Effect Handlers in Haskell, Evidently

Ningning Xie
Microsoft Research
USA
nnxie@cs.hku.hk

Daan Leijen
Microsoft Research
USA
daan@microsoft.com

Abstract

Algebraic effect handlers offer an alternative to monads to incorporate effects in Haskell. In recent work Xie et al. show how to give semantics to effect handlers in terms of plain polymorphic lambda calculus through evidence translation. Besides giving precise semantics, this translation also allows for potentially more efficient implementations. Here we present the first implementation of this technique as a library for effect handlers in Haskell. We show how the design naturally leads to a concise effect interface and how evidence translation enables evaluating tail resumptive operations in-place. We give detailed benchmark results where our library performs well with respect to other approaches.

CCS Concepts: • Software and its engineering → Control structures; Polymorphism.

Keywords: Algebraic Effects, Handlers, Evidence Passing Translation

ACM Reference Format:

1 Introduction

Algebraic effects handlers [Plotkin and Power 2003; Plotkin and Pretnar 2013] provide an alternative to monads to incorporate effectful programs in Haskell [Kammar et al. 2013; Kiselyov and Ishii 2015; Wu and Schrijvers 2015a]. Effect handlers can express any free monad in a concise and composable way, and can be used to express complex control-flow, like exceptions, asynchronous I/O, local state, backtracking, and much more.

In recent work Xie et al. [2020] show how to give semantics to effect handlers in terms of plain polymorphic lambda calculus through evidence translation. Besides giving precise semantics, this translation also allows for potentially more efficient implementations – a handler is now passed as evidence to the call site of an operation where it can be invoked immediately without needing to search for it. Here we present the first implementation of this technique as a library for effect handlers in Haskell. In particular,

• We give an implementation of effect handlers based on the target language $F^\forall$ in [Xie et al. 2020]. This implements effect handler semantics faithfully and in particular enforces the scoped resumptions restriction (although at runtime only).
• The library interface (Figure 1) is concise and arguably simpler than other library interfaces for effect handlers. In particular, effects are defined as a regular data type with a field for each operation. For example,

```haskell
data Reader a e ans = Reader{ ask :: Op () a e ans }
```

declares a Reader effect with one operation ask from () to a (in effect context e with answer type ans). Other libraries typically require GADT’s [Kiselyov and Ishii 2015], data types à la carte [Swierstra 2008; Wu et al. 2014], or Template Haskell [Kammar et al. 2013] to create new effects. Being effect handlers, there are also of course the usual advantages with respect to a monadic interface: effects can be composed freely (as effects always form a free monad), and there is no need to lift operations into a particular monad (as they are all part of the single effect monad).
• Since evidence of each handler is passed explicitly, we can directly invoke operations on a handler. For example, the function greet:

```haskell
greet :: (Reader String [: e) => Eff e String
      greet = do s <- perform ask ()
                return ("hello " ++ s)
```

performs an ask operation. Here the qualified type Reader String [: e ensures the reader effect is in the effect context e and its dictionary allows perform to directly select the actual Reader handler from the effect context evidence (passed in the effect monad Eff e) without needing to search for the correct handler. It then uses ask to select the operation field directly from the handler data type and invokes it. This is quite different from most effect libraries that typically propagate the operations through a handler stack. Moreover,
since the evidence supplies the actual handler instance, we can optimize \textit{tail-resumptive} operations to evaluate \textit{in-place} without needing to capture a continuation (Section 5.3). Since handlers and operations are regular datatypes, often the compiler can deduce the particular operation as well and inline the definition.

- The performance of the library is quite good. We give detailed benchmark results in Section 6, comparing its performance with the monad transformer library and other Haskell libraries. We show when combining multiple effects our library tends to outperform monads and alternative effect libraries.
- The implementation follows the translation in \cite{xie2020reader} closely where we separate the implementation of evidence passing (Section 5.2), from the underlying monad for multi-prompt delimited control (Section 5.1). Our implementation of multi-prompt delimited control is type safe except for the generation of unique markers. The technique of unique markers has been done before \cite{dyvig2007} but we believe our implementation is particularly concise.
- Finally, we also show how we can efficiently represent \textit{local isolated state} without needing higher ranked types through handler hiding (Section 4.3 and 5.7).

The source code of our implementation and benchmarks is available as an artifact and online at https://github.com/xnning/eveff.

## 2 A Tour of Reader

Let’s start with an in-depth example of dynamic binding, also known as the \textit{reader} effect.

Using evidence based effect handlers is straightforward – in contrast to monads, there is no need to create classes or instances to introduce a new effect. Moreover, the focus is on the \textit{operations} of the effect, not the plumbing (in the form of return and \texttt{(>>=)} for monads). As shown in the introduction, to define new effects, we just declare a new data type:

\begin{verbatim}
data Reader a e ans = Reader( ask :: Op () a e ans )
\end{verbatim}

The data declaration defines a new effect \texttt{Reader} with three type parameters: \texttt{a} for the reader value, and \texttt{e} and \texttt{ans} for the effect context and answer type where the reader is handled. The \texttt{e} and \texttt{ans} are always present in effect declarations and will be explained in detail later. The \texttt{Reader} “effect” is just a data type where its constructor \texttt{Reader} has a single \textit{operation} field \texttt{ask} with type \texttt{Op () a e ans} which denotes operations from type \texttt{(e)} to \texttt{a} (in an effect context \texttt{e} with answer type \texttt{ans}).

To perform operations, we pass the operation selector \texttt{ask} with its argument to the \texttt{perform} function. In the introduction we showed a greeting program which asks for a \texttt{String} and returns the greeting message:

\begin{verbatim}
greet = do s <- perform ask ()
          return ("hello " ++ s)
\end{verbatim}

Effectful computations run in the \texttt{Eff e a} monad where \texttt{e} is the effect context and \texttt{a} the result type. The type class constraint \texttt{h} :: \texttt{e ans} indicates that effect handler \texttt{h} is a member of the effect context \texttt{e} and can thus be used to perform operations. In this case, \texttt{perform ask ()} returns a value \texttt{s} of type \texttt{String}, and thus \texttt{greet} is qualified with a string reader effect, \texttt{Reader String :: e}. Operations are given semantics by their handlers. Handlers are defined as instances of effect datatypes, giving implementations for each operation. We can handle a reader effect in an action using the \texttt{handler} function:

\begin{verbatim}
reader :: Eff (Reader String :: e) ans -> Eff e ans
reader action
  = handler (Reader{ ask = value "world" }) action
\end{verbatim}

The first argument to \texttt{handler} is a concrete \texttt{Reader} data type with an implementation of the ask operation. In this case we use \texttt{value \textit{a} \textit{Op () a e ans}} to always resolve with a constant value "world" when \texttt{ask} is performed. The value function is just one way to define operations but there are various other ways which we discuss in Section 3.4.

The reader action has type \texttt{Eff (Reader String :: e) ans}, where the type \texttt{h :: e ans} represents the effect context as a type level \texttt{list}, which has effect \texttt{h} as the head and effect context \texttt{e} as the tail. We can view \texttt{handler} as an elimination rule where the \texttt{Reader String} effect is eliminated from the action context, resulting in \texttt{Eff e ans}, (i.e., \textit{it handles the effect}).

Since action has \texttt{Reader} in its effect context, it can perform operations in the \texttt{Reader} effect – in particular, we can evaluate greet under our \texttt{reader} handler, where the constraint \texttt{Reader a :: e ans} is satisfied:

\begin{verbatim}
helloWorld :: Eff e String
helloWorld = reader greet
\end{verbatim}

Finally, we can run effectful computations using \texttt{runEff} with type \texttt{Eff () a \rightarrow a}:

\begin{verbatim}
> runEff helloWorld
"hello world"
\end{verbatim}

The empty effect context in \texttt{Eff () a} ensures that we can only run effect computations where all effects have been handled.

## 3 Overview of the Effect Interface

Before we present other common examples, we show a short overview of the full library interface as defined in Figure 1.

### 3.1 Defining Effects

We have seen that effects are simply datatypes of a particular shape. In general, an effect type declaration has the general form

\begin{verbatim}
data Effect a₁ ... aₙ e ans
  = Effect ( op₁ :: forall x₁ ... xₙ. Op b₁ c₁ e ans
              , ...
\end{verbatim}
data Eff e a, instance Monad (Eff e) -- effect monad in an effect context e
data Op a b e ans -- operation from a to b (in a context e with answer type ans)
data (h :* e) -- handler h in front of e
class (h :? e) -- is handler h in e?

runEff :: Eff () a -> a
perform :: (h :? e) => (forall e1 ans1. h e1 ans1 -> Op a b e ans) -> a -> Eff e b

value :: a -> Op () a e ans -- value operation
function :: (a -> Eff e b) -> Op a b e ans -- function operation
operation :: (a -> (b -> Eff e ans) -> Eff e ans) -> Op a b e ans -- general operation with resumption

handler :: h e ans -> Eff (h :* e) ans -> Eff e ans -- handle h in an action
handlerRet :: (a -> ans) -> h e ans -> Eff (h :* e) a -> Eff e ans

data Local a e ans = Local{ lget :: Op () a e ans, lput :: Op a () e ans } -- local state effect
handlerLocal :: a -> h (Local a :* e) ans -> Eff (h :* e) ans -> Eff e ans
handlerLocalRet :: a -> (b -> a -> ans) -> h (Local a :* e) ans -> Eff (h :* e) b -> Eff e ans

Fig. 1. The Control.Ev.Eff library interface

value x = function (\ () -> return x)

-- operation :: a -> (b -> Eff e ans) -> Eff e ans -> Op a b e ans. This is the most general way to create an operation. It takes a function that expects two arguments: the operation argument of type a and a resumption of type b to Eff e ans. An operation does not return like function but instead returns directly from the handler. However, it can explicitly call the resumption to resume to the original call-site with a result of type b (and possibly multiple times). We can again define the ask operation in terms of an operation as well:

Reader( ask = operation (\ () k -> k "world") )
and more generally:

function f = operation (\ x k -> k (f x))

We call operations that are defined using function (or value) tail-resumptive. Using function (or value) does not only provide a more concise way for users to define operation implementations, it also has a much more efficient implementation where operations can evaluate in place without needing to capture a continuation. We will discuss this further in Section 5.3.

3.4 Defining Handlers

Handlers are instances of the effect datatypes, providing concrete implementations for operations. As these are just normal datatype values, we can take them as inputs, return them as results, or assign them to variables. There are four common ways to define a handler:

• handler :: h e ans -> Eff (h :* e) ans -> Eff e ans. The handler function takes a concrete handler h e ans and an action, and installs the handler to handle any
operations that are performed in the action. The effect context $h : e$ of the action signifies that $h$ is the top handler in the effect context of the action. Handling an effect $h$ eliminates $h$ from the effect context $\text{Eff}(h : e) \rightarrow \text{Eff}(e)$. Here we also see that we use a partially applied type $h$ in the effect context $h : e$: the fully applied type $h : e : e'$ represents a concrete implementation of a handler. In contrast, a partially applied $h$ is abstract in the effect context and answer type where the actual handler is defined, and can thus be seen as an effect type (i.e. the abstract interface we can use).

- **handlerRet** $h$ takes an extra argument to transform the result type of the action from $a$ to $\text{ans}$, where the answer type of the operations in $h$ is now $\text{ans}$. We will see an example of it in Section 4.1.

- **handlerLocal**. The $\text{handlerLocal}$ function provides locally isolated state to the operations in handler $h$. As apparent from the type, this local state is not exposed outside the operations of the handler. The local effect has two operations $\text{lget}$ and $\text{lput}$ to get and set the local state. We will see how to use it to define the state effect in Section 4.2.

- **handlerLocalRet** is a combination of $\text{handlerRet}$ and $\text{handlerLocal}$ where one can transform the result of an action given the local handler state.

4 Examples

So far we have described the interface of our library and we have seen how to define the reader effect and its handler. Now we move on to more complex examples.

4.1 Exception

The exception effect $\text{Exn}$ has one operation $\text{failure}$ with type $\text{Op}() \rightarrow a : e : \text{ans}$ for any $a$.

```haskell
data Exn e ans = Exn { failure :: forall a. Op () a e ans }
```

Here we see the use of a rank-2 type with $\text{forall a}$ to define an operation that is polymorphic in the result type. Using $\text{Exn}$, we can define partial functions in a type-safe way.

```haskell
safeDiv :: (Exn :? e) \Rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Eff}(\text{e}) \text{Int}
safeDiv x 0 = \text{perform} \text{failure}()
safeDiv x y = \text{return}(x \div y)
```

Exception handlers are special kinds of handlers that abort the execution, by discarding the current resumption. There are different ways to handle exceptions. The following implementation reifies the exceptions effect to the $\text{Maybe}$ monad:

```haskell
toMaybe :: \text{Eff}(\text{Exn} : e) \Rightarrow \text{Eff}(\text{e}) \rightarrow \text{Maybe(a)}
toMaybe
  = \text{handlerRet} \ \text{Just}\$ \ \text{Exn}(
      \text{failure} = \text{operation}(\text{\lambda } x \rightarrow \text{return} \ \text{Nothing})
    )
```

Here by partial application we leave out the function argument, which is an effectful computation $\text{Eff}(\text{Exn} : e) \rightarrow \text{Maybe(a)}$ with the answer type $a$. Note that we use $\text{handlerRet}$ to attach $\text{Just}$ to the program result if it returns normally, which, consistent with the result $\text{Nothing}$ if the program fails, turns the computation result type to $\text{Maybe(a)}$.

```haskell
> runEff (\\text{toMaybe } \text{\$ } \text{safeDiv} \ 42 \ 2)
\text{Just} 21
> runEff (\\text{toMaybe } \text{\$ } \text{safeDiv} \ 42 \ 0)
\text{Nothing}
```

Another common implementation is to return a default value when the computation fails:

```haskell
\text{exceptDefault} :: a \rightarrow \text{Eff}(\text{Exn} : e) \Rightarrow a \rightarrow \text{Eff}(e)\ a
\text{exceptDefault} \ x
  = \text{handler}\$
    \text{Exn}(\text{failure} = \text{operation}(\text{\lambda } x \rightarrow \text{return} \ x))
```

We can run it in the same examples:

```haskell
> runEff (\text{exceptDefault} \ 0 \ \text{safeDiv} \ 42 \ 2)
21
> runEff (\text{exceptDefault} \ 0 \ \text{safeDiv} \ 42 \ 0)
0
```

4.2 State

The state effect comes with an operation $\text{get}$ for reading the state value, and $\text{put}$ for setting the state value.

```haskell
data State a e ans = State{ get = \text{Op}() a e ans } , put = \text{Op} a () e ans }
```

For example, we can get a boolean state value and invert it:

```haskell
\text{invert} :: (\text{State} \text{Bool} :? e) \Rightarrow \text{Eff} e \text{Bool}
\text{invert} = \text{do} \ b \leftarrow \text{perform get}()
  \text{perform put}(\text{not b})
  \text{perform get}()
```

The definition of state takes an initial value of the state, and uses $\text{handlerLocal}$ for handling.

```haskell
\text{state} :: a \rightarrow \text{Eff}(\text{State} a : e) \Rightarrow a \rightarrow \text{Eff} e\ a
\text{state init}
  = \text{handlerLocal init}$
    \text{State}(\text{get} = \text{function}(\text{\lambda } x \rightarrow \text{perform lget}())
      , \text{put} = \text{function}(\text{\lambda } x \rightarrow \text{perform lput} x))
```

The function $\text{handlerLocal}$ offers a local variable, together with two operations $\text{lget}$ and $\text{lput}$ that get and set the local variable (Figure 1). The first argument of $\text{handlerLocal}$ is the initial value and the second argument is the handler implementation whose operations have $\text{Local} a$ in the effect context (and can thus use $\text{lget}$ and $\text{lput}$). In this case, get and put directly correspond to the local state operations.

```haskell
> runEff (\text{state} \text{True} \text{invert})
False
```

4.3 State as a Function

Of course, we cheated a bit in the state example by using the builtin $\text{Local}$ state. However, we can implement the local state from scratch as well, where we use the usual monadic representation of state as a function from the current state to a final result: $s \rightarrow \text{Eff} e\ a$ [Kammar and Pretnar 2017]:

```haskell
invert :: \text{State} \text{Bool} :? e \Rightarrow \text{Eff} e \text{Bool}
```
4.4 Ambiguity

We have seen handlers that resume computations once (e.g., handlerHide where f), and handlers that abort computations (e.g., exceptDefault) more than once.

In this example, we show handlers that resume computations of type a → Eff e a ans, and that function also isolates the local state to just the operations in the handler, and not outside that. This is done using the more primitive handlerHide function as:

```
handlerLocal :: a → (h (Local a :* e) ans) → Eff (h :* e) ans
handlerLocal init h action
    = do f <- handler (Local
        (flip = operation (\ () k → return $
            \s → do (r < k s; r s))
        ,lput = operation (\ s k → return $
            \_ → do (r < k (); r s))
    )
        (do x ← action
            return (\s → return x))
f init
```

where f has type a → Eff e ans, and the resumption k has type a → Eff e (a → Eff e ans). The builtin Local state has exactly the same semantics as defined here, but is implemented internally more efficiently using an STRef [Peyton Jones and Launchbury 1995].

This is still not quite the same as handlerLocal though, as that function also isolates the local state to just the operations in the handler, and not outside that. This is done using the more primitive handlerHide function as:

```
handlerLocal :: a → (h (Local a :* e) ans) → Eff (h :* e) ans
handlerLocal init h action
    = do f <- handler (handlerHide h action)
        local init (handlerHide h action)
where handlerHide hides the top most handler h0 (which is Local a in our example):

```
handlerHide :: h (h0 :* e) ans → Eff (h0 :* e) ans → Eff (h0 :* e) ans
```

giving locally isolated state without needing rank-2 types! [Peyton Jones and Launchbury 1995] We discuss the implementation of handler hiding further in Section 5.7.

4.5 Parser

As a larger example, we show how to implement a parser combinator. A similar example has been shown in [Wu et al. 2014] and [Leijen 2017]. Both of them defined one effect type (called Nondet and Many respectively) for handling exceptions and ambiguity at the same time. Here we do it in a slightly different way: we simply reuse the existing Exn and Amb effects.

With Amb we can non-deterministically choose from two computations:

```
choice :: (Amb :? e) ⇒ Eff e a → Eff e a → Eff e a
choice p1 p2 = do b ← perform flip ()
    if b then p1 else p2
```

We define the parser combinator many that parses zero or more p parsers, and many1, which parses one or more p parsers:

```
many :: (Amb :? e) ⇒ Eff e a → Eff e a
many p = choice (many1 p) (return [])
many1 :: (Amb :? e) ⇒ Eff e a → Eff e a
many1 p = do x ← p; xs ← many p; return (x:xs)
```

For parsing, we define the Parser effect with a satisfy operation to test if the current input satisfies a predicate:

```
data Parse e ans = Parse {
    satisfy :: forall a.
        Op (String → (Maybe (a, String))) a e ans
}
```

A handler for Parser gets a local variable to keep track of the current input string, and applies the predicate to the input. If the predicate is satisfied, it then applies the resumption to the result, or otherwise it fails.

```
parse :: (Exn :? e) ⇒
    String → Eff (Parser :* e) b → Eff e (b, String)
parse input
    = handlerLocalRet input (\s → (x, s)) $
```

This handler transforms computations with a single result a to a computation with a list of all possible results a. Note that the result type of the resumption function k is also Eff e [a] and that we need to use a return clause (\x → [x]) to convert the results to singleton lists (which are appended inside flip).

Or, we can implement backtracking by first trying True, and if it fails, we try False:

```
firstResult :: Eff (Amb :* e) (Maybe a) → Eff e (Maybe a)
firstResult = handler Amb
    (flip = operation (\ () k →
        do xs ← k True
case xs of
            Just _ → return xs
            Nothing → k False)
```
We can also change the parse strategy to get only the first parse result without touching the parse implementation:

eager = perform (firstResult (toMaybe action))

We have seen that we can handle `Parse`, so it transforms the effect from `Parse :: e` to `e`; in the meantime parse itself performs failure, so it requires `Exn :: e`.

Now we can define basic parsers with predicates for symbols and digits:

digit = perform (satisfy (
  t input \rightarrow case input of
    Nothing \rightarrow perform failure ()
    Just (x, rest) \rightarrow do perform input rest
  ) x)

We can use these parsers to build parsers for simple arithmetic expressions:

expr = choice (do i <- term; symbol ';'; j <- term return (i + j))

term = choice (do i <- factor; symbol ';'; j <- factor

factor = choice (do symbol '('; i <- expr; symbol ')'

number = do xs <- many; digit return $ foldl (\n d -> **10** * n + d) 0 xs

We have seen that we can handle `Exn` using `toMaybe`, and handle `Amb` using `allResults` or `firstResult`. Combining those handlers, we give one parse strategy that gets all possible parse results:

solutions = perform (allResults (toMaybe action))

We can also change the parse strategy to get only the first parse result without touching the parse implementation:

eager = perform (firstResult (toMaybe action))

5 Implementation

Our effect implementation is directly based on the target language *F* in the formal evidence translation described by Xie et al. [2020] (Section 5, Figure 9). Their translation has two parts: an implementation of multi-prompt control combined with the propagation of evidence in the form of the effect context. Our library is structured like this where we build upon a generic implementation of a multi-prompt control monad. Embedding delimited control in Haskell has been investigated in much detail [Dyvbig et al. 2007], but we believe our implementation is particularly concise and closely based on the definition given by Xie et al. [2020].

5.1 Multi-prompt Control

First, we define a control monad `Ctl` for multi-prompt delimited continuations:

data Ctl a = Pure { result :: a }

Yield { marker :: Marker ans,

op :: \(b \rightarrow Ctl b\) \rightarrow Ctl ans,

cont :: \(b \rightarrow Ctl a\) \rightarrow Ctl ans,

return = Pure { result :: a }

yield m op = Yield m op Pure

We start with an identity continuation (as `Pure`). When yielding to a prompt, `Yield` stores three components: (1) a unique marker indicating the particular prompt to which it yields, with the answer type `ans` for type-safety (we discuss this shortly); (2) an operation implementation `op` that is already partially applied to its operation argument, so it just needs a resumption (`b \rightarrow Ctl ans`) to evaluate its action; and (3) the partially built up continuation `cont`. Since `cont` is partially built, its result type `a` can be different from the final answer type `ans`. Once the right prompt is reached, the markers will ensure that at that point `a` will be equal to `ans`. The `yield` function is a thin wrapper around `Yield`:

\[
\text{yield} :: \text{Marker} \rightarrow \text{Marker} \rightarrow \text{Yield} \rightarrow \text{Marker}
\]

instance Monad Ctl where

return x = Pure x -- just Pure

(Pure x >>= f) = f x -- pass on the result

(Yield m op cont >>= f) = Yield m op (f \{ kcompose \} cont) -- keep yielding

For efficiency in our library, we expand the definition of `(>>=)` in `kcompose` to break the mutual dependency. When
binding a \texttt{Yield}, we keep yielding but extend the continuation with the continuation of the bind \texttt{f}.

When we install a \texttt{prompt}, we use an internal \texttt{freshMarker} function to generate a \texttt{unique} marker as the name of the current prompt:

\begin{verbatim}
prompt :: (Marker ans \rightarrow Ctl ans) \rightarrow Ctl ans

prompt action
    = freshMarker (\m \rightarrow mprompt m (action m))
\end{verbatim}

The prompt function connects its \texttt{answer type \texttt{ans}} to its marker with type \texttt{Marker \texttt{ans}}. It passes the marker on to \texttt{mprompt} which checks if it is yielded to:

\begin{verbatim}
mprompt :: Marker a \rightarrow Ctl a \rightarrow Ctl a

mprompt m (Pure x) = Pure x

mprompt m (Yield n op cont')
    = let cont' x = mprompt m (cont x) in
        case \texttt{mmatch} m n of
            Nothing \rightarrow \texttt{Yield} n op cont' -- keep yielding
            \texttt{Just} \texttt{Refl} \rightarrow op cont'
\end{verbatim}

In the case of \texttt{Pure}, the action returns a value \texttt{x}, and prompt simply passes on the value. In the case of \texttt{Yield}, the action yields to a prompt with marker \texttt{n}, with the operation implementation \texttt{op} and the current continuation \texttt{cont}. In this case, it first extends the continuation with our own prompt getting a new continuation \texttt{cont'}. Now we need to decide if it is yielding to this particular prompt, by checking whether \texttt{m} is equivalent to \texttt{n}. The function \texttt{mmatch} checks the equality of two markers. If two markers are not equal (\texttt{Nothing}), then we keep yielding with \texttt{cont'}. When they are equal (\texttt{Just}), we are the target of the yield, and our continuation \texttt{cont'} is exactly the \texttt{resumption} that \texttt{op} expects, and we directly apply \texttt{op cont'}.

Note that the \texttt{resumption} \texttt{cont'} resumes under its own \texttt{mprompt} again, and thus we implement deep handlers (as opposed to \texttt{shallow} handlers [Kammar et al. 2013]).

However, the astute reader may have noticed that \texttt{op cont'} cannot type-check in the \texttt{Yield} definition, \texttt{op} requires an existential \texttt{answer type \texttt{ans}}, while both \texttt{cont} and \texttt{cont'} have the \texttt{answer type \texttt{a}}. The key to ensure type-safety is the markers. Markers have two important properties: they are unique, and they carry their \texttt{answer type \texttt{a}}.

One of the key properties of markers is that they have the \texttt{answer type \texttt{a}} which is unique, and they carry their \texttt{answer type \texttt{a}}. In this case, \texttt{m} is \texttt{Marker \texttt{a}} and \texttt{n} is \texttt{Marker \texttt{ans}}. When \texttt{mmatch} confirms that two markers are equal, it also returns the proof term \texttt{Refl} that asserts to the compiler that these two types are equal via a type equality \texttt{a \equiv \texttt{ans}}. Once we have this type equality in the environment, the application \texttt{op cont'} type checks. The creation of markers \texttt{freshMarker} and the equality check with \texttt{mmatch} requires unsafe primitives (the full implementation is in the anonymous supplement).

### 5.2 Evidence Passing Effects

Now that we have a control monad \texttt{Ctl} for multi-prompt control, we can proceed to implement the evidence passing effect monad \texttt{Eff \texttt{e a}}. This is simply a reader monad from the current \texttt{effect context Context \texttt{e}} to a computation \texttt{Ctl \texttt{a}}:

\begin{verbatim}
newtype Eff e a = Eff (Context e \rightarrow Ctl a)

instance Monad (Eff e) where
    return x = Eff (\ctx \rightarrow pure x)
    (Eff eff) >>= f = Eff (\ctx \rightarrow do \texttt{ctl} \leftarrow eff \texttt{ctx}
        \under ctx (f \texttt{ctl}))
\end{verbatim}

The effect context is a \texttt{heterogeneous list} of pairs of markers and a corresponding handler, where the top handler is at the head of the list. We represent the context using a GADT [Xi et al. 2003]:

\begin{verbatim}
data Context e where
    CNil :: Context ()
    CCons :: Marker ans \rightarrow h e ans \rightarrow
        Context e \rightarrow Context (h :* e)
\end{verbatim}

The \texttt{CNil} constructor is an empty context, while \texttt{CCons} takes a marker, a handler, and the tail of the context. The type constructor (\texttt{:*}) is a phantom datatype [Leijen and Meijer 2000] and has no runtime representation:

\begin{verbatim}
data (h :: * \rightarrow * \rightarrow *) :* e
\end{verbatim}

The \texttt{:*} type constructor is only used to maintain the correspondence between the effect type \texttt{e} and the runtime context \texttt{Context \texttt{e}}. In particular, we can only \texttt{CCons} a \texttt{handler \texttt{h e ans}} in front of a \texttt{Context \texttt{e}} where both have the same effect context \texttt{e}, and the resulting runtime context is now typed as \texttt{Context (h :* e)}. This is reflected in the definition for \texttt{handler} as:

\begin{verbatim}
handler :: \texttt{h e ans} \rightarrow \texttt{Eff (h :* e) ans} \rightarrow \texttt{Eff e ans}

handler \texttt{h action}
    = \texttt{Eff $ \ctx \rightarrow}
        \texttt{prompt (\m \rightarrow \under (CCons m h ctx) action)}
\end{verbatim}

which first installs a \texttt{prompt} for our \texttt{handler} (so it can be yielded to), and then \texttt{CCons} the \texttt{resulting marker \texttt{m} and handler \texttt{h in front of the current context}}, and evaluates \texttt{action} under that new context. As such, the \texttt{effect context \texttt{e}} in \texttt{Eff \texttt{e a}} always reflects the actual \texttt{order} of the \texttt{installed handlers} at runtime.

The dual of handling is masking [Biernacki et al. 2017; Convent et al. 2020; Leijen 2014; Wu et al. 2014], where we remove the top \texttt{handler} on a sub-computation \texttt{eff}:

\begin{verbatim}
mask :: \texttt{Eff e ans} \rightarrow \texttt{Eff (h :* e) ans}

mask eff = \texttt{Eff \texttt{\texttt{\}} (CCons m h ctx) \rightarrow \under ctx eff)}
\end{verbatim}

Note that the \texttt{type \texttt{Eff (h :* e) ans}} guarantees that the match on \texttt{CCons} never fails.

We call the pair of a marker \texttt{m} and a handler \texttt{h} the \texttt{evidence}, where a \texttt{handler} can be uniquely identified by the \texttt{marker}. So a context is a list of evidence (called \texttt{evidence vector} in [Xie et al. 2020]). Returning to the \texttt{Eff} definition, we see that our \texttt{effectful computations} are \texttt{evidence passing}. That is, when evaluating \texttt{effectful computations}, we pass the current \texttt{evidence} down into the computation.
5.3 Perform

To perform an operation, we need to be able to select the correct handler from the effect context. The $(?:)$ type class constraint exposes the subContext function to do exactly that:

```haskell
class (h :? e) where
    subContext :: Context e -> SubContext h
```

data SubContext h
    = forall e. SubContext (Context (h :* e))

We define the instances for subContext further on, but it selects a tail of the context where $h$ is the current head. We can use this method to concisely define perform:

```haskell
perform :: (h :? e) =>
    forall e' ans. h e' ans -> Op a b e' ans) 
    -> a -> Eff e b

perform selectOp x
    = Eff (
        \ctx ->
        case subContext ctx of
            SubContext (CCons m h ctx')
                -> case (selectOp h)
                    of
                        Op f -> f m ctx' x)
```

Here we apply subContext to the effect context $ctx$ to select our handler context as $\text{CCons} \ m \ h \ ctx'$. So we know the handler is $h$, its prompt marker is $m$, and it was itself defined under an effect context $ctx'$. First we select the desired operation from the handler with selectOp. This function needs a higher-rank type [Leijen 2008; Peyton Jones et al. 2007] in the effect context $e'$ and answer type $ans$ as the effect context and answer type of the sub context are abstract (and existentially quantified), and therefore selectOp must be polymorphic with respect to those.

We define an operation $\text{Op}$ as:

```haskell
data Op a b e ans
    = Op (Marker ans -> Context e -> a -> Ctl b)
```

i.e., just a function that gets the marker, the handler context, and the argument to the operation. For example, we can define function as:

```haskell
function :: (a -> Eff e b) -> Op a b e ans
function f = Op (\m ctx x -> under ctx (f x))
```

Here we see the advantage of explicit evidence passing: since the handler is passed down to the call site of perform, we can skip an expensive yield back to the handler but directly evaluate $f$ in-place under the handler context $ctx$ (and ignoring the marker $m$). Since all data is explicit, GHC is often able to fully inline the definition of a function operation completely using regular compiler optimizations.

For a general operation though, we need to yield back to the handler. Still, this is efficient as we know the exact marker $m$ to yield to:

```haskell
operation :: (a -> (b -> Eff e ans) -> Eff e ans) 
    -> Op a b e ans
operation f
    = Op \m ctx x 
```

As shown in Section 5.1, the yield $m \ op$ expression yields up to the prompt marked as $m$ building up a continuation which is passed to $op$ at that point. Here we receive the resumption as $ctk$. We may think we can pass this directly to $f$ (i.e., under $ctx$ ($f \ x \ ctk$)) but there is one more step: to maintain a proper correspondence to effect handler semantics, the evidence passing translation of effects requires that all resumptions are scoped resumptions. That is: a resumption can only be resumed in the same evidence context as the context of the handler. Therefore, we need to guard the way the resumption is used, by requiring the context in which it was captured, $ctx$, and the context in which it was resumed, $ctx'$, are equivalent. Xie et al. [2020] prove that this check is sufficient to maintain coherent semantics.

```haskell
yield m $ \ctlk ->
    let k y = Eff (\ctx' ->
        guard ctx ctx' ctk y)
    in under ctx (f x k)
```

As argued by Xie et al. [2020], all important effect handlers in practice can be defined in terms of scoped resumptions (including all examples in this paper), and they argue little expressiveness is lost. They also show that guaranteeing scoped resumptions prevents the construction of difficult to reason about programs where handlers can change the behaviour of other handlers even if defined orthogonally. In particular, they present the follow `evil` example which exhibits this behaviour. Suppose we have the following program that uses a reader effect together with an as yet unknown `Evil` effect:

```haskell
ebody :: (Reader Int :? e, Evil :? e) => Eff e Int
ebody = do x <- perform ask ()
    -> x == 1
    perform evil ()
    y <- perform ask ()
    -> y == 2
    return (x+y)
```

We may reason that, if we evaluate this under the previously defined `reader` handler, that the two ask operations return the same value. But such is not always the case if non-scoped resumptions are allowed. With non-scoped resumptions we can define a handler for `Evil` that is fully orthogonal to the well behaved reader handler, but can still change what value ask returns:

```haskell
data Evil e ans = Evil { evil :: Op () () e ans }
```

```haskell
hevil :: Eff (Evil :* e) a -> Eff e () -> Eff e a
hevil = handlerRet (\x -> (\_ -> return x)) (Evil( 
    evil = operation \_ k ->
        return \_ \_ do f <- k (); f ()
))
```
The `hevil` handler captures the current resumption `k` under a lambda, and returns that directly as the result. This lambda can now be used to resume again but this time under a new reader handler:

```haskell
nonscoped :: Eff e Int
nonscoped = do f <- reader (1::Int) (hevil ebody)
             reader (2::Int) (f ())
```

In our library, nonscoped fails at runtime in the `guard` with a non-scoped resumption section since the resumption `k` that was captured in `hevil` is resumed under a different handler context. We refer the reader to Xie et al. [2020] for further details and examples.

### 5.5 Selecting a Sub Context

We still need to define an instance for `subContext`. In principle, we would like to simply define two instances: one where the head of the context matches the handler type, and one where we need to recurse into the tail:

```haskell
instance (h :? (h :* e)) where
  subContextEq (CCons m h ctx) = subContext ctx

instance (h :? e) where
  subContextEq (CCons _ _ ctx) = subContextEq (CCons _ _ ctx)
```

Unfortunately, this does not quite work as we cannot directly express the type inequality `h /= h'` and thus the instances overlap (and need further extensions as well). However, this is a known problem in Haskell [Kiselyov et al. 2004] and many solutions have emerged over the years. Recently, the use of type families [Schrijvers et al. 2008] to declare the (in)equality of types helps a lot for type inference and we use that technique here [Eisenberg 2012; Eisenberg and Weirich 2012; Xia 2018]. We use the function `HEqual` as a type-level equality function for handlers:

```haskell
type family HEqual (h1 :: *) (h2 :: *) where
  HEqual h1 h2 = 'True
  HEqual h1 h2 = 'False
```

The type function `HEqual` returns `true` only if the handler types `h1` and `h2` are equivalent. Datatype promotion (`'`) [Yorgey et al. 2012] lifts the boolean values to the type-level.

We also use a helper class `InEq` with a single method `subContextEq`. Its instances depend on whether `h1`, the handler we are looking for, and `h2`, the current head of the context, are equivalent.

```haskell
class (heq ~ HEqual h1 h2) => InEq heq h1 h2 e where
  subContextEq :: Context (h2 :* e) -> SubContext h1
```

We define a single instance for `(?:)` in terms of `subContextEq`:

```haskell
instance (InEq (HEqual h1 h2) h1 h2 e) =>
  (h1 :: ? (h2 :* e)) where
  subContext = subContextEq
```

There are now two instances for `InEq` that distinguish whether `h1` and `h2` are equivalent:

```haskell
instance (h1 ~ h2) => InEq 'True h1 h2 e where
  subContextEq (CCons m h ctxx) = SubContext m h ctxx

instance ('False ~ HEqual h1 h2, h1 :: ? e) =>
  InEq 'False h1 h2 e where
  subContextEq (CCons _ _ ctx) = subContext ctx
```

Tricky, but it works well with type inference in our experience since there are no overlapping instances. Nevertheless, having proper row-types [Gaster and Jones 1996; Leijen 2005] would still be preferable in our opinion to this complex encoding.

### 5.6 Return Clauses

We have shown the implementation of `handlerRet` in Section 5.2. The definition of `handlerRet` is simply built upon `handler`:

```haskell
handlerRet :: (ans -> a) -> h e a ->
            Eff (h :*: e) ans -> Eff e a
```

```haskell
handlerRet h action
= handler h (do x <- action; return (ret x))
```

However, this form of return clause is more restricted than allowed by the full effect handler semantics [Xie et al. 2020], as a return clause can perform operations and should have type `ans -> Eff e a`. However, we can `not` define this as:

```haskell
handlerRet ret h action
= handler h (do x <- action; ret x)
```

as that would cause any operation in `ret` to be potentially handled by the handler `h` itself! The correct definition needs to use mask to mask out the current handler [Biernacki et al. 2017; Leijen 2018, Section 4.2]:

```haskell
handlerRetEff :: (ans -> Eff e a) -> h e a ->
               Eff (h :*: e) ans -> Eff e a
```

```haskell
handlerRetEff ret h action
= handler h (do x <- action; mask (ret x))
```

Nevertheless, the generalized definition is almost never needed in practice and most of the time `handlerRetEff` suffices.

### 5.7 Handler Isolation

As shown in Section 4.3, we used the function `handlerHide` to "hide" a handler and only make it available locally. This proved essential to implement local state isolation. The type of `handlerHide` is:

```haskell
handlerHide :: h (h g :*: e) ans -> Eff (h :*: e) ans ->
            Eff (h g :*: e) ans
```

where the top most handler `h g` is only made visible to the operations in handler `h`, but not to the action it handles (of type `Eff (h :*: e) ans`). Unfortunately, we cannot implement this directly: since the context type of the action is `h :*: e`, we have a concrete context of the form `CCons m h ctx` where `ctx :: Context e` and `h :: h e ans` – which does not match the required `h (h g :*: e) ans`. We like to hide the `h g` handler, but need to push it back on the context `e` before we can handle any of its operations. To do this, we extend the context datatype with a `context transformer`:

```haskell
data Context e where
```
The context transformer is a function of type `Context e` to `Context e'` and we can now use handlers with context `e'` (instead of `e`). Usually, the transformer is the identity function where `e = e':`

```
handler :: h e ans → Eff (h ::* e) ans → Eff e ans
```

Now, we can use the context transformer to implement `handlerHide` where we transform the evidence context by `handler h action` just before handling its operations:

```
handlerHide :: h (h0 ::* e) ans → Eff (h0 ::* e) ans → Eff (h0 ::* e') ans
```

In the actual implementation we do not use a function for the context transformer but instead represent it explicitly as a GADT. This is done to improve compiler optimizations where the explicit constructors allow better inlining. We also believe that the context transformers are essential to implement versions of `mask` and `handlerHide` that are not restricted to the top handler, but we leave this to future work.

6.1 Counter

As a basic check, we use the counter benchmark [Kammar et al. 2013; Kiselyov and Ishii 2015] which recursively counts down, with `10^7` as the initial value for the state.

```
runCount :: (State Int :? e) ⇒ Eff e Int
runCount = do i <- perform get ()
  if (i==0) then return i
  else do perform put (i - 1)
       runCount
```

The pure implementation of the counter is simply a tight loop for counting down. The results are given in Figure 2b. The Pure, MTL, and FE versions are all fully inlined and recur directly over a decreasing parameter. Here we can see that the state monad is highly optimized in GHC, and that the build rules in FE are triggered. Our EV implementation is about 5.5 times slower than those. However, as it uses internally an `STRef` for the local state and it performs very close to a plain `runST` implementation, it is close to optimal (and only limited by the performance of updateable references in GHC). EV is respectively 7 and 18 times faster than EE and EV NT. The EV NT is a non tail version: it uses our library but uses an operation instead of a function to define state operations. This performs badly here, as every time it needs to yield up and restore the resumption – evaluating tail-resumptive operations in-place is really effective.

6.2 Realistic Counter

The counter benchmark is a bit unrealistic as it can be heavily optimized as a special case. Kiselyov and Ishii [2015] present a variation that is perhaps more indicative of performance in real programs:

```
runCount5 :: (State Integer :? e) ⇒ Eff e Integer
runCount5 n = foldM f 1 [n, n - 1 .. 0] where f acc x | x ‘mod’ 5 == 0 = do i <- perform get ()
  perform put (i+1)
  return (max acc x)
  f acc x = return (max acc x)
```

Here the program folds over `n` numbers to find its maximum, and performs a get and put whenever it hits a multiple of five. This time we use `10^6` as the initial state. The pure version models state as a tuple. Figure 2c shows the new results over this benchmark.

Now the performance of all libraries is more aligned. Our library EV performs best here, and is about 1.5x faster than the next contenders EE and EF which perform similarly, and each about twice as fast as the “pure” version. We usually expect the direct pure version to be the fastest, but in this case
6.3 Multi Layer Counter

Kiselyov and Ishii [2015] use the realistic state counter to further evaluate performance when there are multiple layers of effects (handlers or monad transformers). Some implementations will increase non-linearly in such case.

In the benchmark, we put many Reader layers under or over the target State layer. For EV, we again tested both the tail-resumptive version (with value and function), and the non-tail resumptive one as EV NT.

Figure 2a presents the results, where the layer 0 results correspond to the previous benchmark. When we put the state layer at the bottom of the layer stack (the left of Figure 2a), as we increase the number of the reader layers over the state layer, our EV runs in constant time, EE runs in linear time and MTL seems to run in quadratic time. On the other hand, if we put the state layer at the top of the layer stack (the right of Figure 2a), as we increase the number of the reader layers under the state layer, our EV still runs in constant time, and this time EE also runs in constant time while MTL seems to run in linear time.

For this benchmark, again EV is faster than the other alternatives. Notably, there is little extra cost to adding more reader layers over- or under the target state layer. We believe this is due to those operations being tail-resumptive, and thus they are always evaluated in place and never need yield up. On the other hand, when the reader layers are over the state, EE and FE still need to yield up to find the target state layer, which starts to take linear time in the number of reader layers. In both cases, MTL suffers severely for deep stacks.

---

**Fig. 2. Benchmark Results.** Time is in milliseconds, while speed is the relative performance with respect to our library (EV).
6.4 Error Effect

The single error effect benchmark [Kiselyov and Ishii 2015] calculates the product of $10^7$ copies of one followed by one zero, and throws an error when it hits the zero. The pure version simply returns the product.

```haskell
runError :: (Except :? e) ⇒ Eff e Int
runError = foldM f 1 (replicate 1000000 1 ++ [0])
  where f acc θ = perform E.throwError (θ::Int)
        f acc x = return $ acc * x
```

The results are given in Figure 2d. For the error effect all versions have similar performance. Note there is no separate EV NT as exceptions are already not tail-resumptive.

6.5 Non-determinism

The non-determinism benchmark [Kiselyov and Sivaramakrishnan 2017] searches for Pythagorean triples up to 250 with non-deterministic brute-force.

```haskell
pyth :: (Choose :? e) ⇒ Int → Eff e (Int, Int, Int)
pyth upbound = do
  x <- perform choose upbound
  y <- perform choose upbound
  z <- perform choose upbound
  if (x*x + y*y == z*z) then return (x,y,z)
  else perform none ()
```

Our EV version uses a Choose effect which chooses from 1 to upbound. The MTL version uses the continuation monad transformer `ContT` with `MonadPlus` instance to collect all results. The EE library provides a (specially implemented) non-determinism effect `NDet`. Figure 2e presents the results. The results show that EV is competitive with EE, while both are (much) slower than FE and MTL.

Like the Counter benchmark, the plain version is very amenable to specific GHC optimizations for lists and continuations. A variant of this benchmark performs one other operation: it counts the number of all attempted choices using a state effect. Figure 2f presents the results. This time EV is quite a bit faster than the alternatives. Again we feel this is more indicative of real-world performance as one usually needs to combine various effects.

6.6 Other Approaches

Kammar et al. [2013] describe the “Handlers in action (HIA)” implementation of effect handlers. Since their implementation has only been maintained up to GHC 7.8.2 (released April 2014), our benchmarks do not support HIA. Instead, we briefly discuss the benchmark results of EE and HIA presented in Kiselyov and Ishii [2015] where they show that EE performs similarly, or slightly faster than HIA for most benchmarks, except for the state counter and the non-counting pythagorean triple benchmark (where HIA has similar performance as MTL). As noted by Kiselyov and Ishii, this is because GHC is good at optimizing simple CPS code employed in simple instances of HIA. In this sense, we believe that like EE, our EV outperforms HIA in most benchmarks except for those two.

Also note that EE has been evolving after Kiselyov and Ishii [2015], and our benchmarks run on the latest version of EE. For example, the curve of EE in the deep layer benchmark is more smooth than the one in Kiselyov and Ishii [2015].

The eff library [King 2020] is a work-in-progress implementation of an extensible effect system. The library uses a fork of the GHC compiler that is extended with native support for delimited continuations, and there is currently an open GHC proposal to include the new primitives in GHC proper. Based upon the primitives, it is shown that the library is as fast as MTL in the single state benchmark. There is yet no direct way to run eff on all benchmarks. We are excited though by the prospect of having native delimited continuations as it possibly enables a more efficient multi-prompt implementation for our `Ctl` monad as well.

7 Related Work

Our Haskell library EV follows the formalization of F\* [Xie et al. 2020]. The most notable difference is that F\* encodes effects using a row type system which is suitable for Hindley-Milner style type inference, while EV encodes effects using a combination of a type list (:*) and type class constraints (:?). This leads to semantic differences in several aspects. Unlike EV, in F\*, when there are multiple instances of an effect, only the first instance is (directly) accessible. For example, for a row effect `(Reader Int, Reader String)`, the reader operation `ask` always corresponds to the first instance, i.e., `Reader Int`, and thus always returns an integer, without need for type annotations. In contrast, EV is able to distinguish same effect of different types. Consider:

```haskell
greetOrExit::(Reader String :? e, Reader Bool :? e) ⇒ Eff e String
greetOrExit = do s <- perform ask ()
               isExit <- perform ask ()
               if isExit then return ("goodbye " ++ s)
                else return ("hello " ++ s)
```

Here we have two asks of different types: `s` asks for a `String` and `isExit` asks for a `Bool`. This is also reflected in two type class constraints `Reader String :? e` and `Reader Bool :? e`. When the reader type is not obvious from the context, by annotating the return type of `ask` we can indicate which reader we want. Multiple reader constraints of the same type, e.g., multiple `Reader Int :? e`, are allowed but are the same as a single constraint. To support multiple readers of the same type, we can use the standard newtype trick.

Kiselyov et al. [2013] first described the extensible effect library (EE) implemented as a free monad. It models a list of effects as open union, but uses overlapping instances and requires effect components to be `Typeable`. Kiselyov and Ishii [2015] improve EE using a `freer` monad, which removes these drawbacks. Furthermore, it improves the performance
building the monadic bind continuation as type-aligned sequence to accumulate the request continuation. Currently, this type-aligned sequence is user-facing: when defining handlers, users get the current sequence and need to explicitly accumulate the sequence to ensure the right semantics. An advantage of our library is that it exposes no internals of effect handling even to the definition of the operations (which also guarantees that one cannot deviate from the effect handler semantics as implemented by our library). It may be possible to integrate type-aligned sequences into our Ctl module as well to create continuations more efficiently but we have not yet investigated this in detail.

Kammar et al. [2013] implement HLA based on a free monad and a continuation monad. They abstract over operations and thus one operation can be handled by multiple effects. However, abstraction over operations requires HIA to manage both effects and handlers using type class constraints which requires all handlers to be top-level. HIA relies on Template Haskell to generate the boilerplate code of the type-class encoding for handler definitions, which unfortunately increases the programming barrier.

Wu et al. [2014] implement extensible effects using a Data types à la carte approach [Swierstra 2008], where handlers are folds over algebras. This further developed by Wu and Schrijvers [2015a] where handlers can be fused together to avoid constructing intermediate datatypes—a technique that is used in the fused effects library. Also, like EE, it exposes the internals of effect handling to the operation and handler definitions (for example, a handler needs to propagate other operations explicitly).

Closely related to algebraic effects is the monad transformer library (MTL) [Liang et al. 1995]. However, MTL is not as composable as effects. In MTL, the semantics is fixed by the order of the transformers, and it is necessary to explicitly lift operations through the transformer stack. Implicit lifting is also possible but it is very restricted. For example, the type class MonadReader allows to use the reader operation ask without specifying its position in the transformer stack. However, it relies on functional dependency and thus can only support up to one layer of Reader. See also Kiselyov et al. [2013] for an extensive comparison between effects and MTL.

8 Conclusion

In this paper, we have introduced a new effect handlers library in Haskell based on evidence passing, which offers a concise library interface with good performance. In the future, we plan to investigate ways to abstract over the repeated effect context and answer type when defining an effect. Also, currently mask and handlerHide are restricted to the top-level handler and we would like to extend those to work on arbitrary handlers in the effect.

Acknowledgments

We would like to thank Simon Peyton Jones for the inspiring discussion on an early version of the Ctl monad.

References


