Effect Handlers in Haskell, Evidently

Ningning Xie
Microsoft Research
nnxie@cs.hku.hk

Daan Leijen
Microsoft Research
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.

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^v 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,

```
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`:

```
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 *tail-resumptive* operations to evaluate *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 [Xie et al. 2020] 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 [Dyvbig et al. 2007] but we believe our implementation is particularly concise.

- Finally, we also show how we can efficiently represent *local isolated state* without needing higher ranked types through handler hiding (Section 4.3 and 5.6).

The source code of our implementations and benchmarks are available as the anonymous supplementary material.

2 A Tour of Reader

Let's start with an in-depth example of dynamic binding, also known as the *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 *operations* of the effect, not the plumbing (in the form of `return` and `(>>=)` for monads). As shown in the introduction, to define new effects, we just declare a new data type:

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

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

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

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

Effectful computations run in the `Eff e a` monad where `e` is the effect context and `a` the result type. The type class constraint `h :? e` indicates that effect handler `h` is a member of the effect context `e` and can thus be used to perform operations. In this case, `perform ask ()` returns a value `s` of type `String`, and thus `greet` is qualified with a string reader effect, `(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 `handler` function:

```
reader :: Eff (Reader String :* e) ans → Eff e ans
reader action
  = handler (Reader{ ask = value "world" }) action
```

The first argument to `handler` is a concrete `Reader` data type with an implementation of the `ask` operation. In this case we use `value :: a → Op () a e ans` to always resume with a constant value `"world"` when `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 `Eff (Reader String :* e) ans`, where the type `h :* e` represents the effect context as a type level *list*, which has effect `h` as the head and effect context `e` as the tail. We can view `handler` as an elimination rule where the the `Reader String` effect is eliminated from the `action` context, resulting in `Eff e ans`, (i.e., it *handles* the effect).

Since `action` has `Reader` in its effect context, it can perform operations in the `Reader` effect – in particular, we can evaluate `greet` under our `reader` handler, where the constraint `Reader a :? (Reader a :* e)` is satisfied:

```
helloWorld :: Eff e String
helloWorld = reader $
               greet
```

Finally, we can run effectful computations using `runEff` with type `Eff () a → a`:

```
> runEff helloWorld
"hello world"
```

The empty effect context in `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

```
data Effect a₁ ... aₙ e ans
  = Effect { op₁ :: forall x₁ ... xₗ. Op b₁ c₁ e ans
           , ...
           , opₘ :: forall x₁ ... xₖ. Op bₘ cₘ e ans }
```

which declares an effect `Effect a₁ ... aₙ` with universal variables `a₁ ... aₙ`, the effect context `e` and the answer context `ans`. Each operation `op` can have its own universally quantified variables `x`'s, and has type `Op` from an argument of type `b` to a result of type `c`, under the same effect and answer context.

3.2 Performing Operations

Operations are performed by passing the operation selector and its argument to the `perform` function:

```
foo :: (h :? e) ⇒ Eff e a
foo = perform op arg
```

If `op` belongs to an effect `h`, then performing `op` induces the constraint `h :? e`, indicating that `h` must be in the effect context of `Eff e a`, e.g., we can only perform operations that are handled in our context.

3.3 Defining Operations

There are three ways to create operations of type `Op a b e ans`:

```
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 e1 ans1) → 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) → 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 :: a → Op () a e ans`. We have seen `value` in the `Reader` example (repeated below for reference). This is used to create operations that always resume with a constant value.

  ```
  Reader{ ask = value "world" }
  ```

- `function :: (a → Eff e b) → Op a b e ans`. In practice, most effects can be defined using `function`, it takes a function from `a` to `b` in the effect context `e` of the handler. We can define the `ask` operation of the reader in terms of `function` as well:

  ```
  Reader{ask = function (\ () → return "world")}
  ```

  and more generally:

  ```
  value x = function (\ () → return x)
  ```

- `operation :: a → (b → Eff e ans) → Eff 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 → Eff e ans`. An `operation` does not resume 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 the 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 `Eff (h :* e) ans` to `Eff e ans`.
  Here we also see that we use a partially applied type `h` in the effect context `h :* e`: the fully applied type `h e ans` 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`. This takes an extra argument to transform the result type of the action from `a` to `ans`, where the answer type of the operations in `h` is now `ans`. We will see an example of it in Section 4.1.
- `handlerLocal`. The `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 operation `lget` and `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 `handlerRet` and `handlerLocal` where one can provide a function to transform the result of 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 `Exn` have one operation `failure` with type `Op () a e ans` for any `a`.

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

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

```
safeDiv :: (Exn :? e) ⇒ Int → Int → Eff e Int
safeDiv x 0 = perform failure ()
safeDiv x y = 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 `Maybe` monad:

```
toMaybe :: Eff (Exn :* e) a → Eff e (Maybe a)
toMaybe
  = handlerRet (Just) $ Exn{
      failure = operation (\ () _ → return Nothing) }
```

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

```
> runEff (toMaybe $ safeDiv 42 2)
Just 21
> runEff (toMaybe $ safeDiv 42 0)
Nothing
```

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

```
exceptDefault :: a → Eff (Exn :* e) a → Eff e a
exceptDefault x
  = handler $
    Exn{ failure = operation (\ () _ → return x) }
```

We can run it in the same examples:

```
> runEff (exceptDefault 0 $ safeDiv 42 2)
21
> runEff (exceptDefault 0 $ safeDiv 42 0)
0
```

4.2 State

The state effect comes with an operation `get` for reading the state value, and `put` for setting the state value.

```
data State a e ans = State { get :: Op () a e ans
                           , put :: Op a () e ans }
```

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

```
invert :: (State Bool :? e) ⇒ Eff e Bool
invert = do b <- perform get ()
            perform put (not b)
            perform get ()
```

The definition of `state` takes an initial value of the state, and uses `handlerLocal` for handling.

```
state :: a → Eff (State a :* e) ans → Eff e ans
state init
  = handlerLocal init $
    State{ get = function (\ () → perform lget ())
         , put = function (\ x → perform lput x) }
```

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

```
> runEff (state True invert)
False
```

4.3 State as a Function

Of course, we cheated a bit in the state example by using the builtin `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 → Eff e a` [Kammar and Pretnar 2017]:

```
local :: a → Eff (Local a :* e) ans → Eff e ans
local init action
  = do f <- handler (Local{
              lget = 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
```

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 → Eff e ans
handlerLocal init h action
```

```
  = local init (handlerHide h action)
```

where `handlerHide` hides the top most handler h_0 (which is `Local a` in our example):

```
handlerHide :: h (h0 :* e) ans → Eff (h :* 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 5.6.

4.4 Ambiguity

We have seen handlers that resume computations once (e.g., `reader`), and handlers that abort computations (e.g., `except`). In this example, we show handlers that resume computations more than once.

The `Amb` effect has one operation `flip`, which takes a unit and returns some boolean.

```
data Amb e ans
     = Amb { flip :: forall b. Op () Bool e ans }
```

As an example, we can define the exclusive or of two booleans.

```
xor :: Amb :? e ⇒ Eff e Bool
xor = do x <- perform flip ()
         y <- perform flip ()
         return ((x && not y) || (not x && y))
```

The concrete implementations decides which boolean value to use. For example, one implementation can collect all results from both choices:

```
allResults :: Eff (Amb :* e) a → Eff e [a]
allResults = handlerRet (\x → [x]) (Amb{
  flip = operation (\ () k →
           do xs <- k True
              ys <- k False
              return (xs ++ ys)) })

> runEff (allResults xor)
[False,True,True,False]
```

This handler transforms computations with results `a` to a computation to a list of all possible results `a`. Note that the result type of the resumption function `k` is also `[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) }
```

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 (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 $\texttt{many}_1$, 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 (Parse :* e) b → Eff e (b, String)
parse input
  = handlerLocalRet input (\x y → (x, y)) $
    Parse { satisfy = operation $ \p k →
      do input <- localGet
         case (p input) of
           Nothing → perform failure ()
           Just (x, rest) → do localPut rest
                               k x }
```

Note the type signature of `parse`: it handles `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:

```
symbol :: Parse :? e ⇒ Char → Eff e Char
symbol c = perform satisfy (\input → case input of
    (d:rest) | d == c → Just (c, rest)
    _ → Nothing)

digit :: Parse :? e ⇒ () → Eff e Int
digit c = perform satisfy (\input → case input of
    (d:rest) | isDigit d → Just (digitToInt d, rest)
    _ → Nothing)
```

Parsers for simple arithmetic expressions are built upon those basic parsers:

```
expr :: (Parse :? e, Amb :? e) ⇒ Eff e Int
expr = choice (do i <- term; symbol '+'; j <- term
```

```
                  return (i + j))
              term

term :: (Parse :? e, Amb :? e) ⇒ Eff e Int
term = choice (do i <- factor; symbol '*'; j <- factor
                  return (i * j))
              factor

factor :: (Parse :? e, Amb :? e) ⇒ Eff e Int
factor = choice (do symbol '('; i <- expr; symbol ')'
                    return i)
                number

number :: (Parse :? e, Amb :? e) ⇒ Eff e Int
number = do xs <- many1 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 :: Eff (Exn :* Amb :* e) a → Eff e [a]
solutions action
  = fmap catMaybes (allResults (toMaybe (action)))

> runEff (solutions (parse "1+2*3" expr))
[(7,""),(3,"*3"),(1,"+2*3")]
```

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

```
eager :: Eff (Exn :* Amb :* e) a → Eff e (Maybe a)
eager action = firstResult (toMaybe (action))

> runEff (eager (parse "1+2*3" expr))
Just (7,"")
```

5 Implementation

Our effect implementation is directly based on the target language F^v in the formal evidence translation described by Xie et al. [2020] (Section 5, Figure 9). Their translation has two parts: a 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 a` for multi-prompt delimited continuations:

```
data Ctl a
  = Pure { result :: a }
  | forall ans b.
    Yield{ marker :: Marker ans,
           op     :: (b → Ctl ans) → Ctl ans,
           cont   :: b → Ctl a }
```

The `Pure` case has a value result, and the `Yield` case reflects 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 → 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`:

```
yield :: Marker ans →
         ((b → Ctl ans) → Ctl ans) → Ctl b
yield m op = Yield m op Pure
```

and starts with an identity continuation (as `Pure`). When yielding to a prompt, the continuation keeps being extended using the *Kleisi* composition over the control monad (`kcompose`), which is defined together with the monadic binding of `Ctl`:

```
kcompose::(b → Ctl c) → (a → Ctl b) → a → Ctl c
kcompose g f x = (f x >>= g)

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 `Yield`, we keep yielding but extend the continuation with the continuation of the bind `f`.

When we install a *prompt*, we use an internal `freshMarker` function to generate a *unique* marker as the name of the current prompt:

```
prompt :: (Marker ans → Ctl ans) → Ctl ans
prompt action
  = freshMarker (\m → mprompt m (action m))
```

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

```
mprompt :: Marker a → Ctl a → Ctl a
mprompt m (Pure x) = Pure x
mprompt m (Yield n op cont)
  = let cont' x = mprompt m (cont x) in
    case mmatch m n of
      Nothing → Yield n op cont'  -- keep yielding
      Just Refl → op cont'
```

In the case of `Pure`, the action returns a value `x`, and `prompt` simply passes on the value. In the case of `Yield`, the action yields to a prompt with marker `n`, with the operation implementation `op` and the current continuation `cont`. In this case, it first extends the continuation with our own prompt

getting a new continuation `cont'`. Now we need to decide if it is yielding to this particular prompt, by checking whether `m` is equivalence to `n`. The function `mmatch` checks the equality of two markers. If two markers are not equal (`Nothing`), then we keep yielding with `cont'`. When they are equal (`Just`), we are the target of the yield, and our continuation `cont'` is exactly the *resumption* that `op` needs to evaluate, and we apply directly `op cont'`.

However, the astute reader may have noticed that `op cont'` cannot type-check: in the `Yield` definition, `op` requires an existential answer type `ans`, while both `cont` and `cont'` have the answer type `a`. The key to ensure type-safety is the markers. Markers have two important properties: they are unique, and they carry their answer types. In this case, `m` is `Marker a` and `n` is `Marker ans`. When `mmatch` confirms that two markers are equal, it also returns the proof term `Refl` that asserts to the compiler that these two types are equal via a type equality `a ~ ans`. Once we have this type equality in the environment, the application `op cont'` type checks. The creation of markers `freshMarker` and the equality check with `mmatch` requires unsafe primitives (the full implementation is in the anonymous supplement).

5.2 Evidence Passing Effects

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

```
newtype Eff e a = Eff (Context e → Ctl a)

instance Monad (Eff e) where
  return x         = Eff (\ctx → pure x)
  (Eff eff) >>= f  = Eff (\ctx → do ctl <- eff ctx
                                    under ctx (f ctl))

under :: Context e → Eff e a → Ctl a
under ctx (Eff eff)  = eff ctx
```

The effect context is a *heterogeneous list* of pairs of markers and 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]:

```
data Context e where
  CNil  :: Context ()
  CCons :: Marker ans → h e ans →
           Context e → Context (h :* e)
```

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

```
data (h :: * → * → *) :* e
```

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

```
handler :: h e ans → Eff (h :* e) ans → Eff e ans
handler h action
  = Eff $ \ctx →
    prompt (\m → under (CCons m h ctx) action)
```

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

The dual of handling is masking [Biernacki et al. 2017], where we remove the top handler on a sub-computation `eff`:

```
mask :: Eff e ans → Eff (h :* e) ans
mask eff = Eff (\ (CCons m h ctx) → under ctx eff)
```

Note that the type `Eff (h :* e) ans` guarantees that the match on `CCons` never fails.

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

```
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`:

```
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 `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 `Op` as:

```
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:

```
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:

```
operation :: (a → (b → Eff e ans) → Eff e ans) →
             Op a b e ans
operation f
  = Op $ \m ctx x →
    yield m $ \ctlk →
      let k y = Eff (\ctx' → guard ctx ctx' ctlk y)
      in under ctx (f x k)
```

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 `ctlk`. We may think we can pass this directly to `f` (i.e., `under ctx (f x ctlk)`) 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.

```
guard :: Context e → Context e →
         (b → Ctl a) → b → Ctl a
guard ctx1 ctx2 k x
  = if ctx1 == ctx2 then k x
    else error "unscoped resumption"
```

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 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.

5.4 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:

```
instance (h :? (h :* e)) where
  subContext ctx = SubContext ctx

instance (h :? e) ⇒ (h :? (h' :* e)) where
  subContext (CCons _ _ ctx) = subContext 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:

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

The type function `HEqual h1 h2` returns true only if the handler types h_1 and h_2 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 h_1, the handler we are looking for, and h_2, the current head of the context, are equivalent.

```
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`:

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

There are now two instances for `InEq` that distinguish whether h_1 and h_2 are equivalent:

```
instance (h1 ~ h2) ⇒ InEq 'True h1 h2 e where
  subContextEq (CCons m h ctx)  = SubContext m h ctx

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.

5.5 Return Clauses

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

```
handlerRet :: (ans → a) → h e a →
              Eff (h :* e) ans → Eff e a
handlerRet ret 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:

```
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]:

```
handlerRetEff :: (ans → Eff e a) → h e a →
                 Eff (h :* e) ans → Eff e a
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 `handlerRet` suffices.

5.6 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:

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

where the top most handler h_0 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 (h₀ :* e) ans`. We like to hide the h_0 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*:

```
data Context e where
  CNil  :: Context ()
  CCons :: Marker ans → h e' ans →
           (Context e → Context e') →
             Context e → Context (h :* e)
```

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
handler h action
  = Eff $ \ctx →
    prompt (\m → under (CCons m h id ctx) action)
```

The transformer is applied in the `perform` to transform the evidence context that was passed down into a context that is required by the handler:

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

Now, we can use the context transformer to implement `handlerHide` where we transform the evidence context by restoring the hidden handler h_0 just before handling its operations:

```
handlerHide :: h (h₀ :* e) ans →
               Eff (h :* e) ans →
                 Eff (h₀ :* e) ans
handlerHide h action
  = Eff $ \ (CCons m' h' g' ctx') →
    prompt $ \m →
      let g = CCons m' h' g'
      in under (CCons m h g ctx') action
```

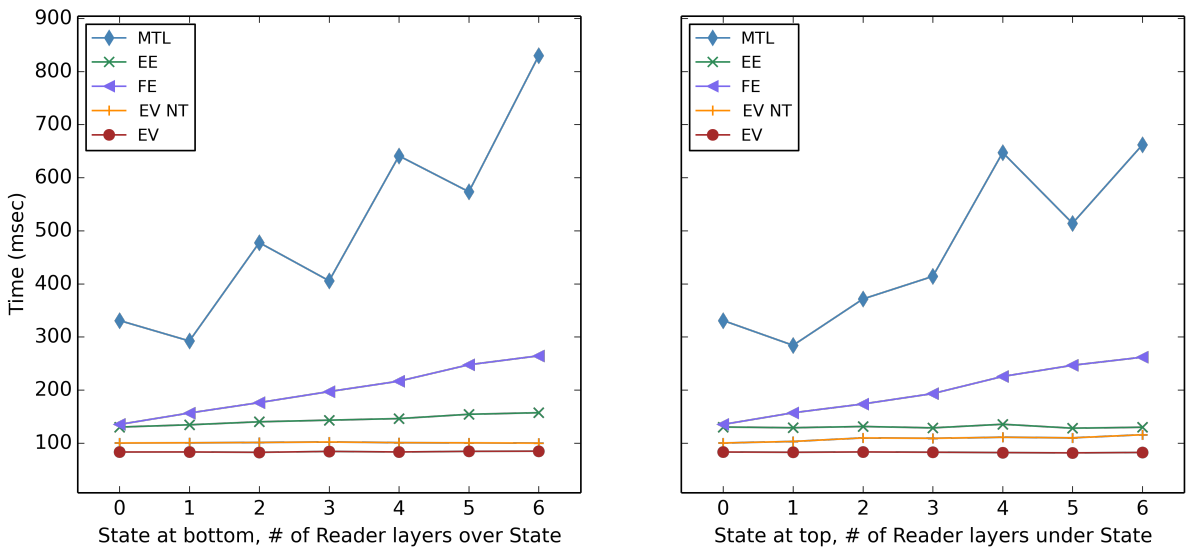
In the actual implementation we do not use a function for the context transformer but instead represent it explicitly as a GADT. This is done to improve compiler optimizations where the explicit constructors allow better inlining. We also believe that the context transformers are essential to implement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

6 Benchmarks

This section evaluates the performance of our library, by implementing the benchmarks from Kiselyov and Ishii [2015]. We compare the performance of our library (**EV**) relative to (1) the latest Extensible Effects library (**EE**) [Kiselyov and Ishii 2015]; (2) the latest Fused Effect library (**FE**), which follows the techniques described in [Schrijvers et al. 2019; Wu and Schrijvers 2015b; Wu et al. 2014]; and (3) the monad transformer library **MTL**.

The benchmark code was compiled using GHC 8.6.5 with the compile flag `-O2`. The benchmarks were run on a HP-Z4 workstation with a 4-core Intel Xeon processor at 3.60GHz and 32 MiB memory. The performance results were collected using O'Sullivan's Criterion library.

Figure 2 summarizes our benchmark results.



(a) Deep Layer Stack

	Time	Speed
Pure	9	5.44×
MTL	9	5.44×
RunST	41	1.20×
EE	339	0.14×
FE	10	4.90×
EV NT	867	0.06×
EV	49	**1.00**×

(b) Counter

	Time	Speed
Pure	247	0.34×
MTL	327	0.25×
RunST	256	0.32×
EE	129	0.64×
FE	136	0.61×
EV NT	99	0.84×
EV	83	**1.00**×

(c) Counter5

	Time	Speed
Pure	57.2	1.01×
MTL	62.2	0.93×
EE	61.9	0.93×
FE	59.5	0.97×
EV	57.6	**1.00**×

(d) Error

	Time	Speed
MTL	141	4.23×
EE	574	1.04×
FE	229	2.60×
EV	596	**1.00**×

(e) Pythagorean Triples

	Time	Speed
MTL	3230	0.39×
EE	1698	0.75×
FE	4974	0.26×
EV	1272	**1.00**×

(f) Pythagorean Triples (C)

Fig. 2. Benchmark Results. Time is in milliseconds, while speed is the relative performance with respect to our libray (EV).

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. A pure counter is simply a tight loop for counting down.

```
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 results are given in Figure 2b. The Pure, MTL, and FE versions are all fully inlined and recurse 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) ⇒
                Integer → 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 1.5× 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 it needs to fold with an extra state which causes allocation of tuples.

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.

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.

```
runError :: (Except :? e) ⇒ Eff e Int
runError = foldM f 1 (replicate 10000000 1 ++ [0])
  where f acc 0 = perform E.throwError (0::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.

```
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 is built upon an open GHC proposal, which proposes to add new low-level primitives for capturing slices of GHC RTS stack. 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 these GHC extensions 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^v [Xie et al. 2020]. The most notable difference is that F^v 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 semantics differences in several aspects. Unlike EV, in F^v, 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:

```
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 `ask`s 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] improves 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 HIA 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.

References

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Dec. 2017. Handle with Care: Relational Interpretation of Algebraic Effects and Handlers. *Proc. ACM Program. Lang.* 2 (POPL'17 issue): 8:1–8:30. doi:10.1145/3158096.

R Kent Dyvbig, Simon Peyton Jones, and Amr Sabry. 2007. A Monadic Framework for Delimited Continuations. *Journal of Functional Programming* 17 (6). Cambridge University Press: 687–730. doi:10.1017/S0956796807006259.

Richard Eisenberg. Dec. 2012. Decidable Propositional Equality in Haskell. https://typesandkinds.wordpress.com/2012/12/01/decidable-propositional-equality-in-haskell.

Richard Eisenberg, and Stephanie Weirich. 2012. Dependently Typed Programming with Singletons. In *Proceedings of the 2012 Haskell Symposium*, 117–130. Haskell '12. Copenhagen, Denmark. doi:10.1145/2364506.2364522.

Ben R. Gaster, and Mark P. Jones. 1996. *A Polymorphic Type System for Extensible Records and Variants*. NOTTCS-TR-96-3. University of Nottingham.

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in Action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, 145–158. ICFP '13. ACM, New York, NY, USA. doi:10.1145/2500365.2500590.

Ohad Kammar, and Matija Pretnar. Jan. 2017. No Value Restriction Is Needed for Algebraic Effects and Handlers. *Journal of Functional Programming* 27 (1). Cambridge University Press. doi:10.1017/S0956796816000320.

Alexis King. May 2020. Eff: Screaming Fast Extensible Effects for Less . https://github.com/hasura/eff.

Oleg Kiselyov, and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, 94–105. Haskell'15. Vancouver, BC, Canada. doi:10.1145/2804302.2804319.

Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly Typed Heterogeneous Collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, 96–107. Haskell '04. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1017472.1017488.

Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible Effects: An Alternative to Monad Transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, 59–70. Haskell '13. Boston, Massachusetts, USA. doi:10.1145/2503778.2503791.

Oleg Kiselyov, and KC Sivaramakrishnan. Dec. 2017. Eff Directly in OCaml. In *ML Workshop 2016*. http://kcsrk.info/papers/caml-eff17.pdf. Extended version.

Daan Leijen. 2005. Extensible Records with Scoped Labels. In *Proceedings of the 2005 Symposium on Trends in Functional Programming*, 297–312.

Daan Leijen. Sep. 2008. HMF: Simple Type Inference for First-Class Polymorphism. In *Proceedings of the 13th ACM Symposium of the International Conference on Functional Programming*. ICFP'08. Victoria, Canada. doi:10.1145/1411204.1411245.

Daan Leijen. Jan. 2017. Type Directed Compilation of Row-Typed Algebraic Effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*, 486–499. Paris, France. doi:10.1145/3009837.3009872.

Daan Leijen. 2018. First Class Dynamic Effect Handlers: Or, Polymorphic Heaps with Dynamic Effect Handlers. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Type-Driven Development*, 51–64. TyDe 2018. St. Louis, MO, USA. doi:10.1145/3240719.3241789.

Daan Leijen, and Erik Meijer. 2000. Domain Specific Embedded Compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, 109–122. DSL '99. Austin, Texas, USA. doi:10.1145/331960.331977.

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 333–343. POPL '95. Association for Computing Machinery, New York, NY, USA. doi:10.1145/199448.199528.

Simon Peyton Jones, and John Launchbury. 1995. State in Haskell. *Lisp and Symbolic Comp.* 8 (4): 293–341. doi:10.1007/BF01018827.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Jan. 2007. Practical Type Inference for Arbitrary-Rank Types. *J. Funct. Program.* 17 (1). Cambridge University Press, USA: 1–82. doi:10.1017/S0956796806006034.

Gordon D. Plotkin, and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11 (1): 69–94. doi:10.1023/A:1023064908962.

Gordon D. Plotkin, and Matija Pretnar. 2013. Handling Algebraic Effects. In *Logical Methods in Computer Science*, volume 9. 4. doi:10.2168/LMCS-9(4:23)2013.

Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. 2008. Type Checking with Open Type Functions. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, 51–62. ICFP '08. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1411204.1411215.

Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. 2019. Monad Transformers and Modular Algebraic Effects: What Binds Them Together. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, 98–113. Haskell 2019. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3331545.3342595.

Wouter Swierstra. Jul. 2008. Data Types à La Carte. *Journal of Functional Programming* 18 (4): 423–436. doi:10.1017/S0956796808006758.

Nicolas Wu, and Tom Schrijvers. 2015a. Fusion for Free: Efficient Algebraic Effect Handlers. In *Proceedings of the International Conference on Mathematics of Program Construction*. MPC'15. doi:10.1.1.723.5577.

Nicolas Wu, and Tom Schrijvers. 2015b. Fusion for Free: Efficient Algebraic Effect Handlers. In *Proceedings of the 12th International Conference on Mathematics of Program Construction*, 9129:302. Springer, Königswinter, Germany.

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect Handlers in Scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, 1–12. Haskell '14. Göthenburg, Sweden. doi:10.1145/2633357.2633358.

Li-yao Xia. Jun. 2018. Heterogeneous Lists with Dependent Types in Haskell. https://blog.poisson.chat/posts/2018-06-06-hlists-dependent-haskell.html.

Ningning Xie, Jonathan Brachthäuser, Phillip Schuster, Daniel Hillerström, and Daan Leijen. Aug. 2020. Effect Handlers, Evidently. In *Conditionally Accepted at the 25th ACM SIGPLAN International Conference on Functional Programming (ICFP'2020)*. Jersey City, NJ.

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 224–235. POPL '03. Association for Computing Machinery, New York, NY, USA. doi:10.1145/604131.604150.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 53–66. TLDI '12. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2103786.2103795.