Monads

Let’s remind ourselves of the issues around IO. Imagine we have functions such as:

- `primGetChar :: Char`
- `primPutChar :: Char -> ()`

For these functions to be meaningful they would be performing some side-effecting IO operations whenever they are evaluated. A clear violation of referential transparency!

It doesn’t take much to see the problem. Do we know what this will do:

\[
f1 = (\text{getChar}, \text{getChar})
\]

How about this:

\[
f2 = \text{let } x = \text{getChar} \text{ in } (x, x)
\]

If we draw the graphs of \(f1\) and \(f2\) we can see the problem.

But by referential transparency they should be the same value!
One solution to this problem is to require some sort of “token” which will enforce the evaluation we want. We can wrap up the (unsafe) primGetChar function in a (safe) function that takes into account the side effects:

\[ \text{getChar} :: \text{World} \rightarrow (\text{Char}, \text{World}) \]

The “World” here is a parameter which represents the state of the world from one moment to another. The actual details of how that is encoded, and of how getChar might use it are not important right now, but we assume that it is not possible to make copies of the world!

Now we can write our hazardous function differently:

\[ f3 \mathfrak{w} = ( w2, (ch1, ch2) ) \]
\[
\text{where}
\]
\[
(ch1, w1) = \text{getChar} \mathfrak{w}
\]
\[
(ch2, w2) = \text{getChar} \mathfrak{w1}
\]

Having to “thread” the various \( \mathfrak{w} \) parameters forces the evaluation to happen the way we want, as long as we are careful never to make more than one reference to any given state of the world.

### Enforcing safety

We need to ensure that this sort of thing never happens:

\[ f3 \mathfrak{w} = ( w2, (ch1, ch2) ) \]
\[
\text{where}
\]
\[
(ch1, w1) = \text{getChar} \mathfrak{w}
\]
\[
(ch2, w2) = \text{getChar} \mathfrak{w}
\]

We would also like to make it a bit easier to write functions that use this style (manually threading the “\( \mathfrak{w} \)”s around will get tedious quickly).

### Type definition for IO

Since all our functions will have a type similar to:

\[ \text{World} \rightarrow (a, \text{World}) \]

we will declare a type to capture this:

\[ \text{type IO } a = \text{World} \rightarrow (a, \text{World}) \]

(again, not worrying about what World actually is at this time)

Giving the familiar looking types:

\[ \text{getChar} :: \text{IO } \text{Char} \]
\[ \text{putChar} :: \text{IO } () \]
Joining two computations

Given two computations we can write a combinator which will sequence them, safely threading the World value through in the correct manner:

\[ (\gg) :: \text{IO } a \to \text{IO } b \to \text{IO } b \]

which we can use to sequence two actions:

\[ f4 = (\text{putChar } 'a') (\gg) (\text{putChar } 'b') \]

This operator will do the first action, throw away the result, and then do the second action.

\[ (\ggg) l r = \lambda w \rightarrow \text{let } (w1,_) = l w \text{ in } r w1 \]

Further operations

We could also join two computations together passing the result of the first on to the second:

\[ (\ggg) :: \text{IO } a \to (a \to \text{IO } b) \to \text{IO } b \]

This operation is called “bind”. In fact, if we have “bind” we can write \(\gg\) easily:

\[ (\gg) l r = l (\ggg) (\lambda _ \rightarrow r) \]

Further operations

Finally, we will need a computation which does nothing and produces a given result:

\[ \text{return} :: a \rightarrow \text{IO } a \]

This represents a sort of “unit” operation. We can see the need for it when we try to combine several values:

\[ f5 = \text{getChar} (\ggg) (\lambda \ ch1 \rightarrow \text{getChar} (\ggg) (\lambda \ ch2 \rightarrow \text{return } (\text{ch1},\text{ch2}) )) \]

The IO Monad

What we have seen so far is a possible implementation for a monad that performs IO. A monad is an abstraction which represents a computation. The computations have results (reflected in the type). The monad provides at least the basic operations:

- \(\text{return}\) which produces a result
- \(\ggg\) which binds together two computations.

Generally a monad will also provide a collection of primitive operations (like \text{getChar}) to make it useful.
Do notation

Programs written in a monadic style will typically contain long chains of >> and >>= operations. Haskell provides some syntactic sugar, called the “do-notation” that allows us to write the previous program as follows:

```haskell
f5 = do
    ch1 <- getChar
    ch2 <- getChar
    return (ch1,ch2)
```

“do” is just sugar!

There is a mechanical translation from the do-notation form to the combinator form, which we can summarize:

```haskell
do x
    y =
    x >>= \a -> do y

do a <- x
    y =
    x >>= \a -> do y

do x = x
```

The Monad laws

In order to retain the semantics that we want any implementation of a monad is required to follow these rules:

```haskell
(return v >>= f) == f v

f >>= return == f

(x >>= f) >>= g == x >>= (\a -> f a >>= g)
```

These laws are not checked by the compiler. But they are assumed by it!

Any monad?

Any implementation of a monad?

Yes, monads represent something fundamental in computation, the idea of connecting two computations and threading results. There are more monads than IO. For example, another monad which we have seen already is Maybe!

Imagine a function:

```haskell
f tree = case (findEntry "foo" tree) of
    Nothing -> Nothing
    Just x -> case (findEntry "bar" tree) of
        Nothing -> Nothing
        Just y -> Just (x,y)
```

We can clean this up because “Maybe” is a monad!
Monads in Haskell

Monads in Haskell are represented by a type class:

```haskell
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    return :: a -> m a
    fail :: String -> m a
```

Since `>>` can be defined in terms of `>>=` we usually only need to provide instances for `return` and `>>=`.

(sidebar: The fourth member of the class is an error handling operation which takes an error message and causes the chain of functions to fail, usually by using `error` to halt the program).

Quick terminology overview:

- **XYZ Monad**: A data type “XYZ” which is an instance of Monad
- **Monadic**: Pertaining to monads (of a programming style or a value, usually)

IO uses a monad instance to manage the “state” value that we represented with the word `World` in the previous lecture. Whenever we have a state parameter like this we could make use of the monadic style to manage it safely.

This approach is so common that it is worth the time to abstract the overall pattern of managing state.
State Monad

First we need a data type to be the instance of Monad. The standard Haskell state data type used for this is:

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

newtype State s a = State {
    runState :: s -> (a, s)
}
```

This is a Haskell record type.

Records in Haskell

A record is really just a standard data type with one constructor:

```haskell
data Pair a b = Pair a b
```

To use a type like this you might provide some “accessor” functions:

```haskell
first :: (Pair a b) -> a
first (Pair a _) = a

second :: (Pair a b) -> b
second (Pair _ b) = b
```

Records in Haskell

Defining the data type as a record type is just a syntactic convenience for creating those accessors:

```haskell
newtype Pair a b = Pair {
    first :: a,
    second :: b
}
```

We can either create values using the usual constructor, or using the named fields:

```haskell
f = Pair 1 'a'

g = Pair { second = 'a', first = 1 }
```

(Notice the order doesn’t matter when we use the fields).

Records in Haskell

We can look them up using the field names:

```haskell
h :: (Pair Int b) -> Int
h p = (first p) + 1
```

There’s also a convenient syntax for creating a new value based on an old one:

```haskell
newpair p = p { second = 'b' }```
We need to provide an instance of `return` to inject a value into the `State` monad:

```hs
returnState :: a -> State s a
returnState a = State (\s -> (a, s))
```
Alternatively:

```hs
returnState a s = State (a, s)
```

Joining two `State` actions together is simple enough. We need to use `runState` to extract the actual functions.

```hs
bindState :: State s a -> (a -> State s b) -> State s b
bindState m k = State (\s -> let (a, s') = runState m s
                        in runState (k a) s')
```
This is enough: the other instances in the `Monad` class will be derived automatically.

Almost all that remains is to make `State` an instance of the `Monad` class:

```hs
instance Monad (State s) where
  return = returnState
  (>>=) = bindState
```

Set and Get the state

There is actually one more job: `return` will inject a value into the `State` monad, but it won’t actually touch the state parameter itself! We need to add a couple of transitions which will set and get the state parameter:

```hs
getSt :: State s s
getSt = State $ \s -> (s, s)

putSt :: s -> State s ()
putSt s = State $ \_ -> ((), s)
```
Perhaps unsurprisingly, there is another class that we can instantiate once we have these:

```hs
instance MonadState (State s) s where
  get = getSt
  put = putSt
```
Example - threading Random generators

```
data Val = Val Int Bool Char Int
    deriving Show

makeRandomValue :: StdGen -> (Val, StdGen)
makeRandomValue = let (n,g1) = randomR (1,100) g
                (b,g2) = random g1
                (c,g3) = randomR ('a', 'z') g2
                (m,g4) = getOne (-n, n) g3
                in (Val n b c m, g4)
```

Ugly, no?
This is what is meant when we talk about having to thread values through a computation.

Using State

```
A classic use of a State monad is to thread a random number generator:

getAny :: (Random a) => State StdGen a
getAny = do
    g <- get
    (x,g') <- return $ random g
    put g'
    return x
```

```
getOne :: (Random a) => (a,a) -> State StdGen a
getOne bounds = do g <- get
    (x,g') <- return $ randomR bounds g
    put g'
    return x
```

Using State

```
data Val = Val Int Bool Char Int
    deriving Show

makeRandomValueST :: StdGen -> (Val, StdGen)
makeRandomValueST = runState (do n <- getOne (1,100)
                                 b <- getAny
                                 c <- getOne ('a', 'z')
                                 m <- getOne (-n, n)
                                 return (Val n b c m))
```
Using it

```haskell
main = do
    g <- getStdGen
    print $ fst $ makeRandomValue g
    print $ fst $ makeRandomValueST g
```

List as a monad

The Maybe type can be thought of as a monad. Anything else? Lists!

```haskell
instance Monad [] where
    return a = [a]
    lst >>= f = concat (map f lst)
    fail _ = []
```

What could it mean?

What does it mean to say that lists form a monad? It represents the type of computations that may return 0, 1, or more results. More specifically, it combines actions by applying the operations to all possible values.

How does it work?

Take this code:

```haskell
cart xs ys = do
    x <- xs
    y <- ys
    return (x,y)
```

What will an application like `cart [1,2,3] [97,98,99]` do?

```haskell
Prelude> cart [1,2,3] [97,98,99]
[(1,97),(1,98),(1,99),(2,97),(2,98),(2,99),
  (3,97),(3,98),(3,99)]
```
Some practical Monad notes

- Functors and Monads are closely related. When a programmer defines a Monad instance they usually define a Functor instance as well. Therefore `fmap` usually works on Monadic values.

- Writing code which unwraps a Monadic value and then applies it right away is fairly common (and tiresome):
  ```haskell
  m >>= \l -> return (sum l)
  ```
  In these cases we can use `liftM`
  ```haskell
  liftM :: (Monad m) => (a -> b) -> m a -> m b
  liftM f m = m >>= \i -> return (f i)
  ```

- In fact, many more control structures are abstracted this way. There is a whole library: `Control.Monad` which contains such things (check out `sequence_`, for example, and there are nearly 30 others).