# IO Monad (3C)

Young Won Lim
12/11/17

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

This document was produced by using OpenOffice.

# Based on

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

# IO ( )

```
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 is to run in main

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

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

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

https://wiki.haskell.org/Introduction_to_IO

# IO ( )

(>>) :: IO a -> IO b -> IO b

where if x and y are IO actions, then (x >> y) is the action that performs x, dropping the result, then performs y and returns its result.

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

to use the result of the first in order to affect what the second action will do

Now, x >>= f is the action that first performs the action x, and captures its result, passing it to f, which then computes a second action to be performed. That action is then carried out, and its result is the result of the overall computation.

x >> y = x >>= const y

https://wiki.haskell.org/Introduction_to_IO

# IO ( )

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

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

```
return :: a -> IO a
```

Note that there is no function:

```
unsafe :: IO a -> a
```

https://wiki.haskell.org/Introduction_to_IO

# Basic IO

```
getChar          ::   IO Char

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

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

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

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

# Basic IO

```
return              ::   a -> IO a

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

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

# Basic IO

```
f   ::  Int -> Int -> Int
```

absolutely cannot do any I/O since IO does not appear in the returned type.

Basically, it is not intended to place print statements liberally throughout their code during debugging in Haskell.

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.

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

# Actions

```
todoList :: [IO ()]

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

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

```
do x;y
x >> y
```

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

# Actions

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

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

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

# Exception Handling

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.

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

If the action succeeds,

its result is returned without invoking the handler.

If an error occurs, it is passed to the handler as a value of type IOError and the action associated with the handler 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

# 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.

The getChar function used previously can be defined as:

```
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

# IO ( )

**put :: s -> State s ( )**

**put :: s -> (State s) ( )**

one value input type **s**
the effect-monad **State s**
the value output type **( )**

the operation is used *only for its effect*;
the *value* delivered is *uninteresting*

**putStr :: String -> IO ()**

delivers a string to stdout but does not return anything exciting.

https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

# Monadic Effect

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

https://en.wikibooks.org/wiki/Haskell/Understanding_monads/IO
https://stackoverflow.com/questions/2488646/why-are-side-effects-modeled-as-monads-in-haskell
https://stackoverflow.com/questions/7840126/why-monads-how-does-it-resolve-side-effects
https://stackoverflow.com/questions/2488646/why-are-side-effects-modeled-as-monads-in-haskell

https://www.cs.hmc.edu/~adavidso/monads.pdf

# IO ( )

Monadic operations tend to have types which look like

val-in-type-1 -> ... -> val-in-type-n -> effect-monad val-out-type

where the return type is a type application:
the function tells you which effects are possible
and the argument tells you what sort of value
is produced by the operation

https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

# Side Effects in Haskell

Generally, a monad cannot perform side effects in Haskell.
there is one exception: **IO monad**

Suppose there is a type called World,
which contains all the state of the external universe

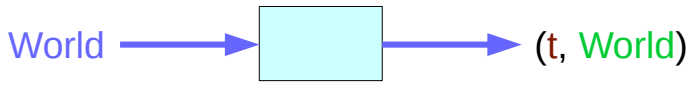
A way of thinking what IO monad does

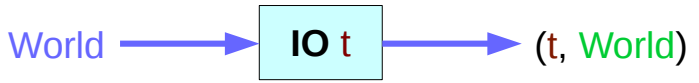
```
type  IO t  =  World  ->  (t, World)
```

type synonym

**IO** t is a function
*input* : a World
*output*: the t it's supposed to contain,
a new, updated World obtained
by modifying the given World
in the process of computing the t.

World -> (t, World)



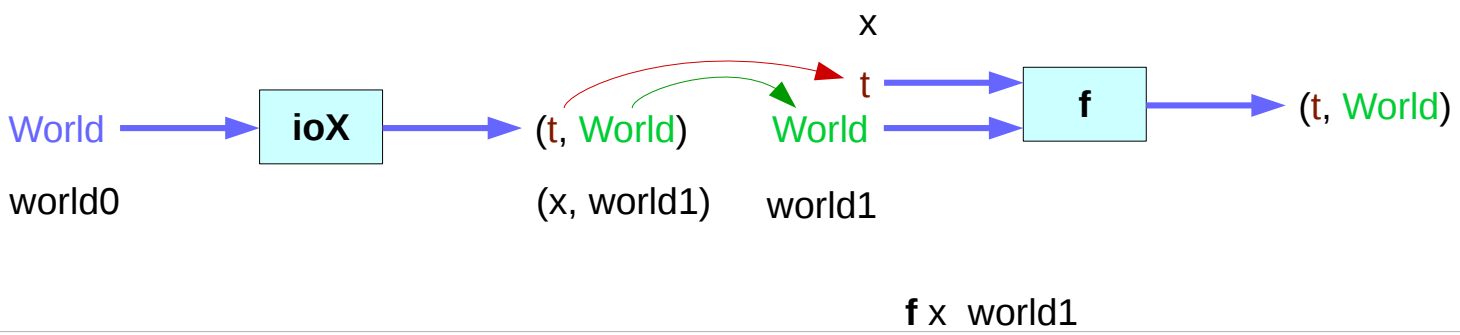




**IO** x world0 (x, world1)

https://www.cs.hmc.edu/~adavidso/monads.pdf

# Side Effects in Haskell

```
instance Monad IO where
   return x world = (x, world)
   (ioX >>= f) world0  =
   let
      (x, world1) = ioX world0
   in
      f x world1                                  -- Has type (t, World)
```



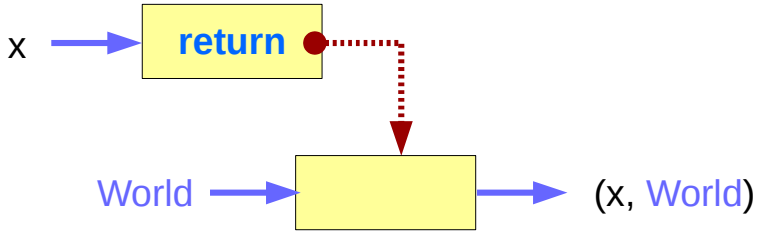

https://www.cs.hmc.edu/~adavidso/monads.pdf

# Side Effects in Haskell

The return function takes x
and gives back a function
    that takes a World
    and returns x along with the “new, updated” World
    formed by not modifying the World it was given

.

**return** x world = (x, world)

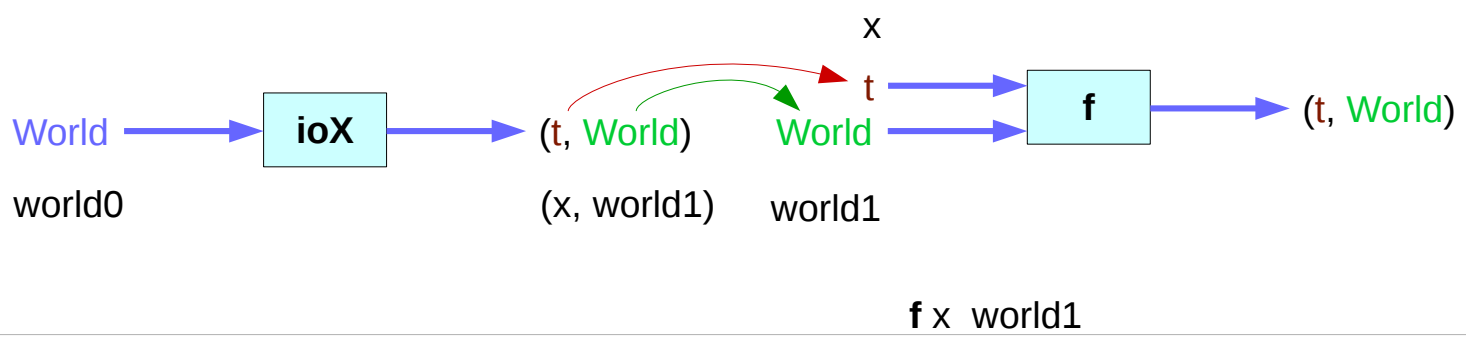



https://www.cs.hmc.edu/~adavidso/monads.pdf

# Side Effects in Haskell

the expression (**ioX >>= f**) has type World -> (t, World)
a function that takes a World, called world0,
which is used to extract x from its **IO** monad.
This gets passed to **f**, resulting in another **IO** monad,
which again is a function that takes a World
and returns a x and a new, updated World.
We give it the World we got back from getting x out of its monad,
and the thing it gives back to us is the t with a final version of the World
.

**the implementation of bind**




https://www.cs.hmc.edu/~adavidso/monads.pdf

## References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf
[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf