IO Monad (3D)

Young Won Lim
1/18/18

Copyright (c) 2016 - 2018 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

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

This document was produced by using LibreOffice/OpenOffice.

Based on

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

Pure Function

A **pure** function

- **no state**
- **no** access to **external states**
- **no side effects**

A **pure** function returns exactly the same result every time it's called with the same set of arguments.

Calling a **pure** function once is the same as calling it twice and discarding the result of the first call.

laziness

- **easily parallelized**
- **no data races**

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Actions

Haskell **runtime**

- first evaluates **main** (an expression)
  - not to a **simple value**
  - but to an **action**.
- then executes this **action**.

- the **program** itself has no side effects
- the **action** does have side effects

the functional nature of the **program** is maintained
(no side effects)

Evaluation
Execution

Program
Actions

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Simple IO

main calls functions like **putStrLn** or **print**,
which return **IO actions**.

there is only one non-trivial source of **IO actions**:

- **primitives** built into the language.
- **return** converts any **value** into an **IO action**.

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

IO Actions in main

IO action is invoked, after the Haskell **program** has run

- We can never execute an **IO action** inside the **program**
- once created, an **IO action** keeps percolating up
  until it ends up in **main** and is executed by the **runtime**.
- can also discard an **IO action**,
  but that means it will never be evaluated

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Laziness

Haskell will not calculate anything
unless it's strictly necessary
or is forced by the programmer

won't even evaluate arguments to a function before calling it.
assumes that the arguments will not be used by the function
procrastinates as long as possible.

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Laziness Example 1

Division by zero : **undefined -** never be evaluated.

```
main = print $ undefined + 1
```

the compiler doesn't complain

a runtime error resulting from an attempt to evaluate undefined.

```
foo x = 1
main = print $ (foo undefined) + 1
```

Haskell calls **foo** but never evaluates its argument **undefined**

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Laziness Example 2

not result from optimization: The compiler sees the definition of **foo** and figures out that **foo** and discards its argument.

But the result is the same

if the definition of **foo** is hidden from view in another module.

```
{-# START_FILE Foo.hs #-}
-- show
module Foo (foo) where
foo x = 1

{-# START_FILE Main.hs #-}
-- show
import Foo
main = print $ (foo undefined) + 1
```

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Laziness with infinity

laziness allows it to deal with

- **infinity** (like an infinite list)
- the **future** that hasn't materialized yet

Laziness or not, a program is going to be executed at some point.

why an expression would have to be evaluated --
there are several reasons
the fundamental one – somebody wants to display its result.

without I/O, nothing would ever be evaluated in Haskell.

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Do Notation

Larger IO actions are built from smaller IO actions.

- the **order** of composition
- **sequence** of **IO actions**.

special syntax for sequencing : the **do** notation.

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Do Notation Example

```
main = do
    putStrLn "The answer is: "
    print 43
```

sequencing two **IO actions**

- one **IO action** returned by **putStrLn**
- another **IO action** returned by **print**

inside a **do** block with proper **indentation**.

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Do Notation – input action (1)

whatever you receive from the user or from a file
you assign to a variable and use it later.

```
main = do
   str <- getLine
   putStrLn str
```

- not really a variable
- not really an assignment

when executed,
creates an **action** that will take the input from the user.
then pass this input to the rest of **actions** of the **do** block
under the name **str** when the rest is executed.

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Do Notation – input action (2)

```
str <- getLine
```

In Haskell you never assign to a variable,
instead you bind a name to a value.

it binds the name **str** to the value returned
by executing the **action** that was produced by **getLine**.

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Monadic Operation

the do block is used for

- **IO actions**
- **sequencing** a more general set of **monadic operations**

IO is just one example of a **monad**

a **monad** has an **imperative** feel.

A **monadic do** block

- really looks like chunks of **imperative** code.
- also behaves like **imperative** code

all **imperative** programming is at its core **monadic**.

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

Semicolon Overloading

The way the **actions** are glued together
is the essence of the **Monad**.

Since the glueing happens between the lines,
the **Monad** is sometimes described as
an "**overloading of the semicolon**."

Different **monads** overload it differently.

https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell/3-pure-functions-laziness-io

IO Monad

the type signature **IO a** looks remarkably similar to **Maybe a**.

- **IO** doesn't expose its constructors
- only be "run" by the Haskell runtime system
- a **Functor**
- a **Monad**

https://stackoverflow.com/questions/18808258/what-does-the-just-syntax-mean-in-haskell

IO Monad

Haskell separates **pure functions** from **computations** where **side effects** must be considered

by encoding those **side effects**

as **values** of a particular type (**IO a**)

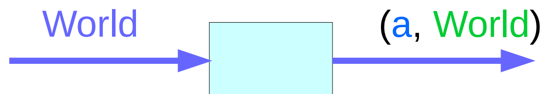
Specifically, a value of type (**IO a**) is an **action**,

which if executed would produce a **value** of type **a**.

Execution → Value

IO a

a type of action



https://wiki.haskell.org/Introduction_to_IO

Functions 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** is to run in **main**

https://wiki.haskell.org/Introduction_to_IO

Functions 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

>> of IO Monad

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

(x >> y)

- if **x** and **y** are IO actions
- then it is the action that first performs **x**
- dropping the result
- then performs **y**
- returns its result.

```
putStrLn "Hello" >> putStrLn "World"
         IO ()   ->          IO ()   ->    IO ()
```

https://wiki.haskell.org/Introduction_to_IO

>>= of IO Monad

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

(x >>= f)

- to use the result of the first action (**x**)
- in order to affect what the second action will do

- the action that first performs the action x
- and captures its result
- passing it to **f**
- then **f** computes a second action to be performed.
- this second action is then carried out,
- its result is the result of the overall computation.

```
x >> y = x >>= const y
```

```
getLine   >>= \name -> putStrLn ("Hello, " ++ name ++ "!")
  IO a     ->     (a -> IO b)                          -> IO b
```

https://wiki.haskell.org/Introduction_to_IO

randomRIO

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

takes a range **(lo,hi)** and a random number generator **g**,
and returns a random value uniformly distributed
in the closed interval **[lo,hi]**, together with a new generator.

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

A variant of **randomR** that uses the global random number generator

See System.Random

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

RandomRIO Example

```haskell
import System.Random

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

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

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

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

Bind operator >>= and do Block

```
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

Return IO Bool

```
getChar   ::  IO Char

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

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

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

```
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

Each do, a single 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

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

absolutely cannot do any I/O

since no **IO a** 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

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

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.

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

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.

```
[ putChar ‘a’,  putChar ‘b’,  putChar ‘c’ ]
```

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

putStr via putChar

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

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

[putChar ‘a’, putChar ‘b’, putChar ‘c’]

The **foldr** operation in **sequence_** uses the **>>** function
to combine all of the individual actions into a single action.
The **return** () used here is quite necessary –
foldr needs a null action at the end of the chain of actions
(especially if there are no characters in the string!).

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 <u>cannot</u> perform side effects in Haskell.
there is one exception: **IO monad**

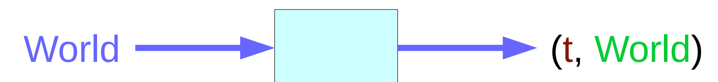
Suppose there is a type called World,
which contains all the state of the external universe

A way of thinking what IO monad does

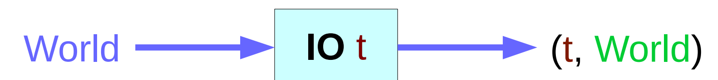
```
type    IO t   =   World   ->   (t, World)
```

type synonym

World -> (t, World)



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.

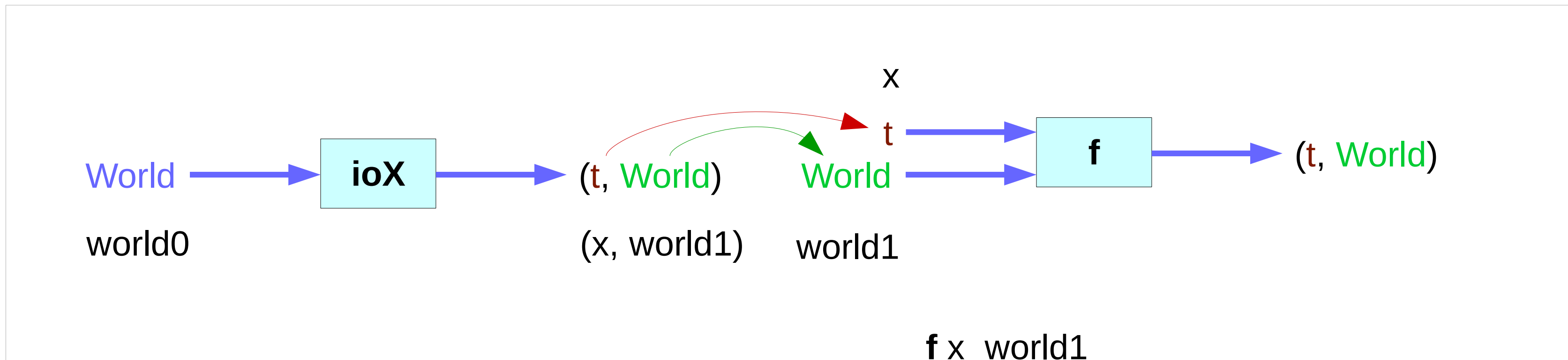


IO x world0 (x, world1)

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)
```



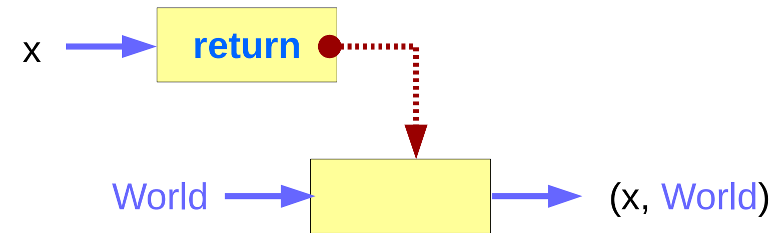
https://www.cs.hmc.edu/~adavidso/monads.pdf

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)

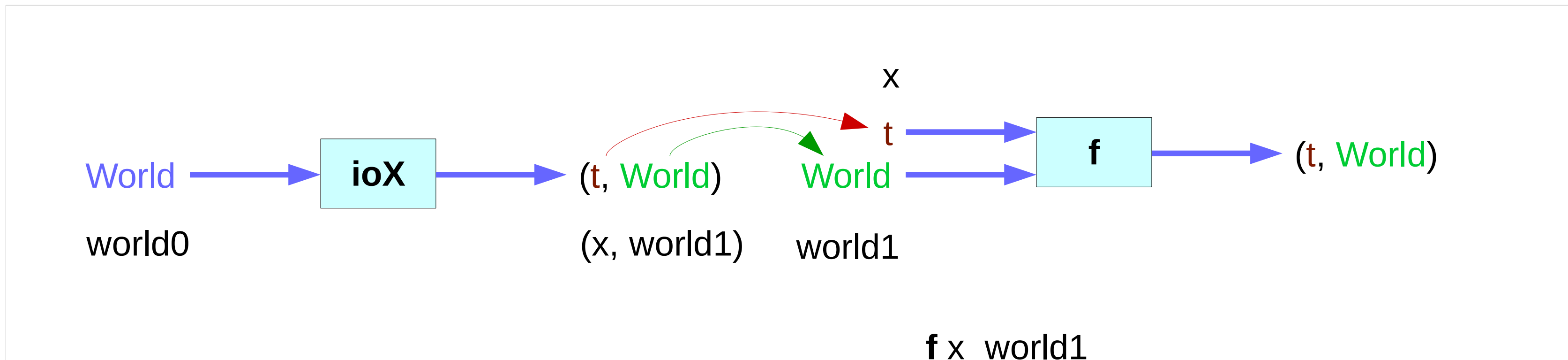


https://www.cs.hmc.edu/~adavidso/monads.pdf

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