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 computations 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 language:

\[
data \text{ Exp } = \text{ Con } \text{ Int } \mid \text{ Add } \text{ Exp } \text{ Exp } \mid \text{ Div } \text{ Exp } \text{ Exp}
\]

The evaluator is defined in Haskell as the function \text{eval}:

\[
\text{eval} :: \text{ Exp } \rightarrow \text{ Int}
\]

\[
\text{eval (Con i)} = \text{i}
\]

\[
\text{eval (Add e1 e2)} = \text{Eval e1 + Eval e2}
\]

\[
\text{eval (Div e1 e2)} = \text{Eval e1 / Eval e2}
\]

Now suppose we wanted \text{eval} itself to handle the error situation produced by a division by 0. Then the modified definition of \text{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, \texttt{unit} and \texttt{then}, which have the following types.

\texttt{unit :: a -> ErrorMonad a}
\texttt{then :: ErrorMonad a -> (a -> ErrorMonad b) -> ErrorMonad b}

In the case of of the \texttt{ErrorMonad}, the functions \texttt{unit} and \texttt{then} are:

\texttt{unit i = N i}
\texttt{then m k = case m of}
  \texttt{Error -> Error}
  \texttt{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'
  \texttt{\_i1 -> eval e2 'then'
  \_i2 -> unit (i1 + i2)}
eval (Div e1 e2) = eval e1 'then'
  \texttt{\_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 \texttt{++} like construct to change
the state:

\begin{verbatim}
data Exp = V Var | PP Var | Add Exp Exp | Div Exp Exp
data Var = A | B | C
\end{verbatim}

To interpret this language we have to introduce states.

\begin{verbatim}
type State = Var -> Int
\end{verbatim}

Now, apart from producing a value, \texttt{eval} also changes the state. Ignoring the pro-
duction of error values, \texttt{eval} can be written as

\begin{verbatim}
eval :: State -> (Int, State)
eval (V v) s = (s v, s)
eval (PP v) s = let \texttt{i = s v}
in (i, \texttt{\textbackslash v' \textbackslash . if v \texttt{== v' \textbackslash then i+1 \textbackslash 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)
\end{verbatim}

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

\begin{verbatim}
type State = Ide -> Int
type StateMonad a = State -> (a, State)
\end{verbatim}

\begin{verbatim}
unit :: a -> StateMonad a
then :: StateMonad a -> (a -> StateMonad b) -> StateMonad b
\end{verbatim}

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has been replaced by \texttt{StateMonad}.

\begin{verbatim}
unit i s = (i, s)
then m k = \texttt{\textbackslash s \textbackslash -> let (i1, s1) = m s}
in k i1 s1
\end{verbatim}

The action of \texttt{then} can be explained by the following diagram:
The monadic form of the evaluator is:

```
val :: Exp -> StateMonad a

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

As an exercise, find the value of

```
val (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

4
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
4 A third example - IO monad

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

```haskell
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.

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

Now we make IOMonad as an instance of Monad.

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

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

```haskell
eval :: Exp -> IOMonad
eval (Con i) = return i
eval Read = \(i, is, os) \rightarrow (i, (is, os))
eval (Print e) = do
  o <- eval e
  \(is, os) \rightarrow ((), (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.

```haskell
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.

\[
\text{type } \text{IOMonad } a = \text{IOState } \rightarrow (a, \text{IOState})
\]

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

\[
\text{eval } \text{exp } \text{initialState}
\]

This can be summarized by the diagram shown below.

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

\[
\text{World is a datatype modeling the state of the Haskell runtime system.}
\]

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

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

\[
(\text{eval}) \text{main initialWorld}
\]
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.

\[
\begin{align*}
\text{getChar} & : \text{IO Char} \\
\text{putChar} & : \text{Char} \rightarrow \text{IO ()}
\end{align*}
\]

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

```
  w :: World
  getChar
  w' :: World
  c :: Char
```

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

```
  w :: World
  c :: Char
  putChar
  w' :: World
  () :: ()
```

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

\[
\begin{align*}
\text{echo} & : \text{IO ()} \\
\text{echo} & = \text{do}
\end{align*}
\]

```
c <- \text{getChar}
\text{putChar} c
```

And another to echo the character twice.

\[
\begin{align*}
\text{echoDup} & : \text{IO ()} \\
\text{echoDup} & = \text{do}
\end{align*}
\]

```
c <- \text{getChar}
\text{putChar} c
\text{putChar} c
```

getTwoChars gets two characters.

\[
\begin{align*}
\text{getTwoChars} & : \text{IO (Char, Char)} \\
\text{getTwoChars} & = \text{do}
\end{align*}
\]

```
c1 <- \text{getChar}
c2 <- \text{getChar}
\text{return} (c1, c2)
```
and `getLine` reads an entire line.

```haskell
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
```

```haskell
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)
```

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

`getContents` operation returns all user input as a single string, which is read lazily as it is needed.

```haskell
getContents :: IO [Char]
```

```haskell
interact :: (String -> String) -> IO ()
interact f = do
  s <- getContents
  putStrLn(f s)
```

Here is an example of `interact`

```haskell
main = interact(unlines . map f . takeWhile (\= "quit") . lines)
   where f = map toUpper
```
Here are some reading functions:

\[
\text{readLn :: Read a \Rightarrow I0 a}
\]

\[
\text{readIO :: Read a \Rightarrow String \Rightarrow I0 a}
\]

\[
\text{readIO s = case [x | (x,t) <- reads s, ("","") <- lex t] of}
\]

\[
\begin{align*}
[x] & \rightarrow \text{return x} \\
\emptyset & \rightarrow \text{ioError (userError "Prelude.readIO: no parse")} \\
- & \rightarrow \text{ioError (userError "Prelude.readIO: ambiguous parse")}
\end{align*}
\]

\[
\text{readLn :: Read a \Rightarrow I0 a}
\]

\[
\text{readLn = do l <- getLine}
\]

\[
\text{r <- readIO l}
\]

\[
\text{return r}
\]

\[
\text{print :: Show a \Rightarrow a \Rightarrow I0 ()}
\]

6 Single-threaded-ness and implementation

Consider the program

\[
\text{getChar >>= \"c \rightarrow (putChar c >> putChar c)\"}
\]

This rewrites to:

\[
\lambda world \rightarrow \text{let (c, world') = getChar world}
\]

\[
\begin{align*}
\text{in let (cw, world'') = putChar c world'} \\
\text{in let (v, world''') = putChar c world'''}
\end{align*}
\]

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 implemention in which every copy of getChar and putChar is replaced with a corresponding C function.

Suppose the compiler rewrites this to:

\[
\lambda world \rightarrow \text{let (c, world') = getChar world}
\]

\[
\begin{align*}
\text{in let (cw, world'') = putChar c world'} \\
\text{in let (v, world''') = putChar (fst (getChar world)) world'''}
\end{align*}
\]

This is not possible.
6.1 IO specific to a given type

\[
\text{readIO} :: \text{Read } a \Rightarrow \text{String } \Rightarrow \text{IO } a \\
\text{readIO } s = \\
\quad \text{case } [x \mid (x,t) \leftarrow \text{reads } s, ("","") \leftarrow \text{lex } t] \text{ of} \\
\quad [x] \Rightarrow \text{return} \\
\quad [] \Rightarrow \text{ioError (userError "Prelude.readIO: no parse")("Prelude.readIO: ambiguous parse")}
\]

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

\[
\text{readLn} :: \text{Read } a \Rightarrow \text{IO } a \\
\text{readLn} = \text{do } l \leftarrow \text{getLine} \\
\quad r \leftarrow \text{readIO } l \\
\quad \text{return } r
\]

\[
\text{print} :: \text{Show } a \Rightarrow a \Rightarrow \text{IO } ()
\]

6.2 File IO

\[
\text{type } \text{FilePath} = \text{String}
\]

\[
\text{writeFile} :: \text{FilePath } \Rightarrow \text{String } \Rightarrow \text{IO } () \\
\text{appendFile} :: \text{FilePath } \Rightarrow \text{String } \Rightarrow \text{IO } () \\
\text{readFile} :: \text{FilePath } \Rightarrow \text{IO } \text{String}
\]

\[
\text{main } = \text{do} \\
\quad \text{putStr } \"\text{Input file: } \" \\
\quad \text{ifile } \leftarrow \text{getLine} \\
\quad \text{putStr } \"\text{Output file: } \" \\
\quad \text{ofile } \leftarrow \text{getLine} \\
\quad s \leftarrow \text{readFile } \text{ifile} \\
\quad \text{writeFile } \text{ofile } (\text{filter } \text{isAscii } s) \\
\quad \text{putStr } \"\text{Filtering successful}\n\"}
\]