Why Data is the Better Monad

Using Freedom to Great Effect

Justus Adam
justus.adam@tu-dresden.de

ABSTRACT

Side effects are the sore spot for pure languages. The mathematical beauty of referential transparency is broken to allow outside interactions that are unpredictable and hard to reason about. Interactions such as network and filesystem access for example, that depend on invariants in the context they are executed in and may unexpectedly fail, but are necessary to write useful programs.

Fortunately, with the Monad, an abstraction was discovered that allowed safe and transparent, sequential interaction with an arbitrary ambient environment.

Yet it too is flawed. Versatile though the Monad is, composing effects in a reusable fashion proved difficult and has performance implications which increase in severity as the granularity of effects gets finer.

With the free monad we now finally not only obtain a way to compose arbitrary effects together in an efficient way but can also easily implement effects via transformation into other effects and even implement the same effect in different ways depending on needs and constraints of its application.

This paper provides an overview of the origins of the Monad, how its shortcomings lead to the invention of transformers, then the free monad, and eventually the freer monad. How free and freer monads work and how they can be used to realise our collective dream of pure, efficient, composable and versatile ambient interactions.

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, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Imperative languages, as well as many functional ones, make it decidedly easy to execute side effects. While many see this as an advantage in the sense that it "makes programming easier", particularly members of the functional programming community are of a different opinion. The term "side effect" characterises all such interactions that are not evident from a functions type signature, which includes interactions with the operating system or modification of global structures in the program. What makes this complicated in particular is that programs become much harder to reason about in the presence of arbitrary side effects due to the increased degree of freedom of all function. The alternative is a concept called "purity", which removes arbitrary side effects and forces functions to declare all dependencies and results in the type signature. Due to the useful nature of side effects however even languages that are "pure" must incorporate them.

The most prominent representative of pure functional languages is Haskell. It has incorporated a concept called a *Monad*, which has since been adopted by other, similar languages, to capture the sequential nature of computations involving side effects, encode effectful computations as values and thus allow them back into the language in a controlled and safe manner.

However using monads for effectful computations has certain disadvantages. A major one being that monads themselves do not compose well. What makes side effects so convenient in impure languages is that any type of effect can be used together with any other type of effect. Achieving the same in pure languages has been the subject for over two decades of research with one of the most influential papers being [8] by Jones. Their approach of using "monads with a hole", in this paper referred to a *transformers* or *monad transformers*, has shaped the library and application design in Haskell ever since.

However these monad transformers are unwieldy in many situations and do no exhibit good performance as the number of composed effects increases. A new approach is receiving a lot of interest in recent years: "extensible effects". Here a single monad is used parameterised with a set of effects, as opposed to the monad stack used by transformers.

Extensible effect systems encode computations as data which, on one hand, makes it easier to compose various effects, and on the other allows for effects to be handled in context dependent ways.

This paper explores in greater detail how monads are used (section 2) and what makes them unwieldy, how transformers attempted to solve the problem (section 3) and succeeded only partially. The largest section is attributed to extensible effect systems, beginning with the parts they are built out of (section 4) how these systems were improved to achieve competitive performance to transformers (section 5 and 6) and finally how this interface makes monadic effects more usable. This part is largely based on the two iterations of extensible effects as implemented by Kiselyov and Ishii, as it showcases nicely the progression of the system. Section 8 presents a select few similar systems, comparing and contrasting with the system from Kiselyov and Ishii, showing both the potential for new features in the future as well as providing evidence for the universality of the approach to the underlying implementation.

2 MONADS

Purity, often also called referential transparency, is the property of a function to always produce the same result when called with the same arguments. It is a desirable property as it makes it considerably easier to reason about the behaviour of a program, particularly with

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

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

respects to refactoring. But it also allows optimisations, such as common subexpression elimination and memoization.

However, many tasks that we wish programs to perform cannot be expressed in terms of pure functions as they entail an interaction with the world outside of the program, such as accessing a database or querying the file system. Even inside of the program it is often useful to define some ambient environment in which certain tasks are performed, from this object oriented programming was conceived where each function, or method, carries around the implicit environment of an object.

To perform the aforementioned tasks many programming languages opt to sacrifice purity and allow arbitrary side effects in any function. The designers of the Haskell programming language however found a way to enable side effects in programs without sacrificing purity. **Monads**, in the Haskell sense, are a class of types that describe some sort of environment, which can be interacted with. The concrete interactions a particular monad allows differ but they have a common notion of sequentiality, encoded with the **bind** operator (`(≫=) :: Monad m ⇒ m a → (a → m b) → m b`), which connects an action performed with the monad and a continuation that requires the result of this interaction, and the ability to embed pure values into the monad (`pure :: Monad m ⇒ a → m a`).

Of particular importance is the sequentiality of actions that is enforced by ≫=. Since the left hand argument is a continuation the right hand action must be performed before the program can advance. This structure allows ordering of side effects that may or may not produce actual results, such as writing to a table in a database.

The Monad proved to be a very successful concept in describing sequential interactions with ambient environments and thus many different monads had soon been developed. Monads for interacting with the system (`IO`[1]), for interacting with databases (`Redis` [2]) or handling web requests in a server (`HandlerFor`[3]) as well as monads that defined environments embedded in the program such as `Writer`, which collects outputs, `Reader`, which adds a static environment and `Except`[4], which adds throwing and handling of user defined errors.

Though each of these monads are well suited for interacting with the various environments they describe, they are rather unwieldy, if not impossible to use when we wish to interleave their effects. We may desire to read some data from the `Redis` database, perform some network `IO` afterwards and finally send some computed result as part of being a `HandlerFor`, all the while tracking potential `Except` errors.

This however is not possible with a simple monad. Most of these monads are opaque types, results from which can only be obtained by performing complex set-up and tear-down operations, and some, like the `HandlerFor` monad offer no way for the user to extract the pure data directly. This makes sense of course as the creation of this data entails certain opaque interactions with the environment from which it cannot be easily untangled. Though the power of the monad lies precisely in leaving those interactions opaque it poses a challenge when trying to achieve composability and interleaving of effects.

The next section offers an overview of the current most popular solution for achieving composability: effect classes and monad transformers.

3 TRANSFORMERS AND CLASSES

The need for ambient effects is different for each program. Whereas effect monads like `IO` and `HandlerFor` are absolutely required, as they provide the only facility to interact with certain resources, additional "convenience" monads such as `Reader`, `Writer`, `State`[5] and `Except` are "optional" and provide easier interfaces for describing certain types of computation.

For each domain there is a different set of effect combinations particularly suited to describe it. When terminal monads are used, like those mentioned in the last paragraph, an entirely new monad would have to be implemented for every combination of effects we desire. Alternatively one large monad, which includes **all** effects could be used, but then each effect would necessarily have to be handled when the monad is run, regardless of whether the computation actually uses it. And how would the domain specific effects like `IO` and `HandlerFor` be added to this monolithic monstrosity?

Clearly neither of these approaches is well suited to solve the need for interleaved effects. A system is needed whereby larger, more complicated, monads can be composed of smaller, elemental ones. Ideally this could be done dynamically, as some sections of the code may require additional effects to be added, like adding a `Writer` to accumulate the results of some embedded computation.

The most popular solution currently is the one implemented by the `mtl`[5] and `transformers`[6] library and it revolves around using stacks of so called **monad transformers** and **monad classes**. The approach was inspired by a paper from Mark P. Jones [8]. The idea is that, rather than defining terminal monads, such as `Reader`, the defined monad is parameterized by an additional *inner monad* to which other effects can be delegated. The resulting structure is called a *transformer* as it transforms a given monad by adding an additional type of effect. Transformers are typically named after their effect with a capital "T" prepended. Thus the transformer for the `Reader` monad would be `ReaderT`.

This process of wrapping a monad with a transformer can be done arbitrarily often as the result of wrapping a monad with a transformer again produces a monad which can become the inner monad to another transformer. The resulting structure is often also referred to as a transformer *stack* that is terminated at the last level by some terminal monad which cannot be used as a transformer, such as `IO` or `HandlerFor`.

When the computation is executed the layers of transformers are removed individually, the outermost layer always first until a computation in the base monad remains. Transformer stacks can even be used to describe pure computations by choosing `Identity` as the inner monad, which resolves to a pure value. Additionally, during a computation, new (outer) layers can be dynamically added

[1] Part of the `base` library
[2] Part of the `hedis`[13] package for interacting with the Redis Key-Value-Store
[3] Part of the Yesod [15] web framework
[4] `Except` does not actually exist. Only the transformer `ExceptT` does as `Except` would be the same as `Either`. `Except` is only used here to make the connection to its transformer more obvious

[5] Similar to the `Reader` monad but the environment is not static and can be altered during the computation

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

Figure 1: The monad class for the `State` effect

```
writeState :: ( MonadState s m
              , MonadWriter s m
              , Monoid s )
           ⇒ m ()
writeState = get ≫= tell ≫ put mempty

writeState1 :: ( Monad m
               , Monoid s )
            ⇒ StateT s (WriterT s m) ()
writeState = get ≫= tell ≫ put mempty
```

Figure 2: Polymorphic and explicit effect signatures

and removed from the monad. As such for instance a subcomputation can be run with an additional `Writer` effect, that is then separately unwrapped to obtain the written results.

The characteristic action of a transformer, as captured by the `MonadTrans` class, is `lift :: (MonadTrans n, Monad m) ⇒ m a → n m a` which can be seen as embedding an action of the inner monad in the outer monad, or, alternatively, as delegating (lifting) an effect to be handled by the inner monad. Unfortunately this lifting operation poses a problem when the transformer stack get larger. Performing an effect which is far up the stack will require several lifting operations to reach the transformer that can actually satisfy it.

To deal with this issue another concept is used in the form of **monad classes**. Rather than defining effects on a transformer directly a type class is defined with the interface for the effect in question, see for instance the effect as provided by the `State` monad in figure 1. Transformers responsible for handling the effect are then made instances of this class and implement it accordingly. Additionally all other transformers are also made instances of this class but use `lift` to delegate the effect up the stack. Thus when such an effect is used it automatically propagates up the stack until it reaches the transformer capable of handling it (should one exist[6]).

An additional advantage of monad classes is that effectful computations can be defined solely in terms of the set of effects it uses, rather than the concrete monad it operates in. As a result they can be reused in entirely different monad stacks without modification of the code so long as it satisfies the same effect interface. An example of this can be seen in figure 2. Whereas the second function, `writeState1`, can only be used in the explicit `StateT`, `WriterT` stack, the first function, `writeState`, could also be used in a stack where the two transformers are reversed, or if it were wrapped by additional transformers.

While this provides a clean interface for interacting with the various effects in our stack, using the class-based effect functions like `get`, it also poses some issues. In the interface definition for the `State` effect, for instance, the type of state, which can be handled, is fixed to the monad, as expressed by the `m → s` functional constraint. This means a stack can only ever handle the state effect for *one single type of state s*. Adding additional `StateT` transformer will shadow any `State` effect of its inner monad and render it unreachable using the `MonadState` class [7]. This becomes particularly troublesome when libraries are used that implement certain interactions in terms of effects on a specific type, like `Reader` on a specific type `t` but the monad it is supposed to run in is already a `Reader` for some *other* type. In these cases it becomes rather tedious to wrap and unwrap every library interaction with another `ReaderT` to shadow ones own `Reader` effect. This problem can be eased using lenses but requires that the library be defined both in terms of lenses as well as monad classes.

Monad class instances themselves also pose issues. In general it is beneficial to define custom effects in terms of new monad classes and transformers. This makes these effects extensible, composable and reusable, properties which are desired, particularly in large projects which are frequently subjected to refactoring and expansion.

However defining you own transformers is tedious. Not only does one have to implement the transformer itself but additionally provide a monad class instance for every *type of effect* one could possibly combine with the new transformer. Additionally a new class for the custom effect as well es instances of this class for every *existing transformer* is necessary to achieve the maximum amount of composability. In case of GHC some of these can be automatically derived by the compiler, but only when `newtype` wrapping is used. When the granularity of the custom effects is decreased, that is they are broken into smaller and smaller pieces to achieve modularity, more and more of such definitions and instances are required, bloating the code.[8]

A truly extensible system should not require changes to existing effects in order to incorporate new ones. Ideally we would only have to define our effect interface and a means to perform them and the system would implicitly know how to combine it all other effects.

The next section introduces a first such system by first removing the boilerplate of defining monads in the first place and then moves on to extend this into a composable system of effects.

4 FREEDOM TO THE RESCUE

In Section 2 the monad was introduced as a way to sequence effectful computations. Then various types of monads were found, encoding different effects.

4.1 Decoupling Monad and Effect

One of the most important discoveries [16] was that the effect and its sequential nature can be separated and the latter encoded with a generic data structure. The essence of the monad, ≫= and `pure` is captured using the so called **free monad** (see figure 4).

[6] Should no capable handler exist the type checker will report a class instance resolution error.

[7] Theoretically it is still reachable using a combination of explicit `lift` and the effect function but using `lift` reintroduces the issues from before that prompted the addition of monad classes in the first place.

[8] One particularly good example of this I believe is the `monad-logger` library, paticularly this module https://github.com/snoyberg/monad-logger/blob/master/Control/Monad/Logger.hs where they even resort to the use of preprocessor macros to reduce the code duplication.

```haskell
class Functor f where
  fmap :: (a → b) → f a → f b
```

Figure 3: The `Functor` interface

```haskell
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
  return = 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 4: The simple free monad

```haskell
join :: Monad m ⇒ m (m a) → m a
join ma = ma >>= id

(>>=) :: Monad m ⇒ m a → (a → m b) → m b
ma >>= cont = join $ fmap cont ma
```

Figure 5: The equivalence of ≫= and `join`

This generic monad is parameterized by an effect type `f` which itself only needs to be a `Functor`. Functors are much simpler structures than monads. They have no notion of sequentiality and continuations which makes them easier to implement. A Functor only needs to allow the application of a function to a contained value as illustrated by the type class `Functor` displayed in figure 3.

The two constructors for the free monad represent the two characteristic functions of the `Monad` in category theory, `pure` and `join`, the latter represented by the `Impure` constructor. The `join :: Monad m => m (m a) -> m a` function allows flattening of two stacked, identical monadic contexts. It, in conjuction with `pure` is sufficient to characterize a monad the same way that ≫= does since either one can be implemented in terms of the other, see figure 5.

In case of the `Impure` constructor the continuation (`Free f a`) is contained in a layer of the effect functor `f` meaning that proceeding with the computations requires execution of one instance of the effect. Only then is an interpretation of the inner (later) computation possible, including the effects used later in the computation. This ensures effects are executed in the same order that they are used in, in the program.

Using this data structure removes the need define a `Monad` instance for the effect type `f`, as the free monad is a true `Monad` for *any* `Functor f`, see figure 4, this also means all the useful combinators defined for monads are available to an effect system defined

```haskell
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 lines cio = go lines [] cio
  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 6: Implementing a simple console I/O interaction using `Free`

like this. Effects are entirely delegated to the functor `f` allowing completely independent and different kinds of effect systems to be implemented using `Free`.

To run a computation expressed in a free monad effect system is done by a function typically called an *interpreter*. This interpreter implements how the effects should be handled in a certain environment. One additional advantage of using an interpreter is that effects can be handled differently depending on the concrete interpreter. They can even run in different environments entirely without requiring a change in the computations code.

An example can be seen in figure 6 where a simple console interaction is implemented using the free monad. Here two very different interpreters are provided. One where the interpretation happens in the IO monad and the two requests `ReadLine` and `WriteLine` are directly delegated to the usual I/O functions and a second one where the requests are served from a list of input strings and the written lines are again recorded in a list. This power of the free monad effects to be interpreted in different ways makes it ideal to implement mock interpreters for testing functions as well as making code more reusable in different scenarios. Transformers for instance do not offer the same degree of flexibility.

4.2 Composing Effects

Decoupling the monad from its effect enables the composition of effects using functor composition. Two effect systems implemented using the free monad can be composed into an effect system implementing both types of effects by composing the two effect functors

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

Figure 7: Then 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 :<: sum) ⇒
f :<: (g :+: sum) where
  inj = Inr . inj
  prj (Inr sum) = prj sum
  prj _ = Nothing
```

Figure 8: Dispatch classes for effects in a simple open union

into a single one and using the result functor to parameterize the free monad. When handling the effect the two functors are decomposed again and the individual effects are dispatched to corresponding handlers.

The simplest way to combine effects is the sum functor (`:+:`) as seen in figure 7. If combines two functors `f` and `g`, dispatching requests accordingly. We can use this simple union of two types to express functors that combine arbitrary types by nesting `:+:` and add projection (`prj`) and injection (`inj`) via a subtype relation class `:<:`, see figure 8. This way we form a so called *Open Union* of functors. The sum functor and the open union is a technique developed by Wouter Swierstra [16] and based on the `OR` type from Liang, Hudak and Jones [12], lifted to the higher kinded functors.

Using an open union like this as the base functor for the free monad allows us to parameterize functions over the types of effects they use without directly describing the structure of the functor and without the need for additional type classes and class instances for every new effect. An extensible version of the `readLine` and `writeLine` functions from the earlier example can be seen in figure 9. Whereas the earlier version from figure 6 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.

As the functions are now independent of the concrete type of underlying functor we can combine effects freely. In figure 10 for instance we can see another effect which retrieves the current time. A function that wishes to use both effects can now do so, again without having to specify any particular structure of the functor but simply gaining a constraint, see figure 11.

In addition the combination of effects we can also implement an `interpose` combinator which can be used to alter effects,

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

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

Figure 9: An extensible version of the console interations

```
  CurrentTime
  deriving (Show)

data Time a =
  GetTime (CurrentTime → a)

instance Functor Time where
  fmap f (GetTime g) = GetTime (f . g)

getTime :: (Functor f, Time :<: f) ⇒ Free f CurrentTime
getTime = Impure $ inj $ GetTime pure
```

Figure 10: An effect for retrieving the current time

```
reportCurrentTime :: (Console :<: f, Time :<: f) ⇒ Free f ()
reportCurrentTime = getTime ≫= writeLine . show
```

Figure 11: A function using both `Console` and `Time`

```
interpose ::
     (f :<: sum, Functor f, Functor sum)
  ⇒ (forall b. f b → Free sum b)
  → Free sum a
  → Free sum 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 12: The interpose combinator

see figure 12. This can be used for instance to attach a timestamp to each message sent to the console by use of the `Time` effect or implement input filtering, see figure 13.

4.3 Handling Effects

While it is not necessary to know about the concrete structure of the effect functor to use its effects it is necessary to handle them. The nesting of sum functors forms a linked list and effects have to interpreted starting from the head of the list. Various combinators are available to handle these effects. In figure 14 the `interpret` combinator is shown which interprets an effect in terms of other effects present in the list. Similarly it shows the signatures of other combinators. Some effects, like the `Reader` effect shown in figure 15 can be interpreted directly. Others may need to be delegated. The idea is that we either handle effects or delegate until we have unrolled the whole effect list and are left with some base monad (often `IO`) which we run directly using the `runM` or with no effect at all and we can simply return the pure value with `run`, both can be seen in figure 14.

```
addTimestamp ::
     (Functor sum, Time :<: sum, Console :<: sum)
  ⇒ Free sum a
  → Free sum a
addTimestamp =
  interpose $ \case
    WriteLine line cont → do
      ts ← getTime
      writeLine (show ts ++ line)
      pure cont
    other → Impure $ inj $ fmap pure other

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

Figure 13: Two examples for using the interpose combinator

```
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 14: Combinators for interpreting effects

```
data Reader e a =
  Ask (e → a)

ask :: (Reader e :<: effs, Functor effs) ⇒ Free effs e
ask = Impure $ inj $ Ask pure

interpretReader :: e → Free (Reader e :+: effs) a → Free effs a
interpretReader env = interpret $ \(Ask cont) → pure $ cont env
```

Figure 15: The reader effect and its interpretation

```
  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) = liftIO (putStrLn str) ≫ pure cont
```

Figure 16: Interpreting the extensible console effect

It may not seem immediately obvious, but this retains the same flexibility we saw earlier in the `Free` monad whereby effects could be interpreted differently based on context. In this case it is even easier to do this. For instance in the example of handling the console effect our computation will have a type of `(Console :<: sum, Functor sum) => Free sum a`. If this is to be interpreted in the `IO` monad `sum` can be instantiated to be `Console :+: IO` and interpreted with the `interpret` function. All console effects are delegated using the `liftIO` helper and the two handlers `putStrLn` and `getLine` which were also used in the earlier example in figure 6. A computation of type `Free IO a` remains, which we can be run using the `runM` function. If instead the interpretation should happen in a pure context, similar to the example before, `sum` can instead be instantiated to `Console :+: Identity`. Using the `reinterpret` function, the `Console` effect is turned into a `State ([String], [String])` effect. `State` is a standard effect and its implementation omitted here. It can be run using the `runState :: s -> Free (State s :+: sum) a -> Free sum (a, s)` handler. After interpreting `State` only `Free Identity (a, s)` is left, which fits the type signature for `run`. Using `run` the final type will be `(a, ([String], [String]))`, which is the result of our computation, as well as the remaining, unread lines of input and the lines written to output. Code for this example can be viewed in figure 16. In short we retain the flexibility of context-dependent interpretation because our computation is unaware of the underlying effect layers and thus we are free to instantiate them as necessary.

4.4 Summary

Using a free monad we can decouple the computation using an effect from the way that effect is handled. Effects themselves need only concern themselves with encoding the effect in a functor and the free monad provides the sequentiality "for free". By using an open union the functions using the effect can be implemented unaware of the concrete structure of the underlying effect functor. This means that new effects can be added in a modular fashion without the need to change the functions using them.

Interpreting the various effects is highly flexible and can be changed depending on the context of the computation, allowing

easy mocking for instance. Furthermore using the open union allows the interpretation of effects in terms of other effects that can be handled easier or are generic effect types.

What was set out to do has been achieved, a highly modular, easy to extend and flexibly interpretable system for defining effectful computations.

However an issue remains with regards to efficiency. Particularly the implementation of an open union using a linked list of functors means that some part of the list has to be traversed on every `fmap`, which is used ubiquitously in the implementation. This also means that the performance degrades as the number of different effect types grows. Smaller effect granularity, which leads to greater reusability and makes it easier to reason about effects, is punished by degrading performance, an undesirable effect. Furthermore the ≫= implementation also recurses into the continuation on every bind, leading to bad asymptotic performance. Section 6 will explore an alternative way to implement the free monad to improve the performance of bind. Section 5 shows ways to implement a more efficient open union.

5 A BETTER OPEN UNION

Monad transformers have been a part of the Haskell ecosystem for a long time. They are the de-facto standard method used to implement modular effect systems. Given how useful and necessary, if not unavoidable, effect systems are for "real world applictions" it is not surprising that monad transformers are now widely used in libraries and even more so in applications and frameworks. being the biggest player in the game and ubiquitously used comes with benefits for a library. Namely, in this case, the attention of the compiler writers. According to Kiselyof and Ishii [10] the `State` monad in particular enjoys "preferential treatment by GHC" as the authors put it, "with dedicated optimisation passes". For an alternative effect system to be effective therefore it must provide at least comparable performance to the `mtl` library, it will not see significant adoption.

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 linked list is not only slow but also degrades as the number of distinct effects increases, discouraging modularity and reusability. This section will explore alternative ways of representing an open union with the aim to improve its efficiency. The next section will deal with improving the asymptotic performance of the bind operator ≫= for free monads.

In general union types, also called algebraic data types, are implemented as variably sized structures, prefixed by a tag of known size (typically a byte or machine word) which tells the inspecting party which concrete shape, also called constructor, inhabits this value.

For a closed union the possible shapes the value can assume are known at compile time, enabling efficient code generation and statically checked pattern matches. To implement an extensible effect system however the union cannot be closed. Neither the concrete types nor their order is known when the effect monad itself is implemented. Furthermore it is desirable to be able to add additional effects for subcomputations only. Therefore the union must support dynamically adding constructors. A key idea from Kiselyov, Sably and Swords [11] is that open unions can be implemented similarly to closed unions by coercing the concrete value to an untyped one and pairing it with a tag to discern its type at runtime. The first published version of this technique can be seen in figure 17.

```
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 17: A single value open union based on reflection
9

In its first instance the authors used the builtin reflection mechanism provided by the GHC compiler[17]. It generates a runtime representation of a type called `TypeRep`. Each such `TypeRep` structure carries a compiler generated 128 bit unique MD5 hash of that concrete type. `TypeRep`'s are static constants and support fast comparison by comparing their fingerprints.

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 et al. [10] 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 [10] 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. As a result matching values of this union type is almost as efficient as matching a native, closed union value.

6 ASYMPTOTIC PERFORMANCE OF ≫=

As mentioned before the asymptotic performance of ≫= for the `Free` monad is not good. As can be seen in figure 4 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.

6.1 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. A solution to this problem was independently developed in 2008 by Janis Voigtländer [18] in an attempt to make free monads more efficient. The idea is rather simple and again related to the problem

```
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 18: The codensity monad

of list appends. For list appends the solution is the so called *difference list* [7] which really is not a list but a function of type `[a] -> [a]`. This function will, when called with a list, append it to its end and return the resulting list. Normal list operations, such as appending and prepending are realised via function composition. This is often also described as building the list while leaving a "hole" at its end to be plugged with a terminating value once the construction is complete. One this value is provided, which is usually just an empty list, it can be built in a single step. In imperative programming this is known as the *builder pattern*. A hole is left to the end of the list and all altering operations are realised with function composition.

Asymptotic performance improvement of the free monad employs a similar idea. It uses a generic structure from category theory, which is now known as the *Codensity monad*. The original paper by Voigtländer [18] called it `C`. The codensity monad, shown in figure 18, is also a function which, given a continuation, builds some monad `m`. The *extensible effects* library, developed by Kiselyov et al [11] uses this codensity monad, specialised to a variant of `Free`. The figure also shows the new implementation for >>= which does not push down the tree anymore, but builds two lambda functions, which are passed as continuations, a much cheaper operation in Haskell.

6.2 Improving interpreter performance using type aligned sequences

The builder pattern, or programming with holes and continuations can drastically improve the construction of a value. However this only work when the structure is built in its entirety and then inspected in a second step. If the structure needs to be read *during* its construction this approach cannot be used. Because a function is an opaque object, the intermediate values of a list, or in this case a forming monad chain, cannot be read, necessitating an evaluation of the structure before reading. Subsequent constructions can be sped up again, but an intermediate structure was allocated to facilitate the reads, diminishing the achieved gains.

For the effect system, implemented by our free monads, in particular, this situation occurs in the interpreter. In order an effect we first must inspect the computation. When the effect has been handled the continuation has to be pushed through the monad, necessitating a full traversal of the (potentially growing) effect chain. The crucial point where this traversal is performed can be seen in figure 19 where the continuation is not simply appended, as was the case in figure 18, but traverses the entire stack, similar to figure 4.

```
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 19: Relay function as implemented in [11]

```
{-# LANGUAGE GADTs #-}

data TList a b where
  Nil :: TList a a
  (><) :: (a → w) → TList w b → TList a b

toFun :: TList a b → (a → b)
toFun Nil = id
toFun (f >< l) = f . toFun l
```

Figure 20: A type aligned list of functions

The principal problem her is that new continuations, in the form on lambda functions are created and partial applications are pushed into the functor. This is a typical case where allocation occurs in Haskell. While these are eventually reduced to smaller and simpler expressions, the intermediate allocations never the less occur which increases both the time needed and the space. As the evaluation from the extensible effects paper [10] shows this regression in time and space only occurs when a *used* effect is threaded through the computation, meaning the second clause in figure 19 is used. In the case where the reader effect is *under* the state, this clause is triggered much less often, explaining the linear performance. Such a performance degradation caused by the order of effects is highly undesirable. It would be difficult to a user of this system to utilise it correctly in the presence of such subtle differences influencing the performance.

A recent paper by van der Ploeg and Kielyov [14] designed a solution to this problem in the form of so called *type aligned sequences*. The approach is applicable much more broadly than only free monads. Van der Ploeg and Kiselyov represent chains of computations with data structures via the use of Generalized Algebraic Data Types or GADTs. 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. A small example of one such structure can be seen in figure 20. 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. Now the handler computations inspect only the head of the queue and push new types of effects back onto the front without altering the tail.

With the two improvements from these last two sections the computational complexity of extensible effects improves dramatically. Whereas the naive implementation from section 4 worse than quadratic in both time and space, the improved version performs

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

hoistCoyoneda ::
     (forall a. f a → g a) → Coyoneda f b → Coyoneda g b
hoistCoyoneda trans (Coyoneda alter f) = Coyoneda alter (trans f)

lowerCoyoneda :: Functor f ⇒ Coyoneda f a → f a
lowerCoyoneda = lowerWith fmap

lowerWith ::
     (forall a b. (a → b) → f a → f b) → Coyoneda f c → f c
lowerWith map (Coyoneda alter f) = map alter f
```

Figure 21: The coyoneda functor

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

Figure 22: The augmented free monad with decoupled effect

linearly with respect to both. This is not only an improvement over the MTL library. According to the performance evaluation in [10] the MTL library is *faster* for a single state effect, due to GHC specific optimisations deliberately targeted at the MTL `State` monad. However MTL does not scale well with more effects. It scales quadratically in both time and space, regardless of the ordering of effects.

7 A FREER MONAD

The "freer" effect system brings another advantage over the older "extensible effects". As mentioned in section 4 the definition of an effect system requires an effect `Functor`. This is not strictly true. It is convenient for the definition of a simple effect system, but can be elided. One way to do so is using yet another kan extension, a relative of to the *codensity monad* seen in section 6.1. This structure is called the *coyoneda functor* and it is able to lift an arbitrary `* → *` kinded type to a functor. Simply put it accumulates alterations applied with `fmap` in the first field. The contained structure can be extracted, if a function is provided to apply the accumulated alterations.

While this structure can remove the functor constraint it still couples the continuation with the effect. Kiselyov and Ishii [10] take it one step further and completely uncouple the two. To achieve this the free monad is augmented such that the `Impure` case contains *two* fields, one for the effect used and one for its continuation as can be seen in figure 22. In fact only having this decoupled monad is what enables the use of the type aligned sequence in section 6.2. 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

```
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 23: Freer version of the console IO effect

GADTs to express which forms an effect may assume. Thus an effect is associated with some potential inputs, which are later passed to the effect handler, via the fields of the effect value, as well as with the type of the data returned by the effect to the computation via the type variable parameterising the effect type. For instance in figure 23 is the console effect from section 4 and figure 6. 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 23.

With the `Typeable` requirement gone (see section 5) 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.

8 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.

8.1 Handlers in Action

Kammar et al. [9] Developed a an alternative system, building on free monads, similar to the one from Kiselyov and Ishii [10]. The system, called "Handlers in Action" (HIA), also leverages an augmented free monad, see figure 22, 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 [10] 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 [9]) is the representation of mutable references by a set of dynamically created `Put` and `get` operations.

8.2 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 7, 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.[10]

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

[10] 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.

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.

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

9 CONCLUSION

Extensible effect systems have enjoyed a significant amount of attention by the scientific community in recent years. They provide means to write code which is very expressive at the type level via fine grained compositions of effects and even improved type safety via resource tracking through the environment. Since such computations are expressed in data structures rather than opaque functions the can be inspected, altered, filtered or otherwise interpreted depending on context and requirements. This aids reusability of the code on one hand, and simplifies mocking and testing on the other. It also makes it easier to refactor code and change implementations, because the functions using the effects are implemented against abstract interfaces rather than concrete effects. The PureScript language is an example of a language that already employs an extensible effects system to structure its computations.

Extensible effects may finally provide a solution to the problem of monolithic monads such as Haskell's `IO`.

A popular solution is the effectful, extensible monad based on the *free monad* which has the desired characteristics when combined with an open union, or equivalent structure.

Thanks to additional effort being put into improving the performance of constructing a free monad and reflecting on the computations, extensible effect systems begin to outperform traditional systems such as MTL.

Regardless of the concrete implementation extensible effect systems have certainly succeeded in simplifying the definition of monadic effects as well as means to test, mock and reuse them. Extensible effect systems have certainly come to stay.

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] Andy Gill. 2006. Monad classes, using functional dependencies. http://hackage.haskell.org/package/mtl

[6] Andy Gill and Ross Paterson. 2009. Concrete functor and monad transformers. http://hackage.haskell.org/package/transformers

[7] 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

[8] 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.

[9] 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

[10] 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

[11] 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

[12] 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

[13] Falko Peters. 2011. Client library for the Redis datastore. http://hackage.haskell.org/package/hedis

[14] 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

[15] Michael Snoyman. 2010. Creation of type-safe, RESTful web applications. http://hackage.haskell.org/package/yesod-core

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

[17] The GHC Team. [n. d.]. The Glasgow Haskell Compiler. https://www.haskell.org/ghc/
[18] 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.