Why Data is the Better Monad

Using Freedom to Great Effect

Justus Adam

July 4, 2018

Table of Contents

Table of Contents

Why Monads?

- Answer to the question “How to do pure side effects?”
- Associates values with side effects used for its production
- Can be thought of as an environment for a value
- Most iconic Monad: `IO`

Desugaring

```
f :: IO ()
f = do {
  putStrLn "Abort?␣[y/N]";
  answer <- getLine;
  when (answer == "y") (exitWith ExitSuccess);
  ... }
```

Desugaring

```
f :: IO ()
f = do {
  putStrLn "Abort?␣[y/N]";
  answer <- getLine;
  when (answer == "y") (exitWith ExitSuccess);
  ... }

f =
  putStrLn "Abort?␣[y/N]" >>
  getLine >>= \answer ->
  when (answer == "y") (exitWith ExitSuccess) >>
  ...
```

Overloading example

```
ssa :: Expression -> CompileM Expression
ssa (Assign vname expression) = do {
  exists <- isNameInScope vname;
  newName <-
    if exists
      then do {
        generated <- generateUniqueName vname;
        recordRenaming vname generated;
        pure generated }
      else pure vname
  pure (Assign newName expression) }
ssa (Var vname) = do {
  replacement <- lookupRenaming vname;
  let newName = fromMaybe vname replacement;
  pure (Var newName) }
```

Overloading

- Overloading `>>`, `>>=` and `pure` changes the meaning of the `do` block.
- Various effects possible, such as encoding failure, non-determinism and associated state.

Overloading

- Overloading `>>`, `>>=` and `pure` changes the meaning of the `do` block.
- Various effects possible, such as encoding failure, non-determinism and associated state.

Problem

Monads do not compose well.

Table of Contents

The Basic Transformer Framework

- Transformers [Jon95a, LHJ95] combine several, reusable effects.
- Achieved by layering single-purpose structures.
- `MonadTrans` defines `lift`, which delegates effects up the stack.
- Classes like `MonadState` are used to overload effect dispatch, removing the need to explicitly call `lift`. [Jon95b]
  - Also allows for programming against effect interfaces.

The Basic Transformer Framework

- Transformers [Jon95a, LHJ95] combine several, reusable effects.
- Achieved by layering single-purpose structures.
- `MonadTrans` defines `lift`, which delegates effects up the stack.
- Classes like `MonadState` are used to overload effect dispatch, removing the need to explicitly call `lift`. [Jon95b]
  - Also allows for programming against effect interfaces.

```
comp :: StateT s (ExceptT e IO) a

comp :: (MonadState s m, MonadError e m, MonadIO m) => m a
```

Issues

- Transformers are expensive. >>= and >> traverse the entire stack.
- They are unwieldy. Each new effect requires a new class and instances for all transformers. Each new transformer requires instances for each former effect.

Issues

- Transformers are expensive. >>= and >> traverse the entire stack.
- They are unwieldy. Each new effect requires a new class and instances for all transformers. Each new transformer requires instances for each former effect.

$\Rightarrow$ $2n$ new instances for each new transformer - effect class pair.

Defining a new effect

```haskell
newtype CountT m a = CountT
  { runCountT :: Int -> m (Int, a) }

instance Monad m => Monad (CountT m) where
  pure a = CountT (\i -> pure (i, a))
  CountT f >>= g = CountT (\i -> do {
    (i', a) <- f i
    runCountT (g a) i' })

instance MonadTrans CountT where
  lift m = CountT (\i -> fmap (i,) m)
```

The Necessary Effect Class

```haskell
class MonadCount m where
  increment :: m ()
  getCount :: m Int

instance Monad m => MonadCount (CountT m) where
  increment = CountT (\i -> pure (i + 1, ()))
  getCount = CountT (\i -> pure (i, i))
```


CHAIR FOR COMPILER CONSTRUCTION

Lifting Instances for the Effect Class

```
instance MonadCount m => MonadCount (StateT s m) where
  increment = lift increment
  getCount = lift getCount
instance MonadCount m => MonadCount (ReaderT e m) where ...
instance MonadCount m => MonadCount (WriterT w m) where ...
instance MonadCount m => MonadCount (RWST e w s m) where ...
instance MonadCount m => MonadCount (ExceptT err m) where ...
instance MonadCount m => MonadCount (ListT err m) where ...
instance MonadCount m => MonadCount (ContT err m) where ...
instance MonadCount m => MonadCount (ResourceT err m) where ...
```

Lifting Instances for Preexisting Effects

```
instance MonadState s m => MonadState s (CountT m) where
  get = lift get
  put = lift . put
instance MonadReader e m => MonadReader e (CountT m) where ...
instance MonadWriter w m => MonadWriter w (CountT m) where ...
instance MonadRWS e w s m => MonadRWS e w s m (CountT m) where ...
instance MonadError err m => MonadError err (CountT m) where ...
instance MonadCont m => MonadCont (CountT m) where ...
instance MonadIO m => MonadIO (CountT m) where ...
instance MonadResource m => MonadResource (CountT m) where ...
```

Table of Contents

CHAIR FOR COMPILER CONSTRUCTION

Basics

- Combines effects in a single monad parameterized with an effect set.
- No need for effects to implement the `Monad` typeclass.
- Generic `Member` constraint, no need for separate effect classes or a `MonadTrans` class.
- Effects are captured as data, can be interpreted freely depending on context.

Basics

- Combines effects in a single monad parameterized with an effect set.
- No need for effects to implement the `Monad` typeclass.
- Generic `Member` constraint, no need for separate effect classes or a `MonadTrans` class.
- Effects are captured as data, can be interpreted freely depending on context.

```
comp :: Eff [State s, Error e, IO] a

comp :: Members [State s, Error e, IO] effs => Eff effs a
```

Defining Effects in EE

```
data Count a where
  Increment :: Count ()
  GetCount :: Count Int

increment :: Member Count effs => Eff effs ()
increment = send Increment

getCount :: Member Count effs => Eff effs Int
getCount = send GetCount
```

Interpreting Effects

```
runCount :: Eff (Count ': effs) a -> Eff effs (Int, a)
runCount =
  handleRelayS 0
  (\i a -> pure (i, a))
  (\i eff cont ->
    case eff of
      Increment -> cont (i + 1) ()
      GetCount -> cont i i)
```

Interpreting Effects

```haskell
runCount :: Eff (Count ': effs) a -> Eff effs (Int, a)
runCount =
  handleRelayS 0
  (\i a -> pure (i, a))
  (\i eff cont ->
    case eff of
      Increment -> cont (i + 1) ()
      GetCount -> cont i i)

runCount :: Eff (Count ': effs) a -> Eff effs (Int, a)
runCount = runState 0 $ reinterpret $ \case
  Increment -> modify (+1)
  GetCount -> get
```

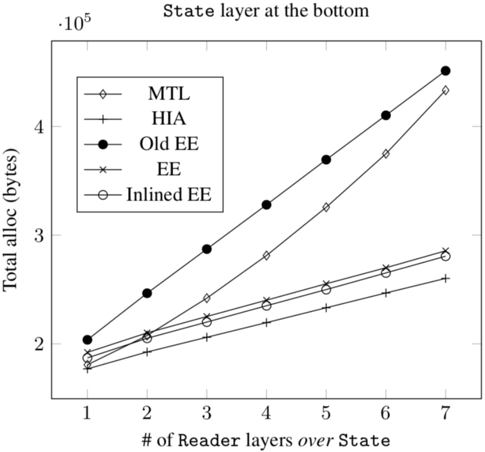
Improvements on transformers

- Mo more `instance` boilerplate, no distinction between effect type and class.
- `reinterpret` combinator to implement interpreting in terms of other effects.
- Also a `interpose` combinator to partially handle effects independent of the concrete interpreter used.

Structure and Performance

- Comprises of
  - An augmented *free monad* [Swi08] which “records” the effect chain.
  - An *open union* to combine effects in a flat structure.
  - A type aligned sequence [PK14] of continuations for efficient reflection.
- Recording effects as data allows for the modular interpreters.
- Using a flat union and a type aligned sequence has linear performance, unlike transformers, which is quadratic.

Performance Comparison



[Kl15]

CHAIR FOR COMPILER CONSTRUCTION

Table of Contents

CHAIR FOR COMPILER CONSTRUCTION

Similar Systems

Handlers in Action
: Is is similar to extensible effects, but does not group individual effects. [KLO13]

Effects
: Is a library in idris and additionally allows effects to change their types during computation. Enables the use of type level computations. [Bra13]

PureScript
: Is a language with a built-in `Eff` monad for fine grained control over FFI effects.

Table of Contents

The Free Monad

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

instance Functor f => Monad (Free f a) where
  pure = Pure
  Pure a >>= g = g a
  Impure m >>= g = Impure (fmap (>>= g) m)

iter :: Functor f => (f a -> a) -> Free f a -> a
```

- Treat any `Functor` like a `Monad`.
- Several combinators are available to quickly create interpreters.
- Makes it easy to overload the `do` notation.

Coyoneda

```
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

hoistCoyoneda :: (f ~> g) -> Coyoneda f b -> Coyoneda g b

lowerCoyoneda :: Functor f => Coyoneda f a -> f a
```

- Treat any type like a `Functor`.
- Concrete interpretation can be decided later.

Table of Contents

CHAIR FOR COMPILER CONSTRUCTION

Lessons Learned

- Many effects can be expressed in data and handled generically, which makes them compose.
- Using generic, transparent data structures rather than stacks removes boilerplate and improves performance.
- Free structures build complicated structures out of simple, less powerful ones.

References I

Edwin Brady.
Programming and Reasoning with Algebraic Effects and Dependent Types.
In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 133–144, New York, NY, USA, 2013. ACM.

Mark P. Jones.
Functional programming with overloading and higher-order polymorphism.
In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, pages 97–136, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

CHAIR FOR COMPILER CONSTRUCTION

References II

Mark P. Jones.
A system of constructor classes: overloading and implicit higher-order polymorphism.
Journal of Functional Programming, 5(1):1–35, 1995.

Oleg Kiselyov and Hiromi Ishii.
Freer monads, more extensible effects.
In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, pages 94–105, New York, NY, USA, 2015. ACM.

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

CHAIR FOR COMPILER CONSTRUCTION

References III

- Sheng Liang, Paul Hudak, and Mark Jones.
  Monad Transformers and Modular Interpreters.
  In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 333–343, New York, NY, USA, 1995. ACM.
- Atze van der Ploeg and Oleg Kiselyov.
  Reflection Without Remorse: Revealing a Hidden Sequence to Speed Up Monadic Reflection.
  In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 133–144, New York, NY, USA, 2014. ACM.
- Wouter Swierstra.
  Data types à la carte.
  Journal of Functional Programming, 18(4):423–436, 2008.