Monads and IO in Haskell

Amitabha Sanyal

April 25, 2005

Abstract

In certain situations a program has to handle more than normal values. Examples are computations that can produce errors and computations that have to handle state change. A monad provides a mechanism of handling such situations without cluttering the program.

Haskell models IO as compuations which change the state of the external world.

In certain situations we have to handle more than normal values. Examples are computations that can produce errors and computations that have to handle state change. A monad provides a mechanism of handling such situations without cluttering the program.

1 A first example - The Error monad

Consider writing an evaluator for the following langugage:

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

The evaluator is defined in Haskell as the function `eval`:

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

Now suppose we wanted `eval` itself to handle the error situation produced by a division by 0. Then the modified definition of `eval` will be:

```
data Value = N Int | Error   -- N for normal values.

eval :: Exp -> Value
eval (Con i) = N i
eval (Add e1 e2) = case eval e1 of
                        Error -> Error
                        N i1  -> case eval e2 of
                                      Error -> Error
                                      N i2  -> N (i1 + i2)
eval (Div e1 e2) = case eval e1 of
                        Error -> Error
                        N i1  -> case eval e2 of
                                      Error -> Error
                                      N 0   -> Error
                                      N i2 -> N (i1 / i2)
```

The code which has become messy can be cleaned by factoring out frequently occurring patterns by the use of a monad.

```
data ErrorMonad a = N a | Error
```

A monad is a datatype which extends normal value along two functions `unit` and `then`, which have the following types.

```
unit :: a -> ErrorMonad a
then :: ErrorMonad a -> (a -> ErrorMonad b) -> ErrorMonad b
```

In the case of of the `ErrorMonad`, the functions `unit` and `then` are:

```
unit i = N i
then m k = case m of
                Error -> Error
                N i  -> k i
```

So that the monadic definition of the evaluator becomes

```
eval :: Exp -> ErrorMonad
eval (Con i) = unit i
eval (Add e1 e2) = eval e1 'then'
                        \i1 -> eval e2 'then'
                          \i2 -> unit (i1 + i2)
eval (Div e1 e2) = eval e1 'then'
                        \i1 -> eval e2 'then'
                          \i2 -> if (i2 == 0) then Error
                                 else unit (i1 / i2)
```

2 A second example - state monad

Now consider the language which has variables and a x++ like construct to change the state:

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

To interpret this language we have to introduce states.

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

Now, apart from producing a value, eval also changes the state. Ignoring the production of error values, eval can be written as

```
eval :: State -> (Int, State)
eval (V v) s = (s v, s)
eval (PP v) s = let i = s v
                 in (i, \v'. if v == v' then i+1 else s v')
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)
```

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

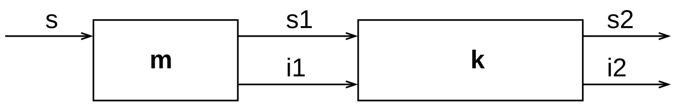
```
type State = Ide -> Int
type StateMonad a = State -> (a, State)

unit :: a -> StateMonad a
then :: StateMonad a -> (a -> StateMonad b) -> StateMonad b
```

Notice that the types of unit and then remain unchanged except that ErrorMonad has been replaced by StateMonad.

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

The action of then can be explained by the following diagram:



The monadic form of the evaluator is:

```
eval :: Exp -> StateMonad a

eval (V v) = \s -> (s v, s)
eval (PP v) = \s -> let i = s v
                     in (i, \v'. if v == v' then i+1 else s v')
eval (Add e1 e2) = eval e1 'then'
                    \i1 eval e2 'then
                     \i2 unit (i1 + i2)
eval (Div e1 e2) = eval e1 'then'
                    \i1 eval e2 'then
                     \i2 unit (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
```

3 Haskell support for monads

In Haskell, there is a predefined class called `Monad`

```
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
```

The second form of `then` is useful in situations when the value produced by the first argument of `then` is not required by the second. `(>>)` can be defined as

```
  (>>) m k =  m >>= \_ -> k
```

We shall see examples of use of (¿¿) in defining IO functions.
We can now define `ErrorMonad` and `StateMonad` to be instances of `Monad`

```
instance Monad ErrorMonad where
   (>>=) m k  = case m of
                 Error -> Error
                 N i  -> k i
   return i = N i
```

Now the monadic evaluator for `eval` can be written as:

```
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 Error
                                 else return (i1 / i2)
```

In fact, Haskell provides a notation called `do` to express the above very conveniently.

```
eval (Con i) = return i
eval (Add e1 e2) = do
                      i1 <- eval e1
                      i2 <- eval e2
                      return (i1 + i2)
eval (Div e1 e2) = do
                      i1 <- eval e1
                      i2 <- eval e2
                      if (i2 == 0) then Error else return (i1 / i2)
```

In summary.

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

is a shorthand for

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

whereas

```
do
  m1
  m2
  m3
```

is a shorthand for

```
  m1 (>>) m2 (>>) m3
```

4 A third example - IO monad

We now add features to perform IO in our example language.

```
data Exp = (Con i) | Read | Print Exp | Add Exp Exp
```

IO is modeled as a changes in state, where the state consists of a pair of lists representing input and an output streams.

```
type IOState = ([Int], [Int])
type IOMonad a = IOState -> (a, IOState)
```

Now we make `IOMonad` as an instance of `Monad`.

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

The details of the state has not afftected the `unit` and the `then` definition.
The evaluator for this language is

```
eval :: Exp -> IOMonad
eval (Con i) = return i
eval Read = \(i:is, os) -> (i, (is,os))
eval (Print e) = do
                   o <- eval e
                  \(is,os) -> ((), (is, o:os))
eval (Add e1 e2) = do
                     i1 <- eval e1
                     i2 <- eval e2
                     return (i1 + i2)
```

To enable a program in the example language to perform IO, we have to call the evaluator with the program and supply it with an initial state. As an exercise, find the value of the program below.

```
eval (Print (Add Read Read)) ([4,6,3,3], [])
```

5 The Haskell IO model

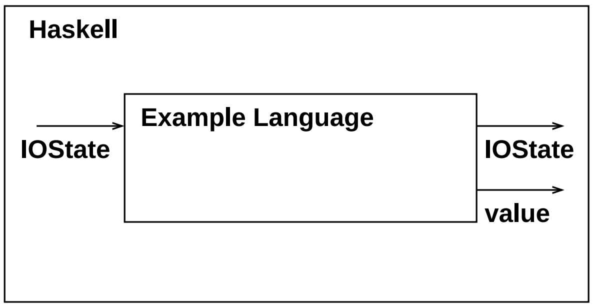
We just modeled a small IO-capable language on top of Haskell.

```
type IOMonad a = IOState -> (a, IOState)
```

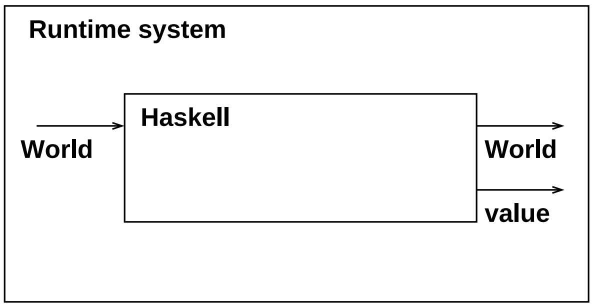
To evaluate a program in this language, we create an initial state at the Haskell level, and pass it to the program.

```
eval exp initialState
```

This can be summarized by the diagram shown below.



The IO model of Haskell can be understood in terms of a similar diagram:



`World` is a datatype modeling the state of the Haskell runtime system.

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

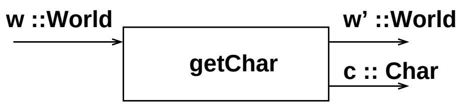
The runtime system passes an initial `World` to Haskell. This happens when we call the function `main`.

```
(eval) main initialWorld
```

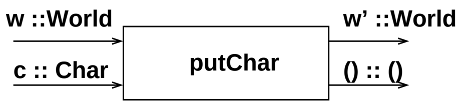
Regard `IO a` as an "action" (script) that, when performed, may do some input/output, before delivering a value of type a. Here "performed" means supplied with a World, and "do some input output" means change the World.
World is an abstract data type. It cannot be defined in or created inside Haskell. It can only be modified through a set of given IO functions.
Here are some functions defined in Haskell.

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

`getChar` takes a `World` and reads a character from the keyboard, thereby changing it. In terms of a diagram:



`putChar` on the otherhand, takes a character and a `World` and changes the `World` by writing the character on the console. It returns the void value.



Using these, we can write a program to echo a character from the keyboard.

```
echo :: IO () echo = do
                        c <- getChar
                        putChar c
```

And another to echo the character twice.

```
echoDup :: IO ()
echoDup = do
           c <- getChar
           putChar c
           putChar c
```

`getTwoChars` gets two characters.

```
getTwoChars :: IO (Char, Char)
getTwoChars = do
                c1 <- getChar
                c2 <- getChar
                return (c1, c2)
```

and getLine reads an entire line.

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

forever performs an IO action for ever.

```
forever :: IO () -> IO ()
forever a = do
               a
               forever a


sequence :: [IO a] -> IO [a]
sequence       :: Monad m => [m a] -> m [a]
sequence       =  foldr mcons (return [])
                      where mcons p q = do
                                          x <- p
                                          y <- q
                                          return (x:y)

sequence_      =  foldr (>>) (return [])
```

getContents operation returns all user input as a single string, which is read lazily as it is needed.
getContents :: IO [Char]

```
interact :: (String -> String) -> IO ()

interact f = do
              s <- getContents
              putStr(f s)
```

Here is an example of interact

```
main = interact(unlines . map f . takewhile (\= ‘‘quit’’) . lines)
         where f = map toUpper
```

Here are some reading functions:

```
readLn      :: Read a => IO a

readIO    :: Read a => String -> IO a
readIO s =  case [x | (x,t) <- reads s, ("","") <- lex t] of
              [x] -> return x
              []  -> ioError (userError "Prelude.readIO: no parse")
              _   -> ioError (userError "Prelude.readIO: ambiguous parse")


readLn           :: Read a => IO a
readLn           =  do l <- getLine
                       r <- readIO l
                       return r

print       :: Show a => a -> IO ()
```

6 Single-threaded-ness and implementation

Consider the program

```
getChar >>= \c -> (putChar c >> putChar c)
```

This rewrites to:

```
\world -> let (c, world') = getChar world
            in let (_, world'') = putChar c world'
                 in let (v, world''') = putChar c world''
                      in (v, world''')
```

This is single threaded. The same copy of the world passes through the program getting modified in the process. This admits a feasible and efficient impementation in which every copy of getChar and putChar is replaced with a corresponding C function.
Suppose the compiler rewrites this to:

```
\world -> let (c, world') = getChar world
            in let (_, world'') = putChar c world'
                 in let (v, world''') = putChar (fst (getChar world))
                                                  world''
                      in (v, world''')
```

Since the world is duplicated, the efficient implementation through C is not possible.

6.1 IO specific to a given type

```
readIO    :: Read a => String -> IO a
readIO s =
       case [x | (x,t) <- reads s, ("","") <- lex t] of
        [x] -> return
        []  -> ioError (userError "Prelude.readIO: no parse")
        _   -> ioError (userError "Prelude.readIO: ambiguous parse")
```

From a string s reads a value of a type a. After reading the character there should not be any Haskell lexeme left in the string. And the value should not have an ambiguous parse.
A user defined type, such as a binary tree could have an ambiguous parse

```
readLn      :: Read a => IO a

readLn           :: Read a => IO a
readLn           =  do l <- getLine
                       r <- readIO l
                       return r

print        :: Show a => a -> IO ()
```

6.2 File IO

```
type FilePath =  String

writeFile     ::  FilePath -> String -> IO ()
appendFile    ::  FilePath -> String -> IO ()
readFile      ::  FilePath -> IO String

main = do
         putStr "Input file: "
         ifile <- getLine
         putStr "Output file: "
         ofile <- getLine
         s <- readFile ifile
         writeFile ofile (filter isAscii s)
         putStr "Filtering successful\n"
```