Handlers in Action

Ohad Kammar
University of Cambridge
ohad.kammar@cl.cam.ac.uk

Sam Lindley
University of Strathclyde
Sam.Lindley@ed.ac.uk

Nicolas Oury
nicolas.oury@gmail.com

Abstract

Plotkin and Pretnar's handlers for algebraic effects occupy a sweet spot in the design space of abstractions for effectful computation. By separating effect signatures from their implementation, algebraic effects provide a high degree of modularity, allowing programmers to express effectful programs independently of the concrete interpretation of their effects. A handler is an interpretation of the effects of an algebraic computation. The handler abstraction adapts well to multiple settings: pure or impure, strict or lazy, static types or dynamic types.

This is a position paper whose main aim is to popularise the handler abstraction. We give a gentle introduction to its use, a collection of illustrative examples, and a straightforward operational semantics. We describe our Haskell implementation of handlers in detail, outline the ideas behind our OCaml, SML, and Racket implementations, and present experimental results comparing handlers with existing code.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]; D.3.1 [*Formal Definitions and Theory*]; D.3.2 [*Language Classifications*]: Applicative (functional) languages; D.3.3 [*Language Constructs and Features*]; F.3.2 [*Semantics of Programming Languages*]: Operational semantics

Keywords algebraic effects; effect handlers; effect typing; monads; continuations; Haskell; modularity

1. Introduction

Monads have proven remarkably successful as a tool for abstraction over effectful computations [4, 30, 46]. However, monads as a programming language primitive violate the fundamental encapsulation principle: program to an *interface*, not to an *implementation*.

Modular programs are constructed using abstract interfaces as building blocks. This is *modular abstraction*. To give meaning to an *abstract* interface, we instantiate it with a *concrete* implementation. Given a composite interface, each sub-interface can be independently instantiated with different concrete implementations. This is *modular instantiation*.

The monadic approach to functional programming takes a *concrete* implementation rather than an *abstract* interface as primitive. For instance, in Haskell we might define a state monad:

$$\begin{array}{l}
\mathbf{newtype}\ \mathit{State}\ s\ a = \mathit{State}\ \{\mathit{runState} :: s \to (a, s)\} \\
\mathbf{instance}\ \mathit{Monad}\ (\mathit{State}\ s)\ \mathbf{where} \\
\quad \mathit{return}\ x = \mathit{State}\ (\lambda s \to (x, s)) \\
\quad m \gg\!= f\ = \mathit{State}\ (\lambda s \to \mathbf{let}\ (x, s') = \mathit{runState}\ m\ s\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad \mathit{runState}\ (f\ x)\ s')
\end{array}$$

This definition says nothing about the intended use of $\mathit{State}\ s\ a$ as the type of computations that read and write state. Worse, it breaks abstraction as consumers of state are exposed to its concrete implementation as a function of type $s \to (a, s)$. We can of course define the natural get and put operations on state, but their implementations are fixed.

Jones [18] advocates modular abstraction for monads in Haskell using type classes. For instance, we can define the following interface to abstract state computation[1]:

$$\begin{array}{l}
\mathbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadState}\ s\ m \mid m \to s\ \mathbf{where} \\
\quad \mathit{get} :: \qquad m\ s \\
\quad \mathit{put} :: s \to m\ ()
\end{array}$$

The $\mathit{MonadState}$ interface can be smoothly combined with other interfaces, taking advantage of Haskell's type class mechanism to represent type-level sets of effects.

Monad transformers [25] provide a form of modular instantiation for abstract monadic computations. For instance, state can be handled in the presence of other effects by incorporating a state monad transformer within a monad transformer stack.

A fundamental problem with monad transformer stacks is that once a particular abstract effect is instantiated, the order of effects in the stack becomes concrete, and it becomes necessary to explicitly lift operations through the stack. Taming the monad transformer stack is an active research area [16, 17, 38, 42].

Instead of the top-down monad transformer approach, we take a bottom-up approach, simply adding the required features as language primitives. We want modular abstraction, so we add abstract effect interfaces, in fact *abstract operations*, as a language primitive. Abstract operations compose, yielding modular abstraction. We also want modular instantiation, so we add *effect handlers* as a language primitive for instantiating an abstract operation with a concrete implementation. A handler operates on a specified subset of the abstract operations performed by an abstract computation, leaving the remainder abstract, and yielding modular instantiation.

By directly adding the features we require, we obtain modular abstraction and modular instantiation while avoiding many of the pitfalls of monad transformers.

Our first inspiration is the *algebraic theory of computational effects*. Introduced by Plotkin and Power [33–35], it complements Moggi's monadic account of effects by incorporating abstract effect interfaces as primitive. Our second inspiration is the elimination construct for algebraic effects, *effect handlers* [36]. In Plotkin and Power's setting, one defines algebraic effects with respect to an equational theory. As with other handler implementations [2, 6, 29],

[1] From the Monad Transformer Library [12].

ICFP '13, September 25–27, 2013, Boston, MA, USA.

ACM 978-1-4503-2326-0/13/09…$15.00.
http://dx.doi.org/10.1145/2500365.2500590

in this paper we always take the equational theory to be the free theory, in which there are no equations.

We argue that algebraic effects and effect handlers provide a compelling alternative to monads as a basis for effectful programming across a variety of functional programming languages (pure or impure, strict or lazy, statically typed or dynamically typed). Our position is supported by a range of handler libraries we have implemented for Haskell, OCaml, SML, and Racket, backed up by a core calculus of effect handlers with a straightforward operational semantics. This paper focuses on the Haskell library.

Our key contributions are the following:

- A collection of novel features for practical programming with handlers in the presence of effect typing, illustrated through a series of examples.
- A small-step operational semantics for effect handlers.
- A type and effect system for effect handlers.
- Effect handler libraries for Haskell, OCaml, SML, and Racket.
- A performance comparison between our Haskell library and equivalent non-handler code.

The rest of the paper is structured as follows. Section 2 presents handlers in action through a series of small examples. Section 3 introduces our core calculus of effect handlers, λ_{eff}, its effect type system, and small-step operational semantics. Section 4 presents in detail the Haskell effect handlers library, and sketches the design of our libraries for OCaml, SML, and Racket. Section 5 reports on the baseline performance of handlers in comparison with existing (less abstract) code. Section 6 discusses related work, and Section 7 concludes.

2. Handlers in Action

We present our examples in a monadic style, using an extension to Haskell syntax implemented using the Template Haskell [40] and quasiquote [28] features of GHC. Haskell's features allow for a relatively simple, user-friendly and efficient implementation of effects handlers with effect typing in an existing language. (None of our other libraries support effect typing.)

However, all of the examples could in fact be implemented with the same level of expressivity and flexibility in a direct-style, without any monadic boilerplate. Indeed, handlers can be direct-style with or without effect typing, and provide a natural abstraction for adding more controlled effectful computations to the ML family of languages or even to the Lisp family of languages as witnessed by the libraries we have implemented in OCaml, SML, and Racket. In Section 4.3 we outline a practical method to prototype an implementation of handlers for a large set of such languages.

A key reason for focusing on our Haskell library in this paper is that it is the only one that supports effect typing. Effect typing is crucial to the termination of our core calculus (Section 3). More importantly, it it is crucial for our notion of soundness, as it statically ensures that operations do not accidentally go unhandled. Anecdotally, we have observed that such static guarantees are important in practice: when we added effect typing to an early version of our Haskell library, it revealed a number of bugs in our test examples.

The code for our effect handler libraries, examples, and benchmarks is available in the GitHub repository at:

```
http://github.com/slindley/effect-handlers/
```

This repository also includes many other other examples. For instance, we have reimplemented Kiselyov and Shan's HANSEI DSL for probabilistic computation [22], Kiselyov's iteratees [21], and Gonzalez's Pipes library [14], all using handlers.

Template Haskell and Quasiquotes Template Haskell [40] is a facility for compile-time meta programming in Haskell. It provides a data type for manipulating Haskell code as an abstract syntax tree. It also provides constructs for splicing chunks of Haskell together.

Quasiquotes [28] extend Template Haskell splicing functionality with user-defined syntax. Each syntax extension (also known as *quasiquoter*) is associated with a name. To define syntax extension, **ext**, a library writer supplies a parser for **ext** expressions as a function that takes a string and returns a Haskell abstract syntax tree. To invoke the syntax extension, a programmer writes an expression e in quasiquote brackets $[\mathbf{ext} \mid e \mid]$, a *quasiquote*. At compile-time, GHC runs the parser on e and splices in the resulting abstract syntax tree in place of the quasiquote brackets.

2.1 State and Handlers

We introduce the primitives for abstract operations and handlers in our Haskell library through the example of global state.

We define abstract operations for state with the following **operation** quasiquotes

$$\begin{array}{l}[\mathbf{operation} \mid \mathit{Get}\ s :: \quad s \ \mid] \\ [\mathbf{operation} \mid \mathit{Put}\ s :: s \to () \mid]\end{array}$$

which declare a $\mathit{Get}\ \ s$ operation that takes no parameters and returns values of type s, and a $\mathit{Put}\ s$ operation that takes a single parameter of type s and returns values of type $()$. In general an abstract operation declaration has the form

$$[\mathbf{operation} \mid \forall u_1 \ldots u_l . \mathit{Op}\ e_1 \ldots e_m :: A_1 \to \ldots \to A_n \to A \mid]$$

where Op is the name of the operation, $u_1, \ldots, u_l$ are *universal* type variables, $e_1, \ldots, e_m$ are *existential* type variables, $A_1, \ldots, A_n$ are parameter types, and A is the return type. (We will discuss the role of universal and existential type variables in Section 2.2.)

The declarations above automatically derive wrappers get and put for actually invoking the operations. Ideally, we would like their types to be

$$\begin{array}{l}\mathit{get} :: \qquad \mathit{Comp}'\ \{\mathit{Get}\ s\}\ s \\ \mathit{put} :: s \to \mathit{Comp}'\ \{\mathit{Put}\ s\}\ ()\end{array}$$

where $\mathit{Comp}'\ \ \varepsilon\ \ a$ is the type of abstract computations that can perform abstract operations in the set ε and return a value of type a. But GHC does not have a built-in effect type-system, so we simulate one, encoding sets of operations as type class constraints. Thus, get and put actually have slightly more complicated types:

$$\begin{array}{l}\mathit{get} :: [\mathbf{handles} \mid h\ \{\mathit{Get}\ s\} \mid] \Rightarrow \qquad \mathit{Comp}\ h\ s \\ \mathit{put} :: [\mathbf{handles} \mid h\ \{\mathit{Put}\ s\} \mid] \Rightarrow s \to \mathit{Comp}\ h\ ()\end{array}$$

We can think of the type variable h as standing for a set of operations, an *effect set* ε. Membership of operation Op in ε is denoted by the quasiquotation $[\mid \mathbf{handles} \mid h\ \{\mathit{Op}\} \mid]$ (which is desugared into a corresponding type class constraint). The reason we write **handles** instead of **contains**, and h instead of ε, is that h is more than just an effect set; it actually ranges over *handlers* for ε. Thus $\mathit{Comp}\ h$ represents computations interpreted by handlers of type h, and the constraint $[\mid \mathbf{handles} \mid h\ \{\mathit{Op}\} \mid]$ really means: handler h handles operation Op. (We will introduce the syntax for handlers shortly.)

We define the type of abstract state computations as follows[2]:

$$\begin{array}{l}\mathbf{type}\ \mathit{SComp}\ s\ a = \\ \quad \forall h.([\mathbf{handles} \mid h\ \{\mathit{Get}\ s\} \mid], \\ \qquad [\mathbf{handles} \mid h\ \{\mathit{Put}\ s\} \mid]) \Rightarrow \mathit{Comp}\ h\ a\end{array}$$

For example:

[2] We would ideally like to write $[\mathbf{handles} \mid h\ \{\mathit{Get}\ \ s, \mathit{Put}\ \ s\} \mid]$ as a single constraint, but Template Haskell currently only allows us to generate one type class constraint per quasiquote.

```
comp :: SComp Int Int
comp = do {x ← get; put (x + 1);
           y ← get; put (y + y); get}
```

Because Haskell is lazy we still require notation for explicitly sequencing computations. We take advantage of the existing **do** notation, and $Comp\ h$ is implemented as a certain kind of universal monad (see Section 4).

We can provide many concrete interpretations of stateful computation, which is where handlers come in. First, we interpret state in the standard monadic way:

```
1 [handler |
2   RunState s a :: s → (a, s)
3     handles {Get s, Put s} where
4       Return x    s → (x, s)
5       Get       k s → k s s
6       Put     s k _ → k () s |]
```

We describe the syntax line by line.

Line 1 begins a **handler** quasiquote.

Line 2 specifies the name, type parameters, and type signature of the handler. Notice that the type signature $s \to (a, s)$ is that of the state monad. The type signature indicates that this handler takes one parameter of type s, which is threaded through the handler, and returns a result of type (a, s).

Line 3 specifies the set of operations handled by the handler.

Line 4 is a *return clause*. It expresses how to return a final value from a computation. In general, a return clause takes the form $Return\ \ x\ \ y_1\ ...\ y_n\ \to\ e$, where x binds the value returned by the computation and $y_1, \ldots, y_n$ bind the handler parameters. Here, the final value is paired up with the single state parameter.

Lines 5 and 6 are *operation clauses*. They express how to handle each operation. In general, an operation clause takes the form $Op\ \ x_1\ ...\ x_m\ \ k\ \ y_1\ ...\ y_n\ \to\ e$, where $x_1, \ldots, x_m$ bind the operation parameters, k binds the continuation of the computation, and $y_1, \ldots, y_n$ bind the handler parameters, here the single state parameter. The continuation k is a curried function which takes a return value followed by a sequence of handler parameters, and yields the interpretation of the rest of the computation. For Get, the return value is the current state, which is threaded through the rest of the computation. For $Put\ s$, the existing state is ignored, the return value is $()$, and the state parameter is updated to s.

Analogously to abstract operation declarations, a handler declaration generates a convenient wrapper, whose name is derived from that of the handler by replacing the first letter with its lower case counterpart.

```
runState :: s → SComp s a → (a, s)
*Main> runState 1 comp
(4, 4)
```

If we do not need to read the final contents of the state, then we can give a simpler interpretation to state [36], using the type that a Haskell programmer might normally associate with a read-only state monad:

```
[handler |
  EvalState s a :: s → a handles {Get s, Put s} where
    Return x    s → x
    Get       k s → k s s
    Put     s k _ → k () s |]

*Main> evalState 1 comp
4
```

More interestingly, we can give other interpretations:

```
[handler |
  LogState s a :: s → (a, [s])
    handles {Get s, Put s} where
      Return x    s → (x, [])
      Get       k s → k s s
      Put     s k _ → let (x, ss) = k () s in (x, s : ss) |]
```

This handler logs the history of all writes to the state. For instance,

```
*Main> logState 1 comp
(4, [2, 4])
```

2.2 State and Open Handlers

The three handlers of Section 2.1 are *closed*. They each handle Get and Put, but cannot interpret computations that might perform other operations. Thus they do not support modular instantiation.

Open handlers extend closed handlers by automatically forwarding all operations that are not explicitly handled. For instance, the following defines a handler that forwards all operations other than Get and Put:

```
[handler |
  forward h.
    OpenState s a :: s → a handles {Get s, Put s} where
      Return x    s → return x
      Get       k s → k s s
      Put     s k _ → k () s |]
```

The type variable h is an artefact of the Haskell implementation. It represents an abstract *parent handler* that will ultimately handle operations forwarded by $OpenState$. It is implicitly added as the first type argument to $OpenState$ (yielding $OpenState\ \ h\ \ s\ \ a$) and $Comp\ \ h$ is implicitly applied to the return type (yielding $Comp\ \ h\ \ a$). Any operations other than Get or Put will be automatically forwarded to h.

To illustrate the composability of open handlers, we return to the logging example. In Section 2.1, we demonstrated how to log Put operations using a special handler. We now factor the logging in such a way that we can *refine* any abstract stateful computation into an equivalent abstract computation that also performs logging, such that both logging and state can be subsequently interpreted in arbitrary ways using suitable handlers.

First we define a new operation for logging each Put:

```
[operation | LogPut s :: s → () |]
```

Now we can define an open handler that inserts a $LogPut$ operation before every Put operation in the original computation, but otherwise leaves it unchanged:

```
[handler |
  forward h handles {Put s, LogPut s}.
    PutLogger s a :: a handles {Put s} where
      Return x    → return x
      Put     s k → do {logPut s; put s; k ()} |]
```

For instance, the computation $putLogger\ \ comp$ is equivalent to:

```
do {x ← get; logPut (x + 1); put (x + 1);
    y ← get; logPut (y + y); put (y + y); get}
```

The constraint $(h\ \ \textbf{handles}\ \ \{Put\ \ s, LogPut\ \ s\})$ asserts that the parent handler h must also handle the $Put\ \ s$ and $LogPut\ \ s$ operations[3].

To obtain the original behaviour of $LogState$, we can define the following open handler:

```
[handler |
  forward h.
    LogPutReturner s a :: (a, [s])
      handles {LogPut s} where
        Return x     → return (x, [])
        LogPut s k → do (x, ss) ← k (); return (x, s : ss) |]
```

[3] Under the hood this aids GHC type inference.

and compose several handlers together:

$$\begin{array}{l} \mathit{stateWithLog} :: s \to \mathit{SComp}\ s\ a \to (a, [s]) \\ \mathit{stateWithLog}\ s\ \mathit{comp} = (\mathit{handlePure} \circ \mathit{logPutReturner} \circ \\ \qquad\qquad\qquad\qquad \mathit{openState}\ s \circ \mathit{putLogger})\ \mathit{comp} \end{array}$$

where $\mathit{HandlePure}$ is a canonical top-level closed handler:

$$\begin{array}{l} [\mathbf{handler}\ | \\ \quad \mathit{HandlePure}\ a :: a\ \mathbf{handles}\ \{\,\}\ \mathbf{where}\ \mathit{Return}\ x \to x\ |] \end{array}$$

which interprets a pure computation as a value of type a.

An alternative interpretation of logging is to output logging messages as they arrive:

$$\begin{array}{l} [\mathbf{handler}\ | \\ \quad \mathbf{forward}\ h\ \mathbf{handles}\ \{\mathit{Io}\}.(\mathit{Show}\ s) \Rightarrow \\ \quad\quad \mathit{LogPutPrinter}\ s\ a :: a\ \mathbf{handles}\ \{\mathit{LogPut}\ s\}\ \mathbf{where} \\ \quad\quad\quad \mathit{Return}\ x \quad \to \mathit{return}\ x \\ \quad\quad\quad \mathit{LogPut}\ s\ k \to \\ \quad\quad\quad\quad \mathbf{do}\ \mathit{io}\ (\mathit{putStrLn}\ (\texttt{"Put: "} +\!\!+\ \mathit{show}\ s)); k\ ()\ |] \end{array}$$

Now we can plug everything together:

$$\begin{array}{l} \mathit{statePrintLog} :: \mathit{Show}\ s \Rightarrow s \to \mathit{SComp}\ s\ a \to \mathit{IO}\ a \\ \mathit{statePrintLog}\ s\ \mathit{comp} = (\mathit{handleIO} \quad \circ \mathit{logPutPrinter} \circ \\ \qquad\qquad\qquad\qquad \mathit{openState}\ s \circ \mathit{putLogger})\ \mathit{comp} \end{array}$$

where $\mathit{HandleIO}$ is another top-level closed handler for performing arbitrary operations in the IO monad with the Io operation:

$$\begin{array}{l} [\mathbf{operation}\ |\ \forall a.\mathit{Io} :: \mathit{IO}\ a \to a\ |] \\ [\mathbf{handler}\ | \\ \quad \mathit{HandleIO}\ a :: \mathit{IO}\ a\ \mathbf{handles}\ \{\mathit{Io}\}\ \mathbf{where} \\ \quad\quad \mathit{Return}\ x \quad \to \mathit{return}\ x \\ \quad\quad \mathit{Io}\ m \qquad k \to \mathbf{do}\ \{x \leftarrow m; k\ x\}\ |] \end{array}$$

The *universal* quantifier in the Io operation declaration indicates that it must be handled polymorphically in a. This is in contrast to the declaration of Get, for instance:

$$[\mathbf{operation}\ |\ \mathit{Get}\ s :: s\ |]$$

where the type parameter s is *existential*, in the sense that for any handler that handles Get, there must exist a fixed type for s. Correspondingly, in any given abstract computation, Io can be used at arbitrary types a whereas Get must be used at a fixed type s.

Comparing the outputs on our sample computation we obtain:

```
*Main> stateWithLog 1 comp
(4, [2, 4])
*Main> statePrintLog 1 comp
Put: 2
Put: 4
4
```

The pattern of precomposing one closed top-level handler with a sequence of open handlers is common when using our library. The order in which open handlers are composed may or may not change the semantics. For instance, if we were to swap the order of $\mathit{openState}\ s$ and $\mathit{putLogger}$ in $\mathit{statePrintLog}$ then all of the Put operations would be handled before any PutLog operations could be generated, so no logging information would ever be output. On the other hand, if we were to swap the order of $\mathit{logPutPrinter}$ and $\mathit{openState}\ s$ then the semantics would be unchanged as their actions are orthogonal.

Open handlers allow us to handle a subset of the effects in an abstract computation, thus supporting modular instantiation.

The next three subsections present more involved examples, demonstrating the interaction of user-defined effects with various Haskell features.

2.3 Choice and Failure

Consider abstract operations Choose and $\mathit{Failure}$:

$$\begin{array}{l} [\mathbf{operation}\ |\ \forall a.\mathit{Choose} :: a \to a \to a\ |] \\ [\mathbf{operation}\ |\ \forall a.\mathit{Failure} :: \qquad\quad a\ |] \end{array}$$

The idea is that Choose should select one of its two arguments of type a, and $\mathit{Failure}$ just aborts. Abstract choice computations are interesting because they admit a range of useful interpretations. Archetypal handlers which we consider include those that enumerate all choices ($\mathit{AllResults}$ below) and random sampling ($\mathit{RandomResult}$ below). The former is notable as it takes full advantage of the ability for an operation clause to invoke the continuation more than once.

The type of abstract computations over Choose and $\mathit{Failure}$ is:

$$\begin{array}{l} \mathbf{type}\ \mathit{CF}\ a = \forall h.([\mathbf{handles}\ |\ h\ \{\mathit{Choose}\}\ |], \\ \qquad\qquad\qquad [\mathbf{handles}\ |\ h\ \{\mathit{Failure}\}\ |]) \Rightarrow \mathit{Comp}\ h\ a \end{array}$$

As a simple example, consider the following program:

$$\begin{array}{l} \mathbf{data}\ \mathit{Toss} = \mathit{Heads}\ |\ \mathit{Tails}\ \mathbf{deriving}\ \mathit{Show} \\ \mathit{drunkToss} :: \mathit{CF}\ \mathit{Toss} \\ \mathit{drunkToss} = \mathbf{do}\ \{\mathit{caught} \leftarrow \mathit{choose}\ \mathit{True}\ \mathit{False}; \\ \qquad\qquad\qquad \mathbf{if}\ \mathit{caught}\ \mathbf{then}\ \mathit{choose}\ \mathit{Heads}\ \mathit{Tails} \\ \qquad\qquad\qquad \mathbf{else}\ \mathit{failure}\} \\ \mathit{drunkTosses} :: \mathit{Int} \to \mathit{CF}\ [\mathit{Toss}] \\ \mathit{drunkTosses}\ n = \mathit{replicateM}\ n\ \mathit{drunkToss} \end{array}$$

The abstract computation $\mathit{drunkToss}$ simulates a drunk performing one coin toss. If the drunk catches the coin then the result of tossing the coin (Heads or Tails) is returned. If the coin falls into the gutter then no result is returned. The $\mathit{drunkTosses}$ function repeats the process the specified number of times.

We can write a handler that returns all possible results of a CF computation as a list, providing a model of non-determinism.

$$\begin{array}{l} [\mathbf{handler}\ | \\ \quad \mathit{AllResults}\ a :: [a]\ \mathbf{handles}\ \{\mathit{Choose}, \mathit{Failure}\}\ \mathbf{where} \\ \quad\quad \mathit{Return}\ x \qquad \to [x] \\ \quad\quad \mathit{Choose}\ x\ y\ k \to k\ x +\!\!+\ k\ y \\ \quad\quad \mathit{Failure} \qquad k \to [\,]\ |] \end{array}$$

This is the first handler we have seen that uses the continuation non-linearly. The Choose operation is handled by concatenating the results of invoking the continuation with each alternative. The $\mathit{Failure}$ operation is handled by returning the empty list and ignoring the continuation. For example:

```
*Main> allResults (drunkCoins 2)
[[Heads, Heads], [Heads, Tails], [Tails, Heads], [Tails, Tails]]
```

Rather than returning all of the results of a CF computation, we might wish to sample a single result at random. In order to keep the implementation of randomness abstract, let us declare a new operation for generating random numbers.

$$[\mathbf{operation}\ |\ \mathit{Rand} :: \mathit{Double}\ |]$$

We first give a handler for Choose alone.

$$\begin{array}{l} [\mathbf{handler}\ | \\ \quad \mathbf{forward}\ h\ \mathbf{handles}\ \{\mathit{Rand}\}. \\ \quad\quad \mathit{RandomResult}\ a :: a\ \mathbf{handles}\ \{\mathit{Choose}\}\ \mathbf{where} \\ \quad\quad\quad \mathit{Return}\ x \qquad \to \mathit{return}\ x \\ \quad\quad\quad \mathit{Choose}\ x\ y\ k \to \mathbf{do}\ \{r \leftarrow \mathit{rand}; \\ \qquad\qquad\qquad\qquad\qquad k\ (\mathbf{if}\ r < 0.5\ \mathbf{then}\ x\ \mathbf{else}\ y)\}\ |] \end{array}$$

Unlike in the $\mathit{AllResults}$ handler, the Choose operation is handled by supplying one of the arguments to the continuation at random. We can implement randomness using the IO monad.

$$\begin{array}{l} [\mathbf{handler}\ | \\ \quad \mathit{HandleRandom}\ a :: \mathit{IO}\ a \\ \quad\quad \mathbf{handles}\ \{\mathit{Rand}\}\ \mathbf{where} \\ \quad\quad\quad \mathit{Return}\ x \qquad \to \mathit{return}\ x \\ \quad\quad\quad \mathit{Rand} \qquad\quad k \to \mathbf{do}\ \{r \leftarrow \mathit{getStdRandom}\ \mathit{random}; k\ r\}\ |] \end{array}$$

Let us now define another open handler for handling $\mathit{Failure}$, interpreting the result of a possibly failing computation with a Maybe type.

```
[handler |
  forward h.
    MaybeResult a :: Maybe a handles {Failure} where
      Return x → return (Just x)
      Failure k → return Nothing |]
```

As the body of the handler is pure, there is no need to constrain h with a **handles** clause. We now compose the above three handlers:

```
sampleMaybe :: CF a → IO (Maybe a)
sampleMaybe comp =
  (handleRandom ∘ maybeResult ∘ randomResult) comp
```

The *sampleMaybe* function[4] first uses *randomResult* to handle *Choose* using *Rand*, forwarding *Failure*. Then it uses *maybeResult* to handle *Failure*, forwarding *Rand*. Finally, at the top-level, it uses *handleRandom* to handle *Rand* in the *IO* monad. Here are some example runs:

```
*Main> sampleMaybe (drunkTosses 2)
Nothing
*Main> sampleMaybe (drunkTosses 2)
Just [Heads, Heads]
```

We might decide that rather than stopping on failure, we would like to persevere by trying again:

```
[handler |
  forward h.
    Persevere a :: Comp (Persevere h a) a → a
      handles {Failure} where
        Return x    _ → return x
        Failure    k c → persevere c c |]
```

The parameter to the *Persevere* handler is a computation that must be handled recursively by the handler itself. The *Failure* operation is handled by reinvoking the handler.

We can now persevere until we obtain a sample.

```
sample :: CF a → IO a
sample comp = handleRandom (persevere comp′ comp′)
  where comp′ = randomResult comp
```

For instance:

```
*Main> sample (drunkTosses 5)
[Heads, Tails, Heads, Tails, Heads]
```

In practice, one might use a more sophisticated sampling approach to improve performance by avoiding failures [22].

2.4 Word Count

The *wc* program counts the number of lines, words and characters in an input stream. We first present the abstract operations required to implement this functionality:

```
[operation | ReadChar :: Maybe Char |]
[operation | Finished  :: Bool       |]
```

The *ReadChar* operation reads a character if available. The *Finished* operation checks whether the input is finished. Given these operations, we can implement a function that reads a line:

```
readLine :: [handles | h {ReadChar} |] ⇒ Comp h String
readLine =
  do mc ← readChar
     case mc of
       Nothing   → return []
       Just '\n' → return []
       Just c    → do cs ← readLine; return (c : cs)
```

Of course, this implementation does not specify where the input is to be read from. We define a handler that reads from a string:

```
[handler |
  forward h.
    StringReader a :: String → a
      handles {ReadChar, Finished} where
        Return x     _        → return x
        ReadChar k []         → k Nothing []
        ReadChar k (c : cs)   → k (Just c) cs
        Finished   k []       → k True []
        Finished   k cs       → k False cs |]
```

and another that reads from standard input:

```
[handler |
  forward h handles {Io}.
    StdinReader a :: a
      handles {ReadChar, Finished} where
        Return x     → return x
        ReadChar k →
          do b ← io (hIsEOF stdin)
             if b then k Nothing
                  else do c ← io getChar; k (Just c)
        Finished   k → do b ← io (hIsEOF stdin); k b |]
```

With the *readLine* function, we can count the number of lines in an input stream, but *wc* additionally provides facilities to count characters and words. For doing so, we give two handlers, each of which instruments the *readChar* operation in a different way. The first handler counts the number of characters read by enumerating each call to *readChar*:

```
[handler |
  forward h handles {ReadChar}.
    CountChar0 a :: Int → (a, Int)
      handles {ReadChar} where
        Return x     i → return (x, i)
        ReadChar k i → do mc ← readChar
                          case mc of
                            Nothing → k mc i
                            Just _ → k mc $! i + 1 |]
countChar = countChar0 0
```

The second handler counts the number of words read by tracking space characters:

```
[handler |
  forward h handles {ReadChar}.
    CountWord0 a :: Int → Bool → (a, Int)
      handles {ReadChar} where
        Return x     i _ → return (x, i)
        ReadChar k i b → do
          mc ← readChar
          case mc of
            Nothing    →
              (k mc $! (if b then i + 1 else i)) $ False
            Just c     →
              if  ( c ≡ ' '   ∨ c ≡ '\t'
                  ∨ c ≡ '\n' ∨ c ≡ '\r') then
                (k mc $! (if b then i + 1 else i)) $ False
              else k mc i True |]
countWord = countWord0 0 False
```

Combining these two handlers, we write a general *wc* function:

```
wc ::
  ([handles | h {ReadChar} |], [handles | h {Finished} |])
     ⇒ Comp h (Int, Int, Int)
wc =
  do ((l, w), c) ← countChar (countWord (loop 0))
     return (c, w, l)
  where loop i = do b ← finished
                    if b then return i
                    else do _ ← readLine; loop $! (i + 1)
```

Here is a version of *wc* that takes a string as input:

[4] We cannot η-reduce *sampleMaybe* as doing so upsets the GHC type checker.

$$
\begin{array}{l}
\mathit{wcString} :: \mathit{String} \to \mathit{IO}\ () \\
\mathit{wcString}\ s = \\
\quad \mathbf{let}\ (c, w, l) = \mathit{handlePure}\ (\mathit{stringReader}\ s\ \mathit{wc})\ \mathbf{in} \\
\quad \mathit{putStrLn}\ \$\ (\mathit{show}\ l) +\!\!+ \texttt{" "} +\!\!+ (\mathit{show}\ w) +\!\!+ \texttt{" "} +\!\!+ (\mathit{show}\ c)
\end{array}
$$

Here is a version of wc that uses standard input:

$$
\begin{array}{l}
\mathit{wcStdin} :: \mathit{IO}\ () \\
\mathit{wcStdin} = \mathbf{do} \\
\quad (c, w, l) \leftarrow \mathit{handleIO}\ (\mathit{stdinReader}\ \mathit{wc}) \\
\quad \mathit{putStrLn}\ \$\ (\mathit{show}\ l) +\!\!+ \texttt{" "} +\!\!+ (\mathit{show}\ w) +\!\!+ \texttt{" "} +\!\!+ (\mathit{show}\ c)
\end{array}
$$

In practice, one might define other handlers in order to support file input, network input, or different forms of buffering.

2.5 Tail

The $tail$ program takes an argument n and prints the last n lines of a text file. In order to implement the functionality of $tail$, we make use of $readLine$ as well as two additional abstract operations: the first to record a line, and the second to print all recorded lines.

$$
\begin{array}{l}
[\mathbf{operation} \mid \mathit{SaveLine} :: \mathit{String} \to ()\ |] \\
[\mathbf{operation} \mid \mathit{PrintAll} :: \qquad\qquad ()\ |]
\end{array}
$$

With these two operations, implementing an abstract tail computation $tailComp$ is straightforward.

$$
\begin{array}{l}
\mathit{tailComp} :: \\
\quad ([\mathbf{handles} \mid h\ \{\mathit{ReadChar}\}\ |], [\mathbf{handles} \mid h\ \{\mathit{Finished}\}\ |], \\
\quad\ [\mathbf{handles} \mid h\ \{\mathit{SaveLine}\}\ |], [\mathbf{handles} \mid h\ \{\mathit{PrintAll}\}\ |]) \\
\qquad \Rightarrow \mathit{Comp}\ h\ () \\
\mathit{tailComp} = \\
\quad \mathbf{do}\ s \leftarrow \mathit{readLine}; \mathit{saveLine}\ s \\
\qquad\ b \leftarrow \mathit{finished}; \mathbf{if}\ b\ \mathbf{then}\ \mathit{printAll}\ \mathbf{else}\ \mathit{tailComp}
\end{array}
$$

We now just need to handle the $SaveLine$ and $ReadLine$ operations. A naive handler might store all saved lines in memory, and print the last n as required. In practice, a more efficient implementation might store only the last n lines, using a circular array, say.

2.6 Pipes and Shallow Handlers

The behaviour of handlers we have described thus far is such that the continuation of an operation is handled with the current handler (though the parameters passed to the continuation may differ from the current parameters).

Another possible behaviour is for the continuation to return an unhandled computation, which must then be handled explicitly. We call such handlers *shallow handlers* because each handler only handles one step of a computation, in contrast to Plotkin and Pretnar's *deep handlers*. Shallow handlers are to deep handlers as case analysis is to a fold on an algebraic data type.

Shallow handlers sometimes lead to slightly longer code. For example, the $EvalState$ handler from Section 2.1 becomes:

$$
\begin{array}{l}
[\mathbf{shallowHandler} \mid \\
\quad \mathit{EvalStateShallow}\ s\ a :: s \to a \\
\qquad \mathbf{handles}\ \{\mathit{Get}\ s, \mathit{Put}\ s\}\ \mathbf{where} \\
\qquad\quad \mathit{Return}\ x \quad s \to x \\
\qquad\quad \mathit{Get} \qquad k\ s \to \mathit{evalStateShallow}\ (k\ s)\ s \\
\qquad\quad \mathit{Put} \qquad s\ k\ \_ \to \mathit{evalStateShallow}\ (k\ ())\ s\ |]
\end{array}
$$

The need to call the handler recursively in most clauses is characteristic of the style of program one writes with shallow handlers.

In some situations, it is helpful to have access to the unhandled result of the continuation. Consider *pipes* as exemplified by Gonzalez's `pipes` library [14]. A *pipe* is a data structure used to represent composable producers and consumers of data. A consumer can $await$ data and a producer can $yield$ data. A pipe is both a consumer and a producer. It is straightforward to provide such an abstraction with the following operations[5]:

$$
\begin{array}{l}
[\mathbf{operation} \mid \mathit{Await}\ s :: \qquad s\ \ |] \\
[\mathbf{operation} \mid \mathit{Yield}\ \ s :: s \to ()\ |]
\end{array}
$$

To define a plumbing operator that combines a compatible consumer and producer we write two handlers: one handles the downstream consumer and keeps a suspended producer to resume when needed, the other handles the upstream producer and keeps a suspended consumer. These two handlers are straightforward to write using shallow handlers:

$$
\begin{array}{l}
[\mathbf{shallowHandler} \mid \\
\quad \mathbf{forward}\ h.\mathit{Down}\ s\ a :: \mathit{Comp}\ (\mathit{Up}\ h\ a)\ a \to a \\
\qquad \mathbf{handles}\ \{\mathit{Await}\ s\}\ \mathbf{where} \\
\qquad\quad \mathit{Return}\ x \qquad \_ \qquad \to \mathit{return}\ x \\
\qquad\quad \mathit{Await} \qquad\quad k\ \mathit{prod} \to \mathit{up}\ k\ \mathit{prod}\ |]
\end{array}
$$

$$
\begin{array}{l}
[\mathbf{shallowHandler} \mid \\
\quad \mathbf{forward}\ h.\mathit{Up}\ s\ a :: (s \to \mathit{Comp}\ (\mathit{Down}\ h\ a)\ a) \to a \\
\qquad \mathbf{handles}\ \{\mathit{Yield}\ s\}\ \mathbf{where} \\
\qquad\quad \mathit{Return}\ x \quad \_ \qquad \to \mathit{return}\ x \\
\qquad\quad \mathit{Yield}\ s \quad\ k\ \mathit{cons} \to \mathit{down}\ (k\ ())\ (\mathit{cons}\ s)\ |]
\end{array}
$$

However, transforming these handlers into deep handlers requires some ingenuity. Indeed, we need to work with continuations that are fully handled and we cannot keep the simple mutually recursive structure of the two handlers. Instead, we introduce two mutually recursive type definitions

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Prod}\ \ s\ r = \mathit{Prod}\ \ (() \to \mathit{Cons}\ s\ r \to r) \\
\mathbf{data}\ \mathit{Cons}\ s\ r = \mathit{Cons}\ (s \to \mathit{Prod}\ s\ r \to r)
\end{array}
$$

which we use to encode the suspended partner of each computation

$$
\begin{array}{l}
[\mathbf{handler} \mid \\
\quad \mathbf{forward}\ h.\mathit{Down}\ s\ a :: \mathit{Prod}\ s\ (\mathit{Comp}\ h\ a) \to a \\
\qquad \mathbf{handles}\ \{\mathit{Await}\ s\}\ \mathbf{where} \\
\qquad\quad \mathit{Return}\ x \quad\ \_ \qquad\qquad\quad \to \mathit{return}\ x \\
\qquad\quad \mathit{Await} \qquad k\ (\mathit{Prod}\ \mathit{prod})\ \to \mathit{prod}\ ()\ (\mathit{Cons}\ k)\ |] \\
[\mathbf{handler} \mid \\
\quad \mathbf{forward}\ h.\mathit{Up}\ s\ a :: \mathit{Cons}\ s\ (\mathit{Comp}\ h\ a) \to a \\
\qquad \mathbf{handles}\ \{\mathit{Yield}\ s\}\ \mathbf{where} \\
\qquad\quad \mathit{Return}\ x \quad\ \_ \qquad\qquad\quad \to \mathit{return}\ x \\
\qquad\quad \mathit{Yield}\ s \quad\ k\ (\mathit{Cons}\ \mathit{cons}) \to \mathit{cons}\ s\ (\mathit{Prod}\ k)\ |]
\end{array}
$$

resulting in a more complex program. We believe both deep and shallow handlers are useful. For clarity of presentation, we focus on deep handlers in the rest of this paper. In Section 3.4 and Section 4.2 we outline how shallow handlers differ from the main presentation.

2.7 Other Perspectives

In this paper we primarily treat handlers as a flexible tool for interpreting abstract effectful computations. Before we proceed with the rest of the paper we highlight some alternative perspectives on what handlers are.

Generalised exception handlers. Benton and Kennedy [3] introduced the idea of adding a return continuation to exception handlers. Their return continuation corresponds exactly to the return clause of an effect handler. Effect handler operation clauses generalise exception handler clauses by adding a continuation argument, providing support for arbitrary effects. An operation clause that ignores its continuation argument behaves like a standard exception handler clause.

Taming delimited continuations. A handler invocation delimits the start of a continuation. Each operation clause captures the continuation of the computation currently being handled, that is, the continuation up to the invocation point of the handler. Effect handlers modularise delimited continuations by capturing particular patterns of use. As Andrej Bauer, the co-creator of the Eff [2] lan-

[5] These operations have exactly the same signatures as Get and Put, but their intended interpretation is different. For instance, $yield\ x; yield\ y$ is in no way equivalent to $yield\ y$.

(values)	A, B	$::= 1 \mid A_1 \times A_2 \mid 0 \mid A_1 + A_2 \mid U_E C$
(computations)	C	$::= FA \mid A \to C \mid \top \mid C_1 \,\&\, C_2$
(effect signatures)	E	$::= \{\mathrm{op} : A \to B\} \uplus E \mid \emptyset$
(handlers)	R	$::= A \,{}^{E}{\Rightarrow}^{E'}\, C$
(environments)	Γ	$::= x_1 : A_1, \ldots, x_n : A_n$

Figure 1. Types and Effects of λ_{eff}

(values)	V, W	$::= x \mid () \mid (V_1, V_2) \mid \mathbf{inj}_i\, V \mid \{M\}$
(computations)		
	M, N	$::= \mathbf{split}(V, x_1.x_2.M) \mid \mathbf{case_0}(V)$
		$\mid\ \mathbf{case}(V, x_1.M_1, x_2.M_2) \mid V!$
		$\mid\ \mathbf{return}\, V \mid \mathbf{let}\, x \leftarrow M \,\mathbf{in}\, N$
		$\mid\ \lambda x.M \mid M\, V$
		$\mid\ \langle M_1, M_2 \rangle \mid \mathbf{prj}_i\, M$
		$\mid\ \mathrm{op}\, V(\lambda x.M) \mid \mathbf{handle}\, M \,\mathbf{with}\, H$
(handlers)	H	$::= \{\mathbf{return}\, x \mapsto M\} \mid H \uplus \{\mathrm{op}\, p\, k \mapsto N\}$

Figure 2. Syntax of λ_{eff} Terms

guage puts it (personal communication, 2012):

$$\frac{\text{“effects + handlers” : “delimited continuations”}}{\text{“while” : “goto”}}$$

Folds over free monads. As we shall see in Section 4, we can define an algebraic data type for a collection of operations, and a handler is exactly a fold (otherwise known as *catamorphism*) over that data type.

3. The λ_{eff}-calculus

In this section, we present a small-step operational semantics and a sound type and effect system for λ_{eff}, a higher-order calculus of effect handlers. Following related work on effect operations [19], effect handlers [36], and monadic reflection [11], which takes Levy's call-by-push-value [24] as the underlying paradigm, we extend Levy's calculus with effect operations and effect handlers. For our purposes, the important features of call-by-push-value are that it makes an explicit distinction between values and computations, and that thunks are first-class objects distinct from functions.

3.1 Syntax and Static Semantics

The types and effects of λ_{eff} are given in Figure 1, the terms are given in Figure 2, and the typing rules are given in Figure 3. In all figures, the interesting parts are highlighted in grey. The unhighlighted parts are standard.

Call-by-push-value makes a syntactic distinction between values and computations. Only value terms may be passed to functions, and only computation terms may compute.

Value types (Figure 1) comprise the value unit type (1), value products ($A_1 \times A_2$), the empty type (0), sums ($A_1 + A_2$), and thunks ($U_E C$). The latter is the type of suspended computations. The effect signature E describes the effects that such computations are allowed to cause.

Computation types comprise value-returning computations (FA), functions ($A \to C$), the computation unit type ($\top$), and computation product types ($C_1 \,\&\, C_2$). Call-by-push-value includes two kinds of products: computation products, which are eliminated by projection, and value products, which are eliminated by binding.

An effect signature is a mapping from operations to pairs of value types, written as a set of type assignments. Each type assignment $\mathrm{op} : A \to B$, specifies the parameter type A and return type B of operation op.

A handler type $A \,{}^{E}{\Rightarrow}^{E'}\, C$ has a input value type A, output computation type C, input effect signature E, and output effect signature E'. A handler of this type handles value-returning computations of type FA that can only perform operations in E. The body of the handler itself may only perform operations in E', and its computation type is C. Type environments are standard.

Value terms (Figure 2) include variables and value introduction forms. We write $\{M\}$ for the thunk that represents the suspended computation M as a value. All elimination occurs in computation terms, as is standard for call-by-push-value. We write $\mathbf{split}(V, x_1.x_2.M)$ for the elimination form for value products, which binds the components of the product value V to the variables x_1 and x_2 in the computation M. We write $\langle M_1, M_2 \rangle$ for a computation pair and $\mathbf{prj}_i\, M$ for the i-th projection of M. We write $V!$ for the computation that *forces* the thunk V, that is, runs the computation suspended in V. A lambda-abstraction $\lambda x.M$ is *not* a value, so must be suspended to be passed as an argument. Function application, products, and projections are standard.

In λ_{eff} operation applications are in continuation-passing-style. An operation application $\mathrm{op}\, V(\lambda x.M)$ takes a parameter V and a continuation $\lambda x.M$. The intuition is that the operation op is applied to the parameter V, returning a value that is bound to x in the continuation computation M. We restrict the continuation to be a lambda abstraction in order to simplify the operational semantics.

While programming with effects, it is more convenient to work with direct-style operation application. Direct-style application can be defined in terms of continuation-passing-style application: $\widehat{\mathrm{op}}\; V = \mathrm{op}\; V(\lambda x.\mathbf{return}\, x)$, and vice versa: $\mathrm{op}\; V(\lambda x.M) = \mathbf{let}\; x \leftarrow \widehat{\mathrm{op}}\; V \;\mathbf{in}\; M$. Plotkin and Power call the function $\lambda x.\widehat{\mathrm{op}}\; x$ underlying a direct-style application the *generic effect* of op [35].

A handled computation $\mathbf{handle}\; M \;\mathbf{with}\; H$ runs the computation M with the handler H. A handler H consists of a return clause $\mathbf{return}\; x \mapsto M$, and a set of operation clauses of the form $\mathrm{op}\; p\; k \mapsto N$. The return clause $\mathbf{return}\; x \mapsto M$ specifies how to handle a return value. The returned value is bound to x in M. Each operation clause $\mathrm{op}\; p\; k \mapsto N$ specifies how to handle applications of the distinct operation name op. The parameter is bound to p and the continuation is bound to k in N. The body of the continuation continues to be handled by the same handler.

The typing rules are given in Figure 3. The computation typing judgement $\Gamma \vdash_E M : C$ states that in type environment Γ the computation M has type C and effect signature E. Only operations in the current effect signature can be applied. Handling a computation changes the current effect signature to the output effect signature of the handler. The effect signature and type of the handled computation must match up exactly with the input type and effect signature of the handler.

In the handler typing judgement $\Gamma \vdash H : R$, all clauses must have the same output type and effect signature. The input type is determined by the return clause. The effect annotation on the thunked continuation parameter k in an operation clause $\mathrm{op}\; p\; k \mapsto N$ is annotated with the output effect rather than the input effect. The reason for this is that when handling an operation, the handler is automatically wrapped around the continuation.

Syntactic Sugar For convenience, we define syntactic sugar for projecting out the return clause and operation clauses of a handler. For any handler

$$\{\mathbf{return}\; x \mapsto M\} \uplus \{\mathrm{op}_i\; p\; k \mapsto N_i\}_i$$

we write

$$H^{\mathbf{return}} \equiv \lambda x.M \qquad\qquad H^{\mathrm{op}_i} \equiv \lambda p_i\, k_i.N_i$$

Value typing $\boxed{\Gamma \vdash V : A}$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{}{\Gamma \vdash () : 1} \qquad \frac{\Gamma \vdash V_1 : A_1 \qquad \Gamma \vdash V_2 : A_2}{\Gamma \vdash (V_1, V_2) : A_1 \times A_2}$$

$$\frac{\Gamma \vdash V : A_i}{\Gamma \vdash \mathbf{inj}_i\, V : A_1 + A_2} \qquad \frac{\Gamma \vdash_E M : C}{\Gamma \vdash \{M\} : U_E C}$$

Computation typing $\boxed{\Gamma \vdash_E M : C}$

$$\frac{\Gamma \vdash V : A_1 \times A_2 \qquad \Gamma, x_1 : A_1, x_2 : A_2 \vdash_E M : C}{\Gamma \vdash_E \mathbf{split}(V, x_1.x_2.M) : C}$$

$$\frac{\Gamma \vdash V : 0}{\Gamma \vdash_E \mathbf{case_0}(V) : C}$$

$$\frac{\Gamma \vdash V : A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vdash_E M_1 : C \qquad \Gamma, x_2 : A_2 \vdash_E M_2 : C}{\Gamma \vdash_E \mathbf{case}(V, x_1.M_1, x_2.M_2) : C}$$

$$\frac{\Gamma \vdash V : U_E C}{\Gamma \vdash_E V! : C} \qquad \frac{\Gamma \vdash V : A}{\Gamma \vdash_E \mathbf{return}\, V : FA}$$

$$\frac{\Gamma \vdash_E M : FA \qquad \Gamma, x : A \vdash_E N : C}{\Gamma \vdash_E \mathbf{let}\, x \leftarrow M\, \mathbf{in}\, N : C}$$

$$\frac{\Gamma, x : A \vdash_E M : C}{\Gamma \vdash_E \lambda x.M : A \to C} \qquad \frac{\Gamma \vdash_E M : A \to C \qquad \Gamma \vdash V : A}{\Gamma \vdash_E M\, V : C}$$

$$\frac{}{\Gamma \vdash_E \langle\rangle : \top} \qquad \frac{\Gamma \vdash_E M_1 : C_1 \qquad \Gamma \vdash_E M_2 : C_2}{\Gamma \vdash_E \langle M_1, M_2 \rangle : C_1 \,\&\, C_2}$$

$$\frac{\Gamma \vdash_E M : C_1 \,\&\, C_2}{\Gamma \vdash_E \mathbf{prj}_i\, M : C_i}$$

$$\frac{(\mathrm{op} : A \to B) \in E \qquad \Gamma \vdash V : A \qquad \Gamma, x : B \vdash_E M : C}{\Gamma \vdash_E \mathrm{op}\, V (\lambda x.M) : C}$$

$$\frac{\Gamma \vdash_E M : FA \qquad \Gamma \vdash H : A \stackrel{E}{\Rightarrow}^{E'} C}{\Gamma \vdash_{E'} \mathbf{handle}\, M\, \mathbf{with}\, H : C}$$

Handler typing $\boxed{\Gamma \vdash H : A \stackrel{E}{\Rightarrow}^{E'} C}$

$$\frac{\begin{array}{c} E = \{\mathrm{op}_i : A_i \to B_i\}_i \\ H = \{\mathbf{return}\, x \mapsto M\} \uplus \{\mathrm{op}_i\, p\, k \mapsto N_i\}_i \\ [\Gamma, p : A_i, k : U_{E'}(B_i \to C) \vdash_{E'} N_i : C]_i \\ \Gamma, x : A \vdash_{E'} M : C \end{array}}{\Gamma \vdash H : A \stackrel{E}{\Rightarrow}^{E'} C}$$

Figure 3. Typing Rules for λ_{eff}

3.2 Operational Semantics

The reduction relation ($\longrightarrow$) for λ_{eff} is defined in Figure 4. We use *reduction frames* as an auxiliary notion to simplify our presentation. The β-rules are standard: each β-redex arises as an introduction followed by an elimination. The first three β-rules eliminate value terms; the last three eliminate computation terms.

The $hoist.\mathrm{op}$-rule hoists operation applications through hoisting frames. Its purpose is to forward operation applications up to the nearest enclosing handler, so that they can be handled by the $handle.\mathrm{op}$-rule.

The $handle.F$-rule returns a value from a handled computation. It substitutes the returned value into the return clause of a handler

Reduction frames

$$\begin{array}{lrcl} \text{(hoisting frames)} & \mathcal{H} & ::= & \mathbf{let}\, x \leftarrow [\,]\, \mathbf{in}\, N \mid [\,]\, V \mid \mathbf{prj}_i\, [\,] \\ \text{(computation frames)} & \mathcal{C} & ::= & \mathcal{H} \mid \mathbf{handle}\, [\,]\, \mathbf{with}\, H \end{array}$$

Reduction $\boxed{M \longrightarrow M'}$

$$\begin{array}{lrcl} (\beta.\times) & \mathbf{split}((V_1, V_2), x_1.x_2.M_1) & \longrightarrow & M[V_1/x_1, V_2/x_2] \\ (\beta.+) & \mathbf{case}(\mathbf{inj}_i\, V, x_1.M_1, x_2.M_2) & \longrightarrow & M_i[V/x_i] \\ (\beta.U) & \{M\}! & \longrightarrow & M \\ \\ (\beta.F) & \mathbf{let}\, x \leftarrow \mathbf{return}\, V\, \mathbf{in}\, M & \longrightarrow & M[V/x] \\ (\beta.\to) & (\lambda x.M)\, V & \longrightarrow & M[V/x] \\ (\beta.\&) & \mathbf{prj}_i\, \langle M_1, M_2 \rangle & \longrightarrow & M_i \end{array}$$

$(hoist.\mathrm{op})$

$$\frac{x \notin FV(\mathcal{H})}{\mathcal{H}[\mathrm{op}\, V(\lambda x.M)] \longrightarrow \mathrm{op}\, V(\lambda x.\mathcal{H}[M])}$$

$(handle.F)$

$$\frac{H^{\mathbf{return}} = \lambda x.M}{\mathbf{handle}\, (\mathbf{return}\, V)\, \mathbf{with}\, H \longrightarrow M[V/x]}$$

$(handle.\mathrm{op})$

$$\frac{H^{\mathrm{op}} = \lambda p\, k.N \qquad x \notin FV(H)}{\begin{array}{l}\mathbf{handle}\, \mathrm{op}\, V(\lambda x.M)\, \mathbf{with}\, H \\ \qquad \longrightarrow N[V/p, \{\lambda x.\mathbf{handle}\, M\, \mathbf{with}\, H\}/k]\end{array}}$$

$(frame)$

$$\frac{M \longrightarrow M'}{\mathcal{C}[M] \longrightarrow \mathcal{C}[M']}$$

Figure 4. Operational Semantics for λ_{eff}

in exactly the same way that $\beta.F$-reduction substitutes a returned value into the body of a let binding.

The $handle.\mathrm{op}$-rule is the most involved of the reduction rules. A handled operation application $\mathbf{handle}$ op $V(\lambda x.M)$ $\mathbf{with}$ H reduces to the body of the operation clause $H^{\mathrm{op}} = \lambda p\, k.N$ with the parameter V substituted for p and the continuation $\lambda x.M$ substituted for k. Any further operation applications should be handled by the same handler H. Thus, we wrap H around M.

The *frame*-rule allows reduction to take place within any stack of computation frames, that is, inside any *evaluation context*.

The semantics is deterministic, as any term has at most one redex. Furthermore, reduction on well-typed terms always terminates.

Theorem 1 (Termination)**.** *If* $\Gamma \vdash_E M : C$ *then reduction on* M *terminates.*

Proof sketch: The proof is by a relatively straightforward adaptation of Lindley's proof of strong normalisation for sums [26]. The interesting rule is $handle.\mathrm{op}$, which reinvokes the handler, possibly many times, but always on a subterm of the original computation. As with Ariola et al.'s normalisation result for delimited continuations [1], termination depends crucially on the effect type system. □

Theorem 2 (Type soundness)**.** *If* $\vdash_{\{\}} M : FA$ *then* M *reduces to a returned value* $\mathbf{return}\ V$ *of type* A*.*

Proof sketch: Define a *canonical* term to be any computation term of the form: $\mathbf{return}\ V$, $\lambda x.N$, $\langle\rangle$, $\langle V, W\rangle$, or op $V(\lambda x.M)$. Induction on typing derivations shows progress: if $\vdash_E M : C$ then, either there exists M' such that $M \longrightarrow M'$, or M is canonical. By appeal to a substitution lemma, induction on typing derivations shows preservation: if $\Gamma \vdash_E M : C$ and $M \longrightarrow M'$ then $\Gamma \vdash_E M' : C$. $\square$

3.3 Open Handlers

Our presentation of λ_{eff} gives an operational account of closed handlers. We can adapt λ_{eff} to support open handlers by making two small changes. The typing rule for handlers becomes:

$$\frac{\begin{array}{c} E = E' \oplus \{\text{op}_i : A_i \to B_i\}_i \\ H = \{\textbf{return}\ x \mapsto M\} \uplus \{\text{op}_i\ p\ k \mapsto N_i\}_i \\ [\Gamma, p : A_i, k : U_{E'}(B_i \to C) \vdash_{E'} N_i : C]_i \\ \Gamma, x : A \vdash_{E'} M : C \end{array}}{\Gamma \vdash H : A \stackrel{E}{\Rightarrow}\!{}^{E'}\, C}$$

The only change to the original rule is that the input effects are now $E' \oplus E$ instead of just E, where $E' \oplus E$ is the extension of E' by E (where any clashes are resolved in favour of E).

The $handle$.op-rule is refined by extending the meaning of H^{op}, such that it is defined as before for operations that are explicitly handled by H, but is also defined as $\lambda p\, k.\text{op}\ p(\lambda x.k\ x)$ for any other operation op. This means that any operation that is not explicitly handled gets forwarded.

In our simply-typed formalism, it is straightforward to translate any program that uses open handlers into an equivalent program that uses closed handlers. As any open handler handles a bounded number of operations, we can simply write down all of the implicit forwarding clauses explicitly. In practice, it seems desirable to offer both open and closed handlers, as in our Haskell library.

To take full advantage of open handlers in a typed language, one inevitably wants to add some kind of effect polymorphism. Indeed, our Haskell implementation provides effect polymorphism, encoded using type classes. We believe that effect polymorphism can be supported more smoothly using row polymorphism.

We briefly outline one path to supporting row polymorphism. The open handler rule from above can be rewritten as follows:

$$\frac{\begin{array}{c} E = \{\text{op}_i : A_i \to B_i\}_i \uplus E_f \\ E' = E'' \uplus E_f \\ H = \{\textbf{return}\ x \mapsto M\} \uplus \{\text{op}_i\ p\ k \mapsto N_i\}_i \\ [\Gamma, p : A_i, k : U_{E'}(B_i \to C) \vdash_{E'} N_i : C]_i \\ \Gamma, x : A \vdash_{E'} M : C \end{array}}{\Gamma \vdash H : A \stackrel{E}{\Rightarrow}\!{}^{E'}\, C}$$

The key difference is that this version of the rule uses disjoint union $\uplus$ in place of extension $\oplus$, explicitly naming the collection of forwarded effects. One can now instantiate the meta variable E_f with a row variable. We would likely also wish to support polymorphism over E''. This is not difficult to achieve using a Remy-style [37] account of row typing if we insist that E'' only include operations in $\{\text{op}_i\}_i$. We leave a full investigation of effect polymorphism for handlers to future work.

3.4 Shallow Handlers

Our presentation of λ_{eff} gives an operational account of deep handlers. In order to model shallow handlers we can make two small changes. The typing rule for handlers becomes:

$$\frac{\begin{array}{c} E = \{\text{op}_i : A_i \to B_i\}_i \\ H = \{\textbf{return}\ x \mapsto M\} \uplus \{\text{op}_i\ p\ k \mapsto N_i\}_i \\ [\Gamma, p : A_i, k : U_E(B_i \to FA) \vdash_{E'} N_i : C]_i \\ \Gamma, x : A \vdash_{E'} M : C \end{array}}{\Gamma \vdash H : A \stackrel{E}{\Rightarrow}\!{}^{E'}\, C}$$

The only changes with respect to the original rule are to the types of the continuations, which now yield FA computations under the input effect signature E, rather than C computations under the output effect signature E'. The $handle$.op-rule is replaced by the $shallow\text{-}handle$.op-rule:

$$(shallow\text{-}handle.\text{op})\quad \frac{H^{\text{op}} = \lambda p\, k.N \qquad x \notin FV(H)}{\textbf{handle}\ \text{op}\ V(\lambda x.M)\ \textbf{with}\ H \longrightarrow N[V/p, \{\lambda x.M\}/k]}$$

which does not wrap the handler around the continuation. Of course, without recursion shallow handlers are rather weak.

4. Implementation

Any signature of operations can be viewed as a free algebra and represented as a functor. Every such functor gives rise to a free monad (Swierstra [43] gives a clear account of free monads for functional programmers). This yields a systematic way of building a monad to represent computations over a signature of operations.

We use our standard state example to illustrate. Concretely we can define the free monad over state as follows:

```
data FreeState s a =
    Ret a
  | Get () (s  → FreeState s a)
  | Put s  (() → FreeState s a)
instance Monad (FreeState s) where
  return          = Ret
  Ret v      >>= f = f v
  Get () k   >>= f = Get () (λx → k x >>= f)
  Put s  k   >>= f = Put s  (λx → k x >>= f)
```

The type $FreeState\ s\ a$ is a particular instance of a free monad. It can be viewed as a computation tree. The leaves are labelled with $Ret\ v$ and the nodes are labelled with $Get\ ()$ and $Put\ s$. There is one edge for each possible return value supplied to the continuation of Get and Put — a possibly infinite number for Get depending on the type of the state, and just one for Put. The bind operation performs a kind of substitution. To compute $c \gg\!= f$, the tree $f\ v$ is grafted onto each leaf $Ret\ v$ of c.

The generic free monad construction can be defined as follows:

```
data Free f a = Ret a | Do (f (Free f a))
instance Functor f ⇒ Monad (Free f) where
  return     = Ret
  Ret v  >>= f = f v
  Do op >>= f = Do (fmap (>>=f) op)
```

and we can instantiate it with state as follows:

```
data StateFunctor s a = GetF () (s → a) | PutF s (() → a)
  deriving Functor
```

where $FreeState\ s$ is isomorphic to $Free\ (StateFunctor\ s)$.

A handler for state is now simply a unary function whose argument has type $Free\ (StateFunctor\ s)$. For instance:

```
stateH :: Free (StateFunctor s) a → (s → a)
stateH (Ret x)             = λs → x
stateH (Do (GetF () k)) = λs → stateH (k s) s
stateH (Do (PutF s k))  = λ_ → stateH (k ()) s
```

interprets a stateful computation as a function of type $s \to a$.

A limitation of the above free monad construction is that it is *closed* in that it can only handle operations in the signature. A key feature of our library is support for *open handlers* that handle a fixed set of operations in a specified way, and forward any other operations to be handled by an outer handler. To encode openness in GHC we take advantage of type classes and type families.

The code for open free monads is given in Figure 5. We split the type of each operation into two parts: a type declaration that defines the parameters to the operation, and a type family instance that defines the return type of the operation. For instance:

```
[operation | Put s :: s → () |]
```

generates:

```
data Put (e :: ⋆) (u :: ⋆) where
  Put :: s → Put s ()
type instance Return (Put s ()) = ()
```

The first type argument e encodes the *existential* type arguments of an operation, while the second type argument u encodes the *universal* type arguments of an operation. In the case of Put there is a single existential type argument s and no universal arguments. Using GADTs we can encode multiple arguments as tuples. Handler types are generated similarly.

Lines 1–2 of Figure 5 declare type families for operation return types and handler result types.

Lines 3–6 define a ternary type class $(h\ \text{`}Handles\text{`}\ op)\ e$. Each instance of this class defines the clause of handler h that handles operation op with existential type arguments bound to e. Of course, we do not supply the universal arguments, as the clause should be polymorphic in them. The functional dependency $h\ op \to a$ asserts that type a must be uniquely determined by types h and op. This is crucial for correctly implementing open handlers as we discuss further in Section 4.1. The **handles** quasiquoter generates type class constraints on the $Handles$ type class.

Lines 7–14 define the monad $Comp\ h$, which is simply a free monad over the functor defined by those operations op that are handled by h (i.e. such that $(h\ \text{`}Handles\text{`}\ op)\ e$ is defined for some type e).

Lines 15–17 define the auxiliary $doOp$ function, which realises an operation as an abstract computation of the appropriate type. The **operation** quasiquoter uses $doOp$ to automatically generate a function for each operation. For instance, the above declarations for Get and Put generate the following functions:

```
get :: (h `Handles` Get) s ⇒ Comp h s
get = doOp Get
put :: (h `Handles` Put) s ⇒ s → Comp h ()
put s = doOp (Put s)
```

Finally, Lines 18–21 define the one remaining ingredient for the core library, the $handle$ function, which takes a computation, a return clause, and a handler, and returns the result of handling the computation. We supply the return clause independently from the type class mechanism in order to simplify the implementation. The handler is automatically applied to the result of the continuation as specified in the operational semantics. This behaviour contrasts with the closed free monad code presented at the beginning of this subsection, which instead uses explicit recursion as with the shallow handlers for state and pipes.

We are making essential use of the type class mechanism. It is instructive to read the type of $handle$ as taking a return clause, a list of operation clauses, one for each op such that $(h\ \text{`}Handles\text{`}\ op)\ e$ for some e, and returning a result. Thus the second argument of type h, as well as providing parameters to the handler, also, albeit indirectly, encodes the list of operation clauses.

The **handler** quasiquoter automatically generates a convenient wrapper meaning that programmers need never directly manipulate the constructor for a handler type — just as automatically generated operation wrappers mean that they need never directly manipulate the constructor for an operation type.

Limitations Our Haskell implementation of handlers has several limitations. First, because handlers are encoded as type classes and type classes are not first-class, neither are handlers. Second, because we abstract over handlers in order to simulate effect typing the types of the operation wrappers are more complex than necessary. Third, because we explicitly mention a parent handler in the type of open handlers, the order in which open handlers are composed can leak into types (this is not as bad as with monad transformers, as lifting is never required, but it is still undesirable).

All of these limitations arise from attempting to encode handlers in Haskell. None is inherent to handlers. We believe that a row-based effect type system along the lines of Leroy and Pesaux [23], Blume et al [5], or Lindley and Cheney [27] would provide a cleaner design.

```
1  type family Return (opApp :: ⋆) :: ⋆
2  type family Result (h :: ⋆) :: ⋆
3  class ((h :: ⋆) `Handles` (op :: j → k → ⋆)) (e :: j) | h op → e
4    where
5      clause :: op e u →
6        (Return (op e u) → h → Result h) → h → Result h
7  data Comp h a where
8    Ret :: a → Comp h a
9    Do  :: (h `Handles` op) e ⇒
10     op e u → (Return (op e u) → Comp h a) → Comp h a
11 instance Monad (Comp h) where
12   return          = Ret
13   Ret v    >>= f  = f v
14   Do op k  >>= f  = Do op (λx → k x >>= f)
15 doOp :: (h `Handles` op) e ⇒
16           op e u → Comp h (Return (op e u))
17 doOp op = Do op return
18 handle :: Comp h a → (a → h → Result h) → h → Result h
19 handle (Ret v)     r h = r v h
20 handle (Do op k) r h =
21   clause op (λv h′ → handle (k v) r h′) h
```

Figure 5. Free Monad Implementation

```
1  type family Return (opApp :: ⋆) :: ⋆
2  type family Result (h :: ⋆) :: ⋆
3  class ((h :: ⋆) `Handles` (op :: j → k → ⋆)) (e :: j) | h op → e
4    where
5      clause :: op e u → (Return (op e u) → h → Result h) →
6                                                h → Result h
7  newtype Comp h a =
8    Comp {handle :: (a → h → Result h) → h → Result h}
9  instance Monad (Comp h) where
10   return v = Comp (λk → k v)
11   Comp c >>= f = Comp (λk → c (λx → handle (f x) k))
12 doOp :: (h `Handles` op) e ⇒
13           op e u → Comp h (Return (op e u))
14 doOp op = Comp (λk h → clause op k h)
```

Figure 6. Continuation Monad Implementation

The Codensity Monad It is well known that free monad computations can be optimised by composing with the *codensity monad*, which is essentially the continuation monad over a polymorphic return type [45].

The Continuation Monad We can in fact do even better by using the continuation monad directly, meaning that the Ret and Do constructors disappear completely. The continuation monad code is given in Figure 6. The difference from the free monad code begins on Line 7. The type constructor $Comp\ h$ is exactly that of the continuation monad with return type $h \to Result\ h$. We choose not to factor through the continuation monad defined in the standard library as doing so hurts performance. The $handle$ function is now exactly the deconstructor for $Comp$, while the $doOp$ function is just $Comp \circ clause$. We explicitly η-expand $doOp$ because GHC is unable to optimise the pointless version.

4.1 Open Handlers and Forwarding

The key trick for implementing forwarding is to parameterise a handler H by its parent handler h. Without this parameter we would have no way of describing the operations that are handled by both H and h. Now we can define the following type class instance:

$$\textbf{instance}\ (h\ \text{`}\mathit{Handles}\text{`}\ op)\ e \Rightarrow (H\ h\ \text{`}\mathit{Handles}\text{`}\ op)\ e\ \textbf{where}$$
$$\quad \mathit{clause}\ op\ k\ h = \mathit{doOp}\ op \ggg\!= (\lambda x \to k\ x\ h)$$

The **handler** quasiquoter generates this boilerplate automatically for open handlers.

The functional dependency $(H\ \ h)\ \ op\ \to\ e$ is crucial here. Without it GHC would be unable to resolve the *clause* function. For instance, consider the *OpenState* handler. This must select from the following instances for *Get*:

$$\textbf{instance}\ (\mathit{OpenState}\ h\ s\ a\ \text{`}\mathit{Handles}\text{`}\ \mathit{Get})\ s\ \textbf{where}$$
$$\quad \mathit{clause}\ \mathit{Get}\ k\ (\mathit{OpenState}\ s) = k\ s\ (\mathit{OpenState}\ s)$$

$$\textbf{instance}\ (h\ \text{`}\mathit{Handles}\text{`}\ op)\ t \Rightarrow$$
$$\quad (\mathit{OpenState}\ h\ s\ a\ \text{`}\mathit{Handles}\text{`}\ op)\ t\ \textbf{where}$$
$$\qquad \mathit{clause}\ op\ k\ h = \mathit{doOp}\ op \ggg\!= (\lambda x \to k\ x\ h)$$

Without the functional dependency the latter is chosen. This is because GHC tries to ensure that the same version of *clause* is used for $\mathit{Get}\ t$ for any t, and the former is only valid if s is equal to t. The functional dependency asserts that t must be equal to s.

Because the type variable t does not appear in either of the types $\mathit{OpenState}\ \ h\ \ s\ \ a$ or op, and there is a functional dependency which states that $\mathit{OpenState}\ \ h\ \ s\ \ a$ and op uniquely determine e, GHC's default type inference gives up. Enabling the `UndecidableInstances` language option fixes this. We believe that our basic use of `UndecidableInstances` is well-founded (and decidable!), because of the type class constraint $(h\ \text{`}\mathit{Handles}\text{`}\ op)\ t$ which implies that h and op already uniquely determine t.

4.2 Shallow Handlers

It is relatively straightforward to adapt our free monad implementation to implement shallow handlers. The key change is to the type of the continuation argument of the *clause* function which must return a computation. It seems less clear how to adapt the continuation monad implementation.

4.3 Delimited Continuations

We now sketch an implementation of (open) effect handlers in terms of *delimited continuations* [8, 9]. These ideas underlie our OCaml, SML, and Racket implementations.

A variety of different delimited continuation operators are covered in the literature. Shan has recently shown that the four basic choices are straightforwardly inter-definable [39][6]. We choose to describe our implementation in terms of a minor variant of Danvy and Filinski's **shift** and **reset** operators [8] called **shift0** and **reset0**. The behaviour of **shift0** and **reset0** can be concisely summarised through the following reduction rule:

$$\mathbf{reset0}\,(\mathcal{E}[\mathbf{shift0}\,(\lambda k.M)]) \longrightarrow M[(\lambda x.\mathbf{reset0}\,(\mathcal{E}[x]))/k]$$

where $\mathcal{E}$ ranges over call-by-value evaluation contexts.

The **reset0** operator delimits the start of a continuation, and the **shift0** operator captures the continuation up to the nearest enclosing **reset0**. Crucially, the captured continuation is wrapped in a further **reset0**. It is instructive to compare the above rule with the *handle*.op-rule, where $\textbf{handle} - \textbf{with}\ H$ plays a similar role to **reset0**.

The implementations rely on the following key ingredients:

- A global (or thread-local) variable keeps a stack of handlers in the current dynamic scope.
- Each handler includes a map from the handled operations to the corresponding operation clause.

[6] These encodings do not preserve memory behaviour, so can sometimes introduce memory leaks.

To handle an effectful computation $\textbf{handle}\ M\ \textbf{with}\ H$:

- The handler H is added to the top of the stack.
- We invoke $\mathbf{reset0}\,(H^{\mathbf{return}}\ M)$.

To apply an operation (in direct style) $\widehat{\mathrm{op}}\ p$:

1. We invoke **shift0** to capture the continuation k up to (but excluding) the next operation handler.
2. The top-most handler H is popped from the stack.
3. We let $k' = \lambda x.\mathit{push}\,H;\ \mathbf{reset0}\,(k\ x)$, where $\mathit{push}\,H$ pushes H back onto the stack.
4. The clause corresponding to the operation, that is H^{op}, is applied to the parameter p and the continuation k'.
5. If there is no clause corresponding to this operation, it will be forwarded by the handler. If no other handlers enclose the operation, an exception is raised.

To support shallow handlers, one replaces **shift0** and **reset0** with **control0** and **prompt0**, which behave like **shift0** and **reset0**, except no **prompt0** is wrapped around the continuation:

$$\mathbf{prompt0}\,(\mathcal{E}[\mathbf{control0}\,(\lambda k.M)]) \longrightarrow M[(\lambda x.\mathcal{E}[x])/k]$$

4.4 Dynamic and Static Operations

Our Haskell implementation uses one static type per operation. A program execution cannot dynamically create a new operation. Because they do not provide effect typing, our other implementations do support dynamic operations, which can be created, closed upon and garbage collected. This makes some programs easier to write. For example, references can be represented as pairs of dynamically generated *Put* and *Get* operations. With only static operations, one has to parameterise *Put* and *Get* by some representation of a reference, and explicitly manage all of the state in a single handler.

Static effect typing for dynamic operations presents challenges:

- Writing an effect type system for dynamic operations involves some form of dependent types, as operations are now first-class objects of the language.
- Dynamic operations are difficult to implement as efficiently as static operations. In particular, it is not clear how to use the type system to pass only the relevant effects to each scope.

5. Performance Evaluation

To evaluate the performance of our Haskell library, we implemented a number of micro-benchmarks, comparing handler code against monadic code that makes use of existing libraries. The code for the micro-benchmarks can be found in the GitHub repository at:

`http://github.com/slindley/effect-handlers/Benchmarks`

Detailed performance results can be found in Appendix A.

Our primary goal was to check that the handler abstraction does not cripple performance. We rely on GHC to optimise away many of the abstractions we introduce. In the future we envisage building handlers into the core of a programming language. We might reasonably hope to do significantly better than in the library-based approach by tailoring optimisations to be aware of handlers.

The results confirm that the Haskell library performs adequately. The performance of the continuation monad implementation of handlers is typically no worse than around two thirds that of baseline code. In some cases the continuation monad implementation actually outperforms existing implementations. The continuation monad implementation always outperforms the codensity monad implementation (sometimes by more than an order of magnitude), which always outperforms the free monad implementation. Usually standard handlers outperform shallow handlers, pipes being an exception, where for large numbers of nested sub-pipes shallow handlers outperform even the continuation monad implementation of standard handlers.

6. Related Work

Algebraic Effects and Effect Handlers Effect operations were pioneered by Plotkin and Power [35], leading to an algebraic account of computational effects [34] and their combination [15]. Effect handlers were added to the theory in order to support exception handling [36]. Recent work incorporates additional computational effects within the algebraic framework, for example, local state [41], and applies the algebraic theory of effects to new problem domains, such as effect-dependent program transformations [19], and logical-relations arguments [20].

While mostly denotational in nature, operational accounts of algebraic effects (without handlers) do exist. Plotkin and Power [33] gave operational semantics to algebraic effects in a call-by-value setting, and Johann et al. [32] gave operational semantics to algebraic effects in a call-by-name setting.

Effect Handler Implementations Bauer and Pretnar's effectful strict statically-typed language *Eff* [2] has built-in support for algebraic effects and effect handlers. Like our ML implementations, it lacks an effect type system. Eff implements dynamic generation of new operations and effects, which we only consider statically. Visscher has implemented the `effects` library [44] for Haskell inspired by Eff. The core idea is to layer continuation monads in the style of Filinski [10], using Haskell type classes to automatically infer lifting between layers. McBride's language *Frank* [29] is similar to Eff, but with an effect system along the lines of ours. It supports only shallow handlers and employs a novel form of effect polymorphism which elides effect variables entirely. Brady [6] has implemented an effect handlers library for the dependently-typed language Idris. It takes advantage of dependent types for resource tracking. The current design has some limitations compared with the other handler implementations. In particular, the composition of other effects with non-determinism is not well-behaved.

Monadic Reflection and Layered Monads Filinski's work on monadic reflection and layered monads is closely related to effect handlers [11]. Monadic reflection supports a similar style of composing effects. The key difference is that monadic reflection interprets monadic computations in terms of other monadic computations, rather than abstracting over and interpreting operations. Filinski's system is nominal (an effect is the name of a monad), whereas ours is structural (an effect is a collections of operations).

Monad Transformers and Inferred Lifting Jaskelioff and Moggi [17] develop the theory of monad transformers and lifting of effect operations. Jaskelioff's Monatron [16] is a monad transformer library based on this development.

Schrijvers and Oliveira [38] infer lifting in Haskell using a zipper structure at the level of type classes to traverse the monad transformer stack. Swamy et al. [42] add support for monads in ML, inferring not only where and how to lift operations, but also where to insert return and bind statements. In both approaches, once the required monad transformers have been defined, the desired lifting is inferred automatically.

7. Conclusion

Algebraic effects and handlers provide a promising approach for supporting effectful computations in functional languages. By offering a new form of modularity, they create possibilities for library design and reusability that are just beginning to emerge. This paper shows that implementing these constructs is within the reach of current compiler technology.

Acknowledgments

The λ_{eff} calculus stemmed from discussions of the first author with Andrej Bauer and Matija Pretnar in Ljubljana and Swansea.

The authors would like to thank Stevan Andjelkovic, Danel Ahman, Bob Atkey, Andrej Bauer, Brian Campbell, James Cheney, Derek Dreyer, Andrzej Filinski, Ben Kavanagh, Neel Krishnaswami, Conor McBride, James McKinna, Gordon Plotkin, Matija Pretnar, Alex Simpson, Sam Staton, Phil Wadler, and the POPL 2013, PLDI 2013, and ICFP 2013 referees, for useful conversations, comments and suggestions. We are grateful to Gabriel Gonzalez for pointing out a flaw in the pipes benchmarks in an earlier draft of this paper. The type system originated from a visit by the first author to Andrej Bauer and Matija Pretnar in Ljubljana, supported by the Laboratory for Foundations of Computer Science. This work was supported by a Google Research Award, EPSRC grants EP/J014591/1 and EP/H005633/1, a SICSA studentship, an Edinburgh University Informatics School studentship, and an Isaac Newton Trust starter grant.

References

[1] Z. M. Ariola, H. Herbelin, and A. Sabry. A type-theoretic foundation of delimited continuations. *Higher-Order and Symbolic Computation*, 22(3):233–273, 2009.

[2] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *CoRR*, abs/1203.1539, 2012.

[3] N. Benton and A. Kennedy. Exceptional syntax. *J. Funct. Program.*, 11(4):395–410, 2001.

[4] N. Benton, J. Hughes, and E. Moggi. Monads and effects. In *APPSEM 2000*. Springer-Verlag, 2002.

[5] M. Blume, U. A. Acar, and W. Chae. Exception handlers as extensible cases. In *APLAS*. Springer-Verlag, 2008.

[6] E. Brady. Programming and reasoning with algebraic effects and dependent types. In *ICFP*. ACM, 2013.

[7] B. C. d. S. Oliveira, T. Schrijvers, and W. R. Cook. MRI: Modular reasoning about interference in incremental programming. *J. Funct. Program.*, 22(6):797–852, 2012.

[8] O. Danvy and A. Filinski. Abstracting control. In *LFP*. ACM, 1990.

[9] M. Felleisen. The theory and practice of first-class prompts. In *POPL*. ACM, 1988.

[10] A. Filinski. Representing layered monads. In *POPL*. ACM, 1999.

[11] A. Filinski. Monads in action. In *POPL*. ACM, 2010.

[12] A. Gill. The mtl package (2.1.2), 2012. `http://hackage.haskell.org/package/mtl`.

[13] G. Gonzalez. pipes-2.5: Faster and slimmer, 2012. `http://www.haskellforall.com/2012/10/pipes-25-faster-and-slimmer.html`.

[14] G. Gonzalez. The pipes package (3.2.0), 2013. `http://hackage.haskell.org/package/pipes`.

[15] M. Hyland, G. D. Plotkin, and J. Power. Combining effects: Sum and tensor. *Theoret. Comput. Sci.*, pages 70–99, 2006.

[16] M. Jaskelioff. Monatron: An extensible monad transformer library. In *IFL*. Springer-Verlag, 2008.

[17] M. Jaskelioff and E. Moggi. Monad transformers as monoid transformers. *Theoret. Comput. Sci.*, 411(51–52), 2010.

[18] M. P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming*, 1995.

[19] O. Kammar and G. D. Plotkin. Algebraic foundations for effect-dependent optimisations. In *POPL*. ACM, 2012.

[20] S. Katsumata. Relating computational effects by $\top\top$-lifting. *Inf. Comput.*, 222, 2013.

[21] O. Kiselyov. Iteratees. In *FLOPS*. Springer-Verlag, 2012.

[22] O. Kiselyov and C.-c. Shan. Embedded probabilistic programming. In *DSL*. Springer-Verlag, 2009.

[23] X. Leroy and F. Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, pages 340–377, 2000.

[24] P. B. Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*, volume 2 of *Semantics Structures in Computation*. Springer, 2004.

[25] S. Liang, P. Hudak, and M. P. Jones. Monad transformers and modular interpreters. In *POPL*. ACM, 1995.

[26] S. Lindley. Extensional rewriting with sums. In *TLCA*. Springer-Verlag, 2007.

[27] S. Lindley and J. Cheney. Row-based effect types for database integration. In *TLDI*. ACM, 2012.

[28] G. Mainland. Why it's nice to be quoted: quasiquoting for Haskell. In *Haskell*. ACM, 2007.

[29] C. McBride. Frank (0.3), 2012. `http://hackage.haskell.org/package/Frank`.

[30] E. Moggi. Computational lambda-calculus and monads. In *LICS*. IEEE Computer Society, 1989.

[31] B. O'Sullivan. The criterion package (0.8.0.0), 2013. `http://hackage.haskell.org/package/criterion`.

[32] A. S. Patricia Johann and J. Voigtländer. A generic operational metatheory for algebraic effects. In *LICS*. IEEE Computer Society, 2010.

[33] G. D. Plotkin and J. Power. Adequacy for algebraic effects. In *FoSSaCS*. Springer-Verlag, 2001.

[34] G. D. Plotkin and J. Power. Notions of computation determine monads. In *FoSSaCS*. Springer-Verlag, 2002.

[35] G. D. Plotkin and J. Power. Algebraic operations and generic effects. *Appl. Categ. Structures*, 11(1):69–94, 2003.

[36] G. D. Plotkin and M. Pretnar. Handlers of algebraic effects. In *ESOP*. Springer-Verlag, 2009.

[37] D. Rémy. Type inference for records in a natural extension of ML. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[38] T. Schrijvers and B. C. d. S. Oliveira. Monads, zippers and views: virtualizing the monad stack. In *ICFP*. ACM, 2011.

[39] C.-c. Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, 20(4):371–401, 2007.

[40] T. Sheard and S. L. P. Jones. Template meta-programming for Haskell. *SIGPLAN Notices*, 37(12):60–75, 2002.

[41] S. Staton. Two cotensors in one: Presentations of algebraic theories for local state and fresh names. *Electr. Notes Theor. Comput. Sci.*, 249:471–490, 2009.

[42] N. Swamy, N. Guts, D. Leijen, and M. Hicks. Lightweight monadic programming in ML. In *ICFP*. ACM, 2011.

[43] W. Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008.

[44] S. Visscher. The effects package (0.2.2), 2012. `http://hackage.haskell.org/package/effects`.

[45] J. Voigtländer. Asymptotic improvement of computations over free monads. In *MPC*. Springer-Verlag, 2008.

[46] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*. Springer-Verlag, 1995.

A. Performance Results

All performance testing was conducted with the `-O2` compiler flag enabled using a PC with a quad-core Intel i7-3770K CPU running at 3.50GHz CPU and 32GB of RAM, running GHC 7.6.1 on Ubuntu Linux 12.10. We used O'Sullivan's `criterion` library [31] to sample each micro-benchmark ten times.

The code for the micro-benchmarks can be found in the GitHub repository at:

`http://github.com/slindley/effect-handlers/Benchmarks`

A.1 State

As a basic sanity check we tested the following function on state:

$$\begin{array}{l}
count :: SComp\ Int\ Int \\
count = \\
\quad \mathbf{do}\ i \leftarrow get; \\
\qquad \mathbf{if}\ i \equiv 0\ \mathbf{then}\ return\ i \\
\qquad \mathbf{else\ do}\ put\ (i-1); count
\end{array}$$

We used 10^8 as the initial value for the state. We tested implementations using: the state monad, three different versions of standard deep handlers, and one implementation of shallow handlers. As a control, we also tested a pure version of $count$. In each case we used open handlers for interpreting state.

Implementation	Time (ms)	Relative Speed
pure	51	1.00
state monad	51	1.00
handlers (continuations)	77	0.67
handlers (free monad)	5083	0.01
handlers (codensity)	2550	0.02
shallow handlers	5530	0.01

Table 1. State

The results are shown in Table 1. GHC is optimised for monadic programming, so there is no cost to using a state monad over a pure program. The continuation monad implementation runs at two thirds of the speed of the baseline. The free monad implementation is a hundred times slower than the baseline. Using a codensity monad gives a two times speed-up to the free monad implementation. Shallow handlers are implemented using a free monad and are slowest of all.

A.2 Flat Pipes

We tested our pipes implementation using a flat pipeline previously used by the author of the `pipes` library for comparing performance between the `pipes` library and other libraries [13]. The pipeline consists of a producer that yields in turn the integers in the sequence $[1\,..\,n]$, for some n, connected to a consumer that ignores all inputs and loops forever. The `pipes` library optionally takes advantage of GHC rewrite rules to define special code optimisations for `pipes` code. We tested against `pipes` with and without the rewrite rules enabled. The results are shown in Table 2 for $n = 10^8$. The continuation monad implementation is nearly twice as fast as the `pipes` library without rewrite rules enabled. Enabling the rewrite rules makes a significant difference. In this case the `pipes` library is faster than the continuation monad implementation. The free monad and codensity implementations are slower than the `pipes` library, but the differential is much smaller than in the case of the $count$ micro-benchmark. Interestingly, shallow handlers outperform the free monad implementation of standard handlers.

Implementation	Time (ms)	Relative Speed
`pipes` library	3398	1.00
`pipes` library + rewrites	1304	2.61
handlers (continuations)	1820	1.87
handlers (free monad)	6736	0.50
handlers (codensity)	3918	0.87
shallow handlers	5239	0.65

Table 2. Flat Pipes

A.3 Nested Pipes

To test the scalability of handlers we implemented a deeply-nested pipes computation constructed from 2^n sub-pipes, for a range of values of n. The results are shown in Table 5. They are intriguing as the relative performance varies according to the number of sub-pipes. The relative performance is shown graphically in Figure 7. The `pipes` library always out-performs all of our libraries, even with GHC rewrite rules disabled. The continuation monad implementation is relatively constant at around two thirds the speed of the `pipes` library. What is particularly notable is that as the level of nesting increases, the performance of shallow handlers eventually overtakes that of the continuation monad implementation.

One possible reason for the `pipes` library outperforming our continuation monad implementation on nested pipes is that it is in fact based on a free monad, and the implementation takes advantage of the reified representation of computations to optimise the case where an input is forwarded through several pipes.

We are not sure why shallow pipes perform so well on deeply-nested pipes, but suspect it may be due to the simpler definition of pipes for shallow handlers, as compared with that for standard handlers, opening up optimisation opportunities along the lines of those explicitly encoded in the `pipes` library.

We conjecture that the anomalous dip at 2^{11} sub-pipes for the bottom three lines in Figure 7 is due to cache effects.

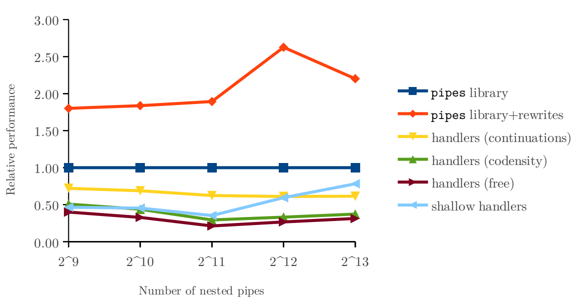

Figure 7. Relative Performance of Nested Pipes

A.4 The n-Queens Problem

To test the performance of handlers that invoke the continuation zero or many times, we implemented the classic n-queens problem in terms of an n-ary *Choose* operation:

$$[\textbf{operation} \mid \forall a. \mathit{Choose} :: [a] \to a \mid]$$

We wrote a handler that returns the first correct solution for the n-queens problem, and tested against a hand-coded n-queens solver. We tested both algorithms with $n = 20$.

Implementation	Time (ms)	Relative Speed
hand-coded	160	1.00
handlers	237	0.67

Table 3. n-Queens

The results are shown in Table 3. The handler version is about two thirds the speed of the hand-coded version.

A.5 Aspect-Oriented Programming

Effect handlers can be used to implement a form of aspect-oriented programming. We tested an expression evaluator taken from Oliveira et al.'s work on monadic mixins [7]. The expression evaluator is extended to output logging information whenever entering or exiting a recursive call, and to output the environment whenever entering a recursive call. We compared a hand-coded evaluator with Oliveira et al.'s mixin-based evaluator and an evaluator implemented using our continuation monad implementation of standard handlers. We tested each evaluator on the same randomly generated expression containing 2^{12} leaf nodes.

Implementation	Time (ms)
hand-coded	6516
mixins	6465
handlers (continuations)	6526

Table 4. Aspect-Oriented Programming

The results are shown in Table 4. The performance is almost identical for each of the three implementations, indicating no abstraction overhead.

2^9 sub-pipes

Implementation	Time (ms)	Relative Speed
`pipes` library	41	1.00
`pipes` library + rewrites	23	1.80
handlers (continuations)	57	0.72
handlers (free monad)	103	0.40
handlers (codensity)	81	0.51
shallow handlers	88	0.47

2^{10} sub-pipes

Implementation	Time (ms)	Relative Speed
`pipes` library	90	1.00
`pipes` library + rewrites	49	1.84
handlers (continuations)	131	0.69
handlers (free monad)	275	0.33
handlers (codensity)	206	0.44
shallow handlers	198	0.45

2^{11} sub-pipes

Implementation	Time (ms)	Relative Speed
`pipes` library	209	1.00
`pipes` library + rewrites	111	1.89
handlers (continuations)	336	0.62
handlers (free monad)	990	0.21
handlers (codensity)	716	0.29
shallow handlers	595	0.35

2^{12} sub-pipes

Implementation	Time (ms)	Relative Speed
`pipes` library	721	1.00
`pipes` library + rewrites	274	2.63
handlers (continuations)	1181	0.61
handlers (free monad)	2701	0.27
handlers (codensity)	2178	0.33
shallow handlers	1216	0.59

2^{13} sub-pipes

Implementation	Time (ms)	Relative Speed
`pipes` library	1931	1.00
`pipes` library + rewrites	877	2.20
handlers (continuations)	3147	0.61
handlers (free monad)	6172	0.31
handlers (codensity)	5188	0.37
shallow handlers	2470	0.78

Table 5. Nested Pipes