Monad P3 : IO Monad Methods (2B)

Young Won Lim
10/22/19

Please send corrections (or suggestions) to youngwlim@hotmail.com.

Based on

Haskell in 5 steps
https://wiki.haskell.org/Haskell_in_5_steps

Examples of Returning **IO a** Value

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

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

c :: Char

```
ready     :: IO Bool
ready     =  do   c <- getChar
                  c == 'y'  -- Bad!!!
```

```
ready     :: IO Bool
ready     =  do   c <- getChar
                  return (c == 'y' )
```

c == 'y' : just a boolean value,

not an action.

need to take this boolean

and create an action

that does nothing

but return the boolean as its result.

https://www.haskell.org/tutorial/io.html

do produces a chain of statements

```
return            ::  a -> IO a

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

Each **do** introduces a single chain of statements.

Any intervening construct, such as the **if**,

must use a new **do** to initiate further sequences of actions.

https://www.haskell.org/tutorial/io.html

Unsafe functions – extracting **a** from **IO a**

f :: Int -> Int -> Int

absolutely cannot do any **I/O**
since no **IO a** in the returned type.

it is not intended to place **print** statements liberally throughout
their code during debugging in Haskell. (not like C programming)

There are some **unsafe functions** available to get around this
problem but these are not recommended.

Debugging packages (like **Trace**) often make liberal use of
these '**forbidden functions**' in an entirely safe manner.

Note that there is no function like this:

unsafe :: IO a -> a

https://www.haskell.org/tutorial/io.html

IO global ordering

No escape from the IO **monad**.

[exception: **unsafePerformIO**. Do not use!]

all the I/O that your program will ever perform

gets bundled up into a giant **single** IO **block**,

thus enforcing a **global ordering** on the operations.

[unless **forkIO** is called]

unsafePerformIO is so **unsafe** because it is impossible

to figure out exactly *when*, *if*, or *how* many times

the enclosed **I/O operations** will happen

Note that there is no function like this:

unsafe :: IO a -> a

https://www.haskell.org/tutorial/io.html

IO Actions: Ordinary Values

```
todoList :: [IO ()]

todoList = [ putChar 'a',
             do putChar 'b'
                putChar 'c',
             do c <- getChar
                putChar c]
```

This list does not actually invoke any **actions**
---it simply holds them.

To join these **actions** into a **single action**,
a function such as **sequence_** is needed:

https://www.haskell.org/tutorial/io.html

Join a list of actions

```
sequence_        :: [IO ()] -> IO ()
sequence_ []      =  return ()
sequence_ (a:as) =  do a
                       sequence_ as
```

```
do x;y

x >> y
```

```
sequence_        :: [IO ()] -> IO ()
sequence_         =  foldr (>>) (return ())
```

https://www.haskell.org/tutorial/io.html

Another Examples of Returning **IO a** (1)

```
getLine :: IO String
putStrLn :: String -> IO ()  -- note that the result value is an empty tuple.
randomRIO :: (Random a) => (a,a) -> IO a
```

Normally Haskell **evaluation** doesn't cause
this **execution** to occur.

A value of type (**IO a**) is almost completely inert.
the only **IO** **action** that can be run is **main**

https://wiki.haskell.org/Introduction_to_IO

Another Examples of Returning IO a (2)

```
main :: IO ()
main = putStrLn "Hello, World!"
```

putStrLn :: String -> IO ()

```
main = putStrLn "Hello" >> putStrLn "World"
```

```
main = putStrLn "Hello, what is your name?"
          >> getLine
          >>= \name -> putStrLn ("Hello, " ++ name ++ "!")
```

getLine :: IO String

putStrLn :: String -> IO ()

https://wiki.haskell.org/Introduction_to_IO

putStr via putChar

```
putStr            :: String -> IO ()
putStr s           =  sequence_ (map putChar s)
```

In an **imperative** language,
mapping an **imperative** version of **putChar** over the **string**
would be sufficient to print it.

In **Haskell**, however,
the map function does not perform any **action**.
Instead it creates a **list** of **actions**,
one for each **character** in the string.

```
do x;y

x >> y
```

```
map putChar "abc"

[ putChar 'a',  putChar 'b',  putChar 'c' ]
```

https://www.haskell.org/tutorial/io.html

foldr

```
foldr :: (a -> b -> b) -> b -> [a] -> b

foldr f z [x1, x2, ..., xn] == x1 `f` (x2 `f` ... (xn `f` z)...)

foldr (+) 5 [1,2,3,4]
(+) :: (a -> b -> b)
5 :: b
[1,2,3,4] :: [a]
```

```
fold (+) [1,2,3,4,5]
1 + 2 + 3 + 4 + 5 = 15

foldr (+) 5 [1,2,3,4]
(1+(2+(3+(4+5)))) = 15

foldr (/) 2 [8,12,24,4]
(8/(12/(24/(4/2))))
(8/(12/(24/2)))
(8/(12/12))
(8/1)
8
```

https://www.haskell.org/tutorial/io.html

putStr via putChar example

```
map putChar "abc"
[ putChar 'a',  putChar 'b',  putChar 'c' ]

sequence_ (map putChar "abc")
foldr (>>) (return ()) (map putChar "abc")
foldr (>>) (return ())  [ putChar 'a',  putChar 'b',  putChar 'c' ]

(putChar 'a'  >> (putChar 'b' >> (putChar 'c'  >> (return ()))))
(putChar 'a'  >> putChar 'b' >> putChar 'c'  >> return ())
```

```
map putChar "abc"

[ putChar 'a',  putChar 'b',  putChar 'c' ]
```

https://www.haskell.org/tutorial/io.html

Files, Channels, Handles

```
type FilePath    =  String  -- path names in the file system
openFile         :: FilePath -> IOMode -> IO Handle
hClose           :: Handle -> IO ()
data IOMode      =  ReadMode | WriteMode
                  | AppendMode | ReadWriteMode
```

Opening a file creates a **handle** (of type **Handle**)

for use in I/O transactions.

Closing the **handle** closes the associated file:

https://www.haskell.org/tutorial/io.html

Files, Channels, Handles

Handles can also be associated with **channels**:

communication ports not directly attached to **files**.

Predefined channel handles :**stdin**, **stdout**, and **stderr**

Character level I/O operations include **hGetChar** and **hPutChar**,

which take a handle as an argument.

```
getChar        = hGetChar stdin
```

Haskell also allows the entire **contents** of a **file** or **channel**

to be returned as a **single string**:

```
getContents     :: Handle -> IO String
```

https://www.haskell.org/tutorial/io.html

Files, Channels, Handles

```
main = do fromHandle <- getAndOpenFile "Copy from: "
ReadMode
          toHandle   <- getAndOpenFile "Copy to: " WriteMode
          contents   <- hGetContents fromHandle
          hPutStr toHandle contents
          hClose toHandle
          putStr "Done."
```

```
getAndOpenFile          :: String -> IOMode -> IO Handle
getAndOpenFile prompt mode =
    do putStr prompt
       name <- getLine
       catch (openFile name mode)
             (\_ -> do putStrLn ("Cannot open "++ name ++ "\n")
                       getAndOpenFile prompt mode)
```

https://www.haskell.org/tutorial/io.html

Functional vs Imperative Programming

```
getLine        = do c <- getChar
                  if c == '\n'
                     then return ""
                     else do l <- getLine
                          return (c:l)
```

```
function getLine() {
  c := getChar();
  if c == `\n` then return ""
          else {l := getLine();
             return c:l}}
```

https://www.haskell.org/tutorial/io.html

IOError Monad

Errors are encoded using a special data type, **IOError**.

This type represents all possible **exceptions**

that may occur within the **I/O monad**.

This is an **abstract** type:

no **constructors** for IOError are available to the user.

isEOFError :: IOError -> Bool

https://www.haskell.org/tutorial/io.html

Exception Handling

An **exception handler** has type **IOError -> IO a**.

The **catch function** associates an **exception handler**

with an **action** or **set of actions**

The **arguments** to catch are an **action** and a **handler**.

```
              action        handler
catch  :: IO a -> (IOError -> IO a) -> IO a
```

If the **action** succeeds,

its result is returned without invoking the handler.

If the **action** fails (an **error** occurs),

the **error** is passed to the **handler** as a **value** of type **IOError**

and the **handler's action** is then invoked

https://www.haskell.org/tutorial/io.html

Exception Handling

```
catch          :: IO a -> (IOError -> IO a) -> IO a
```

```
getChar'       :: IO Char
getChar'       =  getChar `catch` (\e -> return '\n')
```

```
getChar'       :: IO Char
getChar'       =  getChar `catch` eofHandler where
   eofHandler e = if isEofError e then return '\n' else ioError e
```

```
isEOFError     :: IOError -> Bool
ioError        :: IOError -> IO a
```

https://www.haskell.org/tutorial/io.html

Exception Handling

```
getLine'  :: IO String
getLine'  = catch getLine'' (\err -> return ("Error: " ++ show err))
        where
                getLine'' = do c <- getChar'
                               if c == '\n' then return ""
                                            else do l <- getLine'
                                                    return (c:l)
```

https://www.haskell.org/tutorial/io.html

RandomRIO, RandomIO

```
randomR :: RandomGen g => (a, a) -> g -> (a, g)
random  :: RandomGen g =>           g -> (a, g)
```

takes a range **(lo,hi) :: (a, a)** and

a random number generator **g**,

returns a **random value** uniformly distributed

in the closed interval **[lo,hi]**,

together with a new generator g

```
randomRIO :: (a, a) -> IO a
randomIO  ::           IO a
```

A variant of **randomR** / **random**

that uses the global random number generator

See System.Random

https://hackage.haskell.org/package/random-1.1/docs/System-Random.html

RandomRIO Example

```
import System.Random

main = do
          putStr . show  =<< randomRIO (0, 100 :: Int)      :: IO Int
          putStr ", "
          print          =<< randomRIO (0, 100 :: Int)      :: IO Int

          print          =<< (randomIO :: IO Float)         :: IO Float
```

```
randomRIO :: (a, a) -> IO a
randomIO  ::           IO a
```

```
$ runhaskell random-numbers.hs
51, 15
0.2895795
```

https://hackage.haskell.org/package/random-1.1/docs/System-Random.html

IO Monad

```
newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))

instance  Monad IO  where
    m >> k    = m >>= \ _ -> k
    return    = returnIO
    (>>=)     = bindIO
    fail s    = failIO s

returnIO :: a -> IO a
returnIO x = IO $ \s -> (# s, x #)

bindIO :: IO a -> (a -> IO b) -> IO b
bindIO (IO m) k = IO $ \s -> case m s of (# new_s, a #) -> unIO (k a) new_s

unIO :: IO a -> (State# RealWorld -> (# State# RealWorld, a #))
unIO (IO a) = a
```

http://blog.ezyang.com/2011/05/unraveling-the-mystery-of-the-io-monad/

IO Monad

```
newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))

instance  Monad IO  where
    {-# INLINE return #-}
    {-# INLINE (>>)   #-}
    {-# INLINE (>>=)  #-}
    m >> k    = m >>= \ _ -> k
    return    = returnIO
    (>>=)     = bindIO
    fail s    = failIO s

returnIO :: a -> IO a
returnIO x = IO $ \ s -> (# s, x #)

bindIO :: IO a -> (a -> IO b) -> IO b
bindIO (IO m) k = IO $ \ s -> case m s of (# new_s, a #) -> unIO (k a) new_s

unIO :: IO a -> (State# RealWorld -> (# State# RealWorld, a #))
unIO (IO a) = a
```

http://hackage.haskell.org/package/base-4.12.0.0/docs/src/GHC.Base.html#Monad

IO Monad

```
newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))
(>>=)    = bindIO

bindIO :: IO a -> (a -> IO b) -> IO b
bindIO (IO m) k = IO $ \s ->
                                    case m s of
                                      (# s’, a #) -> unIO (k a) s’


        (IO m) >>= k
        IO m :: IO a          m :: State# RealWorld -> (# State# RealWorld, a #)
        k :: a -> IO b        k a :: IO b


          s  :: State# RealWorld
          s’ :: State# RealWorld
        m s :: (# State# RealWorld, a #)
  (# s’, a #) :: (# State# RealWorld, a #)
```

http://blog.ezyang.com/2011/05/unraveling-the-mystery-of-the-io-monad/

IO Monad

```
newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))
(>>=)    = bindIO

bindIO :: IO a -> (a -> IO b) -> IO b
bindIO (IO m) k = IO $ \s ->
                             case m s of
                               (# s', a #) -> unIO (k a) s'

unIO :: IO a -> (State# RealWorld -> (# State# RealWorld, a #))
unIO (IO a) = a

      k :: a -> IO b        k a :: IO b

              unIO (k a) :: State# RealWorld -> (# State# RealWorld, a #)
                     s' :: State# RealWorld
            unIO (k a) s' :: (# State# RealWorld, a #)
       \s -> unIO (k a) s' :: State# RealWorld -> (# State# RealWorld, a #)
   IO $ \s -> unIO (k a) s' :: IO b
```

http://blog.ezyang.com/2011/05/unraveling-the-mystery-of-the-io-monad/

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

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf