Introducing Monads

Lecture 3,

Designing and Using Combinators

John Hughes

Common ”Look and Feel”

We have already seen that **do** and return can be used with many different DSELs:

- Parsers, Wash/CGI
- IO, ST s

These are all examples of *monads*.

A monad is something that supports **do** and return!

The **do** Syntactic Sugar

The **do** syntax is just sugar for using an overloaded operator:

do e ⇒ e

do x <- e
c
⇒ e >>= (\x -> **do** c)

Sequencing.
Common in DSLs.

"bind"

Example

```
do s <- readFile f
   writeFile g s
   return s
```

⇒

```
readFile f >>= \s->
writeFile g s >>= \_->
return s
```

What are the Types?

A monad is always associated with a *parameterised type*, e.g. IO or ST s, the type of *actions*.

Call it m.

return :: a -> m a

```
readFile f >>= \s->
writeFile g s >>= \_->
return s
```

What is the type of >>=?

What are the Types?

A monad is always associated with a *parameterised type*, e.g. IO or ST s, the type of *actions*.

Call it m.

```
readFile f >>= \s->
writeFile g s >>= \_->
return s
```

```
return :: a -> m a
```

What is the type of >>=?

```
(>>=) :: m a -> … -> …
```

First arg is an action

What are the Types?

A monad is always associated with a *parameterised type*, e.g. IO or ST s, the type of *actions*.

Call it m.

return :: a -> m a

readFile f >>= \s->
writeFile g s >>= _->
return s

What is the type of >>=?

(>>=) :: m a -> (a -> ...) -> ...

Second arg receives result of first

What are the Types?

A monad is always associated with a *parameterised type*, e.g. IO or ST s, the type of *actions*.

Call it m.

```
return :: a -> m a
```

```
readFile f >>= \s->
writeFile g s >>= \_->
return s
```

What is the type of >>=?

```
(>>=) :: m a -> (a -> m b)
-> ...
```

Second arg returns an action

What are the Types?

A monad is always associated with a *parameterised type*, e.g. IO or ST s, the type of *actions*.

Call it m.

return :: a -> m a

readFile f >>= \s->
writeFile g s >>= _->
return s

What is the type of >>=?

(>>=) :: m a -> (a -> m b) -> m b

Result is an action returning result of second arg.

The Monad Class

Monad operations are overloaded (hence can be used with many libraries).

class Monad m **where**
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b

instance Monad IO …

instance Monad (ST s) …

The Monad Class

Monad operations are overloaded (hence can be used with many libraries).

class Monad m **where**
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b

instance Monad Parser …

instance Monad CGI …

Why Use Monads?

Why Use Monads?

- A *shared interface* to many libraries
  - allows a common ”look and feel” -- familiarity!
  - allows *shared code*

liftM :: Monad m => (a->b) -> m a -> m b
sequence :: Monad m => [m a] -> m [a]
do syntactic sugar!

Standard library Monad

- functionality the DSEL implementor need not implement
- ... and which users already know how to use.

Why Use Monads?

- A *shared interface* to many libraries

- A *design guideline*: no need to spend intellectual effort on the design of sequencing operations.

Why Use Monads?

- A *shared interface* to many libraries

- A *design guideline*: no need to spend intellectual effort on the design of sequencing operations.

- A *systematic implementation*: saves intellectual effort, encourages code reuse.

Systematic Monad Implementation

- Start with an *underlying* monad (e.g. IO, ST)

- Add features (e.g. state, exceptions, concurrency) one by one.

Systematic Monad Implementation

- Start with an *underlying* monad (e.g. IO, ST)
- Add features (e.g. state, exceptions, concurrency) one by one.

This can be automated!

The Identity Monad

The ”vanilla” monad, supporting no special features.

```
newtype Id a = Id a

instance Monad Id where
    return x = Id x
    Id x >>= f = f x
```

An abstract type: represented by an a, but a different type.

Note (in passing) that >>= is *lazy* -- it doesn’t need its first argument unless f does.

Monadic ”sequencing” doesn’t imply sequencing in time…

Adding Features: Monad Transformers

A *monad transformer* transforms an existing monad (without a particular feature) into a new monad which has it.

A *parameterised monad* -- represent by a parame

```
class (Monad m, Monad (t m)) =>
          MonadTransformer t m where
  ...
```

What should the method(s) be?

Adding Features: Monad Transformers

A *monad transformer* transforms an existing monad (without a particular feature) into a new monad which has it.

A *parameterised monad* -- represent by a parame

```
class (Monad m, Monad (t m)) =>
        MonadTransformer t m where
  lift :: m a -> t m a
```

Anything we can do in the old monad, we can also

Example: Adding State

Consider adding a *state* feature: actions may depend on, and change, a state.

```
class Monad m => StateMonad s m | m -> s where
    update :: (s -> s) -> m s
```

```
readState = update id
writeState s = update (\_->s)
```

```
tick :: StateMonad Integer m
   => m Integer
tick = do n <- readState
          writeState (n+1)
          return n
```

The State Monad

How can we add a state to actions?

Let actions take the state as an argument, and deliver a new state as a result.

```
newtype State s m a = State (s -> m (s,a))
```

Parameterised on both the state and the underlying monad.

The State Monad

```
newtype State s m a = State (s -> m (s,a))
```

The monad operations just pass the state along.

```
instance Monad m => Monad (State s m) where
  return x = State $ \s -> return (s,x)
  State f >>= h = State $ \s ->
    do (s’,a) <- f s
       let State g = h a
       g s’
```

The State Monad is a StateMonad

```
newtype State s m a = State (s -> m (s,a))
```

Of course, we can implement update:

```
instance Monad m => StateMonad s (State s m)
   where
      update f = State $ \s ->
         let s' = f s in
            return (s',s')
```

State is a Monad Transformer

```
newtype State s m a = State (s -> m (s,a))
```

Can we ”lift” actions in the underlying monad?

Yes -- they don’t change the state!

```
instance MonadTransformer (State s) m where
  lift a = State $ \s ->
    do x <- a
       return (s,x)
```

Last Step: Run Functions

We must be able to *observe the result* of an action -- otherwise the action is useless!

(The only exception is the IO monad, which is observable at the top level).

We define a *run function* for every monad (c.f. runST)

runId (Id a) = s
runState s (State f) =
  let (s’,a) = f s **in** a

Main> runId $ runState 0 $
sequence [tick,tick,tick]
[0,1,2]
Main>

Summary: How to Add a Feature

- Define a class representing the feature to be added (StateMonad).
- Define a *type parameterised on an underlying monad* to represent actions supporting the feature (State).
- Define sequencing of actions (**instance** Monad).
- Define how it supports the feature (**instance** StateMonad).
- Define how to lift underlying actions (**instance** MonadTransformer).
- Define how to observe the result of an action

Another Example: Failure

Add a possibility for actions to *fail*, and to handle failure.

```
class Monad m => FailureMonad m where
  failure :: m a
  handle :: m a -> m a -> m a
```

Applications in search and backtracking programs.

Example:

divide x y = **if** y==0 **then** return (x/y) **else**
failure

Actions with Failure

Allow actions to deliver a special result, meaning "I failed".

```
newtype Failure m a = Failure (m (Maybe a))
```

```
data Maybe a = Just a
             | Nothing
```

Sequencing Failure

Sequencing must *check if the first action failed,* and if so abort the second.

```
instance Monad m => Monad (Failure m)
  where
    return x = Failure (return (Just x))
    Failure m >>= h = Failure $
      do a <- m
         case a of
           Nothing -> return Nothing
           Just x -> let Failure m’ = h x in m’
```

Supporting Failure

```
instance Monad m => FailureMonad m where

  failure = Failure $ return Nothing

  Failure m `handle` Failure h = Failure $
      do x <- m
         case x of
           Nothing -> h
           Just a -> return (Just a)
```

Lifting Actions to Failure

Lifting an action just makes it succeed (return Just something).

```
instance Monad m =>
  MonadTransformer Failure m where
     lift m = Failure $ do x <- m
                           return (Just x)
```

Observing Result of a Failure

```
runFailure :: Failure m a -> m a
runFailure (Failure m) =
   do Just a <- m
      return a
```

What happens if the result is Nothing? Should runFailure handle this?

NO! Handle failures in the monad!

```
Main> runId $ runFailure $ return 2 `handle` return 3
2
```

Observing Result of a Failure

```
runFailure :: Failure m a -> m a
runFailure (Failure m) =
   do Just a <- m
      return a
```

What happens if the result is Nothing? Should runFailure handle this?

NO! Handle failures in the monad!

```
Main> runId $ runFailure $ failure `handle` return 3
3
```

Combining Features

Suppose we want to add State *and* Failure to a monad

```
example ::
  (StateMonad Integer m,
   FailureMonad m)
      => m Integer
example = do tick
             failure
      `handle`
        do tick
```

We can build a suitable monad in two ways:

- Failure (State Integer m)

Need new instances:

- FailureMonad (State s m)
- StateMonad (Failure m) s

Sample Runs

```
example ::
   (StateMonad Integer m,
    FailureMonad m)
        => m Integer
example = do tick
                  failure
         `handle`
            do tick
```

```
Main> runId $ runState 0 $ runFailure $ example
1
Main> runId $ runFailure $ runState 0 $ example
0
```

Sample Runs

```
example ::
   (StateMonad Integer m,
    FailureMonad m)
        => m Integer
example = do tick
             failure
     `handle`
        do tick
```

Failure (State Integer Id)

```
Main> runId $ runState 0 $ runFailure $ example
1
Main> runId $ runFailure $ runState 0 $ example
0
```

State Integer (Failure Id)

What are the types?

Always yields a final state

- Failure (State s m) a ≅ s -> m (s, Maybe a)

No state on failure

- State s (Failure m) a ≅ s -> m (Maybe (

Failure (State Integer Id)

Main> runId $ runState 0 $ runFailure $ example
1
Main> runId $ runFailure $ runState 0 $ example
0

State Integer (Failure Id)

Why the Different Behaviour?

Compare the instances of `handle`:

```
nstance FailureMonad m => FailureMonad (State s m) whe
tate m `handle` State m' = State $ \s ->
    m s `handle` m' s
```

Same state in handler.

```
instance Monad m => FailureMonad (Failure m) where
  Failure m `handle` Failure m' =
    Failure (do x <- m
                case x of
                      Nothing -> m'
                Just a -> return (Just a))
```

M is a state monad

So state changes

Example: Parsing

- A parser can fail -- we handle failure by trying an alternative.
- A parser *consumes input* -- has a state, the input to parse.
- A fa… e.

type Parser m tok a = State [tok] (Failure m) a

Type *synonym* -- not abstract

We get for free:

- return x -- accept no tokens & succeed
- **do** syntax -- for sequencing
- failure -- the failing parser

Parsing a Token

```
satisfy p = do s<-readState
               case s of
                [] -> failure
                x:xs -> if p x then do writeState xs
                                       return x
                                  else failure

literal tok = satisfy (==tok)
```

Completing the Library

```
p ||| q = p `handle` q
many p = some p ||| return []
some p = liftM2 (:) p (many p)

runParser p input =
    runFailure $ runState input $ p
```

This completes the basic parsing library we saw in the previous lecture.

Summary

- Monads provide *sequencing*, and offer a general and uniform interface to many different DSELs.

- Monad transformers provide a systematic way to design and implement monads.

- Together with generic monadic code, they provide a lot of functionality “for free” to the DSEL implementor.