Composing UNIX with Effect Handlers (Extended Abstract)

A Case Study in Effect Handler Oriented Programming

Daniel Hillerström
Laboratory for Foundations of Computer Science
The University of Edinburgh, Scotland, UK
daniel.hillerstrom@ed.ac.uk

ABSTRACT

Effect handler oriented programming (EHOP) is a paradigm in which programs are syntax whose semantics are compartmentalised into a collection of effect handlers. The separation of syntax and semantics provides a modular basis for building software, where programs can be retrofitted with more functionality in a backwards compatible way. My talk proposal is about demonstrating EHOP in practice by implementing a tiny UNIX-style operating system.

NOTICE

If you wish to cite this work, I ask that you cite my *PhD thesis* as it is, at the time of writing, the authoritative source on this material.

1 INTRODUCTION

Plotkin and Pretnar's effect handlers [12] provide a versatile abstraction for programming with Plotkin and Power-style computational effects [11], where programs are written with respect to an abstract interface of effect operations they expect to be able to perform in their environment. Programs can be run in any environment that supplies a conforming implementation of this interface. The implementation of this interface is supplied by a user-implementable effect handler. A compelling trait of effect handlers is that multiple handlers can cooperate through seamless composition to implement an effect interface. The ability to seamlessly compose is the key enabler for *effect handler oriented programming*, where implementations of effect interfaces are decomposed into several fine-grained effect handlers, which can be combined in various ways to instantiate the behaviour of programs.

With the imminent arrival of effect handlers in OCaml, effect handler oriented programming will become available to ML programmers [15]. As an illustrative example of effect handler oriented programming I will demonstrate how to implement the essence of Ritchie and Thompson's UNIX operating system by starting from a tiny kernel and incrementally retrofit it with additional functionality by composing ever more effect handlers [14]. Along the way I will discuss the advantages of effect handler oriented programming and point to avenues for further research to address its limitations.

2 THE EFFECTIVE ESSENCE OF UNIX

The key components of Ritchie and Thompson's UNIX include multi-tasking with time sharing of computing resources, multiple user environments, a file system, and a programmable I/O interface [14]. In this section I will briefly describe each of the aforementioned components can be realised in terms of standard textbook effects. For the sake of brevity I will leave some of the more intricate details for the presentation accompanying this extended abstract (nonetheless, I will provide some pointers to more detailed explanations for the impatient reader).

The key to modelling an UNIX-style operating system with effect handlers is to view *system calls as operations of an effect interface*. In addition, we will view computations as processes.

2.1 Exceptions: Process Termination

A process may terminate successfully by running to completion, or it may terminate with success or failure in the middle of some computation by performing an *exit* system call. The exit system call is parameterised by an integer value intended to indicate whether the exit was due to success or failure. By convention, UNIX interprets the integer zero as success and any nonzero integer as failure. We can model the exit system call as a single operation Exit.

$$\mathsf{Status} = \{\mathsf{Exit} : \mathsf{Int} \twoheadrightarrow \forall\alpha.\alpha\}$$

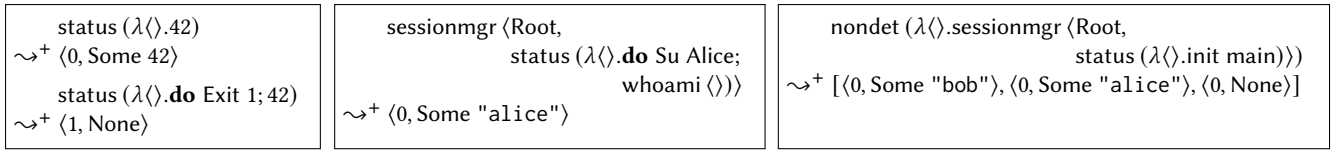
The operation is parameterised by an integer value, however, an invocation of Exit can never return, because its return type is uninhabited. Thus invoking Exit is like throwing an exception in, say, vanilla OCaml. An implementation of Exit is an exception handler.

$$\begin{aligned}&\mathsf{status} : (1 \to \alpha!\mathsf{Status}) \to \langle \mathsf{Int}, \mathsf{Option}\ \alpha\rangle\\ &\mathsf{status}\ m = \ \mathbf{handle}\ m\,\langle\rangle\ \mathbf{with}\\ &\qquad\qquad \mathbf{return}\ x \mapsto \langle 0, \mathsf{Some}\ x\rangle\\ &\qquad\qquad \langle\!|\mathsf{Exit}\ n|\!\rangle \ \mapsto \langle n, \mathsf{None}\rangle\end{aligned}$$

The status function takes as input a thunk which may perform the Exit operation whilst computing a value of type α. Ultimately, status returns a pair consisting of an integer and an optional α. This is a Benton and Kennedy [2] style handler with a **return**-case which tags the return value x with Some and pairs it with the integer zero. The operation case returns the payload n of Exit paired with None. Figure 1a depicts two example usages of status, where the first process terminates successfully, whilst the second process terminates prematurely with exit code 1.

2.2 Dynamic Binding: Environments

When a process is run in UNIX, the operating system makes available to the process a collection of name-value pairs called the *environment*. The name of a name-value pair is known as an *environment variable*. During execution the process may perform a system call to ask the operating system for the value of some environment variable. The value of some environment variables may vary according to which user enquires the environment. For example, an environment may contain the environment variable `USER` that is bound to the name of the enquiring user. Thus we may view environment variables as dynamically bound variables, whose values depend on the dynamic context rather than the lexical context.



(a) Process Termination **(b) Substitute User** **(c) Multiple Processes with Distinct Environments**

Figure 1: Examples with Exceptions, Dynamic Binding, and Nondeterminism

We will make a gross simplification as our environments will contain only the USER environment variable. The interface for environments consists of a single operation.

$$\mathsf{Env} = \{\mathsf{Ask} : 1 \twoheadrightarrow \mathsf{String}\}$$

The intended behaviour of Ask is to return the name of the enquiring user. Thus using this operation we can readily implement the *whoami* utility from the GNU coreutils [10, Section 20.3].

$$\begin{aligned}&\mathsf{whoami} : 1 \longrightarrow \mathsf{String}!\mathsf{Env}\\ &\mathsf{whoami}\ \langle\rangle = \mathbf{do}\ \mathsf{Ask}\ \langle\rangle\end{aligned}$$

For simplicity we fix the users of the operating system to be root, Alice, and Bob, i.e. User = Alice | Bob | Root. The following handler provides an implementation of the Env interface by making use of delimited continuations.

$$\begin{aligned}&\mathsf{env} : \langle \mathsf{User}, 1 \longrightarrow \alpha!\mathsf{Env}\rangle \longrightarrow \alpha\\ &\mathsf{env}\ \langle \mathit{user}, m\rangle = \mathbf{handle}\ m\ \langle\rangle\ \mathbf{with}\\ &\qquad \mathbf{return}\ \mathit{res} \mapsto \mathit{res}\\ &\qquad \langle\!|\mathsf{Ask}\ \langle\rangle \twoheadrightarrow \mathit{resume}|\!\rangle \mapsto\\ &\qquad\quad \mathbf{case}\ \mathit{user}\ \{\mathsf{Alice} \mapsto \mathit{resume}\ \texttt{"alice"}\\ &\qquad\qquad\qquad \mathsf{Bob} \mapsto \mathit{resume}\ \texttt{"bob"}\\ &\qquad\qquad\qquad \mathsf{Root} \mapsto \mathit{resume}\ \texttt{"root"}\}\end{aligned}$$

The handler takes as input the current *user* and a process that may perform the Ask operation. The operation case for Ask expose an additional parameter *resume* which is the current continuation of Ask in *m*. The extent of this continuation is delimited by the handler. The right hand side of the operation case pattern matches on the *user* parameter and invokes the continuation with a string representation of the user. The continuation invocation is implicitly guarded by the handler such that subsequent invocations of Ask get handled by the same handler. This type of handler is said to be *tail-resumptive*, because it invokes the continuation in tail-position [17]

2.2.1 Session management. It is somewhat pointless to have multiple user-specific environments, if the system does not support some mechanism for switching user session. In UNIX the command *substitute user* (su) enables the invoker to impersonate another user account. We will model su as a singleton effect interface.

$$\mathsf{SubUser} = \{\mathsf{Su} : \mathsf{User} \twoheadrightarrow 1\}$$

The operation Su is parameterised by the user to be impersonated.

The intended operational behaviour of an invocation of Su *user* is to load the environment belonging to *user* and continue the continuation under this environment. We can achieve this behaviour by defining a handler for Su that invokes the provided continuation under a fresh instance of the env handler.

$$\begin{aligned}&\mathsf{sessionmgr} : \langle \mathsf{User}; 1 \longrightarrow \alpha!\mathsf{SubUser} \otimes \mathsf{Env}\rangle \longrightarrow \alpha\\ &\mathsf{sessionmgr}\ \langle \mathit{user}; m\rangle\\ &\quad = \mathsf{env}\langle \mathit{user}; (\lambda\langle\rangle.\mathbf{handle}\ m\ \langle\rangle\ \mathbf{with}\\ &\qquad\qquad \mathbf{return}\ \mathit{res} \mapsto \mathit{res}\\ &\qquad\qquad \langle\!|\mathsf{Su}\ \mathit{user}' \twoheadrightarrow \mathit{resume}|\!\rangle \mapsto \mathsf{env}\langle \mathit{user}'; \mathit{resume}\rangle)\rangle\end{aligned}$$

The function sessionmgr manages a user session. It takes two arguments: the initial user (*user*) and a suspended process (*m*) to run in the current session. The suspended process may perform operations from both SubUser and Env interfaces. An initial instance of env is installed with *user* as argument along with a computation whose body is a handler for Su enclosing the process *m*. The Su-case installs a new instance of env, which is the environment belonging to *user*′, and runs the continuation *resume* under this instance. The new instance of env shadows the initial instance, meaning it will intercept and handle residual occurrences of Ask in the continuation. Each subsequent invocation of Su will install another environment instance, which will shadow the previous environment. Figure 1b shows an example of switching user session.

2.3 Nondeterminism: Process Duplication

The process duplication primitive in UNIX is called *fork* [14]. The fork-invoking process is typically referred to as the parent process, whilst its clone is referred to as the child process. Following an invocation of fork, the parent process is provided with a nonzero identifier for the child process and the child process is provided with the zero identifier. This enables processes to determine their respective role in the parent-child relationship, e.g.

$$\begin{aligned}&\mathbf{let}\ i \leftarrow \mathit{fork}\ \langle\rangle\ \mathbf{in}\\ &\mathbf{if}\ i = 0\ \mathbf{then}\ \mathit{child's\ code}\\ &\mathbf{else}\ \mathit{parent's\ code}\end{aligned}$$

We will make another simplification here as we will model fork as an effect operation that returns a boolean value to indicate the process role; by convention we will interpret the return value true to mean that the process assumes the role of parent.

$$\mathsf{ProcDup} = \{\mathsf{Fork} : 1 \twoheadrightarrow \mathsf{Bool}\}$$

In UNIX the parent process *continues* execution after the fork point, and the child process *begins* its execution after the fork point. Operationally, we may understand fork as returning twice to its invocation site. We can implement this behaviour by invoking the continuation of Fork twice: first with true to continue the parent

process, and subsequently with false to start the child process.

$$\begin{aligned}&\mathsf{nondet} : (1 \longrightarrow \alpha!\mathsf{ProcDup}) \longrightarrow \mathsf{List}\ \alpha\\&\mathsf{nondet}\ m = \mathbf{handle}\ m\ \langle\rangle\ \mathbf{with}\\&\qquad \mathbf{return}\ \mathit{res} \qquad\qquad \mapsto [\mathit{res}]\\&\qquad \langle \mathsf{Fork}\ \langle\rangle \twoheadrightarrow \mathit{resume}\rangle \mapsto \mathit{resume}\ \mathsf{true} +\!\!+ \mathit{resume}\ \mathsf{false}\end{aligned}$$

The function nondet accepts a suspended process that may duplicate itself as input. The function ultimately returns a list containing the result of all parent and child processes. The **return**-case returns a singleton list containing a result of running m. The Fork-case invokes the provided continuation *resume* twice. Each invocation of *resume* effectively copies m and runs each copy to completion. Each copy returns through the **return**-case, hence each invocation of *resume* returns a list of the possible results obtained by interpreting Fork first as true and subsequently as false. The results are joined by list concatenation ($+\!\!+$). Thus the handler returns a list of all the possible results of m.

We can implement something akin to the UNIX *init* process [14].

$$\begin{aligned}&\mathsf{init} : (1 \longrightarrow \alpha) \longrightarrow 1!\mathsf{ProcDup} \otimes \mathsf{Status}\\&\mathsf{init}\ m = \mathbf{if\ do}\ \mathsf{Fork}\ \langle\rangle\ \mathbf{then\ do}\ \mathsf{Exit}\ 0\\&\qquad\qquad \mathbf{else}\ m\ \langle\rangle\end{aligned}$$

The init process uses the ProcDup and Status effects, and whatever effects its parameter m uses (effect polymorphism is implicit as in Frank [9]). This implementation does not faithfully replicate the behaviour of the init process as it may terminate before its child processes. For a more accurate implementation of init we require the full generality of UNIX fork (e.g. see [6, Section 5.4.3]). Nevertheless, we can implement a main process that utilises the interfaces we have defined thus far.

$$\begin{aligned}&\mathsf{main} : 1 \longrightarrow \mathsf{String}!\mathsf{ProcDup} \otimes \mathsf{SubUser} \otimes \mathsf{Env}\\&\mathsf{main}\ \langle\rangle = \mathbf{if\ do}\ \mathsf{Fork}\ \langle\rangle\ \mathbf{then\ do}\ \mathsf{Su\ Alice}; \mathsf{whoami}\ \langle\rangle\\&\qquad\qquad \mathbf{else\ do}\ \mathsf{Su\ Bob}; \mathsf{whoami}\ \langle\rangle\end{aligned}$$

Figure 1c depicts a possible instantiation of main.

2.3.1 Time-sharing. Whilst the nondet handler gives us the ability to create multiple processes, it does not give us any way to interleave computation. In order to implement time sharing of computing resources we need two things: 1) some interruption mechanism that injects instances of an operation $\mathsf{Interrupt} : 1 \twoheadrightarrow 1$ to preempt a running process, and 2) a process scheduler that decides which process to run next. There are multiple ways to inject operations. If we have complete control of the runtime, then we can use the model of Ahman and Pretnar [1] and have some external mechanism inject operations. Alternatively, we can inject operations ourselves by bundling them alongside other operation invocations (this akin to Dybvig and Hieb [3], who hide interruptions under λ-abstractions via macro-expansion). The latter approach requires some care, because we risk Interrupt annotations pervading all contexts if we inject operations at the invocation site of other operations. A better approach is to install an intermediate handler that intercepts and reperforms operations, but before reperforming an operation it performs an instance of Interrupt. I will leave the finer details of how to implement time-sharing for the talk. For an in-depth discussion of the aforementioned approaches see [6, Section 5.2.4] and for an implementation of a basic time-sharing system (supporting the full generality of UNIX fork) see [6, Section 5.4.3].

2.4 Global State: File System

The file system is an integral part of UNIX. We can model it as a particular instance of the well-known global mutable state effect.

$$\mathsf{FileSys} = \{\mathsf{Get} : 1 \twoheadrightarrow \mathsf{FileSystem}, \mathsf{Put} : \mathsf{FileSystem} \twoheadrightarrow 1\}$$

An implementation of this interface is the standard state-passing handler [12]. I will leave the structure of FileSystem abstract.

2.4.1 File I/O. To manipulate the file system we provide two interfaces for opening/closing and reading/writing files, respectively.

$$\begin{aligned}\mathsf{COC} &= \{\mathsf{Create} : \mathsf{String} \twoheadrightarrow \mathsf{Int}, \mathsf{Open} : \mathsf{String} \twoheadrightarrow \mathsf{FD},\\&\qquad \mathsf{Close} : \mathsf{FD} \twoheadrightarrow 1\}\\\mathsf{RW} &= \{\mathsf{Read} : \langle \mathsf{FD}, \mathsf{Int}, \mathsf{Int}\rangle \twoheadrightarrow \mathsf{Option\ String},\\&\qquad \mathsf{Write} : \langle \mathsf{FD}, \mathsf{String}\rangle \twoheadrightarrow 1\}\end{aligned}$$

Like UNIX, we are going to throw caution to wind and model file descriptors (FD) as raw pointers into the global FileSystem structure. Implementing these interfaces is not too hard it just involves low-level manipulations of the FileSystem structure (see [6, Section 5.2.5] for an implementation of a basic file system).

2.4.2 I/O Redirection. The file I/O redirection operators > and >> can readily be implemented using Create, Open, and Write.

2.5 Streams: Programmable I/O

UNIX pipes and filters make up an important part of the UNIX programming experience [13, 14]. In previous work, it has already been shown how to implement these facilities using *shallow handlers* [4, 5, 7, 9]. Shallow handlers are an alternative to Plotkin and Pretnar's *deep* handlers. The key difference is that a shallow handler does not wrap the handler around the continuation, which makes it easy to implement a pair of dual handlers that simulate a data stream by alternating between *producing* and *consuming* values.

3 A HANDLER SEMANTICS FOR UNIX

Plotkin and Power [11] attach equational theories to effect interfaces, which govern the behaviour of operations. As with most practical implementations of effect handlers, I am working in the free theory. I have made no effort to prove the full implementation of the system correct with respect to an equational specification of UNIX. The complete system is composed from 12+ handlers, where the '+' covers the handlers used internally by user processes. However, in the preceding section I have alluded to the fact that many of the handlers are instances of standard handlers that are known to respect the equational theories of the effects they implement. Thus I conjecture that it is within the realm of immediate possibilities to give an equational specification of the core components of UNIX and prove the system correct with respect to this specification.

4 RELATED WORK

The idea of using continuations to implement various facets of operating systems is not new. For example, Wand [16] implements a small multi-tasking kernel with support for mutual exclusion and data protection using callcc-style continuations. Kiselyov and

Shan [8] use delimited continuations to implement a tiny operating system with multi-tasking support and a sophisticated transactional file system that makes it possible for user processes to roll back actions such as file deletion and file update.

ACKNOWLEDGMENTS

Thanks to James McKinna, Gordon Plotkin, Sam Lindley, and Philip Wadler for early comments and insightful discussions about this work. This work was supported by the UKRI Future Leaders Fellowship "Effect Handler Oriented Programming" (reference number MR/T043830/1).

REFERENCES

[1] Danel Ahman and Matija Pretnar. 2021. Asynchronous effects. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28.
[2] Nick Benton and Andrew Kennedy. 2001. Exceptional Syntax Journal of Functional Programming. *J. Funct. Program.* 11, 4 (2001), 395–410.
[3] R. Kent Dybvig and Robert Hieb. 1989. Engines From Continuations. *Comput. Lang.* 14, 2 (1989), 109–123.
[4] Daniel Hillerström and Sam Lindley. 2018. Shallow Effect Handlers. In *APLAS (LNCS, Vol. 11275)*. Springer, 415–435.
[5] Daniel Hillerström, Sam Lindley, and Robert Atkey. 2020. Effect handlers via generalised continuations. *J. Funct. Program.* 30 (2020), e5.
[6] Daniel Hillerström. 2021. *Foundations for Programming and Implementing Effect Handlers.* Ph.D. Dissertation. The University of Edinburgh, Scotland, UK.
[7] Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ICFP*. ACM, 145–158.
[8] Oleg Kiselyov and Chung-chieh Shan. 2007. Delimited Continuations in Operating Systems. In *CONTEXT (LNCS, Vol. 4635)*. Springer, 291–302.
[9] Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *POPL*. ACM, 500–514.
[10] David MacKenzie et al. 2020. *GNU Coreutils.* Free Software Foundation. For version 8.32.
[11] Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003), 69–94.
[12] Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013).
[13] Eric Steven Raymond. 2003. *The Art of UNIX Programming.* Pearson Education.
[14] Dennis Ritchie and Ken Thompson. 1974. The UNIX Time-Sharing System. *Commun. ACM* 17, 7 (1974), 365–375.
[15] KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *PLDI*. ACM, 206–221.
[16] Mitchell Wand. 1980. Continuation-Based Multiprocessing. In *LISP Conference*. ACM, 19–28.
[17] Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. 2020. Effect handlers, evidently. *Proc. ACM Program. Lang.* 4, ICFP (2020), 99:1–99:29.