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 its standard library.

[2]https://github.com/koka-lang/libmprompt
[3]https://hackage.haskell.org/package/fused-effects

- Frank [10] is a strict functional language that is *effectful* in that it has first class support for bi-directional effects and effect handlers.
- Links [11] is a functional programming language designed for the web. Out of the box it does not support true algebraic effects, however through an extension [12] it gains first class support for continuations.

2.4 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* [13]. Deep handlers pass a copy of the full handler along with the computation which allows for the handler to be invoked again as the handlers receive themselves as an argument. Shallow handlers do not pass the handler with the computation. There are also *sheep handlers*, which while being shallow implement some of the behaviour of deep handlers leading to the name sheep or shallow + deep. In practice, the type of handler is more of an implementation detail although it can have an effect on how code is structured.

2.5 UNIX

UNIX [14] 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.

2.5.1 The UNIX Philosophy

A phrase often associated with UNIX is the *Unix philosophy*. The UNIX philosophy refers to some of the core principles with which it was developed. The core principles involve composing many small simple programs that accomplish one task well to solve more complex tasks [15]. The idea of many small modular components has spread to many areas of computer science including effect oriented programming.

2.6 Effect Based File Systems

Continuations in operating systems [16].

2.7 Effect Handlers and UNIX

In chapter 2 of his 2022 thesis, Daniel Hillerström [17] outlines a theoretical implementation of UNIX using the effects syntax outlined by Kammar et. al. [18]. In this he provides an implementation of the original UNIX paper [14] 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`[4] from which he only implements string replacement.

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

Chapter 3

Base Implementation

3.1 Effect Oriented Programming in Unison

As is shown in section 2.1.1, effect oriented programming in Unison is composed of an effect definition with an effect signature and a set of effect operations and any handlers for that effect signature.

3.2 Program Status

In `Unix` programs must provide a code when they exit (usually 0 for success and anything else for failure). The effect signature `Status` provides the `exit` operation which takes one argument of type `Nat` [1] and returns the unit type. A better definition would have exit return the empty type but unison does not have an empty type. The argument represents the return code.

```
unique ability Status
       where
           exit: Nat -> ()
```

3.2.1 Unique vs. Structural Types

In Unison, `unique` types are used when the name of the type is semantically important. The alternative is `structural` types which are used when the name of the type is not important and it can be stored as a hash without it's name. `unique` types are used for all effects as it has no real implication given the program is not distributed.

3.2.2 The Handler

The handler for `Status` is defined as:

```
exitHandler : Request {e, Status} x -> Nat
exitHandler request =
```

[1]a positive integer in Unison

```
    match request with
          { result } -> 0
          { exit v -> resume } -> abort
```

The implementation for exit has no effect it simply consumes the exit code and returns. The handler however returns a `Nat` return code. If an exit operation is encountered we return the value given to the exit operation. The return case simply returns 0 as if we reach the end of a function being handled by the handler then we can assume it was successful and return 0.

3.2.3 The Request Type

The `Request` type is a special type in Unison that allows for pattern matching on operations of an effect. In the braces are the effect types for the handler. The `Status` is the effect signature that is explicitly being handled. The `e` allows for any other effects in the computation to be passed through. The `x` is the return type of the computation.

3.3 Basic I/O

The *effect signature* `BasicIO` is used for simple I/O operations. The first and only *effect operation* of `BasicIO` is `echo` which takes an argument of type `Text` and returns the unit type `()`.

```
unique ability BasicIO where
  echo: Text -> ()
```

The handler for `BasicIO` is simply a wrapper for Unison's `putText` function which it uses to print the given text to `stdout`. It then handles the resumption with the same handler to handle any further `echo` calls.

```
basicIO : Request {BasicIO} x ->{IO, Exception} ()
basicIO result =
    match result with
        { echo text -> resume } ->
          putText stdOut text;
          handle resume () with basicIO
        { result } -> ()
```

3.3.1 IO and Exception abilities

The handler for `BasicIO` uses the `putText` function from Unison's standard library because of this we must include the {`IO, Exeption`} in the type signature to indicate that this function needs access to both the `IO` and `Exception` abilities in order to be run. Both of these abilities are built in and used for all input and output in unison.

Program 1 — Hello World

By combining the operations of Status and BasicIO we can write a simple program that prints “Hello, World!” and then exits with the successful error code. Notice that the operations are invoked in the same way as functions. In this case they are being used inside a function. It would be possible to implement a simple shell for these commands however that is outside the scope of this project.

```
greetAndExit : a ->{BasicIO, Status} ()
greetAndExit _ = echo "Hello, World!\n"; exit 0
```

By composing the two handlers in sections 3.2 and 3.3 we can run the program.

```
runGreetAndExit _ = handle (handle !greetAndExit with basicIO)
    with exitHandler
```

By running this function with the unison codebase manager we get

```
Hello, World!

  0
```

3.3.2 Defining Multiple Handlers

Effects are not limited to just one handler. The semantics of echo can be changed without altering it’s definition. For example, the `backwardsIO` handler below.

```
backwardsIO : Request {BasicIO} x ->{IO, Exception} ()
backwardsIO result =
  match result with
  { echo text -> resume } ->
    handle resume () with basicIO
    putText stdOut text;
  { result } -> ()
```

In this case, the resumption is handled first and then the text is printed. The effect of this is best shown by running it side by side with `basicIO` on the following program.

```
helloworld _ = echo "Hello,"; echo " World!\\n"
```

The output of running this program with each handler is shown in Figure ??.

```
> handle !helloworld with
   basicIO

Hello, World!

   ()
```

```
 > handle !helloworld with
   backwardsIO

 World!
Hello,
  ()
```

Figure 3.1: The output of running `helloworld` with each handler.

3.4 Users and Environment

To introduce the concept of a user-space and users we can start by adding some hard coded users. For now, alice, bob and a root user: `unique type User = Alice | Bob | Root`. To add generic users we could replace the definition with `unique type User = Text`, this would allow for the creation of new users.

Next we introduce the `Session` signature for operations involving users. The operation `su` or *substitute user* is used to change the environment to that of a different user. The `ask` operation can be used to access environment variables. Since the only variable we have now is `USER` the argument to ask is a unit.

```
unique ability Session
    where
        su: User -> {Session } ()
        ask: () -> {Session } Text
```

We can now implement the UNIX command `whoami` with a wrapper around ask.

```
whoami: '{Session} Text
whoami _ = ask ()
```

3.4.1 The Apostrophe in Unison

The `'` character in Unison is syntactic sugar for a function with a unit as the type of it's first argument. For example, the type signature of `whoami` could be rewritten as `() ->{Session} Text`. This is equivalent to `'{Session} Text`.

3.4.2 Environment as a handler

The handler for `Session` also takes a user as an argument, this is the user that is currently logged in. To switch user we simply handle the rest of the computation with the new user provided as the argument to `su`. Then when the computation ends we will be back in the environment of the old user.

Due to the single environment variable being `USER`, `ask` performs the action of `whoami`. It keeps the user the same and returns the user as a string.

```
env: User -> Request {Session} a -> a
env user request =
    match request with
        {result} -> result
        { ask () -> resume } -> match user with
            Alice -> handle resume "alice" with env user
            Bob -> handle resume "bob" with env user
            Root -> handle resume "root" with env user
        {su user' -> resume} -> handle resume () with env user'
```

In this way the environment is the handler itself as it contains the information such as which user is logged in. The handler can be extended to have parameterised environment variables making it the complete environment.

Program 2 — Session Management

We can now compose the handlers we have written so far to switch between the users and invoke `whoami`.

```
    session _ = su Alice
                echo (!whoami)
                echo "\n"
                su Bob
                echo (!whoami)
                echo "\n"
                su Root
                echo(!whoami)
                echo "\n"
```

```
    runsession _ = handle (handle (handle !session with env
   Root) with basicIO) with exitHandler
```

The `env` handler requires we give the user that is logged in initially as an argument so we pass `Root`. Notice we also handle the program with the `exitHandler` even though we do not invoke `exit` in the program. This is because by handling code with no `exit`'s with the `exitHandler` we will get the return case which will return 0 indicating the program was successful.

```
    alice
    bob
    root

      0
```

3.5 Nondeterminism

To implement the `fork` command from UNIX we can leverage deliberate non-determinism that is possible with effect handlers. We define the `fork` operation which returns a `Boolean` as a member of the `TimeSharing` signature.

```
unique ability TimeSharing
    where
        fork: Boolean
```

To use `fork` we can simply use it in control flow to create a branch. Where normally only one branch would be executed, the two branches become our two processes. For example:

```
...
if fork then
  process1 ()
else
  process2 ()
...
```

The handler for fork is also fairly simple:

```
nondet : Request {TimeSharing} a -> [a]
nondet request =
    match request with
        { fork -> resume } -> (handle resume true with nondet) lib.
    base.data.List.++ (handle resume false with nondet)
        { result } -> [result]
```

The handler returns a list of values with the type `a` which is the return type of the computation. When we encounter a `fork` we resume with the values true and false and join the two lists that are created. The return case wraps the value in a list so that we can use the ++ operator.

3.5.1 Joining Lists in Unison

Unison's typechecker sometimes struggles inferring the type of ++. For this reason we include the full path to the standard library where ++ is defined i.e. `lib.base.data.List.++`.

3.6 Scheduling

Now that we can create processes through `fork` it would be useful to be able to write scheduling algorithms. Currently `fork` will run the first process to completion, and then run the second process to completion. To begin scheduling we need to give the processes a method of stopping execution and giving control to the other process. We introduce the `Interrupt` signature with one operation also called `interrupt`.

```
unique ability Interrupt
    where
        interrupt: {Interrupt } ()
```

Now that we have `interrupt` we can write an alternative handler for `BasicIO` that will `interrupt` before every IO operation, allowing for the other process to run first.

```
interruptWrite : Request {e, BasicIO} x ->{e, Interrupt, BasicIO} ()
interruptWrite result =
    match result with
        { echo text -> resume } ->
            interrupt
            echo text
            handle resume () with interruptWrite
        { result } -> ()
```

Note that we still need to provide a handler for `echo`, this handler simply injects interrupt in front of every instance of `echo`.

In order to schedule processes we need to introduce state. Each process can either be `Done` (It has produced a return value) or `Paused` (It has been interrupted). `Paused` is a recursive definition as it contains a `PState` in it's type.

```
unique type PState a e = Done a | Paused ('{e} PState a e)
```

The type `a` is the return type of the process and the `e` is an effect variable that represents any effects that are needed to run the `PState`. We can now implement the handler for `interrupt`.

```
reifyProcess : Request {Interrupt, e} a -> PState a e
reifyProcess request =
    match request with
        { interrupt -> resume } -> Paused (_ -> handle !resume with
   reifyProcess )
        { result } -> Done result
```

In the case of an interrupt, the handler suspends the computation by making it an anonymous function with a unit type as it's first argument, and wrapping it in the paused datatype. This means we can run the `Paused` computations later by invoking that function we created. The return case simply wraps the value in the `Done` type.

```
sched : [PState a {e, TimeSharing}] -> [a] ->{e} [a]
sched ps done =
    match ps with
        [] -> done
        (Done res) +: ps' -> sched ps' (res lib.base.data.List.+:
   done)
        (Paused m) +: ps' -> sched (ps' lib.base.data.List.++ (
   handle !m with nondet)) done

timeshare : '{g, Interrupt, TimeSharing} o ->{g} [o]
timeshare m = sched [Paused (_ -> handle !m with reifyProcess)] []
```

3.7 Serial File System

3.8 Pipes

3.9 Unix Fork

Chapter 4

Extensions

4.1 Environment Variables

```
unique type User = Alice | Bob | Root

unique ability Session
    where
        su: User -> ()
        ask: Text -> Text
        export: Text -> Text -> ()

whoami: '{Session} Text
whoami _ = ask "USER"

env: User -> Request {Session} a ->{State [(User, [(Text, Text)])]}
   a
env user request =
    match request with
        {result} -> result

        { ask var -> resume } ->
            match var with
                "USER" ->
                    match user with
                        Alice -> handle resume "alice" with env user
                        Bob -> handle resume "bob" with env user
                        Root -> handle resume "root" with env user
                var ->
                    let st = get ()
                        envs = lookupEnvs user st
                        val = lookupEnvVar var envs
                        handle resume val with env user

        {su user' -> resume} ->
            handle resume () with env user'

        {export var val -> resume} ->
            let st = get ()
                envs = lookupEnvs user st
                envs' = modifyEnvVar var val envs
                put (modifyEnvs user envs' st)
                handle resume () with env user

lookupEnvVar: Text -> [(Text, Text)] -> Text
lookupEnvVar var env =
    match env with
        [] -> ""
        (var', val) +: rest ->
            if var == var' then val
            else lookupEnvVar var rest

modifyEnvVar: Text -> Text -> [(Text, Text)] -> [(Text, Text)]
modifyEnvVar var val env =
    match env with
        [] -> [(var, val)]
        (var', val') +: rest ->
            if var == var' then (var, val) +: rest
            else (var', val') +: modifyEnvVar var val rest

lookupEnvs: User -> [(User, [(Text, Text)])] -> [(Text, Text)]
lookupEnvs user envs =
    match envs with
        [] -> []
        (user', env) +: rest ->
            if userEquals user user' then env
            else lookupEnvs user rest

modifyEnvs: User -> [(Text, Text)] -> [(User, [(Text, Text)])] -> [(
   User, [(Text, Text)])]
modifyEnvs user env envs =
    match envs with
        [] -> [(user, env)]
        (user', env') +: rest ->
            if userEquals user user' then (user, env) +: rest
            else (user', env') +: modifyEnvs user env rest

userEquals: User -> User -> Boolean
userEquals user user' =
    match user with
        Alice ->
            match user' with
                Alice -> true
                _ -> false
        Bob ->
            match user' with
                Bob -> true
                _ -> false
        Root ->
            match user' with
                Root -> true
                _ -> false
```

4.2 Generic Users

```
unique type User = Username Text

unique ability Session
    where
        su: Text -> ()
        ask: Text -> Text
        export: Text -> Text -> ()
        adduser: Text -> ()

whoami: '{Session} Text
whoami _ = ask "USER"

env: User -> Request {Session} a ->{State [(User, [(Text, Text)])]}
   a
env user request =
    match request with
        {result} -> result

        { ask var -> resume } ->
           let st = get ()
               envs = lookupEnvs user st
               val = lookupEnvVar var envs
               handle resume val with env user

        {su user' -> resume} ->
            if userExists (Username user') (get ()) then
                handle resume () with env (Username user')
            else
                handle resume () with env user    -- fail

        {export var val -> resume} ->
            let st = get ()
                envs = lookupEnvs user st
                envs' = modifyEnvVar var val envs
                put (modifyEnvs user envs' st)
                handle resume () with env user

        {adduser user' -> resume} ->
            if userToText user == "root" then
                let st = get ()
                    newuser = (Username user')
                    newvars = [("USER", user')]
                    newenv = modifyEnvs newuser newvars st
                    if (userExists newuser st) then
                        put newenv
                        handle resume () with env newuser
                    else
                        handle resume () with env user
            else
                handle resume () with env user    -- fail

lookupEnvVar: Text -> [(Text, Text)] -> Text
lookupEnvVar var env =
    match env with
        [] -> ""
        (var', val) +: rest ->
```

```
            if var == var' then val
            else lookupEnvVar var rest

modifyEnvVar: Text -> Text -> [(Text, Text)] -> [(Text, Text)]
modifyEnvVar var val env =
    match env with
        [] -> [(var, val)]
        (var', val') +: rest ->
            if var == var' then (var, val) +: rest
            else (var', val') +: modifyEnvVar var val rest

lookupEnvs: User -> [(User, [(Text, Text)])] -> [(Text, Text)]
lookupEnvs user envs =
    match envs with
        [] -> []
        (user', env) +: rest ->
            if userToText user == userToText user' then env
            else lookupEnvs user rest

modifyEnvs: User -> [(Text, Text)] -> [(User, [(Text, Text)])] -> [(
   User, [(Text, Text)])]
modifyEnvs user env envs =
    match envs with
        [] -> [(user, env)]
        (user', env') +: rest ->
            if userToText user == userToText user' then (user, env)
   +: rest
            else (user', env') +: modifyEnvs user env rest

userExists: User -> [(User, [(Text, Text)])] -> Boolean
userExists user envs =
    match envs with
        [] -> false
        (user', env) +: rest ->
            if userToText user == userToText user' then true
            else userExists user rest

userToText: User -> Text
userToText user =
    let (Username username) = user
        username

initialUserspace : [(User, [(Text, Text)])]
initialUserspace = [(Username "root", [("USER", "root")] )]
```

Chapter 5

Evaluation

Chapter 6

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

[5] 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.

[6] Ningning Xie and Daan Leijen. Effect handlers in haskell, evidently. In Tom Schrijvers, editor, *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2020, Virtual Event, USA, August 7, 2020*, pages 95–108. ACM, 2020.

[7] Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. In Ben Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 94–105. ACM, 2015.

[8] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Effect handlers in scope. In Wouter Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 1–12. ACM, 2014.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] Eric S Raymond. *The art of Unix programming*. Addison-Wesley Professional, 2003.

[16] Oleg Kiselyov and Chung-chieh Shan. Delimited continuations in operating systems. In Boicho N. Kokinov, Daniel C. Richardson, Thomas Roth-Berghofer, and Laure Vieu, editors, *Modeling and Using Context, 6th International and Interdisciplinary Conference, CONTEXT 2007, Roskilde, Denmark, August 20-24, 2007, Proceedings*, volume 4635 of *Lecture Notes in Computer Science*, pages 291–302. Springer, 2007.

[17] Daniel Hillerström. Foundations for programming and implementing effect handlers. *Ph.D Thesis*, 2022.

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