Monads and IO in Haskell

Amitabha Sanyal

Department of Computer Science and Engineering
IIT Bombay.
Powai, Mumbai - 400076

as@cse.iitb.ac.in

October-29 2013

Monads

Real world programming is more than producing a value:

1. Handling errors or failures
2. Managing states.
3. Doing IO, Graphics, Controlling Robots
4. Handling non-determinism

How does one do this:

- Without compromising on the pure functional nature of Haskell.
- Without messing up the clean values-only producing code.

Answer: Use Monads

Handling Failures

Here is a small language.

```
data Exp = Con Int | Add Exp Exp | Div Exp Exp
```

Here is an evaluator for the language:

```
eval :: Exp -> Int
eval (Con i) = i
eval (Add e1 e2) = eval e1 + eval e2
eval (Div e1 e2) = eval e1 `div` eval e2
```

Handling Failures

Question: What is `eval (Div (Con 2) (Con 0))`?

An error issued by the *the Haskell interpreter.* (Not our interpreter)

To handle the error, we have to modify *our* interpreter:

```
data Maybe a = Just a | Nothing   -- Just for normal values.

eval :: Exp -> Maybe Int
eval (Con i) = Just i
eval (Add e1 e2) = case eval e1 of
                        Nothing -> Nothing
                        Just i1  -> case eval e2 of
                                        Nothing -> Nothing
                                        Just i2  -> Just (i1 + i2)
eval (Div e1 e2) = case eval e1 of
                        Nothing -> Nothing
                        Just i1  -> case eval e2 of
                                        Nothing -> Nothing
                                        Just 0   -> Nothing
                                        Just i2 -> Just (i1 `div` i2)
```

Can we bring back the simplicity of the earlier code?

Handling Failures

A commonly occuring pattern is:

```
case m of
     Nothing -> Nothing
     Just i -> k i
```

Examples of `m` and `k`

```
case eval e2 of
     Nothing -> Nothing
     Just i2  -> (\i -> Just (i1 + i)) i2
```

```
case eval e1 of
     Nothing -> Nothing
     Just i1  -> k i1
  where k i = case eval e2 of
                   Nothing -> Nothing
                   Just i2  -> Just (i + i2)
```

Another common pattern is `Just i`

The Failure Monad

What are the types of `m` and `k` ?

```
m :: Maybe a
k :: (a -> Maybe b)
```

Abstract out the pattern:

```
case m of
     Nothing -> Nothing
     Just i -> k i
```

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
(>>=) m k = case m of
             Nothing -> Nothing
             Just i -> k i
```

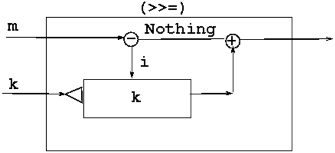
`>>=` is pronounced *then* .

Also write a small abstraction

```
return i = Just i
```

The Failure Monad

Here is (>>=) in pictures:



Now write the interpreter using the abstraction:

```
eval :: Exp -> Maybe Int
eval (Con i) = return i
eval (Add e1 e2) = eval e1 >>=
                    \i1 -> eval e2 >>=
                      \i2 -> return (i1 + i2)
eval (Div e1 e2) = eval e1 >>=
                    \i1 -> eval e2 >>=
                      \i2 -> if (i2 == 0) then Nothing
                             else return (i1 `div` i2)
```

Failure Monad – Summary

- For certain applications we have to handle more than a pure value. Division by zero is an example.
- Introduce a datatype `M a` to capture the extended value.
- Introduce two function:
  - `return :: a -> M a` to to convert ordinary values to monadic values.
  - `(>>=) :: M a -> (a -> M b) -> M b` for monadic application. monadic functions.
- The datatype along with the functions is called a *Monad*.
- Write the application using the monad.
- This methodology is applicable in a very large number of situations.

State Monad

Now consider a language which has variables and state:

```
data Exp = V Var | PP Var | Add Exp Exp | Div Exp Exp
data Var = A | B | C
```

To interpret this language we introduce states.

```
type State = Var -> Int
```

Apart from producing a value, `eval` also changes the state.

```
eval :: Exp -> State -> (Int, State)
```

State Monad

So we introduce `StateMonad a` as a type synonym.

```
type StateMonad a = State  -> (a, State)
```

```
eval :: Exp -> StateMonad Int

eval (V v) = \s -> (s v, s)
eval (PP v) = \s -> let i = s v
                    in (i, update s v (i+1))
eval (Add e1 e2) = \s -> let (i1, s1) = eval e1 s
                             (i2, s2) = eval e2 s1
                         in (i1+i2, s2)
eval (Div e1 e2) = \s -> let (i1, s1) = eval e1 s
                             (i2, s2) = eval e2 s1
                         in (i1/i2, s2)
```

Errors are being ignored.

State Monad

Once again, using monads we can factor out common patterns of code.

```
\s -> let (i1, s1) = m s
      in k i1 s1
```

The patterns occur in the following places:

```
eval (Add e1 e2) = \s -> let (i1, s1) = eval e1 s
                         in k i1 s1
          where k i1 s1 =  let (i2, s2) = eval e2 s1
                             in (i1 + i2, s2)
```

Another place is

```
eval (Add e1 e2) = \s -> let (i1, s1) = eval e1 s
                         in k i1 s1
          where k i1 = \s1 -> let (i2, s2) =  eval e2 s1
                                in k' i2 s2
              where k' i2 s2 = (i1+i2, s2)
```

State Monad

- The types of `m` and `k` are :

```
m :: StateMonad  Int
k :: (Int -> StateMonad Int)
```

- In general:

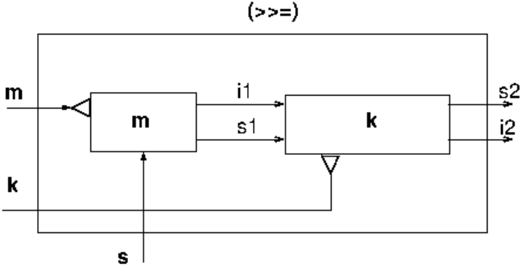
```
m :: StateMonad  a
k :: (Int -> StateMonad b)
```

- The functions `return` and `(>>=)`.

```
return i = \s -> (i, s)
(>>=) m k = \s -> let (i1, s1) = m s
                  in k i1 s1
```

State Monad

`(>>=)` in figures:



State Monad

- Now write the evaluator in terms of these abstractions:

```
eval :: Exp -> StateMonad a

eval (V v) = \s -> (s v, s)
eval (PP v) = \s -> let i = s v
                     in (i, update s v i+1)
eval (Add e1 e2) = eval e1 >>=
                    \i1 eval e2 >>=
                     \i2 return (i1 + i2)
eval (Div e1 e2) = eval e1 >>=
                    \i1 eval e2 >>=
                     \i2 return (i1 / i2)
```

- As an exercise, find the value of

```
eval (Add (Var B) (PP B)) s
    where s v |  v == A = 3
              |  v == B = 6
              |  v == C = 5
```

Haskell support for Monad

- A predefined class definition of the form:

```
class  Monad m  where
    (>>=)  :: m a -> (a -> m b) -> m b  -- then
    (>>)  :: m a -> m b -> m b          -- another form of then
    return :: a -> m a                  -- unit
    (>>) m k = m >>= \_ k
```

- And the syntactic sugars:

```
do
  i1 <- m1
  i2 <- m2
  m3
```

is a shorthand for

```
m1 >>=
 \i1 -> m2 >>=
  \i2 -> m3
```

```
do
  m1
  m2
  m3
```

is a shorthand for

```
m1 >>
  m2 >>
    m3
```

Haskell support for Monads

- The failure monad using Haskell support:

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
   return i = Just i
   m >>= k = case m of
                  Nothing -> Nothing
                  Just i -> k i

eval :: Expr -> Maybe Int
eval (Con i) = return i
eval (Div e1 e2) = do
                      i1 <- eval e1
                      i2 <- eval e2
                      if i2 == 0 then Nothing
                      else return (i1 `div` i2)
```

Haskell support for Monads

- The state monad using Haskell support:

```
type State = Var -> Int

data Var = A | B | C deriving Eq

data StateMonad a = SM (State -> (a, State))

instance Monad StateMonad where
  return i = SM (\st -> (i,st))
  (SM sx) >>= k = SM sx'
    where sx' = \st -> let (i1, st1) = sx st
                           SM sx'' = k i1
                       in sx'' st1
```

Haskell support for Monads

- The state monad using Haskell support:

```
data Exp = V Var | PP Var | Add Exp Exp

eval :: Exp -> StateMonad Int

eval (V v ) = SM (\s -> (s v, s))
eval (PP v) = SM (\s -> (s v, update s v (s v + 1)))
eval (Add e1 e2) = do
                      i1 <- eval e1
                      i2 <- eval e2
                      return (i1 + i2)

mainprog = let (SM sx) = eval (Add (V A) (PP B))
                    in fst (sx initialstate)
```

Exercise

- Rewrite the evaluator so that it also calculates the number of additions and divisions.

Monad Laws

- To qualify as a monad, it is not enough for `return` and `(>>=)` to merely have the types `a -> M a` and `M a -> (a -> M b) -> M b`. They should also satisfy certain *monadic laws*.

  1. `return` **is a left identity of** `(>>=)`**:**

     `return i >>= k` = `k i`

  2. `return` **is a right identity of** `(>>=)`**:**

     `m >>= return` = `m`

  3. `(>>=)` **is associative:**

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

IO in Haskell

- The type `World` captures the state of the Haskell runtime system.

```
type IO a = SM (World -> (a, World))
```

- `IO a` is declared as an instance of the class `Monad`. As a consequence:

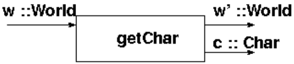
```
return  ::  a -> IO a
(>>=) ::  IO a -> (a -> IO b) -> IO b
```

  come pre-defined.

  - Any reading function returns a value and changes `World`.
  - Any writing function returns `()` and changes `World`.
  - The `initial state` is passed through the distinguished function `main`.

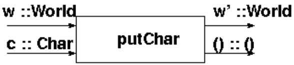
IO in Haskell

- `getChar :: IO Char`



Performs IO action and returns a `Char`

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



Takes a `Char`*, Performs IO action, returns a* `()`

```
echo = do
         c <- getChar
         putChar c
```

IO in Haskell

- `putStr :: String -> IO ()`
  `putStrLn :: String -> IO ()`

```
putStrLn [] = putChar '\n'
putStrLn (x:xs) = do
                    putChar x
                    putStrLn xs
```

- getLine :: IO String

```
getLine= do
           c <- getChar
           if c == '\n' then return []
           else do
                  cs <- getLine
                  return (c:cs)
```

IO in Haskell

- `readLn :: (Read a) => IO a`

  reads any type belonging to the class `Read`
- `print :: Show a => a -> IO ()`

  prints any type belonging to the class `Show`

  ```
  main = do
           i1 <- readLn
           i2 <- readLn
           print (i1 + i2)
  ```
- `getContents :: IO String`

  reads the entire contents of a file *lazily*.

  ```
  main = do
           str <- getContents
           case (reads str) of
              [(i1, str1)] ->
                 case (reads str1) of
                   [(i2, str1)] -> print (i1 + i2)
  ```

IO in Haskell

Now for a game of Hangman

```
main = do
         hSetEcho stdin False
         putStrLn "I am going to give you a word to guess:"
         word <- getLine
         putStrLn $ progress "" word
         hangman ((length word) + 5) [] word
         hSetEcho stdin True


progress sofar answer = map (\c -> if c `elem` sofar then c
                                   else '_')

-- progress ['c', 'd'] "credit" = ['c', '_', '_', 'd', '_','_']
```

IO in Haskell

```
hangman 0 _ _ =  putStrLn "Dead\n"
hangman n sofar ans | ans == progress sofar ans = putStrLn "Well done"
                    |  otherwise = do
                                          putStrLn $ "You have" ++ show n ++
                                          "guesses left. Enter next guess\n"
                                          c <- getChar
                                          respond c n sofar ans


respond c n sofar ans | c `elem` sofar = do
                                              putStrLn $ progress sofar ans
                                              hangman n sofar ans
                      | c `elem` ans = do
                                            putStrLn $ progress (c:sofar) ans
                                            hangman n (c:sofar) ans
                      | otherwise = putStrLn $ progress (c:sofar) ans
                                   hangman (n+1) (c:sofar) ans
```