Introducing The Monads

Aditya Siram

15 September 2010

Outline

Goals

- A monad primer (explain the common interface to all monads)
- A tour of the important monads and how to use them to get stuff done.
- Uses a cookbook approach so no theory or implementation details.
- Show that Haskell can be eminently practical and readable.

Brief IO Example

- This small function writes a text file, reads it back, uppercases its contents and prints them to stdout.

```
import Data.Char
main :: IO ()
main = do
    writeFile "test.txt" "a,b,c,d,e"
    x <- readFile "test.txt"
    y <- return (map toUpper x)
    print y
```

- Programmers with no Haskell knowledge can guess what it does.
- We'll explore this in detail later ...
- First the basics.

Simple Haskell datatypes

A data type in Haskell is essentially one or more tags:

```
data Fruit = Apple | Orange | Grapefruit ...
```

- :type Apple => Fruit
- :type Orange => Fruit

The tags are called "type constructors" because they can take values :

```
data Fruit a = Apple a | Orange a | Grapefruit ...
```

- :type (Apple "hello") => Fruit String
- :type (Orange 1) => Fruit Int

The Maybe Type

The Maybe type is very simple type that models success and failure

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

- Just <some-value> = success,
- Nothing = failure (approx. 'null' in other languages)
- :type (Just 1) => Maybe Int
- :type (Just "hello") => Maybe String
- :type Nothing => Maybe ?? (it is determined in context)

I bring it up because Maybe is also a monad.

Monad implementation basics

- All monads provide the following functions:

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

- Braces around the function name (func) make it infix
  - (»=) x y == x »= y
- And any type that implements these functions is a monad. From one perspective that's it!
- Notice that the type signatures are very simple
- This gives a flexible interface for the monad designer.
- And leads to a consistent and elegant programming experience for the monad user.
- Now we look at each of these functions in detail.

The 'return' function

- The return function takes a value and puts it in the context of the current monad. It is now a "monadic value".
- Another way to think about it: it packs a value into the monad container.
- The way it does this depends entirely on the monad.
- In the context of Maybe, the signature for 'return' is :

  return :: a -> Maybe a

```
- return 1 :: Maybe Int
=> Just 1
- return Nothing :: Maybe Int
=> Nothing
```

The bind (»=) function

The (»=) function

1. Removes a value from the context of the current monad (unpacks a value from the monad)
2. Gives it to a function which does something to it and puts it back into the context of the current monadic. (apply and repack)

- In the context of Maybe, the type signature for (»=) is:

  (»=) :: a -> (a -> Maybe b) -> Maybe b

  ```
  For example,
    (return 1 >>= \x ->
     return (x+1)) :: Maybe Int
     => Just 2
  ```

Chaining with the bind (»=) function (1/2)

- The (»=) function can then be chained together:

```
(return 1 >>= \x ->
 return (x+1) >>= \y ->
 return (y+1)) :: Maybe Int
=> Just 3
```

- The mantra of (»=) is "Unpack, hand over, unpack, hand over . . . "
- This applies to **all** monads.

Chaining with the bind (»=) function (2/2)

- But what about when one of the functions in the chain is Nothing?

```
(return 1 >>= \x ->
 Nothing >>= \y ->
 return (x+1)) :: Maybe Int
=> Nothing
```

- If **any** of the functions is Nothing (fails), the entire chain is Nothing.
- It would have been impossible to guess that from the type signatures alone.
- A lot is left up to the implementation and documentation.
- I guess this is the trade-off for simplicity, consistency and flexibility!

Chaining with (») function

- The (») function is very similar to (»=) but the mantra is slightly different:
  "Unpack, don't hand over, unpack, don't hand over ..."

```
(return 1 >>= \x ->
 return (x+1) >>
 return 1024) :: Maybe Int
=> Just 1024
```

- All computation before 'return 1024' is ignored.
- This is useful when doing something that has side-effects.

```
(print "hello world" >>
print "goodbye world") :: IO ()
=> "hello world"
   "goodbye world"
```

- 'print' just writes to stdout so there is nothing to hand over.

Do-notation (1/2)

- do-notation is a nicer way of writing monadic code without (»=) and (»)
- Here's an example using Maybe :

```
func :: Maybe Int
func = do
  x <- return (1 :: Int)
  y <- return (2 :: Int)
  return (x+y)
```

- is the same as . . .

```
func' :: Maybe Int
func' = return (1 :: Int) >>= \x ->
        return (2 :: Int) >>= \y ->
        return (x+y)
```

Do-notation (2/2)

- And here's an example in the IO monad :

```
func :: IO ()
func =  do
  print "What is your name?"
  name <- getLine
  print ("Hello " ++ name)
```

- is the same as . . .

```
func' = print "What is your name?" >>
        getLine >>= \name ->
        print ("Hello " ++ name)
```

- And speaking of which . . .

IO (1/2)

- The IO monad is Haskell's eyes, ears and mouth.
- Haskell can only communicate with the outside world through the IO Monad.
- Here again is the example where we write a text file, read it back, uppercase its contents and write them to stdout.

```
import Data.Char
main :: IO ()
main = do
    writeFile "test.txt" "a,b,c,d,e" :: IO ()
    x <- readFile "test.txt" :: IO String
    y <- (return :: String -> IO String)
             (map Data.Char.toUpper x :: String)
    print y :: IO ()
-- => "A,B,C,D,E"
```

- The type signatures within "main" are usually omitted. But very useful for learning and debugging.

IO (2/2)

- Querying a Sqlite database

```
import Database.HDBC
import Database.HDBC.Sqlite3
import Control.Exception
testdb emp_id = do
  res <- dbQuery "select * from employees where id=?"
         [toSql emp_id]
  print res
  where
    dbConnect :: IO Connection
    dbConnect = connectSqlite3 "test.sqlite"
    dbQuery :: String -> [SqlValue] -> IO [[SqlValue]]
    dbQuery sql values =
          bracket dbConnect disconnect
            (\conn -> quickQuery' conn sql values)
```

Basic State Manipulation Monads

- In impure languages threading state is the norm

```
func (state) {
  var i = 0;
  i = func1(state);
  i = func2(state);
  return (i,state);
}
```

- 'state' seen by 'func1' may be different from 'state' seen by 'func2'
- In Haskell, 'state' and 'i' are not mutable so output is the original 'i' and 'state' - not what you wanted!
- The three basic state manipulation monads Reader,Writer and State offer (the illusion of) mutable state in Haskell.

Reader (1/2)

- A Reader type consists of some state and the result.
- The state in the Reader type is read-only. It cannot be changed within the monad.
- 'ask' extracts the state from the monad for inspection.
- 'runReader' takes a Reader monad and a state and outputs the final result.
- Now a code example ...

Reader (2/2)

- A simple authentication example:

```
import "mtl" Control.Monad.Reader
type Env = [(String,String)]
simpleAuth :: String -> String -> Reader Env Bool
simpleAuth user pass = do
  env :: Env <- ask :: Reader Env Env
  case (lookup user env)  of
    Nothing -> return False
    Just p -> return (p == pass)
main = print $ runReader auth env
    where
        auth = simpleAuth "deech" "mypass"
        env = [("deech","mypass"),("admin","adminPass")]
-- => True
```

Writer (1/2)

- A Writer type also consists some state and the result.
- State is append-only, cannot be over-ridden.
- New state accumulated during the computation is tacked onto the end of the old.
- State cannot be inspected either. No 'ask' function for the Writer monad.
- 'runWriter' returns a tuple consisting of the new state and the result of the computation.

Writer (2/2)

- State is appended using 'tell'
- A simple example showing how a Writer can be used to maintain a log.

```
  import "mtl" Control.Monad.Writer
  test :: Writer [String] Int
  test = do
    x :: Int  <- return 1 :: Writer [String] Int
    tell $ ["x is " ++ (show x)]
    y :: Int <- return 2 :: Writer [String] Int
    tell $ ["y is " ++ (show y)]
    z :: Int <- return (x + y) :: Writer [String] Int
    tell $ ["z is " ++ (show z)]
    return (z :: Int) :: Writer [String] Int
  main = print $ runWriter test
-- => (3,["x is 1","y is 2","z is 3"])
```

State (1/3)

- Like the Reader and Writer, the State type consists of a state and a final result
- Unlike the Reader and Writer, state can be inspected, modified and replaced mid-stream
- 'get' retrieves the current state
- 'put' replaces the current state with its argument
- 'modify' transforms the current state using the given function.
- Like Writer, 'runState' takes the State computation and returns a tuple consisting of the new state and the result of the computation.
- Now for some code ...

State (2/3)

- Unlike Reader and Writer, 'runState' **has** to be initialized with a starting state, even if you plan on replacing it.

```
  import "mtl" Control.Monad.State
  test :: State Int String
  test = do
      put 2
      modify (\s -> s + 100)
      x <- get
      return "hello"
  main = print $ runState test 1
-- => ("hello",102)
```

State (3/3)

- Simulating an imperative for-loop with mutable counters
- The code finds the index of the minimum element in a list.

```
    import "mtl" Control.Monad.State
-- mapM_ f [1,2,3,4] == f 1 >> f 2 >> f 3 >> f 4
-- zip [0,1,2] ['a','b'] == [(0,'a'),(1,'b')]
    test :: (Ord a) => [a] -> ((),Int)
    test [] = error "Empty List"
    test xs = runState (mapM_ min (zip [0..] xs)) 0
        where
            min (i,n) = do
                curr_min_i <- get
                if (n < (xs !! curr_min_i)) then put i
                  else return ()
    main = print $ test [4,2,5,6,-1,-9]
-- => ((),5)
```

- This is **not** idiomatic Haskell!

Basic State Manipulation Wrap-up (1/2)

- Reader, Writer and State all thread state through the computation
- But they give you different levels of control.
- Reader is read-only state
- Writer is append-only state
- State gives you complete control.

Basic State Manipulation Wrap-up (2/2)

- So technically you only need State!
- But using the other types gives nice guarantees about what a function cannot do.
- For example, given this function :

```
func :: Reader Env Bool
func ...
```

- You **know** that Env isn't changed after running it.

Combining Monads

- So far we've seen single-use monads (IO, or Maybe or State but not all three)
- Monads are combined using "monad transformers"
- They are a lot simpler to use than they sound.
- There are "monad transformer" versions for all the monads we've seen so far - just append a 'T' to the type name.
  - Reader => ReaderT,
  - Writer => WriterT,
  - State => StateT

ReaderT and IO (1/3)

- A ReaderT type takes a state and an embedded monad.
- 'runReaderT' extracts the embedded monad from the ReaderT monad. Analogous to 'runReader'.
- Consider the function :

```
func :: ReaderT [(String,String)] IO ()
```

- 'func' can do two things :
  1. look up an immutable string->string map using 'ask'
  2. actions of type IO using 'liftIO'.
- 'runReaderT func' => something of type IO ()

ReaderT and IO (2/3)

- For example here's an interactive version of the Reader authentication example we saw earlier :

```
import "mtl" Control.Monad.Reader
test :: ReaderT [(String,String)] IO ()
test = do
  user <- liftIO $ print "Enter username" >> getLine
  pass <- liftIO $ print "Enter password" >> getLine
  env <- ask
  case (lookup user env) of
    Nothing -> liftIO $ print "That username isn't found!"
    Just p -> if (p == pass) then liftIO $ print "Welcome!"
                 else liftIO $ print "Incorrect Password!"
main = runReaderT test [("deech","mypass"),
                        ("admin","adminPass")]
```

ReaderT and IO (3/3)

- The result of running the above code is :

```
"Enter username"
deech
"Enter password"
mypass
"Welcome!"
```

WriterT and IO (1/2)

- The idea is exactly same as ReaderT.
- 'runWriterT' extracts a result from the WriterT monad. Analogous to 'runWriter'.
- Given the function

  ```
  func :: WriterT [String] IO ()
  ```

- func can do two things :
  1. append a string to a list of strings using 'tell'
  2. actions of type IO using 'liftIO'
- 'runWriterT func' => something of type IO ()

WriterT and IO (2/2)

- For example, here's a version of the logging function we say earlier.

```
import "mtl" Control.Monad.Writer
test :: String -> WriterT [String] IO Int
test file = do
  x <- return 1
  tellAndLog $ "x is " ++ (show x)
  y <- return 2
  tellAndLog $ "y is " ++ (show y)
  z <- return (x + y)
  tellAndLog $ "z is " ++ (show z)
  return z
  where
    tellAndLog s = tell [s] >>
                   (liftIO $ appendFile file $ s ++ "\n")
main =  runWriterT (test "test.txt")
-- => (3,["x is 1","y is 2","z is 3"])
```

ReaderT, WriterT and IO (1/3)

- The embedded monad can itself be a monad transformer!
- For instance the function:

```
func :: ReaderT [(String,String)] (WriterT [String] IO) ()
```

- It can do three things :
  1. look up an immutable string->string map using ‘ask’
  2. append a string to a list of strings using ‘tell’
  3. actions of type IO using ‘liftIO’
- But how do you extract a value?
  - ‘runReaderT func’ => (WriterT [String] IO ())
  - Therefore ‘runWriterT (runReaderT func)’ => IO ()
- Next a code example . . .

ReaderT, WriterT and IO (2/3)

```
import Control.Monad.Reader
import Control.Monad.Writer
test :: ReaderT [(String,String)] (WriterT [String] IO) ()
test  = do
  user <- liftIO $ print "Username?" >> getLine
  tellAndLog $ "Username entered : " ++ (show user)
  pass <- liftIO $ print "Password?" >> getLine
  tellAndLog $ "Password entered : " ++ (show pass)
  env <- ask
  case (lookup user env) of
      Nothing -> tellAndLog "bad username"
      Just p -> if (p == pass) then tellLogPrint "accepted"
                      else tellLogPrint "denied!"
  where
   tellAndLog s = tell [s] >>(liftIO $ appendFile "test.txt" s)
   tellLogPrint s = tellAndLog s >> (liftIO $ print s)
```

ReaderT, WriterT and IO (3/3)

- And a function to run the computation :

```
main = runWriterT $
           runReaderT test [("deech","mypass"),
                            ("admin","adminPass")]
```

- Results in the following output :

```
"Username?"
deech
"Password?"
mypass
"authenticated"
((),["Username entered : \"deech\"",
     "Password entered : \"mypass\"",
     "authenticated"])
```

Combining Monads wrap up

- Most commonly used monads have monad transformer versions
- Monads transformers can stacked as deep as you want to add functionality.
- For example :

```
func :: StateT (Writer [String] Int)
          (ReaderT [(String,String)] IO) ()
```

  1. Keeps state of type (Writer [String] Int)
     - You now have an inspectable and modifiable running log
  2. Allows the inspection of an environment using 'ask'
  3. Does IO actions
- To extract it
  1. runStateT func <some Writer computation>
  2. runReaderT

  => runReaderT $ runStateT func <some Writer computation>
- The trickiest part is the type signature and knowing how to run the computation.

STM (1/5)

- Software Transactional Memory is basically an in-memory shared database
- Each read and write to the database is thread-safe
- Every thread has a consistent view of the data.
- No IO can be done in the STM monad.
- Any variable of type 'TVar' is an STM variable which can only be read and written in the STM monad.
- Now an example ...

STM (2/5)

- This is an example of a function that waits until a string is non-empty.
- 'retry' is an STM monad function that 'blocks' until a condition has been met.

```
outputMessage :: MVar () -> String -> IO ()
outputMessage lock str = withMVar lock (\_ -> print str)
readShared :: MVar () -> TVar String -> STM String
readShared lock s = do
  s' <- readTVar s
  if (s' == "") then
      do
        unsafeIOToSTM $
           outputMessage lock $ " String is empty " ++
                                 ", waiting for change "
        retry
    else do
      return $ "Got it! " ++ s'
```

STM (3/5)

- Now an example of writing to a transactional string.

```
writeShared :: TVar String -> STM ()
writeShared s = do
  s' <- readTVar s
  writeTVar s (s' ++ "hello world")
  return ()
```

STM (4/5)

- And tying it all together a function that
  1. spawns a thread that creates an empty transactional string
  2. waits 3 seconds and spawns another thread that writes to the transactional string

```
demoRetry :: IO ()
demoRetry = do
  writeLock <- newMVar ()
  a <- atomically $ newTVar ""
  forkIO $ (atomically $ readShared writeLock a) >>= print
  threadDelay 3000000
  forkIO $ atomically $ writeShared a
  return ()
```

STM (5/5)

- The output of running 'demoRetry' is:

```
" String is empty , waiting for change "
<3 seconds passes>
"Got it! hello world"
```

Parsec (1/2)

- Parsec is used to parse complex grammers
- It is readable
- It not only parses a string but can separate it into its tokens in a single step.
- It returns an Either type:

```
data Either a b = Left a | Right b
```

- Successful parse => "Right <parsed-value>"
- Failed parse => "Left <some-error>"
- Some simple examples follow . . .

Parsec (2/2)

- Below is an example that parses (and lexes):
  1. a simple email address, no support for comments etc.
  2. a comma OR dot separated string

```
import Text.ParserCombinators.Parsec
validChar = many $ noneOf "!@#$%^&*()[]{}"
commaOrDot = many alphaNum `sepBy` (char ',' <|> char '.')
simpleEmail = do
    local <- manyTill alphaNum (try (char '@'))
    s <- validChar `sepBy` (char '.')
    eof
    return (local,s)
main = do
   print $ parse simpleEmail "" "alpha123@hotmail.com.org"
   print $ parse commaOrDot "" "a,b,..c,,d"
-- => Right ("alpha123",["hotmail","com"])
-- => Right ["a","b","","","c","","d"]
```

Wrap up

- Monads are:
  1. easy to use once you develop an intuition for 'return' and ($\gg=$)
  2. and a vital part of writing and **reading** Haskell code
  3. used for a wide variety of functionality because of the simple interface
- The monads we talked about will get you most of the way.
- User docs (or source code) will do the rest.

The End

Thanks for listening!