Why Data is the Better Monad

Using Freedom to Great Effect

Justus Adam
justus.adam@tu-dresden.de

ABSTRACT

Monads are an abstraction used to express side effects in pure languages such as Haskell and Idris. Due to their structure they do not compose well, and state of the art solutions to this problem involve lots of boilerplate code.

"Extensible effect" systems utilise a single Monad, parameterised by a set of monadic effects. Effects are expressed as data structures, composed via an open union, allowing dynamic addition, subtraction and rewriting of effects. Unlike previous approaches, computations can be implemented in terms of a *single* generic member constraint on the effect set avoiding per-effect boilerplate classes and instances.

Furthermore new effects are easier to define and effect interpreters can be selected dynamically.

ACM Reference Format:
Justus Adam. 2018. Why Data is the Better Monad: Using Freedom to Great Effect. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Monads are a powerful abstraction to define side effects. Given a sufficiently capable type system, effectful computations can be expressed as first class values with regular, higher kinded types. Monadic values however are inherently opaque, meaning computations only compose within the same monad. As a result interleaving different effects, that is composing different monads, poses a challenge.

Monad transformers [6] use the approach of "monads with a hole", which employs an unfilled type parameter, representing an "inner" monad, to which other effects can be delegated.

While transformers solve the principal composition issue, which will be shown in more detail in §2, they involve a large amount of boilerplate code and the performance degrades quadratically [9] with the number of transformers employed in a given monad.

This paper argues a different approach. Rather than using stacks of monads a single *extensible monad* is employed. It is based on the *free monad*, which captures the essence of what it means to be a Monad, and has no capabilities of its own. Instead effects are encoded as values from a set of effects. Computations are expressed as sequences of effect values and continuations, which are handled by an interpreter. Computations using effects are implemented in terms of member constraints on the effect set, and thus reusable in different concrete effect sets and with different interpreters.

Boilerplate code is reduced substantially, as defining new effects involves only the definition of its encoding values and one or more interpreters.

To implement an extensible effect system first a naive but simple implementation of a free monad and an effect set is shown (§3). Subsequently this approach is improved to be competitive with previous approaches to make it viable for general use (§4, §5) and further simplify the interface (§6) for defining effects. The implemented approach is contextualised with other, similar systems in terms of both how they are implemented and the feature set they offer (§7).

2 CURRENT STATE OF THE ART

Monad transformers are the current state of the art when it comes to composing monads in Haskell. Each type of effect is implemented by what Steele Jr. called a *pseudomonad* [14] and Liang et al. later broadened into the concept of a *transformer* [11], a structure which leaves a "hole" [12] in a monad to embed extensions. Concretely these "incomplete" monads are parameterised by an additional type variable which is instantiated with an arbitrary monad, combining the effect of the transformer and the inner monad. As an example see the `StateT` monad transformer in figure 1 into which an arbitrary monad `m` is embedded. To access effects of the inner monad each transformer implements a `lift :: (MonadTrans m, Monad n) => n a -> m a` operation to embed computations using the inner monad in the transformer. Complex effect systems are built up by nesting multiple transformers into each other, forming a transformer stack which, is terminated by some none-transformer monad, such as `IO` or `Identity`.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
newtype StateT s m a = StateT
  { runStateT :: s → m (a, s)
  }

instance Monad m
  ⇒ Monad (StateT s m) where
  pure a = StateT $ \s → pure $ (a, s)
  v ≫= f =
    StateT $ \s → do
      (a, s') ← runStateT v s
      runStateT (f a) s'
```

Figure 1: The `State` monad transformer

```
class MonadState s m | m → s where
  get :: m s
  put :: s → m ()

instance Monad m
  ⇒ MonadState s (StateT s m) where
  get = StateT $ \s → pure (s, s)
  put a = StateT $ const $ pure ((), a)
```

Figure 2: Monad class for the state effect and its implementation

A key idea of the approach of transformers is to abstract away from the concrete transformer using a type class to define an interface for its primitive operations [7], which the associated transformer implements, such as the example in figure 2. The same primitive operation may now be implemented by different monads, and crucially, other monad transformers. Transformers that do not implement a particular effect delegate its execution up the stack with `lift`. Thus these classes can be used to "auto lift" effects up the stack to the layer capable of handling it.

Furthermore computations using effects ban be implemented in terms of effect interfaces used, by constraining an abstract monad with effect classes, rather than concrete stacks or transformers, which makes them reusable in different monad stacks.

Transformers and classes have two main shortcomings however.

(1) **Boilerplate**. The class machinery to overload effects requires instances of each effect class, for each transformer. Newly defined transformers must thus provide a boilerplate instance of every other effect class delegating the effects using `lift`. Similarly newly defined effects must also provide an instance for their effect class for every preexisting transformer to lift the effects.
(2) **Performance**. Every monadic action such as `pure`, ≫= and the effects must traverse the entire stack adding overhead scaling quadratically with the size of the stack as shown by Kiselyov and Ishii [9, §4.1].

3 BUILDING A NAIVE EXTENSIBLE MONAD

Using monad transformers involves writing significant boilerplate code defining effect instances because every effect is implemented on a separate transformer, which prevents sharing of the common action of lifting an effect. Lifting through a stack can be avoided entirely by instead using a *single* monad parameterised over a variable set of effects.

3.1 Decoupling Monad and Effect

Swierstra has shown a monad which encapsulated the two monadic actions, `pure` and ≫= as a data type with no effect capabilities of its own [15]. It is called a *free* monad, because it extends *any* functor, which is a simpler structure, with the monadic capabilities. Essentially the free monad represents an encoding for the `Monad` typeclass as a recursive data type parameterised with an effect functor. Figure 3 shows the implementation for reference.

By using the free monad the definition of `pure` and ≫= can be elided. Instead a data structure is obtained which has to be *interpreted* later, allowing the precise interpretation to vary freely by using different interpreters. Figure 4 shows a monad for interacting with a console implemented with two different interpreters. Crucially the computation is unaware of the different context/interpreter, which allows free monad based systems do be easily mocked for testing.

There is a caveat to using functors rather than monads for effects however. Certain effects, such as `listen`, an effect belonging to the family of `Writer` effects, cannot be expressed as a functor and thus cannot be handled by a free monad based implementation.

3.2 Composing Effects

Functors unlike monads are composable. A simple functor providing this feature is the `Sum` functor, here called `:+:`, see figure 5. It uses the same technique as the `OR` type by Swierstra [15] lifted to `* → *` kinded types. By nesting `:+:` values arbitrarily many functors can be composed. Such variable composites are called *open union*, as opposed to static composites, such as algebraic datatypes.

Similar to the transformer classes an interface is defined to interact with specific functors inside the composition. The subtype relation class can be seen in figure 6. Unlike with transformer classes the subtype relation can be defined

```
data Free f a
  = Impure (f (Free f a))
  | Pure a

instance Functor f ⇒ Functor (Free f) where
  fmap f (Pure a) = Pure $ f a
  fmap f (Impure eff) =
    Impure $ fmap (fmap f) eff

instance Functor f ⇒ Monad (Free f) where
  pure = Pure

  Pure a >>= cont = cont a
  Impure fa >>= cont =
    Impure $ fmap (>>= cont) fa

iter :: Functor f ⇒ (f a → a) → Free f a → a
iter _ (Pure a) = a
iter handle (Impure fa) =
  handle $ fmap (iter handle) fa

iterM :: (Monad m, Functor f)
      ⇒ (f (m a) → m a) → Free f a → m a
iterM handle (Pure a) = pure a
iterM handle (Impure fa) =
  handle $ fmap (iterM handle) fa
```

Figure 3: The simple free monad

generically without knowing the concrete type of the functor being queried for.

This open union is used as base functor for `Free` to form a monad with a variable set of effects. Computations are parameterised over the types of effects they use, via the subtype relation, without directly describing the structure of the functor. Crucially there is only one subtype relation class which is used for all effects *without* the need for additional instances for new effects. An extensible version of the `readLine` and `writeLine` functions from the earlier example can be seen in figure 7. Whereas the earlier version from figure 4 constrained us to use the explicit type of `Free Console`, the extensible version allows any version of `Free`, so long as the effect functor contains the `Console` effect. Because of the `f :<: f` instance of the dispatch class `:+:` this also includes `Console` itself.

Similar to monad transformers computations using a subset of effects can always be used in functions constrained to a superset of effects.

A unique feature of this extensible monad is the `interpose` combinator which is used to intercept and possibly alter, filter or relay requests. Whereas transformers would

```
data Console a
  = ReadLine (String → a)
  | WriteLine String a

instance Functor Console where
  fmap f (ReadLine cont) =
    ReadLine $ fmap f cont
  fmap f (WriteLine str a) =
    WriteLine str $ f a

type ConsoleM = Free Console

readLine :: ConsoleM String
readLine = Impure (ReadLine pure)

writeLine :: String → ConsoleM ()
writeLine str =
  Impure (WriteLine str (pure ()))

interpretIO :: ConsoleM a → IO a
interpretIO =
  iterM $ \case
    ReadLine f → getLine >>= f
    WriteLine line cont →
      putStrLn line >> cont

interpretPure :: [String] → ConsoleM a
              → (a, [String])
interpretPure ls = go ls []
  where
    go _ outLines (Pure v) =
      (v, reverse outLines)
    go inLines outLines (Impure eff) =
      case eff of
        ReadLine cont →
          go (tail inLines) outLines
            $ cont (head inLines)
        WriteLine str cont →
          go inLines (str : outLines) cont
```

Figure 4: Implementing a simple console I/O interaction using `Free`

```
data (f :+: g) a
  = Inl (f a)
  | Inr (g a)
```

Figure 5: The sum functor `:+:`

```
class sub :<: sup where
  inj :: sub a → sup a
  prj :: sup a → Maybe (sub a)

instance f :<: f where
  inj = id
  prj = Just

instance f :<: (f :+: g) where
  inj = Inl
  prj (Inl f) = Just f
  prj _ = Nothing

instance {-# OVERLAPPABLE #-}
  (f :<: sup) ⇒ f :<: (g :+: sup) where
  inj = Inr . inj
  prj (Inr sum) = prj sum
  prj _ = Nothing
```

Figure 6: Dispatch classes for effects in a simple open union

```
writeLine :: (Console :<: effs, Functor effs)
          ⇒ String → Free effs ()
writeLine str =
  Impure $ inj $ WriteLine str (pure ())

readLine :: (Console :<: effs, Functor effs)
         ⇒ Free effs String
readLine = Impure $ inj $ ReadLine pure
```

Figure 7: An extensible version of the console interations

necessitate the provision of a separate implementation of `interpose` for each transformer, here we are able to implement a generic combinator usable with all effect types and agnostic to the interpreter used.

To illustrate the usefulness of this feature figure 9 shows a function which sanitises the input sent to the console for any monad providing the console effect.

3.3 Handling Effects

With the extensible effects monad generic combinators can be defined to aid the handling of effects. This is possible due to the fact that both the structure of the monad as well as a fair bit about the request structure is known.

```
interpose ::
     (f :<: effs, Functor f, Functor effs)
  ⇒ (forall b. f b → Free effs b)
  → Free effs a
  → Free effs a
interpose _ (Pure a) = Pure a
interpose handler (Impure eff) =
  case prj eff of
    Nothing → Impure eff
    Just f →
      join $ handler $ fmap (interpose handler) f
```

Figure 8: The interpose combinator

```
escapeInput ::
     (Functor effs, Console :<: effs)
  ⇒ (String → String)
  → Free effs a → Free effs a
escapeInput sanitize =
  interpose $ \case
    ReadLine cont → do
      line ← readLine
      pure $ cont (sanitize line)
    other → Impure $ inj $ fmap pure other
```

Figure 9: Two examples for using the interpose combinator

Most useful is the `interpret` and `reinterpret` combinator, shown in figure 10. With `reinterpret` the effect is handled by inserting a new effect and handling the old effect in terms of this new effect and the old effects. New effects can therefore be handles by "remapping" onto another effect for which a (generic) handler already exists. However the computation will remain unaware of the added effect `g`. `interpret` is similar but adds no new effect. Both combinators allow the user to leverage knowledge about other effects present to relay effects or leverage generic effects to implement specialised ones. Figure 11 shows an example how the console effect is interpreted once by delegating to the `IO` monad and once by reinterpreting it as a state effect and using the generic `runState` interpreter.

4 IMPROVING THE OPEN UNION

For extensible effect systems to become viable it is important that its performance is comparable with the currently employed solutions. Monad transformers have been a part of the Haskell ecosystem for a long time and according to Kiselyov and Ishii the `State` monad in particular benefits

```
interpret ::
     (Functor effs, Functor f)
  ⇒ (forall w. f w → Free effs w)
  → Free (f :+: effs) a → Free effs a
interpret _ (Pure a) = pure a
interpret handle (Impure (Inr others)) =
  Impure $ fmap (interpret handle) others
interpret handle (Impure (Inl eff)) =
  join $ handle
    $ fmap (interpret handle) eff

reinterpret ::
     (Functor f, Functor g, Functor effs)
  ⇒ (forall a. f a → Free (g :+: effs) a)
  → Free (f :+: effs) b
  → Free (g :+: effs) b
reinterpret _ (Pure a) = pure a
reinterpret handle (Impure (Inr others)) =
  Impure $ Inr
    $ fmap (reinterpret handle) others
reinterpret handle (Impure (Inl eff)) =
  join $ handle
    $ fmap (reinterpret handle) eff

run :: Free Identity a → a
run = iter runIdentity

runM :: Monad m ⇒ Free m a → m a
runM = iterM join
```

Figure 10: Combinators for interpreting effects

from dedicated optimisation passes in the Haskell compiler GHC.

The free monad in conjunction with the open union from the previous section is unable to compete with the `mtl`. Encoding the open union of types with a chain of sums is inefficient. Its structure is similar to a linked list and hence it has similar computational complexity.

To construct a less space consuming open union, an encoding similar to the encoding of closed unions is used. Efficient closed unions pair the payload with a tag value. At runtime code dispatches based on the value of the tag.

GHC provides a built-in mechanism for generating runtime representations of types called (`TypeRep`). `TypeRep`s support fast comparison via a compiler generated unique 128 bit MD5 hash. Figure 12 shows the implementation of the reflection based union.

```
interpretConsolePure ::
     [String] → Free (Console :+: Identity) a
     → (a, [String])
interpretConsolePure strs =
  run . fmap (second (reverse . snd))
      . runState (strs, []) . reinterpret f
  where
    f :: Console a
      → Free (State ([String], [String])
                :+: Identity) a
    f (ReadLine cont) = do
      (inLines, outLines) ← get
      put (tail inLines
          , outLines :: [String])
      pure $ cont $ head inLines
    f (WriteLine str cont) = do
      (inLines, outLines) ← get
      put (inLines :: [String]
          , str : outLines)
      pure $ cont

interpretConsoleIO :: Free (Console :+: IO) a
                   → IO a
interpretConsoleIO = runM . interpret f
  where
    f (ReadLine cont) =
      cont <$> liftIO getLine
    f (WriteLine str cont) = do
      liftIO (putStrLn str)
      pure cont
```

Figure 11: Interpreting the extensible console effect

```
data Union r v where
  Union :: (Functor t, Typeable t)
        ⇒ t v → Union r v

inj :: (Typeable t, Functor t, t :<: r)
    ⇒ t a → Union r a
inj = Union

prj :: (Typeable t, t :<: r)
    ⇒ Union r a → Maybe (t a)
prj (Union v) = cast v
```

Figure 12: A single value open union based on reflection

While this allows for the implementation of a single-struct open union, as opposed to the linked list from before, resolving the type still involves several steps. (1) Pointer dereference to get to the `TypeRep` structure. (2) tag comparison for the two type reps since `TypeRep` is itself an algebraic data type and (3) comparing the two 128 bit values. Considering the generally small size the union will have a 128 bit fingerprint is unnecessarily large.

The followup paper by Kiselyov and Ishii [9] proposes an alternative source for a tag value, the index in the type level list. Since each union `Union r a` carries with it the type level list `r` each functor type `t` to which we can coerce a value must be present in this list and this have an index that is known at compile time. This index is much smaller than the fingerprint. The original implementation in [9] used an `Int` value, which is at least 30 bit, later implementations use `Word` which is the same size, but unsigned. Unlike the `TypeRep` from before this value can be directly inlined into the `Union` constructor, alleviating the need for both the pointer dereference as well as the tag comparison.

The index based approach also removes the need to derive a `Typeable` instance for the effect type.

5 IMPROVING THE PERFORMANCE OF ≫=

While there is beauty in the simplicity of the bare free monad its performance is lacking. As can be seen in figure 3 the implementation for ≫= pushes the ≫= down the tree using `fmap` until it reaches a `Pure` value. This means that for every use of ≫= the entire tree must be traversed once.

5.1 Better asymptotic performance using continuation passing

This problem is akin to appending to singely linked lists where the entire list has to be traversed to find the final pointer and append. [5] propose a solution whereby an unfinished list is represented as a function `[a] -> [a]` which appends a list to the intermediate result thus retaining a reference to the end of the list. List altering functions, such as appends and prepends are implemented as function composition and thus very cheap. When the construction step is completed the list can be simply realised by passing `[]` to the function.

[16]s solution [16] an adaptation of the idea, applied to moands. He uses a *Codensity monad* shown in figure 13, which is a function that, given a continuation, builds some monad `m`. As the figure shows this ≫= does not traverse the structure. Analogue to the list solution it retains a reference to the end.

```
newtype Codensity m a = Codensity
  { runCodensity ::
      forall w. (a → m w) → m w
  }

instance Monad (Codensity m) where
  return v = Codensity ($ v)
  c ≫= f = Codensity $ \k →
    runCodensity c $ \v →
    runCodensity (f v) k

lowerCodensity :: Applicative f
               ⇒ Codensity f a → f a
lowerCodensity (Codensity f) = f pure
```

Figure 13: The codensity monad

```
handle_relay :: Typeable t
             ⇒ Union (t |> r) v
             → (v → Eff r a)
             → (t v → Eff r a)
             → Eff r a
handle_relay u loop h =
  case decomp u of
    Right x → h x
    Left u → send (\k → fmap k u) ≫= loop
```

Figure 14: Relay function as implemented in [10]

5.2 Improving interpreter performance using type aligned sequences

While a continuation based free monad improves the performance of ≫= it only does so during the initial construction. During interpretation the codensity monad is evaluated to be able to interpret the head of the constructed free monad. In `handle_relay`, when the handler responds to its own effect, see 14, the constructed free monad is traversed similarly to ≫= for `Free`. As a result handling effects is again associated with quadratic complexity, as was measured by Kiselyov and Ishii [9, §4.1].

The principal problem her is that new continuations, in the form on lambda functions are created and partial applications are pushed into the functor. These intermediate allocations cause the increase in both time and space complexity. As the evaluation from the extensible effects paper [9] shows this regression in time and space only occurs when a *used* effect is threaded through the computation. In the case where the reader effect is *under* the state, this clause is triggered when only on the last layer, explaining the linear performance.

```
data Coyoneda f a where
  Coyoneda :: (b → a) → f b → Coyoneda f a

instance Functor (Coyoneda a) where
  fmap f (Coyoneda g v) = Coyoneda (f . g) v
```

Figure 15: The coyoneda functor

A recent paper by van der Ploeg and Kielyov [13] proposes a solution to this problem in the form of so called *type aligned sequences*. The approach is applicable much more broadly than just free monads. Van der Ploeg and Kiselyov represent chains of computations with data structures via the use of Generalized Algebraic Data Types. Whereas usually chains of computations are simply composed, instead they are stored in a data structure that captures how input and return types relates, preserving the type safety. This affords the possibility to inspect and alter individual links of the chain. Type safety is preserved via the GADT which does not allow the storage of an incompatible chain segments. For the free monad in particular a type aligned double-ended queue is used, represented internally as a tree. Using this sequence handler computations inspect only the head of the queue and push new types of effects back onto the front without altering the tail.

6 RELINQUISHING THE FUNCTOR CONSTRAINT WHILE GAINING PERFORMANCE

Upon inspection a pattern can be noticed in the effect values used in extensible effect systems. The "functor" part of the value i.e. the one being transformed upon application of `fmap` tends to be a function of type `t → a`, where `t` is the type of value returned by the effect and `a` is the type parameterising the functor. In cases where the effect returns nothing this field is always type `a` which could be rewritten to `() → a` to fit this pattern.

This is a known pattern exploited by the *coyoneda functor*, shown in figure 15. The coyoneda functor can be thought of like a "free functor" and analogous to the free monad it provides a functor instance for any type of kind `* → *`.

By exploiting this pattern and combining the coyoneda functor with the free monad we can construct a "freer" monad, which works on any higher kinded effect value and requires no functor instance.

Figure 16 shows the augmented free monad with the `Impure` case now containing *two* fields, one for the effect used and one for its continuation as can be seen in figure 16.

Decoupling effect and continuation this way is what enables the use of the type aligned sequence in section 5.2.

```
data Freer f a where
  Pure :: a → Freer f a
  Impure :: f x → (x → Freer f a)
         → Freer f a
```

Figure 16: The augmented free monad with decoupled effect

```
data ConsoleIO a where
  ReadLine :: ConsoleIO String
  WriteLine :: String → ConsoleIO ()

runConsoleIO :: MonadIO (Eff effs) ⇒
Eff (ConsoleIO :+: effs) a → Eff effs a
runConsoleIO =
  interpret $ \case
    ReadLine → liftIO getLine
    WriteLine line → liftIO $ putStrLn line
```

Figure 17: Freer version of the console IO effect

Since the continuation is no longer hidden in the effect functor and has a the general structure of `a → Eff eff b` it no longer needs to be an actual function, but can be replaced by something which can be used *like a function*, in case of the freer monad a sequence of continuations.

Uncoupling effect and continuation results in a much more expressive interface for effects. Whereas before each effect contained some additional fields for continuations, now their mere signatures concisely express *what* the effect actually does. Freer effects use GADTs to express which forms an effect may assume. Thus an effect is associated with some potential inputs, stored as fields in the effect value and later available in the effect interpreter and the type of data returned from the effect to the computation via the type variable parameterising the effect type. For instance in figure 17 is the console effect from section 3 and figure 4. The two constructors of the effect now describe, in types, what to expect, similar to a type signature in traditional effectful computations. `ReadLine` takes no input and returns a string, indicating that this effect will fetch some `String` value *from* the environment, whereas the `WriteLine` effect receives a `String` as input and returns the unit value, meaning no output is produced, which indicates that a `String` value is delivered *to* the environment.

Lastly the handler for an effect no longer has to care about the continuation. Effect handlers are, in their simplest form, functions of type `f a → Eff effs a`, where `f` is the effect type, for instance `ConsoleIO`, and `effs` does not

contain `f` anymore. An example, how simple such a handler may look, can also be seen in figure 17.

With the `Typeable` requirement gone (see section 4) as well as the `Functor` constraint, expressive effect signatures and simple handlers, the interface for defining and handling effects becomes small and easy to implement. Complicated effects can be remapped to standard effects which can be handled generically, such as the `State`, `Reader` or `Error` effect. The combination of these features mean that effects are quick and easy to define. This makes it feasible to implement systems with a much smaller finer effect granularity, which in turn makes the effect reusable, easier to reason about and test. An important contributing factor for the viability of this finely granular approach is the improved performance characteristics as described in the last section. Due to the linear scaling of freer monads finely granular effect systems can be employed without having to expect massive performance penalty as would be the case when using MTL.

An additional feature of the interface of the two systems described in this and the previous section is that multiple of the same effect can be combined in the same computation. Namely parameterised effects, such as `State s`, which is parameterised over the concrete state type `s` can occur multiple times in the effect set, so long as it handles a different `s`. Thus a computation of type `Eff [State Int, State String] a` is possible and will compute as would be expected. If multiple of identically instantiated effects are present, for instance `Eff [State Int, State Int] a`, the effect closer to the head of the list is handled and used first, then the latter. This is seldom used in an actual computation, but may occur due to a handler pushing a new state layer onto the computation, which is subsequently handles. This type of delegation is highly useful when dealing with complex custom effects and thus a feature. One slight problem with allowing multiple types of the same effect is that it becomes more common to have to annotate polymorphic functions such as `put` and `get` with concrete types to disambiguate the used state type, even if according to the constraints of the function using it, only one type of state is in scope. However it does afford finer granularity of effects and composability. Furthermore since defining custom effects that need no disambiguation is so simple with these systems, it should not pose problematic.

7 ALTERNATIVE EFFECT SYSTEMS

Effects are a common topic in the field of pure functional programming languages. Particularly the recent years have enjoyed the development of several new effect systems with a similar aim to the freer monads. Namely providing extensible and performant effect systems.

7.1 Handlers in Action

Kammar et al. [8] Developed a an alternative system, building on free monads, similar to the one from Kiselyov and Ishii [9]. The system, called "Handlers in Action" (HIA), also leverages an augmented free monad, see figure 16, the performance of which is enhanced by using the continuation monad `Cont`, a construct which is very similar to the `Codensity` monad as described before and fulfils a similar role here. Whereas freer dealt with suites of effects, `ConsoleIO` for instance is a suite consisting of two effects, `ReadLine` and `WriteLine`, "Handlers in Action" defines every effect individually. Every effect thus becomes its own top-level construct. Handler functions can choose which set of effects to handle, as well as whether additional effects, which are delegated, should be allowed or not. Handlers that allow additional effects are called "shallow" and the default case for freer. Handlers do not allow additional effects are called "deep" handlers. Deep handlers can provide performance benefits due to fewer dispatches.

To implement this in a user friendly way Kammar et al. leverages the code generation tool "Template Haskell" to generates type classes, constraints and data structures for effects, handlers and functions using effects.

In this approach no open union is needed. Instead each effect is directly paired with its handler via a `Handles` type class. Since each handler function is member of a type class it will be a static top level value, thus reducing the memory overhead because no additional `Union` value is needed as well as improving the speed slightly since not every handler attempts to handle every effect, but only the appropriate handler handles its own effect. However this depends largely on whether or not the compiler inlines the instance dictionaries.

In the performance evaluation in [9] the HIA system performed on-par or better than freer. In fact for the simple example of the state monad it even outperformed MTL. As far as I can tell this is due to the fact that, unlike freer, HIA uses more builtin structures, like the continuation monad and type classes. These structures are more transparent to the compiler, and have been around for a long time, thus enjoy more dedicated optimisations. Type level sequences and generic unions, as used by freer, are opaque to a large degree and recent additions to Haskell and hence harder to optimise by the compiler.

In addition to Haskell HIA also provides implementations for Racket and OCaml. Due to the less powerful type systems of these two languages the implementation of the handlers is slightly different, however the semantics of the ensuing program are similar. They use runtime tracking of handler functions in scope, which is a more dynamic approach. An

advantage of this system is that new operations can be defined at runtime which simplifies certain tasks. One such example (taken from the paper [8]) is the representation of mutable references by a set of dynamically created `Put` and `get` operations.

7.2 Algebraic Effects in Idris

Certain of the features of the effect system depend on the language used to implement it. The Idris [1][3] programming language is fully dependently typed and thus more powerful than Haskell. One result of this is that effects can be even more finely disambiguated. Whereas in freer and extensible effects several effects of a polymorphic type can be disambiguated in the same computation, see section 6, in the effects system implemented by Brady [4] in Idris, called "Effects", even effects instantiated with the exact same types can be disambiguated by means of adding labels to a particular effect. Subsequently the handlers are selected based on the label, not the type alone. An example of how this proves useful is if a computation were to carry two counters which are incremented separately. Rather than having to redefine an effect the same handler can be used and a label to disambiguate which counter is to be incremented.

Another key difference in Idris is that effect handlers are defined via type classes, similar to "Handlers in Action", as opposed to the user-defined handler compositions of freer. In Idris case handlers are parameterised over a context monad in addition to the effect itself. This allows it to be more versatile and implement different handlers for different contexts. A type class based approach can reduce the amount of code necessary to run effects, it does however increase the amount of boilerplate necessary to run handlers in custom contexts, such as when mocking for testing.[2]

The effect type in Idris is also more powerful when it comes to the type of interactions that can be described by it. Effects are parameterised over a type of input resource and a type of output resource. These are type level constructs that live in the environment of the effect monad. An example from the paper by Brady [4] is a `FileIO` effect which allows reading and writing to files. Rather than returning some kind of file handle when opening the file and writing to the handle the `OpenFile` action instead records the file in the computation environment including the mode of opening. Neither reading nor writing takes a handle as argument, but instead retrieves the resource from the environment automatically. Similarly close file retrieves the ambient value but also removes the resource type from the environment. As a result a closed file can never be written to or read from, since the type checker detects that this resource was removed from the environment at compile time. Similarly reading and writing cannot occur on files opened with the incorrect mode, due to the mode also being recorded in the environment of the effect. To open several files in the same computation the aforementioned labels are used to disambiguate the effect targets.

A downside of the Idris approach is that the effect monad is no longer a monad conforming to the `Monad` type class, because effects may change the the environment resources and thus the type of the computation. As an alternative Idris supports operator overloading to leverage the convenient `do` notation regardless. Unlike previous approaches this effect system does not use a simple augmented free monad but a much richer data structure that encodes several other actions in addition to the two monadic actions `pure` and ≫=. This is due to the larger feature set of "Effects" compared to the systems mentioned previously.

7.3 PureScript

One honourable mention at this point goes to the PureScript [2] language. PureScript is a pure functional language, similar to Haskell, that compiles to JavaScript. JavaScript has a lot of different built-in effects, such as console I/O, DOM interaction, Websockets, Browser history modification, and even more when NodeJS is used, adding process spawning, file interactions et cetera. To deal with such myriad of effects, while avoiding the IO-Monad problem, as it exists in Haskell, PureScript has built in support for a an extensible effects monad (`Eff`).

An unfortunate issue in Haskell is that all effects dealing with some kind of external API live in the monolithic `IO` monad. A signature `IO a` does indicate that *some* kind of effect occurs, but not which. It could be reading a file, or launch missiles, an infamous example of how arbitrary effects are from the Haskell community. PureScript addresses this by providing the `Eff` monad as a built-in which is parameterised over an extensible record of effects. The interface is very similar to the freer monad from the previous section. The difference being that all effects in PureScripts `Eff` are native effects. Each effect type is an untouchable empty native type with no runtime representation and it carries no data. Similarly the tagging of the imported native functions with effects is done arbitrarily by the user and no alternative, "pure" implementations directly in PureScript are possible.

However the use `Eff` as a built-in structure as well as using an extensible record to track the composition of effect types means that all of the code using `Eff` can be optimised to such a degree that no overhead remains at runtime. Both effect resolution and ≫= inlining can be done at compile

[2] The reason is that in order to choose different handlers a newtype has to be instantiated as base monad and instances provided for every effect needed. Whereas in freer predefined handlers can simply be composed anew.

time. Furthermore having built-in extensible records result in a concise syntax.

8 CONCLUSION

This paper presents the conception and development of extensible effect systems. It shows how a monad capable of handling extensible sets of effects can be constructed by decoupling the effect from the monad using a free monad and replacing concrete effects with an extensible union type. The implementation is reviewed and optimised to attain competitive performance with transformers and a minimal interface for effects is found resulting in several improvements over current solutions:

- Defining new effects involves minimal boilerplate. Necessary components are only an encoding data type and an interpreter.
- Computations using effects are constrained using the same member constraint regardless of effect, relieving the need for effect classes and instances.
- The effect monad is much more transparent, enabling combinators for effect relaying and generic handling.

Other, similar, solutions are examined and compared, which suggest that even more powerful effect systems can be built upon these foundations and that extensible effects may be a safer, more expressive alternative to monolithic monads such as IO.

REFERENCES

[1] [n. d.]. Idris: A Language with Dependent Types. https://www.idris-lang.org/

[2] [n. d.]. The PureScript Programming Language. http://www.purescript.org/

[3] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

[4] Edwin Brady. 2013. Programming and Reasoning with Algebraic Effects and Dependent Types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2500365.2500581

[5] R. John Muir Hughes. 1986. A novel representation of lists and its application to the function "reverse". *Inform. Process. Lett.* 22, 3 (1986), 141 – 144. https://doi.org/10.1016/0020-0190(86)90059-1

[6] Mark P. Jones. 1995. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming*, Johan Jeuring and Erik Meijer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 97–136.

[7] Mark P. Jones. 1995. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming* 5, 1 (1995), 1–35. https://doi.org/10.1017/S0956796800001210

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

[9] Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*. ACM, New York, NY, USA, 94–105. https://doi.org/10.1145/2804302.2804319

[10] 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 (Haskell '13)*. ACM, New York, NY, USA, 59–70. https://doi.org/10.1145/2503778.2503791

[11] 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 (POPL '95)*. ACM, New York, NY, USA, 333–343. https://doi.org/10.1145/199448.199528

[12] Eugenio Moggi. 1990. *An abstract view of programming languages*. University of Edinburgh. Laboratory for Foundations of Computer Science.

[13] Atze van der Ploeg and Oleg Kiselyov. 2014. Reflection Without Remorse: Revealing a Hidden Sequence to Speed Up Monadic Reflection. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell (Haskell '14)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2633357.2633360

[14] Guy L. Steele Jr. 1994. Building Interpreters by Composing Monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. ACM, New York, NY, USA, 472–492. https://doi.org/10.1145/174675.178068

[15] Wouter Swierstra. 2008. Data types à la carte. *Journal of Functional Programming* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

[16] Janis Voigtländer. 2008. Asymptotic Improvement of Computations over Free Monads. In *Mathematics of Program Construction*, Philippe Audebaud and Christine Paulin-Mohring (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 388–403.