Monad Overview (3B)

Young Won Lim
1/13/18

Based on

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

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

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

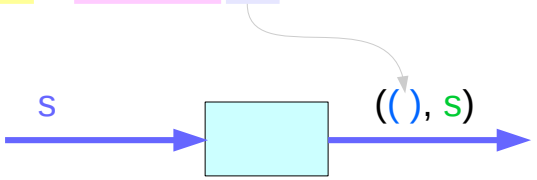
https://www.cs.hmc.edu/~adavidso/monads.pdf

Monadic Operations

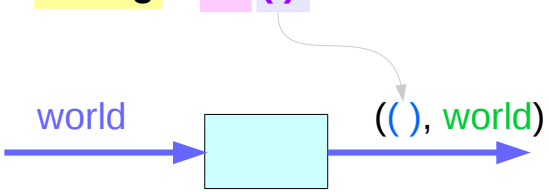
Monadic operations tend to have types which look like

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

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



putStr :: String -> IO ( )



returning a function as a value
executable function
executing an action (**effect-monad**)
produce a result val-out-type

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

Monadic Operations

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

where the **return type** is a type application:

a type with a parameter type

effect-monad

an executable function

giving information about which **effects** are possible

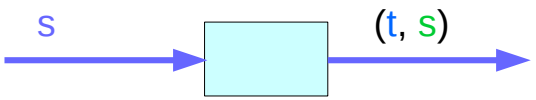
val-out-type

the argument of the executable function

the type of the **result** produced by the function

(the result of executing the function)

returning a function as a value



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

putStr :: String -> IO ( )

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

Monadic Operations – **put**, **putStr**

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 ( )

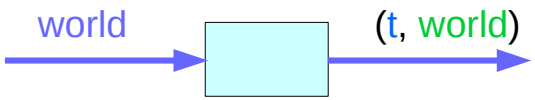
delivers a string to stdout but does not return anything exciting.

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

IO t and State s a types

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

type synonym

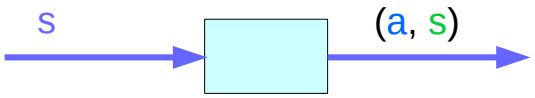



```
newtype State s a = State { runState :: s -> (a, s) }
```

s : the type of the state,

a : the type of the produced result

s -> (a, s) : function type



Monad Definition

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

https://en.wikibooks.org/wiki/Haskell/Understanding_monads

Maybe Monad Instance

```
instance Monad Maybe where
    return x = Just x
    Nothing >>= f = Nothing
    Just x >>= f  = f x
    fail _ = Nothing
```

https://en.wikibooks.org/wiki/Haskell/Understanding_monads

IO Monad Instance

```
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
```

https://stackoverflow.com/questions/9244538/what-are-the-definitions-for-and-return-for-the-io-monad

State Monad Instance

```
instance Monad (State s) where

return :: a -> State s a
return x = state ( \ s -> (x, s) )

(>>=) :: State s a -> (a -> State s b) -> State s b
p >>= k = q where
    p' = runState p        -- p' :: s -> (a, s)
    k' = runState . k      -- k' :: a -> s -> (b, s)
    q' s0 = (y, s2) where  -- q' :: s -> (b, s)
        (x, s1) = p' s0    -- (x, s1) :: (a, s)
        (y, s2) = k' x s1  -- (y, s2) :: (b, s)
    q = State q'
```

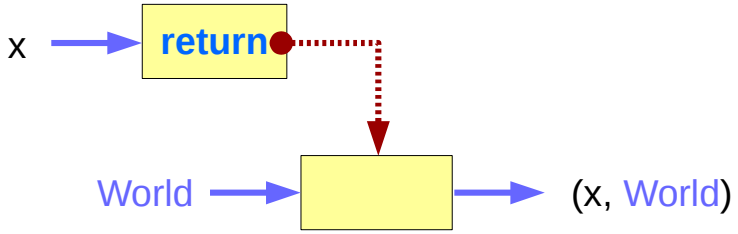
https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

Monad IO - return

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

.

return x world = (x, world)



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

Monad IO - >>=

the expression (**ioX >>= f**) has

type World -> (t, World)

a function **ioX** that takes w0 of the type World,

which is used to extract x from its **IO** monad.

x gets passed to **f**, resulting in another **IO** monad,

which again is a function that takes w1 of the type World

and returns a y and a new, updated World.

the implementation of bind

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

IO Monad >>= Implementation

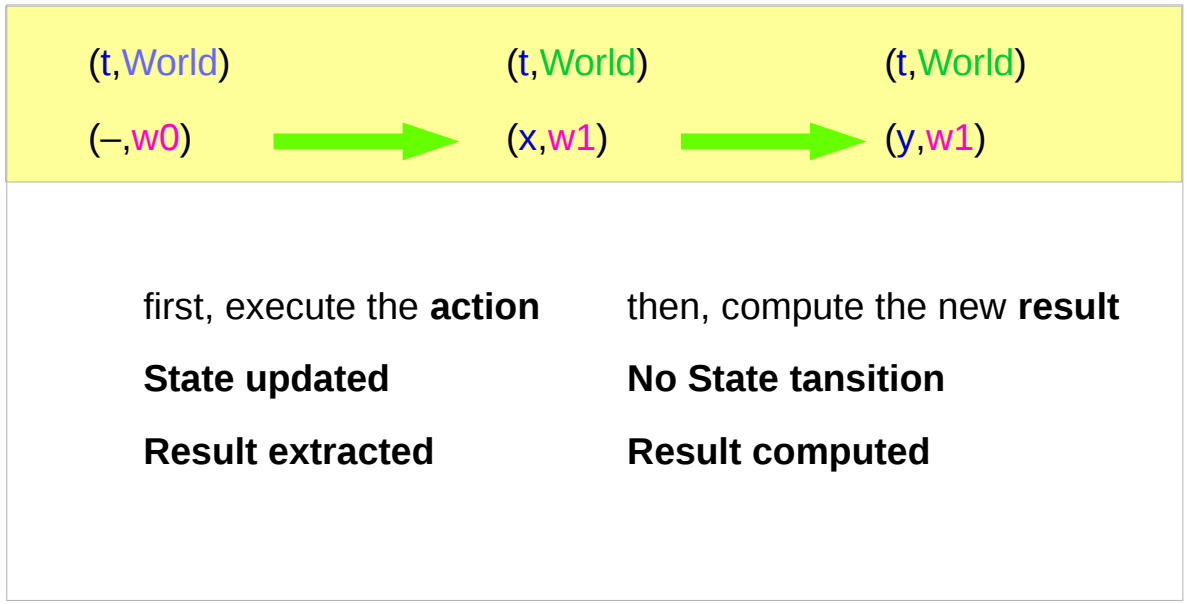
We give **IO** the World	w0 :: World
we got back the World	w1 :: World
from getting x out of its monad,	x :: t
and the thing **IO** gives back to us is	
the y with	y :: t
a final version of the World	w1 :: World

.

the implementation of bind

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

IO Monad >>= Implementation



the implementation of bind

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

IO Monad Implementation

```
instance Monad IO where
   return x w0 = (x, w0)

   (ioX >>= f) w0  =
       let  (x, w1) = ioX w0
       in   f x w1          -- has type (t, World)
```

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

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

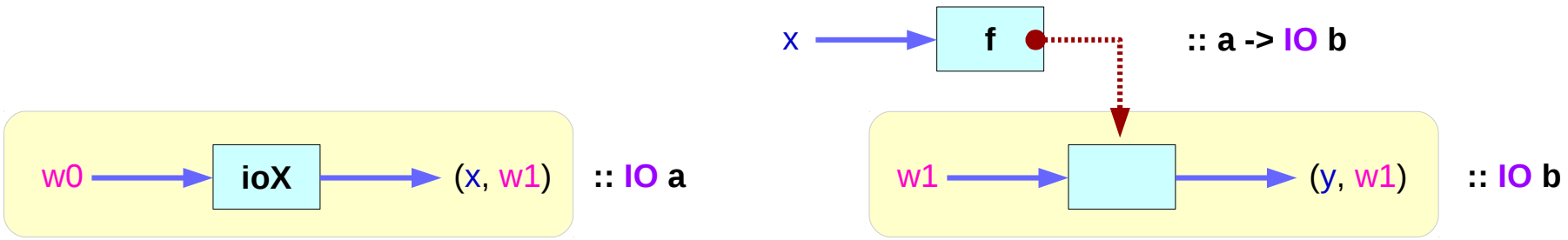
IO Monad Implementation

```
instance Monad IO where
   return x w0 = (x, w0)

   (ioX >>= f) w0  =
      let  (x, w1) = ioX w0
      in   f x w1          -- has type (t, World)
```

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

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

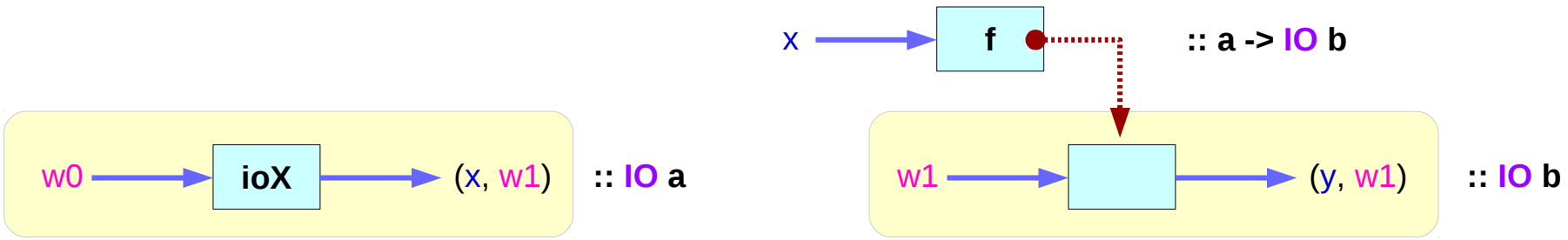
type synonym



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

IO Monad Implementation

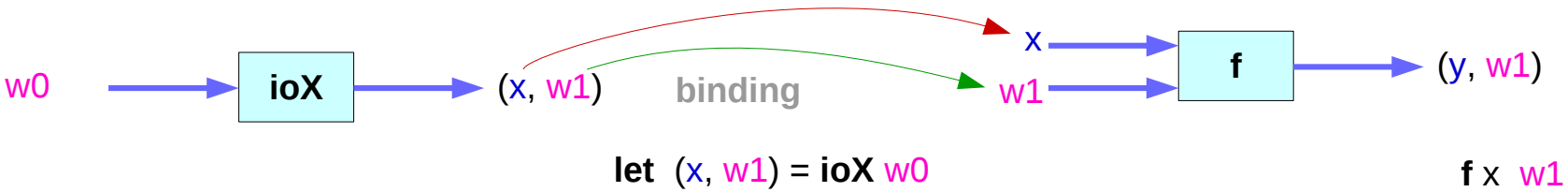
```
ioX >>= f   ::      IO a -> (a -> IO b) -> IO b

ioX :: IO a         w0 :: World       x :: a
f :: a -> IO b      w1 :: World

      ioX w0 :: IO a World          (x, w1)
      f x ::  IO b
      f x w1 :: IO b World          (y, w1)
```



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

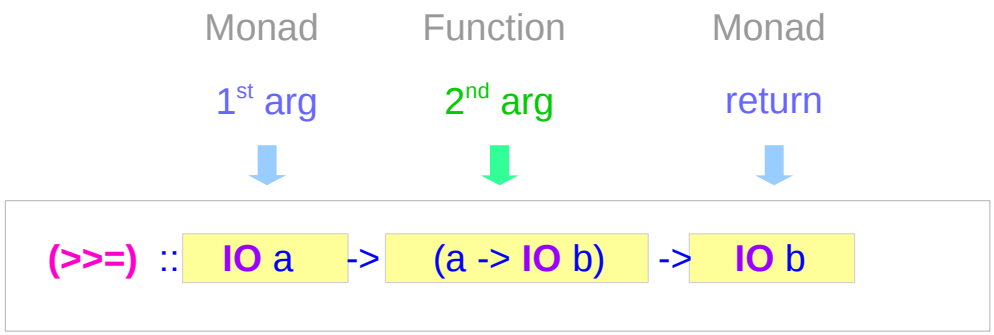
IO Monad Implementation

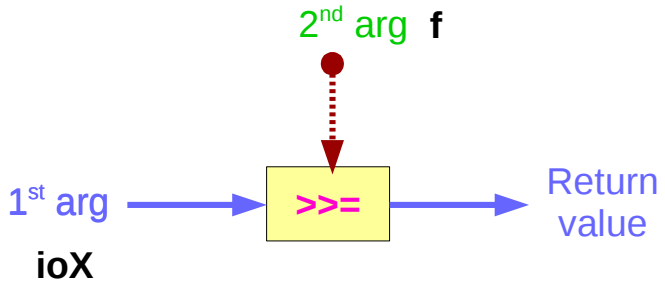
```
ioX >>= f   ::      IO a -> (a -> IO b) -> IO b

ioX :: IO a
f :: a -> IO b      w0 :: World      x :: a
                                     w1 :: World      internal variables

      ioX w0 :: IO a World      ➡      (x, w1)
      f x :: a -> a -> IO b
      f x w1 :: IO b World      ➡      (y, w1)
```



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

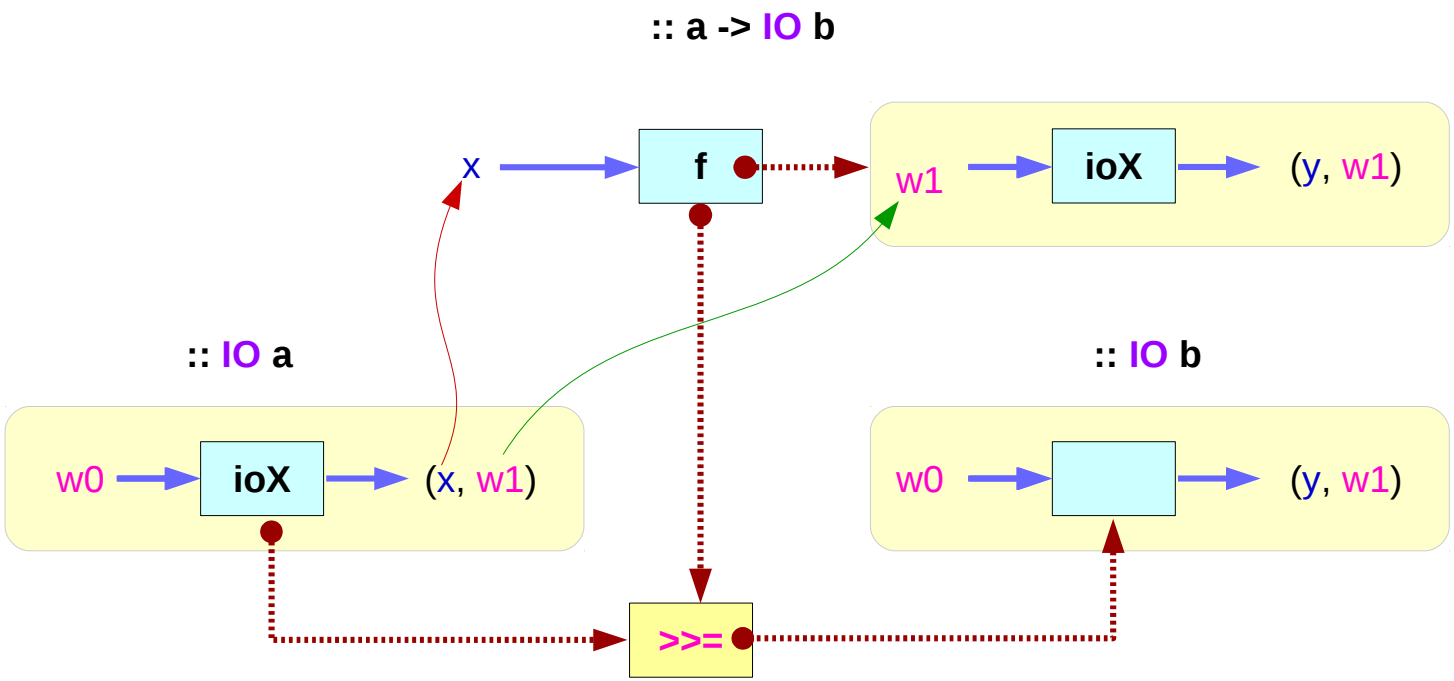
IO Monad






```
(ioX >>= f) w0  =
    let  (x, w1) = ioX w0
    in   f x w1          -- has type (t, World)
```

IO Monad



Monad IO and Monad ST

```
instance Monad IO where
   return x world = (x, world)

   (ioX >>= f) w0  =
       let    (x, w1) = ioX w0
       in     f x w1                -- has type (t, World)
```

```
instance Monad ST where
   -- return :: a -> ST a
   return x  =  \s -> (x,s)

   -- (>>=)  :: ST a -> (a -> ST b) -> ST b
   st >>= f  =  \s -> let (x,s') = st s
                      in f x s'
```

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

type synonym

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

State Transformers ST

```
instance Monad ST where
  -- return :: a -> ST a
  return x  =  \s -> (x,s)

  -- (>>=)  :: ST a -> (a -> ST b) -> ST b
  st >>= f  =  \s -> let (x,s') = st s in f x s'
```

>>= provides a means of sequencing state transformers:
st >>= f applies the state transformer **st** to an initial state s,
then applies the function **f** to the resulting value x
to give a second state transformer (**f** x),
which is then applied to the modified state s' to give the final result:

```
st >>= f  =  \s -> f x s'
        where (x,s') = st s

st >>= f  =  \s -> (y,s')
        where (x,s') = st s
              (y,s') = f x s'
```

```
(x,s') = st s

f x s'
```

https://cseweb.ucsd.edu/classes/wi13/cse230-a/lectures/monads2.html

References

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