8 Monads (IFPH §10)

8.1 Motivation

Monads form *an abstract datatype of computations*.

Computations in general may have *effects*: I/O, exceptions, mutable state, etc.

Monads are a mechanism for cleanly incorporating such impure features in a pure setting.

8.2 An evaluator

Here's a simple datatype of terms:

```
> data Term = Con Integer | Div Term Term
>   deriving Show

> good, bad :: Term
> good = Div (Con 7) (Div (Con 4) (Con 2))
> bad  = Div (Con 7) (Div (Con 2) (Con 4))
```

. . . and an evaluation function:

```
> eval :: Term -> Integer
> eval (Con u)    = u
> eval (Div x y) = eval x `div` eval y
```

8.2.1 Exceptions

Evaluation may fail, because of division by zero.

Let's handle the exceptional behaviour:

```
> data Exc a = Raise Exception | Result a
> type Exception = String

> evalE :: Term -> Exc Integer
> evalE (Con u) = Result u
> evalE (Div x y) =
>   case evalE x of
>     Raise e -> Raise e
>     Result u -> case evalE y of
>       Raise e -> Raise e
>       Result v ->
>         if v==0 then Raise "Division by zero"
>                 else Result (u `div` v)
```

8.2.2 Counting

We could instrument the evaluator to count evaluation steps:

```
> newtype Counter a = C (State -> (a, State))
> type State = Int
> run :: Counter a -> State -> (a,State)
> run (C f) = f

> evalC :: Term -> Counter Integer
> evalC (Con u)   = C (\ n -> (u, n+1))
> evalC (Div x y) = C (\ n ->
>                     let (u, n')  = run (evalC x) (n+1)
>                         (v, n'') = run (evalC y) n'
>                     in (u `div` v, n''))
```

8.2.3 Tracing

…or to trace the evaluation steps:

```
> newtype Trace a = T (Output, a)
> type Output = String

> evalT :: Term -> Trace Integer
> evalT (Con u)   = T (line (Con u) u, u)
> evalT (Div x y) = let
>                      T (s,u)  = evalT x
>                      T (s',v) = evalT y
>                      p = u `div` v
>                    in T (s ++ s' ++ line (Div x y) p, p)

> line :: Term -> Integer -> Output
> line t n = "  " ++ show t ++ " yields " ++ show n ++ "\n"
```

8.2.4 Ugly!

None of these extensions is difficult.

But each is rather awkward, and obscures the previously clear structure.

How can we simplify the presentation? What do they have in common?

8.3 The monad interface

In all cases, there are ways of embedding 'pure' computations, and of sequencing computations.

For type constructor `m`,

```
> lift :: (a -> b) -> (a -> m b)
> comp :: (b -> m c) -> (a -> m b) -> (a -> m c)
```

There are also effect-specific operations.

For exceptions,

```
> liftE :: (a -> b) -> (a -> Exc b)
> liftE f = Result . f

> compE :: (b -> Exc c) -> (a -> Exc b) -> (a -> Exc c)
> compE f g a = case g a of
>   Raise e -> Raise e
>   Result b -> f b
```

There is also an effect-specific operation to throw an exception.

For counters,

```
> liftC :: (a -> b) -> (a -> Counter b)
> liftC f a = C (\ n -> (f a, n))

> compC :: (b -> Counter c) -> (a -> Counter b) -> (a -> Counter c)
> compC f g a = C (\ n -> let (b,n') = run (g a) n
>                          in run (f b) n')
```

There is also an effect-specific operation to increment the counter.

For tracing,

```
> liftT :: (a -> b) -> (a -> Trace b)
> liftT f a = T ("", f a)

> compT :: (b -> Trace c) -> (a -> Trace b) -> (a -> Trace c)
> compT f g a = let T (s,b) = g a
>                   T (s',c) = f b
>               in T (s++s',c)
```

There is also an effect-specific operation to log some output.

8.4 The monad interface in Haskell

Haskell actually chooses a different model of lifting:

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

It's equivalent:

```
  lift f = return . f
  return = lift id
```

Similarly, for composition:

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

with equivalences:

```
  comp f g a = g a >>= f
  ma >>= f = comp f id ma
```

8.4.1 The monad type class

These are the methods of a type class:

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

Technically, there are some laws that should be satisfied.
These are clearest specified in terms of `comp`:

```
f `comp` return = f
return `comp` f = f
f `comp` (g `comp` h) = (f `comp` g) `comp` h
```

(so monads are intimately related to monoids).

8.5 Original evaluator, monadically

```
> evalM :: Monad m => Term -> m Integer
> evalM (Con u)   = return u
> evalM (Div x y) = evalM x >>= \ u ->
>                       evalM y >>= \ v ->
>                         return (u `div` v)
```

Still pure, but written in the monadic style; much easier to extend.

8.5.1 The exception instance

Exceptions instantiate the class:

```
> instance Monad Exc where
>   return = liftE id
>   ma >>= f = compE f id ma
```

That is,

```
> instance Monad Exc where
>   return a = Result a
>   Raise e >>= f = Raise e
>   Result a >>= f = f a
```

The effect-specific behaviour is to throw an exception:

```
> throw :: Exception -> Exc e
> throw e = Raise e
```

8.5.2 Exceptional evaluator, monadically

```
> evalE :: Term -> Exc Integer
> evalE (Con u)    = return u
> evalE (Div x y) = evalE x >>= \ u ->
>                        evalE y >>= \ v ->
>                          if v==0 then throw "Division by zero"
>                                  else return (u ‘div‘ v)
```

8.5.3 The counter instance

Counters instantiate the class:

```
> instance Monad Counter where
>   return = liftC id
>   ma >>= f = compC f id ma
```

That is,

```
> instance Monad Counter where
>   return a = C (\ n -> (a,n))
>   ma >>= f = C (\ n -> let (a,n') = run ma n in run (f a) n')
```

The effect-specific behaviour is to increment the count:

```
> tick :: Counter ()
> tick = C (\ n -> ((),n+1))
```

8.5.4 Counting evaluator, monadically

```
> evalC :: Term -> Counter Integer
> evalC (Con u)    = tick >>= \ () ->
>                      return u
> evalC (Div x y) = tick >>= \ () ->
>                      evalC x >>= \ u ->
>                        evalC y >>= \ v ->
>                          return (u `div` v)
```

8.5.5 The tracing instance

Exceptions instantiate the class:

```
> instance Monad Trace where
>   return = liftT id
>   ma >>= f = compT f id ma
```

That is,

```
> instance Monad Trace where
>   return a = T ("", a)
>   T (s,a) >>= f = let T (s',b) = f a in T (s++s', b)
```

The effect-specific behaviour is to log some output:

```
> trace :: String -> Trace ()
> trace s = T (s, ())
```

8.5.6 Tracing evaluator, monadically

```
> evalT :: Term -> Trace Integer
> evalT (Con u)   = trace (line (Con u) u) >>= \ () ->
>                      return u
> evalT (Div x y) = evalT x >>= \ u ->
>                      evalT y >>= \ v ->
>                        let p = u `div` v in
>                          trace (line (Div x y) p) >>= \ () ->
>                            return p
```

8.6 Do notation

Special syntactic sugar for monadic expressions.

Inspired by (in fact, a generalization of) list comprehensions.

```
do { m }             = m
do { a <- m ; ms } = m >>= \ a -> do { ms }
do {        m ; ms } = m >>= \ _ -> do { ms }
```

where `a` can appear free in `ms`.

8.6.1 Exceptional evaluator, using do notation

```
> evalE :: Term -> Exc Integer
> evalE (Con u)   = do
>                      return u
> evalE (Div x y) = do
>                      u <- evalE x
>                      v <- evalE y
>                      if v==0 then throw "Division by zero"
>                              else return (u ‘div‘ v)
```

8.6.2 Counting evaluator, using do notation

```
> evalC :: Term -> Counter Integer
> evalC (Con u)   = do
>                     tick
>                     return u
> evalC (Div x y) = do
>                     tick
>                     u <- evalC x
>                     v <- evalC y
>                     return (u `div` v)
```

8.6.3 Tracing evaluator, using do notation

```
> evalT :: Term -> Trace Integer
> evalT (Con u)   = do
>                     trace (line (Con u) u)
>                     return u
> evalT (Div x y) = do
>                     u <- evalT x
>                     v <- evalT y
>                     let p = u `div` v
>                     trace (line (Div x y) p)
>                     return p
```

8.7 The IO monad

There's no magic to monads in general: all the monads above are just plain (perhaps higher-order) data, implementing a particular interface.

But there is one magic monad: the `IO` monad. Its implementation is abstract, hard-wired in the language implementation.

```
> data IO a = ...
> instance Monad IO where ...
```

8.7.1 IO-specific operations

```
> putChar :: Char -> IO ()
> getChar :: IO Char

> type FilePath = String
> writeFile :: FilePath -> String -> IO ()
> readFile :: FilePath -> IO String

> data StdGen = ...         -- standard random generator
> class Random where ...    -- randomly generatable
> randomR :: Random a => (a,a) -> StdGen -> (a,StdGen)
> getStdRandom :: (StdGen -> (a,StdGen)) -> IO a
```

among many others.

8.7.2 Character I/O

```
> putStr, putStrLn :: String -> IO ()
> putStr ""      = do { return () }
> putStr (c:s) = do { putChar c ; putStr s }
> putStrLn s     = do { putStr s ; putChar '\n' }

> getLine :: IO String
> getLine = do
>   c <- getChar
>   if c=='\n' then return "" else do
>     s <- getLine
>     return (c:s)
```

8.7.3 File I/O

```
> processFile :: FilePath -> FilePath -> (String->String) -> IO ()
> processFile inFile outFile f = do
>    s <- readFile inFile
>    let s' = f s
>    writeFile outFile s'
```

8.7.4 Random numbers

```
> import Random

> rollDice :: IO Int
> rollDice = getStdRandom (randomR (1,6))

> rollThrice :: IO Int
> rollThrice = do
>   x <- rollDice
>   y <- rollDice
>   z <- rollDice
>   return (x+y+z)
```