Implementing Unix using Effect Handlers

Douglas Torrance

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

2025

Abstract

Algebraic effect provide a structured and modular way to reason about computational effects, such as exceptions, state and concurrency. Most imperative programming languages do not provide a structured method of handling effects, and functional approaches such as Monads lead to poor modularity and composability issues.

Effect handlers have emerged as a flexible, first-class mechanism to define and interpret effects dynamically. It is particularly useful for modelling complex control flows were it has found use in Multicore OCaml.

Unix was developed in the 1970s at Bell Labs to provide a new approach to operating system design. Prior operating systems were typically monolithic and hardware specific. Unix's was designed to be a small, portable and multi-user operating system which emphasised simplicity, modularity and composability. Its key innovations were

- Everything is a file - a unified abstraction for devices, processes and data
- Pipes and redirection - provides a mechanism to compose commands into programs
- Simplified interface - provides a small set of system calls for a consistent way of interacting with; process management, file handling and I/O

This dissertation is composed of two parts. The first part implements the ”Tiny Unix” system described in Hillerström's ”Foundations for Programming and Implementing Effect Handlers”. This is written in the Koka programming language to provide a modular and composable implementation of Unix functionality. The second part uses algebraic equivalence rules from Pretnar's ”An Introduction to Algebraic Effects and Handlers” to perform reasoning on the implementation, we determine the properties of this implementation and how it compares to standard Unix implementations.

Research Ethics Approval

Instructions: *Agree with your supervisor which statement you need to include. Then delete the statement that you are not using, and the instructions in italics.*
Either complete and include this statement:
This project obtained approval from the Informatics Research Ethics committee.
Ethics application number: ???
Date when approval was obtained: YYYY-MM-DD
[If the project required human participants, edit as appropriate, otherwise delete:]
The participants' information sheet and a consent form are included in the appendix.
Or include this statement:
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.

(*Douglas Torrance*)

Acknowledgements

Any acknowledgements go here.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Effect handlers are becoming a more popular way to implement programs. Unix provides an effective subject for us to model with effect handlers. It uses abstract system calls as an interface for the programs to interact with the OS. What we are doing is providing an implementation for them. We can represent each system call as an algebraic effect and provide modular implementations for them. Unix provides us with complicated state and control flow problems which we can demonstrate can be simply modelled using effect handlers, whilst also providing us a way to reason about these. This project will show how effect handlers can be used in a practical context to implement features in a composable way. It show cases one of the main advantages to using effect handlers, the ability reason about our code, this is particularly useful for reasoning about complicated control flows.

1.2 Aims

The objectives of this dissertation are:

- Provide back ground for effect handler oriented programming and how it models the Unix philosophy well
- Implement the ”Tiny Unix” system that Hillertstrom describes in his thesis
- Perform algebraic reasoning techniques on the implementation to show it meets the required specifications.

1.3 Outline

Chapter 2

Background

2.1 Algebraic Effects

Algebraic effects [8] and their handlers [9] are a structured approach to managing computational side effects. They provide a way to explicitly denote computations which produce side effects and handle them non-locally.

We describe an effect by first defining its effect signature, consisting of; the operations it can perform and their input parameters and return types. This provides the interface through which we can perform a side effect. We the abstract "effect signature" and define the implementation details of handling them later. Any function which uses this effect (unless it is handling itself), must include this effect as part of its function signature.

These effects must be handled at some point using an effect handler. When the runtime encounters an effect it searches through successively outer scopes until it finds the appropriate effect handler. This allows the effect to be handled in different ways depending on the scope in which the effect was raised. The effect handler will eventually run the code which carries out the intended side effect. Effect handlers can implement interesting control flow by capturing and managing the program's continuation.

Effect handler's separate the core business logic of the program from the implementation details of handling side effects. This means side-effect handling logic is no longer scattered around the code base, improving the modularity of the code. For example, to implement logging data accesses from a database, one would usually implement logging logic in all data access functions violating the single responsibility principle. Whereas with effect handlers you can define the logging effect and handle all the logging logic in one place.

Modular code allows us to easily compose effectful operations together. When chaining effectful operations in languages like Java it is not clear where the effect is occurring and how errors are handled, making it hard to reason about how the operations will work together. This feature makes effect handler's very useful implementing side effects such as asynchronous operations, state management, logging, exception handling etc.

In the context of functional programming, effects handlers provide a simple and lightweight method of handling side effects. They are a viable alternative to the monad to handle side-effects in a functional setting. It provides a more modifiable approach compared to monads which can be rigid once they are defined. It is easier than applying monad transformers to combine side effects. It also can have better performance compared to chained monads wrapping and unwrapping values into monadic values.

2.2 Support for Effect Handlers

As of the time of writing this paper, effect handlers only have native support in research languages such as Koka and Effect. Some languages have concepts similar to effect handlers, for example the concept of hooks in React are similar to effect handlers in the way they allow you to handle side effects in functional components.

Features required to have a strong support for effect systems include:

- First class effect handlers
- Delimited continuations
- Optimisation for nested effects
- Effect Type System.
    - Ability to write custom effect types
    - Effect signatures greatly eases reasoning about code and indicates purity vs impurity
    - Effect type safety and inference
    - Effect polymorphism

2.2.1 Native Support

- Eff [1]: This language is specifically designed for algebraic effects and effect handlers. It has support for first class effect handlers, delimiting effects and abstract effects
- Koka [7]: This supports effect type inference, strongly distinguishes between pure and impure computations
- MultiCore OCaml [10]: An official extension for OCaml which provides effect typing, first class effect handlers and strong pattern matching. integrates well with handling effect cases

2.2.2 Library Support

Many languages have their own libraries which attempt to implement effect handlers. These implementations often come with downsides compared to native implementations. Languages which don't have first class functions which makes it difficult to compose effects. Languages lacking an advanced type system find it difficult to create flexible and reusable handlers. Often languages' type systems don't provide type inference and ensure type safety for effects. They are also not optimised to track the multiple layers of context that an effect handler may produce. Despite this, there are some languages with effective third party libraries for using effect handlers.

- Haskell [6]: The polysemy library provides a strong implementation of effect handlers, including first class effect handlers, effect polymorphism, effect type inference. However its limited support makes it impractical to use so far.
- Scala [3]: The Cats Effect library has good performance and provides effect type polymorphism. However it doesn't provide first class effect handlers or directly support delimited continuations.

2.3 Syntax for Effect Handlers

2.3.1 Effect System

Effect Handler oriented languages have what is known as an effect system [2]. This describes the computational effects that may occur when a piece of code is executed. An effect system is typically an extension of a type system. Effect systems can be used to enforce effect safety, ensuring all effects are handled and functions only perform the side effects denotd in their effect signature

2.3.2 Effect Type

An effect type provides an explicit way to denote the side effects that a function performs. Each effect type can have multiple effectful operations. Each effectful operation can have parameters with value types and a valued return type. Note that operations can also produce effects and therefore have effect types.

```
effect exception
    ctl exn(error_msg : string) : a

fun safe_div (x : int, y: int) : exn int
    if y == 0 then exn("div by zero") else return x/y
```

Pure functions use the unit effect type. This shows it has no side effects.

```
fun add (x: Int, y: int) : () int {
    return x + y
}
```

2.4 Shallow vs Deep Handlers

Deep handlers [4] handlers can handle all the effects caused by a computation. When the continuation is captured in a handler, the captured continuation is also wrapped in the handler. This means that deep handlers can handle effects, which themselves invoke effects.

In contrast, shallow handlers [5] only manages the first effect caused by a computation. The resumed program no longer includes the handler and the programmer needs to provide a new handler for an effect the resumption may perform. Deep and Shallow handlers can simulate each others behaviour. However it may make more sense to use one type over another in certain contexts. Most languages only support one or the other. As of 2024 deep handlers are the more popular choice amongst effect oriented languages.

2.5 Unix

Unix is an operating system developed by Bell Labs in the 1970s. Its aim is to create a portable, multi-user, multi-tasking system. It is responsible for managing the:

- Processes: creating, managing process communication, terminating processes, forking processes
- Scheduling
- Basic IO
- User and user environment managment

Unix is built on the “Unix Philosophy” which follows the principles of simplicity and modularity. The “everything is a file” philosophy of unix provides a consistent interface for us to interact with system resources. This will allow us to separate functionality into modules and implement them using effect handlers.

2.6 Evaluating Previous work

Daniel Hillerstrom has implemented a theoretical implementation of unix in his 2021 paper “Foundations for Programming and Implementing Effect Handlers” [4]. He makes analogy between operating systems and effect handlers that both interpret a series of abstract commands, in the case of the OS this is system calls, in the case of effect handlers, this is operations. The composition of effect handlers, which he views as “tiny operating systems” can provide semantics for a unix implementation. He uses deep handlers to implement; multiple user sessions, time-sharing and file IO.

This paper will use Koka to create a concrete implementation of the abstract syntax he used in his paper. Then we will implement proof by inductions to show certain properties about effect handlers.

Chapter 3

Unix Implementation

This chapter shows the implementation of the ”Tiny Unix” system written by Hillerstrom in the Koka programming language. In his thesis, Hillertrom uses a pseudo-lambda calculus that can switch between deep and shallow handlers. I only use deep handlers in my Koka implementation.

3.1 Process Status

In Unix, when processes exit they must provide a code. A return code of 0 represents a successful execution and any other number otherwise. In a real unix system, the specific non-zero number represents the type of error, however we won’t implement this level of detail in our system.

We first begin by defining the exit effect with a single exit operation parameterised by an integer.

```
effect exit
  ctl exit(n : int) : a
```

We can now write the exit handler:

```
fun status(action : () -> <exit|e> a) : e int
    with final ctl exit(code) code
        action()
        0
```

In Hillerström’s language we needed to add an absurd function to coerce the valueless return into something that made sense within the language. However, in Koka, we can simply return a value without a resumption. In the case that the function returns without calling an exit(n), we assume that it exited successfully and simply return 0.

3.2 Basic IO

Hillerström first models a simplified form of state. You can see from the effect signature that can label the writes with a file descriptor. However, we don't implement the functionality until later and treat all writes as if they are happening to the same file, which we shall conceptualise and stdout, no matter the file descriptor.

```
effect bio
    fun writeBio( fd : filedesc, s : string ) : ()
```

Here we define the handler:

```
fun bio( action : () -> <bio|e> a ) : e (a,string)
  var buf := ""
  with handler
    return (x) -> (x, buf)
    fun writeBio(fd, s) -> { buf := buf ++ s }
  action()
```

Note that unlike in Hillerström's implementation we don't need explicit `resume` keywords. Effects of type `fun` just perform an effectful operation and continue execution immediately.

3.3 Users and Environment Variables

Environment variables are key-value pairs stored in the environment of a process. In this implementation, provide an example by storing the current user's name as an environment variable.

We store the users as type:

```
type user = Root, Alice, Bob
```

We define effects su (Switch User) parameterised by the user we wish to switch to and whoami() which returns the current user as a string.

```
effect su
  ctl su( u : user ) : ()

effect whoami
  fun whoami() : string
```

```
fun env( user : user, action : () -> <whoami|e> a ) : e a
  with fun whoami()
    match user
      Root  -> "root"
      Alice -> "alice"
      Bob   -> "bob"
  action()
```

Next we define seperate handlers for the `su` and `whoami()` effects

```
fun session-mgr( initial-user : user, action : () -> <su,whoami|e>
  with env(initial-user)
  with ctl su( u : user )
         mask<whoami>
           with env(u)
           resume(())
  action()

fun env( user : user, action : () -> <whoami|e> a ) : e a
  with fun whoami()
    match user
      Root  -> "root"
      Alice -> "alice"
      Bob   -> "bob"
  action()
```

Env is a handler parameterised by the current user, which simply wraps the resumption it is supposed to have scope for, which would represent a process, and returns this value when queried.

session-mgr manages user switches by wrapping the continuation in a new `env` handler. This provides greater modularity than global variables as the state is entirely encapsulated by the handler. This is very useful in the next, section which is concerned with process forking.

3.4 Multi-Process System

Forking

Hillerstrom describes how forking is a nondeterministic operation as from the perspective of the process making the Fork call. It is unclear in which order the processes will execute as this is managed by the operating system's scheduling policy.

```
effect fork
  ctl fork() : bool

fun forking( action : () -> <fork|e> a ) : e list<a>
  with handler
    return(x) [x]
    ctl fork() resume(True) ++ resume(False)
  action()
```

We can see that the forking handler introduces two resumptions, splitting computation down two paths. The resumption with True represents the parent and the resumption with False represents the child. We return these resumptions as a list which can be used by the scheduler.

This is designed to be used in code such as:

```
    if fork() {
        // ... parent specific code
    }
    else{
        /// ... child specific code
    }

    // code executed by the child and the parent
```

This is a simple way of writing code for two programs within the same file.

Scheduling

Fork has given us the capability to spawn new processes, it returns these as list of paused processes. We must implement a scheduling algorithm to interleave the running of these two processes.

First we must be able to differentiate between a process that has finished and one which is just temporarily paused. We introduce a data type to represent these two process states called `pstate`.

```
type pstate<e,a>
  Done(result : a)
  Paused(resumption : () -> e pstate<e,a> )
```

Currently there is no way for a process to indicate it is ready to pause, once it is started it will execute until it has finished. We introduce a way of yielding control to the scheduler to allow the interleaving of process execution.

We introduce an effect called ‘interrupt‘ and an effect handler called ‘reify process‘ which allows us to capture the continuation of a process after it has called interrupt().

```
effect interrupt
  ctl interrupt() : ()

fun reify-process(action : () -> <interrupt|e> a) : e pstate<e,a>
  with handler
    return(x) -> Done(x)
    ctl interrupt() -> Paused(fn() resume(()))
  action()
```

In our simplified system this must be done manually by the programmer. We could introduce wrapped functions such as `interrupt-write` however the programmer must be aware that due to non-determinism, these writes may not occur consecutively in the file:

```
interrupt-write("abc");
interrupt-write("def");
```

Finally we must introduce a scheduler to coordinate the running of our processes:

```
fun scheduler( pstates : list<pstate<<fork,div|e>,a>> ) : <div|e> l
  fun schedule( todos : list<pstate<<fork,div|e>,a>>, done : list<a
    match todos
      Nil -> done
      Cons(Done(x),ps') -> schedule(ps', Cons(x,done))
      Cons(Paused(m),ps') ->
        val ps = forking( m )
        schedule( ps' ++ ps, done )
  schedule(pstates,[])

fun timeshare( action : () -> <fork,interrupt,div|e> a ) : <div|e>
  val p = Paused( fn() reify-process(action) )
  scheduler([p])
```

It maintains two lists of processes, one containing paused processes and one containing finished processes. It works recursively; it runs the first process in the paused until it reaches an interrupt, then it runs the scheduler again with tehe process added to the end of the paused queue. If a process is finished it is added to the done queue. This procedure repeats until there are no processes left in the paused queue.

3.5 Serial File System

Here we implement a more realistic version of the file system we implemented earlier. A unix file system is implemented on top of an array of bytes. Unix structures this medium into several key regions to facilitate file management:

- Directory Region: This stores mappings between human-readable filenames and their corresponding Inode numbers.
- Inode Region: The Inode list contains meta data about each file
- Data Region: This stores the actual contents of file which are accessed via pointers stored in the iNode

We define our file system as a record type, composed of the these three lists and two integers to count where the next inode and data regions should be allocated.

```
type directory
  Directory
    d_list : list<(string, int)>

type dataRegion
  DataRegion
    dr : list<(int, string)>

type iNode
  INode
```

```
    no : int
    loc : int

type iList
  IList
    il : list<(int, iNode)>

struct fileSystem
  dir   : directory
  ilist : iList
  dreg  : dataRegion
  dnext : int
  inext : int
```

We define the initial file system state `fs0` as:

```
val fs0 : fileSystem =
  FileSystem (
    Directory([("stdout", 0)]),
    IList([(0, INode ( 1, loc = 0 ))]),
    DataRegion([(0, "")]),
    1,
    1
  )
```

File Reading and Writing

We define effects for reading and writing for our file system. The effect `read` is parameterised with an iNode number and returns an `Option<String>` type which indicates success or failure to read from the specified iNode. The `write` operation takes an integer to denote the iNode and a string to append to the file associated with this iNode. It does not indicate whether or not the write succeeded.

```
effect fileRW
    fun read( ino : int ) : option<string>
    fun write( ino : int, s : string ) : ()
```

```
fun fread( ino : int, fs : fileSystem ) : <div,fail|e> string
    val inode = lookupINode(ino, fs.ilist)
    val ret = lookupDataRegion(inode.loc, fs.dreg)
    ret
```

The various `lookup` utility functions must account for the possibility that they could fail. We introduce a `fail` effect and an associated `withDefault` handler to

```
effect fail
  ctl fail() : a
```

```
fun withDefault<a>(default: a, m: () -> <div,fail|e>a) : <div|e>a
  with handler
    return(x) -> x
    ctl fail() -> default
  m()
```

```
fun fileRW<a>(m: () -> <fileRW,div,state<fileSystem>|e>a) : <state<
a {
  with handler
    fun read(ino) {
      val cs = withDefault(None, fn() -> Some(fread(ino, get())))
      cs
    }
    fun write(ino, cs) {
      withDefault((), fn() {
          val fsys  = get()
          val fsys' = fwrite(ino, cs, fsys)
          put(fsys')
          ()
      })
    }
  m()
}
```

This is the handler for read and write operations in our system. We wrap `read` and `write` operations with `withDefault` to provide error handling, we make failures in lookup and modify functions correspond to None.

File Creation and Opening

We define effects for creating and opening files:

```
effect fileCO
  ctl create( fname : string ) : option<int>
  ctl open( fname : string ) : option<int>
```

In a real unix system when a process opens a file, they receive a file descriptor that refers to an entry in the open file descriptor table. This is adds an entry to a global 'Open File Table'. The OFT tracks; which files are open, how they are being accessed (read, write), at which offsets they are being accessed. We present a simplified version which means we don't provide shared state management in the same way. There are no restrictions to concurrent reading or writing. Our opening file mechanism simply returns the associated iNode from the file name.

```
fun fopen(fname: string, fs : fileSystem)
    match fs.dir {
        Directory(d_list) ->
        match d_list {
            Nil         -> fail()
            Cons((k, v), rest) ->
            if k == fname then v
            else lookupDirectory(fname, Directory(rest))
        }
    }
```

When creating a file we must handle two possibilities:

- The file already exists: in this case we must remove the file contents and simply return the associated iNode and the modified File System.
- The file doesn't exist: in this case we must first allocate space in the data region, then allocate an iNode and then add the file name to the directory. It returns the iNode of this file and the modified FileSystem as a tuple.

```
fun fcreate(fname: string, fs: fileSystem) : <div,fail> (int, fileS
  if has(fname, fs.dir) then {
    val ino = fopen(fname, fs)
    val inode = lookupINode(ino, fs.ilist)
    val dreg' = modifyDataRegion(inode.loc, "", fs.dreg)

    (ino, FileSystem(fs.dir, fs.ilist, dreg', fs.dnext, fs.inext))
  }
  else {
    val loc = fs.dnext  // Get next free data block
    val dreg' = DataRegion(Cons((loc, ""), fs.dreg.dr))

    val ino = fs.inext  // Get next free i-node index
    val inode = INode(loc, 1)  // Create a new i-node
    val ilist' = IList(Cons((ino, inode) , fs.ilist.il))

    val dir' = Directory(Cons((fname, ino), fs.dir.d_list))

    val fs' = FileSystem(dir', ilist', dreg', fs.dnext + 1, fs.inex
    (ino, fs')
  }

fun has<a, b>(key: string, xs: directory) : <div> bool
    withDefault(False, fn() {
        val disc = lookupDirectory(key, xs)
        True
    })
```

Now we can implement the handler for the Create an Open operations.

```
fun fileCreateOpen<a>(action : () -> <fileCO, div, state<fileSystem
  with handler {
    return(x) -> x
    ctl create(name) {
      val maybeIno = withDefault(None, fn() {
        val fs0 = get()
        val (ino,fs1) = fcreate(name, fs0)
        put(fs1)
        Some(ino)
      })
      resume(maybeIno)
    }

    ctl open(name) {
      val maybeIno = withDefault(None, fn() {
        val fs0 = get()
        val ino = fopen(name, fs0)
        Some(ino)
      })
      resume(maybeIno)}
  }
  action()
```

Stream Redirection

File Linking and Unlinking

We will implement two new operations, linking and unlinking files. There are 2 main types of link in Unix, hard and symbolic. We will implement hard links, this means that multiple file names can point to the same iNode. This allows us to create multiple references to the same underlying file.

```
effect fileLU
  ctl link(src: string, tgt : string) :()
  ctl unlink(tgt : string) :()
```

Chapter 4

Reasoning

In this section we used equational reasoning defined in Pretnar's Handlers Tutorial [9]. We use the algebraic equivalences describing deep effect handlers shown in figure... These allow us to show the observational equivalence of two programs. By this definition, two programs are equivalent if no external observer can distinguish between them. The external observer would have access to external state or communication channels with external systems.

Observational Equivalence is useful for;

- Reducing program complexity, we can replace a complex program with a more simple one
- Proving correctness, we can show that a function is implemented correctly by showing the implementation matches the specification in terms of observable behaviour.

  In this section we will compare Hillerstrom's implementation of "Tiny Unix" to the Unix specification. We will also show interesting properties of Unix

4.1 Simplified State Proofs

I started by reasoning on basic handlers provided in Pretnar's Handler Tutorial [9]. This is a simplified model of state which contains only one file. In this system the set is an rather than an append. So in this sense it is more similar to setting a variable than it is to writing to a file.

I could do reasoning on the four basic state proofs. These proofs are the minimum set needed to capture the basic behaviour of state. They allow us to reason about any sequence of get and sets.

$$\text{state} := \text{handler} \left\{ \begin{array}{l} \text{get}(-,k) \mapsto \lambda s \rightarrow (k\ s)\ s \\ \text{set}(s,k) \mapsto \lambda\_ \rightarrow (k\ ())\ s \\ \text{return}\ x \mapsto \lambda\_ \rightarrow \text{return}\ x \end{array} \right\}$$

$$
\begin{array}{ll}
(1) & \text{do } x \leftarrow \text{return } v \text{ in } c \equiv c[v/x] \\
(2) & \text{do } x \leftarrow \text{op}(v; y.\, c_1) \text{ in } c_2 \equiv \text{op}(v; y.\, \text{do } x \leftarrow c_1 \text{ in } c_2) \\
(3) & \text{do } x \leftarrow c \text{ in return } x \equiv c \\
(4) & \text{do } x_2 \leftarrow (\text{do } x_1 \leftarrow c_1 \text{ in } c_2) \text{ in } c_3 \equiv \text{do } x_1 \leftarrow c_1 \text{ in } (\text{do } x_2 \leftarrow c_2 \text{ in } c_3) \\
(5) & \text{if true then } c_1 \text{ else } c_2 \equiv c_1 \\
(6) & \text{if false then } c_1 \text{ else } c_2 \equiv c_2 \\
(7) & \text{if } v \text{ then } c[\text{true}/x] \text{ else } c[\text{false}/x] \equiv c[v/x] \\
(8) & (\lambda x \rightarrow c)\ v \equiv c[v/x] \\
(9) & \lambda x \rightarrow v\, x \equiv v
\end{array}
$$

In the following rules, we have
$h = \text{handler}\{\text{return } x \rightarrow c_r,\ \text{op}_1(x;k) \rightarrow c_1, \ldots,\ \text{op}_n(x;k) \rightarrow c_n\}$:

$$
\begin{array}{ll}
(10) & \text{with } h \text{ handle } (\text{return } v) \equiv c_r[v/x] \\
(11) & \text{with } h \text{ handle } (\text{op}_i(v; y.\, c)) \equiv c_i[v/x, (\lambda y \rightarrow \text{with } h \text{ handle } c)/k] \quad (1 \leq i \leq n) \\
(12) & \text{with } h \text{ handle } (\text{op}(v; y.\, c)) \equiv \text{op}(v; y.\, \text{with } h \text{ handle } c) \quad (\text{op} \notin \{\text{op}_i\}_{1 \leq i \leq n}) \\
(13) & \text{with } (\text{handler}\{\text{return } x \rightarrow c_2\}) \text{ handle } c_1 \equiv \text{do } x \leftarrow c_1 \text{ in } c_2
\end{array}
$$

Figure 4.1: Pretnar's Algebraic Equivalences

We can state these basic proofs as so:

- Idempotence of set
$$\text{set } a;\ \text{set } b = \text{set } b$$
- Read-After-Write
$$\text{set } a;\ \text{get} = a$$
- Idempotence of get
$$\text{get};\ \text{get} = \text{get}$$
- Write-After-Read
$$\text{get};\ \text{set } a = \text{set } a$$

Idempotence of Set

$$\begin{array}{l}\textbf{with}\ \mathsf{state}\ \textbf{handle}\\ \quad \mathrm{put}\ s\ (x.\,\mathrm{put}\ s'\ (y.C))\\ \equiv\\ \textbf{with}\ \mathsf{state}\ \textbf{handle}\\ \quad \mathrm{put}\ s'\ (y.C)\end{array}$$

Left Hand Side

$$\begin{array}{l}\quad\equiv\quad (11)\\ \mathrm{fun}\ \_ \to (k\ ())\ s\ [a/s,\ \mathrm{fun}\ x \to \textbf{with}\ \mathrm{state}\ \textbf{handle}\ put(b)(y.C)/k]\\ \quad\equiv\quad (\mathrm{subst})\\ \mathrm{fun}\ \_ \to ((\mathrm{fun}\ x \to \textbf{with}\ \mathrm{state}\ \textbf{handle}\ put(b)(y.C))\ ())\ a\\ \quad\equiv\quad (8)\\ \mathrm{fun}\ \_ \to (\textbf{with}\ \mathrm{state}\ \textbf{handle}\ put(b)(y.C))\ a\\ \quad\equiv\quad (11)\\ \mathrm{fun}\ \_ \to (\mathrm{fun}\ \_ \to (k\ ())\ s\ [b/s, \mathrm{fun}\ y \to \textbf{with}\ \mathrm{state}\ \textbf{handle}\ C/k])\ a\\ \quad\equiv\quad (\mathrm{subst})\\ \mathrm{fun}\ \_ \to (\mathrm{fun}\ \_ \to (\mathrm{fun}\ y \to \textbf{with}\ \mathrm{state}\ \textbf{handle}\ C)\ ())\ b)\ a\\ \quad\equiv\quad (8)\\ \mathrm{fun}\ \_ \to (\mathrm{fun}\ \_ \to (\textbf{with}\ \mathrm{state}\ \textbf{handle}\ C)\ b)\ a\\ \quad\equiv\quad (\mathrm{application})\\ \mathrm{fun}\ \_ \to (\textbf{with}\ \mathrm{state}\ \textbf{handle}\ C)\ b\end{array}$$

Right Hand Side

$$\begin{array}{l}\quad\equiv\quad (11)\\ \mathrm{fun}\ \_ \to (k\ ())\ s\ [b/s, \mathrm{fun}\ y \to \textbf{with}\ \mathrm{state}\ \textbf{handle}\ C/k]\\ \quad\equiv\quad (\mathrm{subst})\\ \mathrm{fun}\ \_ \to ((\mathrm{fun}\ y \to \textbf{with}\ \mathrm{state}\ \textbf{handle}\ C)\ ())\ b\\ \quad\equiv\quad (8)\\ \mathrm{fun}\ \_ \to (\textbf{with}\ \mathrm{state}\ \textbf{handle}\ C)\ b\end{array}$$

LHS $\equiv$ RHS

Idempotence of Get

$$\begin{array}{l}\textbf{with}\ \mathsf{state}\ \textbf{handle}\\ \quad \mathsf{get}()\,(\mathsf{x}.\mathsf{get}()\,(y.C))\\ \equiv\\ \textbf{with}\ \mathsf{state}\ \textbf{handle}\\ \quad \mathsf{get}()\,(x.C[x/y])\end{array}$$

LHS

$$\begin{array}{l}\quad\equiv\quad (11)\\ \text{fun } s \rightarrow (k\ s)\ s\ [()/v, \text{fun } x \rightarrow \textbf{with} \text{ state } \textbf{handle } get()(y.C)/k]\\ \quad\equiv\quad (\text{subst})\\ \text{fun } s \rightarrow ((\text{fun } x \rightarrow \textbf{with} \text{ state } \textbf{handle } get()(y.C))\ s)\ s\\ \quad\equiv\quad (8)\\ \text{fun } s \rightarrow (\textbf{with} \text{ state } \textbf{handle } get()(y.C[s/x]))\ s\\ \quad\equiv\quad (11)\\ \text{fun } s \rightarrow ((\text{fun } s' \rightarrow (k\ s')\ s')\ [()/v, \text{fun } y \rightarrow \textbf{with} \text{ state } \textbf{handle}/k])\ s\\ \quad\equiv\quad (\text{subst})\\ \text{fun } s \rightarrow ((\text{fun } s' \rightarrow ((\text{fun } y \rightarrow \textbf{with} \text{ state } \textbf{handle } C[s/x])\ s')\ s')\ s\\ \quad\equiv\quad (8)\\ \text{fun } s \rightarrow (\text{fun } s' \rightarrow (\textbf{with} \text{ state } \textbf{handle } C[s/x][s'/y]s'))\\ \quad\equiv\quad (8)\\ \text{fun } s \rightarrow (\textbf{with} \text{ state } \textbf{handle } C[s/x][s/y])\ s\end{array}$$

RHS

$$\begin{array}{l}\quad\equiv\quad (11)\\ \text{fun } s \rightarrow (k\ s)\ s\ [()/v, C[x/y]]\\ \quad\equiv\quad (\text{subst})\\ \text{fun } s \rightarrow ((\text{fun } x \rightarrow \textbf{with} \text{ state } \textbf{handle } C[x/y]))\\ \quad\equiv\quad (8)\\ \text{fun } s \rightarrow (\textbf{with} \text{ state } \textbf{handle } C[x/y][s/x])\ s\end{array}$$

LHS $\equiv$ RHS

Read-After-Write

$$\begin{array}{l}\textbf{with}\ \textsf{state}\ \textbf{handle}\\ \quad \text{put}\ s'\ (x.\,\text{get}()\ (y.C))\\ \equiv\\ \textbf{with}\ \textsf{state}\ \textbf{handle}\\ \quad \text{put}\ s'\ (x.C[s'/x])\end{array}$$

Left Hand Side

$$\begin{array}{l}\quad\equiv\quad (11)\\ \text{fun}\ \_ \rightarrow (k\ ())\ s\ [a/s, \text{fun}\ x \rightarrow \textbf{with}\ \text{state}\ \textbf{handle}\ get()(y.C)/k]\\ \quad\equiv\quad (\text{subst})\\ \text{fun}\ \_ \rightarrow ((\text{fun}\ x \rightarrow \textbf{with}\ \text{state}\ \textbf{handle}\ get()(y.C))())\ a\\ \quad\equiv\quad (8)\\ \text{fun}\ \_ \rightarrow (\textbf{with}\ \text{state}\ \textbf{handle}\ get()(y.C))\ a\\ \quad\equiv\quad (11)\\ \text{fun}\ \_ \rightarrow (\text{fun}\ s' \rightarrow (k\ s')\ s'[()/s, \text{fun}\ y \rightarrow \textbf{with}\ \text{state}\ \textbf{handle}\ C/k])\ a\\ \quad\equiv\quad (\text{subst})\\ \text{fun}\ \_ \rightarrow ((\text{fun}\ s' \rightarrow ((\text{fun}\ y \rightarrow \textbf{with}\ \text{state}\ \textbf{handle}\ C)\ s')\ s')\ a\\ \quad\equiv\quad (\text{application})\\ \text{fun}\ \_ \rightarrow ((\text{fun}\ y \rightarrow \textbf{with}\ \text{state}\ \textbf{handle}\ C)\ a)\ a\\ \quad\equiv\quad (8)\\ \text{fun}\ \_ \rightarrow (\textbf{with}\ \text{state}\ \textbf{handle}\ C[a/y])\ a\end{array}$$

Right Hand Side

$$\begin{array}{l}\quad\equiv\quad (11)\\ \text{fun}\ \_ \rightarrow (k\ ())\ s\ [a/s, \text{fun}\ x \rightarrow \textbf{with}\ \text{state}\ \textbf{handle}\ C[a/y]/k]\\ \quad\equiv\quad (\text{subst})\\ \text{fun}\ \_ \rightarrow ((\text{fun}\ x \rightarrow \textbf{with}\ \text{state}\ \textbf{handle}\ C[a/y])())\ a\\ \quad\equiv\quad (8)\\ \text{fun}\ \_ \rightarrow (\textbf{with}\ \text{state}\ \textbf{handle}\ C[a/y])\ a\end{array}$$

Write-After-Read

$$\begin{array}{l}\textbf{with}\ \textsf{state}\ \textbf{handle}\\ \quad \text{get}()\ \big(x.\,\text{put}\ s'\ (y.C)\big)\\ \equiv\\ \textbf{with}\ \textsf{state}\ \textbf{handle}\\ \quad \text{put}\ s'\ (y.C[s/x])\end{array}$$

Left Hand Side

$$\equiv \quad (11)$$

$$\text{fun } s \to (k\ s)\ s\ [()/v, \text{fun } x \to \textbf{with} \text{ state } \textbf{handle}\ set(a)(y.C)/k]$$

$$\equiv \quad (\text{subst})$$

$$\text{fun } s \to ((\text{fun } x \to \textbf{with} \text{ state } \textbf{handle}\ set(a)(y.C))\ s)\ s$$

$$\equiv \quad (8)$$

$$\text{fun } s \to (\textbf{with} \text{ state } \textbf{handle}\ set(a)(y.C[s/x]))\ s$$

$$\equiv \quad (11)$$

$$\text{fun } s \to ((\text{fun } \_ \to (k\ ())\ s')[a/s', \text{fun } y \to \textbf{with} \text{ state } \textbf{handle}\ C[s/x]/k])\ s'$$

$$\equiv \quad (\text{subst})$$

$$\text{fun } s \to (\text{fun } \_ \to (\textbf{with} \text{ state } \textbf{handle}\ C[s/x])())\ a$$

$$\equiv \quad (8)$$

$$\text{fun } s \to (\textbf{with} \text{ state } \textbf{handle}\ C[s/x])\ a$$

Right Hand Side

$$\equiv \quad (11)$$

$$\text{fun } \_ \to (k\ ())\ s\ [a/s, \text{fun } y \to \textbf{with} \text{ state } \textbf{handle}\ C[s/x]/k]$$

$$\equiv \quad (\text{subst})$$

$$\text{fun } \_ \to (\text{fun } y \to \textbf{with} \text{ state } \textbf{handle}\ C[s/x])()\ a$$

$$\equiv \quad (8)$$

$$\text{fun } \_ \to (\textbf{with} \text{ state } \textbf{handle}\ C[s/x]\ a)$$

LHS $\equiv$ RHS

4.2 Exceptions

Processes can terminate successfully by running to completion, or they can terminate by performing an exit system call. The exit call is parameterised by a number which indicates the exit status.

An exception does not allow recovery from the error as the process is terminated.

$$\text{status} \overset{\text{def}}{=} \{\text{Exit} : \text{Int} \to 0\}$$

$$\text{status} := \text{handler} \left\{ \begin{array}{r} \text{return } \_ \mapsto 0 \\ \ll \text{Exit } n \gg \mapsto n \end{array} \right\}$$

Absorption

The absorption property states that once an operation like exit is called within a computation, any further computation is ignored and discarded. Formally, we can express this as:

$$\begin{array}{l} \text{with } \textit{status} \text{ handle} \\ \quad \text{Exit (n)(y.C)} \\ \equiv \\ \text{with } \textit{status} \text{ handle} \\ \quad \text{Exit(n)(y.D)} \end{array}$$

Since `Exit n` has a return type of 0, which is the uninhabited type, i.e. a type which has no associated values, the expression `Exit n` can't return a value. The absurd function is used to coerce this to return a value of type α. This is due to the fact that `Exit n` has no resumption. There is no computation left after calling Exit n so it doesn't make sense for it to return a value. The expression Exit n on its own is not well-defined in this type system, because it doesn't return a meaningful value, so we must wrap it in this exit n function so we can use it in the code.

$$\text{exit n := handler}\left\{\text{absurd (do Exit n)}\right\}$$

LHS

(subst)	$\equiv$ absurd (do Exit n) (y. C)
(11)	$\equiv v$ [n/v, fun y $\rightarrow$ with status handle C/k]
(subst)	$\equiv n$

RHS

(11)	$\equiv$ absurd (do Exit n) (y. D)
(11)	$\equiv v$ [n/v, fun y $\rightarrow$ with status handle D/k]
(subst)	$\equiv n$

4.3 Environment Variables

Environment variables are name-value pairs specific to each process. In our system the only environment variable we need to reason about is the user

In this section we only examine intra process reasoning. This can be seen as a specialisation of the state proofs we have done before. The same state laws will apply here. However this case is interesting as the way it implements state is quite different to the previous example. The `BasicIO` models state as a single global file, whereas this models state as a locally scope handler instance. Mutation occurs through handler overloading rather than direct state updates.

This is an effective way modelling state for independent processes, it allows each state to be isolated and for the nested processes to automatically unwind.

We will examine, in a later section, the strength of this approach by reasoning about how it interacts with process forking. For now we will show it holds the same basic state proofs as `BasicIO`

The put operation corresponds to switch user command and the get operation corresponds to the whoami() command. Similarly, it works on a single ”cell” for each process.

$$\text{sessionmgr } \langle user, m\rangle := \text{handler} \left\{ \begin{array}{r} env\langle user; (\lambda\langle\rangle.\ \text{handle } m\langle\rangle \text{ with} \\ \textbf{return}\text{res} \rightarrow res \\ \langle\langle \text{Su } user' \rightarrow resume\rangle\rangle \rightarrow \text{env}\langle user', resume\rangle)) \end{array} \right\}$$

Switch User Idempotence

```
Su(Alice);
Su(Alice);
```

$\equiv$

```
Su(Alice);
```

$$\begin{array}{l} \text{with } sessionmgr(Root) \text{ handle} \\ \quad \text{Su(Alice);Su(Alice;} \\ \equiv \\ \text{with } sessionmgr(Root) \text{ handle} \\ \quad \text{Su(Alice);} \end{array}$$

4.4 Process Forking

Process forking introduces a number of interesting properties for us to consider. We will first examine the properties that the Fork handlers have themselves, then we will go on to anaylse how they interact with other handlers.

Associativity

$$(a \oplus b) \oplus c \quad \equiv \quad a \oplus (b \oplus c)$$

This is an important property to prove for parrallel programming purposes. We can show that despite the fact that the process tree structure are different, the order of execution will remain the same. This allows tree based paralleism. As long as the are independent processes don’t rely on the parent i.e. wait() commands, signals or inherited environment variables child dependencies then this equivalent.

This is a useful property as it allows us as programmers to rearrange or simplify complex structures. For example:

```
Server forks Worker 1
    Worker 1 forks Worker 2
        Worker 2 forks Worker 3
```

```
Server forks Worker 1
Server forks Worker 2
Server forks Worker 3
```

with *nondet* handle
    if Fork() then if Fork() a else b else c

$\equiv$

with *nondet* handle
    if Fork() then a else if Fork() then b else c

LHS

(11)	$\equiv$ resume true ++ resume false [()/x, fun r $\to$ with nondet handle (if r then a else Fork()(s $\to$ if s then b else c)) / k]
(subst)	$\equiv$ (fun r $\to$ with nondet handle (if r then a else Fork()(s $\to$ if s then b else c))) true ++ (fun r $\to$ with nondet handle (if r then a else Fork()(s $\to$ if s then b else c))) false
(8 + evaluate)	$\equiv$ with nondet handle a ++ with nondet handle Fork()(s $\to$ if s then b else c)
(11)	$\equiv$ a ++ (resume true ++ resume false [()/x, fun s $\to$ if s then b else c / k])
(subst)	$\equiv$ a ++ ((fun s $\to$ if s then b else c) true) ++ ((fun s $\to$ if s then b else c) false)
(8 + evaluate)	$\equiv$ a ++ (b ++ c)

RHS

(11)	$\equiv$ resume true ++ resume false [()/x, fun r $\rightarrow$ with nondet handle (if r then Fork()(s $\rightarrow$ if s then a else b) else c) / k]
(subst)	$\equiv$ (fun r $\rightarrow$ with nondet handle (if r then Fork()(s $\rightarrow$ if s then a else b) else c)) true ++ (fun r $\rightarrow$ with nondet handle (if r then Fork()(s $\rightarrow$ if s then a else b) else c)) false
(8 + evaluate)	$\equiv$ with nondet handle Fork()(s $\rightarrow$ if s then a else b) ++ with nondet handle c
(11)	$\equiv$ (resume true ++ resume false [()/x, fun s $\rightarrow$ if s then a else b / k]) ++ c
(subst)	$\equiv$ ((fun s $\rightarrow$ if s then a else b) true) ++ ((fun s $\rightarrow$ if s then a else b) false) ++ c
(8 + evaluate)	$\equiv$ ((fun s $\rightarrow$ if s then a else b) true) ++ ((fun s $\rightarrow$ if s then a else b) false) ++ c
(evaluate)	$\equiv$ (a ++ b) ++ c

$$\textbf{LHS} = \textbf{RHS} \quad \text{and} \quad a++(b++c) \equiv (a++b)++c$$

Distributivity

Distributivity describes how two binary operation interact with one another. Formally, given three elements A,B,C and two binary operations $\diamond, \circ$

$$(B \circ A) \diamond (C \circ A) \equiv (B \diamond C) \circ A$$

Right Distributivity

We can show that non determinism and sequencing (considered a binary operation `P;Q`) are right distributive within our system. We can prove that the code:

```
if Fork()
    then P; R;
    else Q; R;
}
```

$\equiv$

```
if Fork()
    then P;
    else Q;
R;
```

Expressing this property in the continuation passing style of our equivalence rules:

with *nondet* handle
  Fork()(t.(if t then P;R else Q;R)
$\equiv$
with *nondet* handle
  Fork()(t.(if t then P else Q); R)

LHS

$$\equiv \quad (11)$$
$$\text{resume True} ++ \text{resume False}[()/x, \text{fun } t \to (\text{if } t \text{ then } P;R \text{ else } Q;R)/k]$$
$$\equiv \quad (\text{subst})$$
$$\text{fun } t \to (\text{if } t \text{ then } P;R \text{ else } Q;R) \text{ True}$$
$$++$$
$$\text{fun } t \to (\text{if } t \text{ then } P;R \text{ else } Q;R) \text{ False}$$
$$\equiv \quad (8, \text{simplify})$$
$$P;R ++ Q;R$$

RHS

$$\equiv \quad (11)$$
$$\text{resume True} ++ \text{resume False}[()/x, \text{fun } t \to (\text{if } t \text{ then } P \text{ else } Q;R)/k]$$
$$\equiv \quad (\text{subst})$$
$$\text{fun } t \to (\text{if } t \text{ then } P \text{ else } Q);R \text{ True}$$
$$++$$
$$\text{fun } t \to (\text{if } t \text{ then } P \text{ else } Q);R \text{ False}$$
$$\equiv \quad (8, \text{simplify})$$
$$(P;R)$$
$$++$$
$$(Q;R)$$
$$\equiv \quad P;R ++ Q;R$$

Left Distributivity

Unfortunately we can't show the same for left distributivity:

```
R;
if Fork()
    then P;
    else Q;
}
```

Is not equivalent to:

```
if Fork()
    then R; P;
    else R; Q;
}
```

This is because in the first example R is only executed once by the unforked process, whereas in the second example it is executed by both the parent and the child. Mutation operations such as append(x) would be applied a different numbers of times breaking the equivalence.

Whilst right distributivity does not hold in general, it does hold in the special case of idempotent effects, where repeated applications of R are equivalent to applying it once.

```
set(x);
if Fork()
    then P;
    else Q;
}
```

$\equiv$

```
if Fork()
    then set(x); P;
    else set(x); Q;
}
```

Process Suspension and Resumption

We must show that the act of suspending a process through an interrupt and resuming after a number of computations doesn't change any of the processes environment variables. This is difficult to do with the current scheduler

Interaction With Exit

This next property is a specialisation of the absorption proof for exceptions shown earlier.

$$\begin{array}{l}\text{with } \textit{nondet} \text{ handle} \\ \quad \text{with status handle} \\ \quad\quad \text{if Fork() then exit(-1) else } b \\ \equiv \\ \text{with } \textit{nondet} \text{ handle} \\ \quad -1 ++ (\text{with status handle } b)\end{array}$$

LHS

(11, non det)	$\equiv$ resume true ++ resume false [()/x, (fun r $\rightarrow$ with status handle (if r then exit() else b))]
(subst)	$\equiv$ (fun r $\rightarrow$ with status handle (if r then exit() else b)) true ++ (fun r $\rightarrow$ with status handle (if r then exit() else b)) false
(8 + simplify)	$\equiv$ with status handle (exit(-1)) ++ with status handle (b)
(11, status)	$\equiv$ x [-1/x, fun x $\rightarrow$ C / k] ++ with status handle (b)
(subst)	$\equiv$ -1 ++ with status handle (b)

LHS = RHS

Interaction With State

Reasoning about how multiple processes interact with shared state is vital to understanding how our code will behave.

Unix systems typically operate non-backtrackable file systems. The non determinism of programs introduces many problems for us to deal with, we can't trust which order processes (and their associated effects) will be executed. Since state isn't commutative this poses a large problem.

We first examine how the order of our handlers affect how the system interprets the interaction between forking and state.

State Outer, Nondet Inner

This is the approach that Hillerström takes in his thesis to model the file system. This provides a persistent, global state which all processes can interact with. We demonstrate this by showing that set commands in the parent and child processes alter the same global state.

```
with state handle
    with nondet handle
        if Fork() then set(True) else set(False)
```

$$\equiv \quad (12)$$

$$\textbf{with}\ \text{nondet}\ \textbf{handle}\ \text{Fork}()(a.\ \textbf{with}\ \text{state}\ \textbf{handle}\ (\text{if}\ a\ \text{then set(True)}(b.C)\ \text{else set(False)}(d.E)))$$

$$\equiv \quad (11)$$

$$\text{resume True}++\text{resume False}[()/v, \text{fun}\ a \rightarrow \textbf{with}\ \text{state}\ \textbf{handle}\ \text{if}\ a\ \text{then set(True)}(b.\text{get}()(c.D))\ \text{else}$$

$$\text{fun}\ a \rightarrow \textbf{with}\ \text{state if}\ a\ \text{then set(True)}(b.\text{get}()(c.D)\ \text{else set(False)}(d.\text{get}()(e.F)\ \text{True}$$

$$++$$

$$\text{fun}\ a \rightarrow \textbf{with}\ \text{state}\ \textbf{handle}\ \text{if}\ a\ \text{then set(True)}(b.\text{get}()(c.D)\ \text{else set(False)}(d.\text{get}()(e.F)\ \text{False}$$

$$\equiv \quad (8, \text{simplify})$$

$$\textbf{with}\ \text{state}\ \textbf{handle}\ \text{set(True)}(b.\text{get}()(c.D))$$

$$++$$

$$\textbf{with}\ \text{state set(False)}(d.\text{get}()(e.F))$$

$$\equiv \quad (11)$$

$$\text{fun}\ \_ \rightarrow (k\ ())\ s\ [\text{True}/s, \text{fun}\ b \rightarrow \textbf{with}\ \text{state}\ \textbf{handle}\ \text{get}()(c.D)]$$

$$++$$

$$\text{fun}\ \_ \rightarrow (k\ ())\ s\ [\text{False}/s, \text{fun}\ d \rightarrow \textbf{with}\ \text{state}\ \textbf{handle}\ \text{get}()(e.F)]$$

$$\equiv \quad (\text{subst})$$

$$\text{fun}\ \_ \rightarrow (\text{fun}\ b \rightarrow \textbf{with}\ \text{state}\ \textbf{handle}\ \text{get}()(c.D))\ \text{True}$$

$$++$$

$$\text{fun}\ \_ \rightarrow (\text{fun}\ d \rightarrow \textbf{with}\ \text{state}\ \textbf{handle}\ \text{get}()(e.F))\ \text{False}$$

$$\equiv \quad (11, \text{get state})$$

$$\text{fun}\ \_ \rightarrow (\text{fun}\ s \rightarrow (k\ s)\ s\ [()/v, \text{fun}\ c \rightarrow \textbf{with}\ \text{state}\ \textbf{handle}\ D/k])\ \text{True}$$

$$++$$

$$\text{fun}\ \_ \rightarrow (\text{fun}\ s \rightarrow (k\ s)\ s\ [()/v, \text{fun}\ e \rightarrow \textbf{with}\ \text{state}\ \textbf{handle}\ F/k])\ \text{False}$$

$$\equiv \quad (\text{application})$$

$$\text{fun}\ \_ \rightarrow \textbf{with}\ \text{state}\ \textbf{handle}\ D[\text{True}/c]\ \text{True}$$

$$++$$

$$\text{fun}\ \_ \rightarrow \textbf{with}\ \text{state}\ \textbf{handle}\ F[\text{False}/c]\ \text{False}$$

Nondet Outer, State Inner

Instead of global state, this handler configuration gives each process its own independent state. This state only exists for as long as the process is running. This is analogous to processes setting their own local variables or environment variables.

```
with nondet handle
    with status handle
        if Fork() then set(True) else set(False)
```

$\equiv$ (11)

$\textbf{resume True} ++ \textbf{resume False}$

$[()/v, \text{fun } p \to \textbf{with} \text{ nondet } \textbf{handle} \text{ if } p \text{ then set(True)}(a.C) \text{ else get(False)}(b.C)]$

$\equiv$ (subst)

$\text{fun } p \to \textbf{with} \text{ nondet } \textbf{handle} \text{ if } p \text{ then set(True)}(a.C) \text{ else get()}(b.C) \text{ True}$

$++$

$\text{fun } p \to \textbf{with} \text{ nondet } \textbf{handle} \text{ if } p \text{ then set(True)}(a.C) \text{ else get()}(b.C) \text{ False}$

$\equiv$ (8, simplify)

$\textbf{with} \text{ nondet } \textbf{handle} \text{ set(True)}(a.C)$

$++$

$\textbf{with} \text{ nondet } \textbf{handle} \text{ get()}(b.D)$

$\equiv$ (12)

$\text{set(True)}(a.\textbf{with} \text{ nondet } \textbf{handle } C)$

$++$

$\text{get()}(b.\textbf{with} \text{ nondet } \textbf{handle } D)$

(Scheduler executes processes sequentially)

$\text{set(True)}(a.\textbf{with} \text{ nondet } \textbf{handle} \text{ get()}(b.\textbf{with} \text{ nondet } \textbf{handle } D))$

$\equiv$ (11)

$\text{fun } \_ \to (k\ ())\ s\ [\text{True}/s, \text{fun } a \to \textbf{with} \text{ nondet } \textbf{handle} \text{ get()}(b.\textbf{with} \text{ nondet } \textbf{handle } C)/k]$

$\equiv$ (subst)

$\text{fun } \_ \to ((\text{fun } a \to \textbf{with} \text{ nondet } \textbf{handle} \text{ get()}(b.\textbf{with} \text{ nondet } \textbf{handle } D)())\ s)$

$\equiv$ (8, discard a)

$\text{fun } \_ \to (\textbf{with} \text{ nondet } \textbf{handle} \text{ get()}(b.\textbf{with} \text{ nondet } \textbf{handle } C))\text{ True}$

$\equiv$ (12)

$\text{fun } \_ \to \text{get()}(b.\textbf{with} \text{ nondet } \textbf{handle } C) \text{ True}$

$\equiv$ (11)

$\text{fun } \_ \to \text{fun } s \to (ks)s[()/v, \text{fun } y \to \text{get()}(b.\textbf{with} \text{ nondet } \textbf{handle } C)/k] \text{ True}$

$\equiv$ (subst)

$\text{fun } \_ \to \text{fun } s \to ((\text{fun } y \to \text{get()}(b.\textbf{with} \text{ nondet } \textbf{handle } C)s)s \text{ True})$

$\equiv$ (apply)

$\text{fun } \_ \to \text{fun } y \to (\text{get()}(b.\textbf{with} \text{ nondet } \textbf{handle } C) \text{ True}$

$\equiv$ (8)

$\text{fun } \_ \to \text{get()}(b.\textbf{with} \text{ nondet } \textbf{handle } C[\text{True}/y])$

Interaction With Environment Variables

According to the Unix specifications, when a child is forked from its parent, it inherits some of the parent's state including environment variables. However from this point onwards the environments are separate and changes to the child's environment variable shouldn't affect the parent's and vice versa.

The Child Inherits the Parent's Environment Variables

Parent Changing User doesn't Affect the Child

```
#Starts as root

if Fork()
    su(Alice);
    whoami();
}
else {
    whoami();
}

# Should produce ["Alice", "Root"]
```

with *nondet* handle

  with sessionmgr(Root) handle

    Fork()(r.(if r then (su; whoami()) else whoami()))

$\equiv$

with handle

(12) with nondet handle

  Fork()(r $\mapsto$ with sessionmgr handle (if r then (su; whoami()) else whoami()))

(11) $\equiv$ resume(true) ++ resume(false) [()/x, fun r $\rightarrow$ with sessionmgr handle (if r then (su; whoami()) else whoami()) / k]

(subst) $\equiv$ (fun r $\rightarrow$ with sessionmgr handle (if r then (su; whoami()) else whoami())) true

  ++ (fun r $\rightarrow$ with sessionmgr handle (if r then (su; whoami()) else whoami())) false

(8) $\equiv$ with sessionmgr handle (su(”Alice”)(x. whoami()(y.C)))

  ++ with sessionmgr(”root”) handle (whoami()(z.C))

(11) $\equiv$ env(x, k) [(“Alice”/x, fun y $\rightarrow$ with sessionmgr handle whoami()(y.C)/k)]

  ++ (z $\leftarrow$ do Ask()) [()/x, fun z $\rightarrow$ with env handle C/k]

RHS

4.5 Pipe

Pipes are a data stream abstraction that enables communication between processes. The pipe command creates two file descriptors; a read end and a write end. In a typically shell comand `A|B`, the shell forks two processes A — B, it redirects A stdout to the write end (stdout) of the pipe and it redirects B's stdin the read end (stdin) of the pipe.

The Unix kernel handles synchronisation between these two processes by blocking reads and writes under certain conditions. If the write buffer becomes full, the kernel will block the writer process until the consumer reduces the size of the buffer. Similarly, the process will block the reader if the reader buffer is of size 0.

In Hillerstrom's system we handle this producer/consumer modelling using shallow effect handlers. Instead of using a buffer we write ...

Piping can be seen as a form of function composition, if we consider the processes as functions, we take the output from one process and feed it as input into the next one. This means it should follow the same rules as mathematical function composition namely:

- Associativity
-

Chapter 5

Conclusions

5.1 Final Reminder

The body of your dissertation, before the references and any appendices, *must* finish by page 40. The introduction, after preliminary material, should have started on page 1.

You may not change the dissertation format (e.g., reduce the font size, change the margins, or reduce the line spacing from the default single spacing). Be careful if you copy-paste packages into your document preamble from elsewhere. Some LaTeX packages, such as `fullpage` or `savetrees`, change the margins of your document. Do not include them!

Over-length or incorrectly-formatted dissertations will not be accepted and you would have to modify your dissertation and resubmit. You cannot assume we will check your submission before the final deadline and if it requires resubmission after the deadline to conform to the page and style requirements you will be subject to the usual late penalties based on your final submission time.

Bibliography

[1] Andrej Bauer and Matija Pretnar. Programming with Algebraic Effects and Handlers, March 2012. arXiv:1203.1539.

[2] Andrej Bauer and Matija Pretnar. An Effect System for Algebraic Effects and Handlers. In Reiko Heckel and Stefan Milius, editors, *Algebra and Coalgebra in Computer Science*, pages 1–16, Berlin, Heidelberg, 2013. Springer.

[3] Jonathan Immanuel Brachthäuser and Philipp Schuster. Effekt: extensible algebraic effects in Scala (short paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, pages 67–72, Vancouver BC Canada, October 2017. ACM.

[4] Daniel Hillerström. *Foundations for Programming and Implementing Effect Handlers*. PhD thesis, The University of Edinburgh, UK, 2022.

[5] Daniel Hillerström and Sam Lindley. Shallow Effect Handlers. In Sukyoung Ryu, editor, *Programming Languages and Systems*, volume 11275, pages 415–435. Springer International Publishing, Cham, 2018. Series Title: Lecture Notes in Computer Science.

[6] Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, pages 59–70, Boston Massachusetts USA, September 2013. ACM.

[7] Daan Leijen. Koka: Programming with Row Polymorphic Effect Types, June 2014. arXiv:1406.2061.

[8] Gordon D Plotkin and Matija Pretnar. Handling Algebraic Effects. *Logical Methods in Computer Science*, Volume 9, Issue 4:705, December 2013.

[9] Matija Pretnar. An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, 319:19–35, December 2015.

[10] KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. Retrofitting effect handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 206–221, New York, NY, USA, June 2021. Association for Computing Machinery.

Appendix A

First appendix

A.1 First section

Any appendices, including any required ethics information, should be included after the references.

Markers do not have to consider appendices. Make sure that your contributions are made clear in the main body of the dissertation (within the page limit).

Appendix B

Participants' information sheet

If you had human participants, include key information that they were given in an appendix, and point to it from the ethics declaration.

Appendix C

Participants' consent form

If you had human participants, include information about how consent was gathered in an appendix, and point to it from the ethics declaration. This information is often a copy of a consent form.