Monad P3 : IO Monad Basics (2A)

Young Won Lim
9/20/19

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

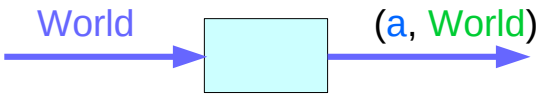
IO Monad

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 **result value** of type **a**.

Execution ➡ Value (result)

IO a — **a type of an action**



https://wiki.haskell.org/Introduction_to_IO

Computations that result in values

Monads like **IO**

map types **t** to a new type **IO t**

that represent "computations that result in values"

a **function** type: **World -> (t, World)**

the **result** type : **t**

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

```
RealWorld -> (a, RealWorld)
```

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

https://wiki.haskell.org/Maybe

Type Synonym **IO t**

IO t is a **parameterized function type**

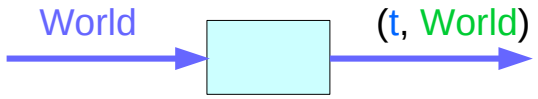
input : a World

output: a result value of the type t and a new updated World
are obtained by modifying the given World
in the process of computing the result value of the type t.

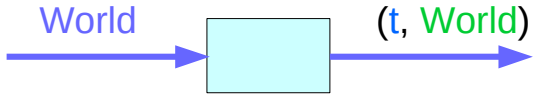
```
type  IO t  =  World  ->  (t, World)
```

type synonym

World -> (t, World)



IO t

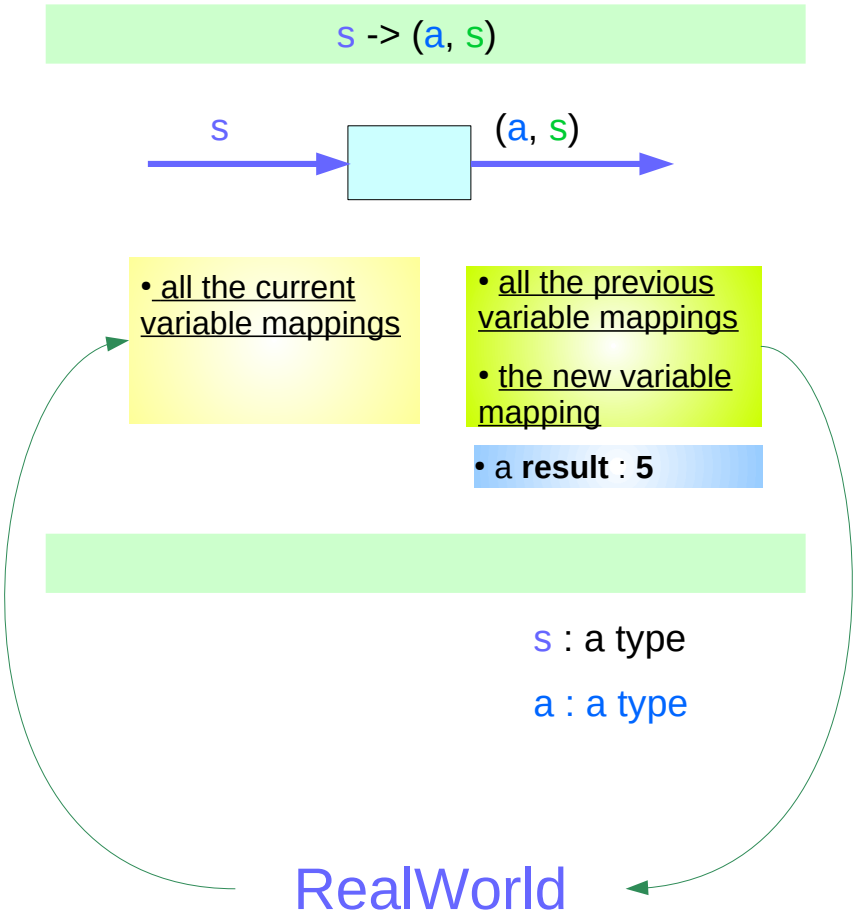


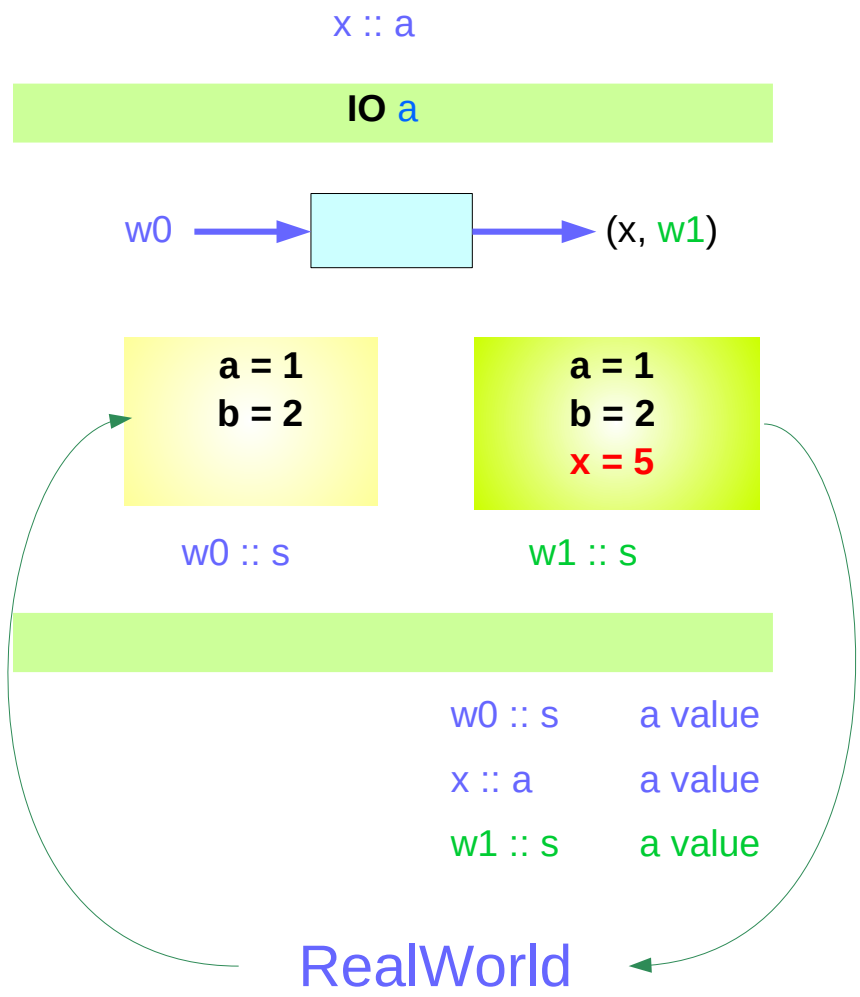
cf) type application

RealWorld

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

Variable Mappings : Context





http://learnyouahaskell.com/for-a-few-monads-more

The value of type IO a

assume the value of type (IO **a**) is **func**

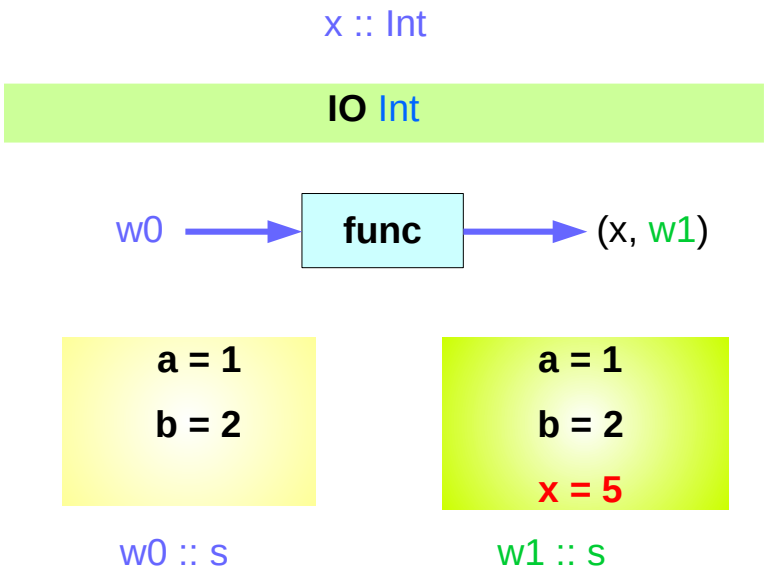
representing an action, which would produce
a **result** value of type **a**, *if executed*

func w0 returns the tuple (**x**, **w1**)
x is the a **result** value of type **a**

When IO **a** is defined as

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

The value **func** of type (IO **a**) is the **name**
of the underlying (state) function



func :: IO a
func :: IO Int

m :: IO a
m :: IO Int

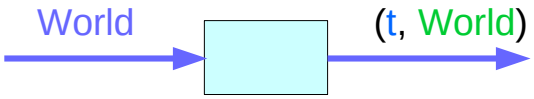
monadic value

IO t is a function type not a function value

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

t ➡ IO t lifted type

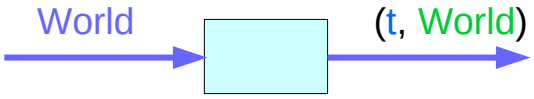
World -> (t, World)

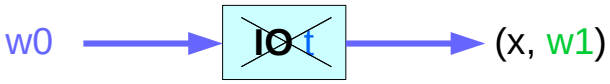


let (x, w1) = IO t w0

IO t type view

value view





x :: t

w1 :: World

(x, w1) :: (t, World)

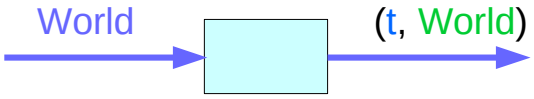
(x, w1) :: IO t World

(t, World) – the return type of the function

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

World -> (t, World)

IO t



func :: **IO** t

func is a monadic value of the type **IO t**, then

func is also the name of the underlying function

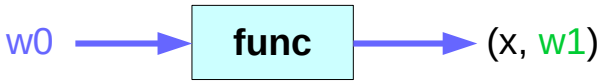
let (x, w1) = **func** w0

x :: t

w1 :: World

(x, w1) :: (t, World)

(x, w1) :: ~~**IO** t World~~

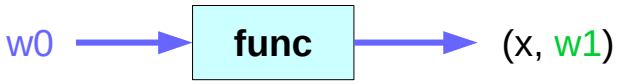


func w0 :: (t, World)

func :: IO t

func :: IO Int type

type IO Int = World -> (Int, World)



(x, w1) = func w0

func	**:: IO** Int	the function type
func w0	**:: IO** ~~Int World~~	the function **return** type

func :: World -> (Int, World)
func w0 **:: (Int, World)**

x	:: Int	the **result** type
w0	:: World	the function **input type – initial state type**

let (x, w1) = **func** w0 the bindings of x and w1

http://learnyouahaskell.com/for-a-few-monads-more

Parameterized type **IO a**

```
type   IO a      =   s   ->   (a, s)
```

```
type   IO Int    =   s   ->   (Int, s)
type   IO Char   =   s   ->   (Char, s)
         ⋮                     ⋮
```

s ← Int

```
type   IO Int    =   Int ->   (Int, Int)
```

func :: IO Int

in practice the type **s** is RealWorld

RealWorld -> (a, RealWorld)

http://learnyouahaskell.com/for-a-few-monads-more

IO Monad Instance (1)

```
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 Instance (2)

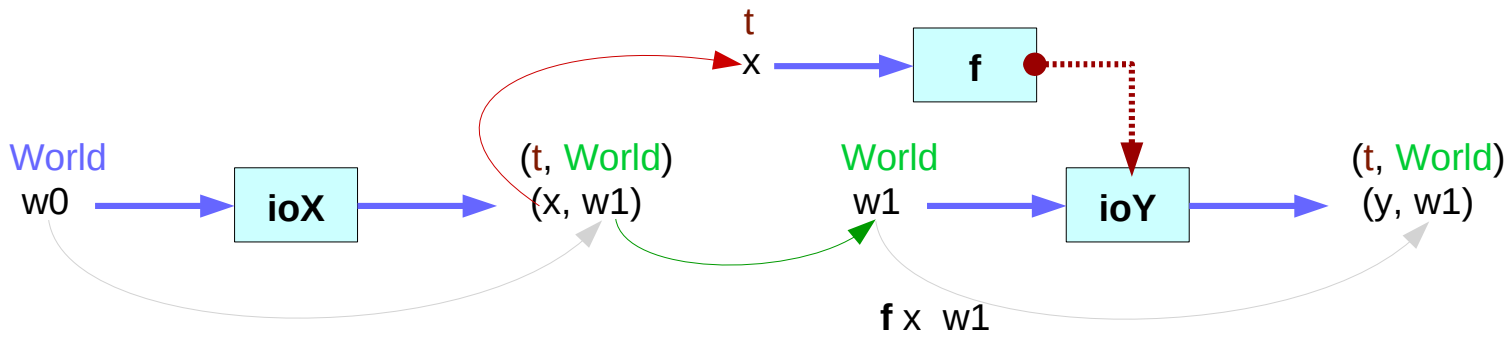
```
instance Monad IO where
   return x w0 = (x, w0)

   (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

ioX :: IO a

f :: (a -> IO b)



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

return

The **return** function takes **x**

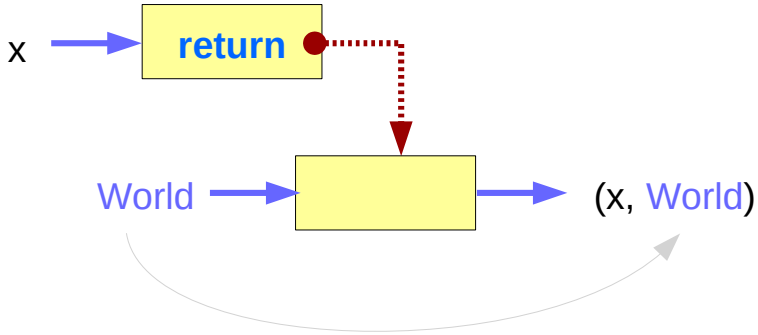
and gives back a function

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)

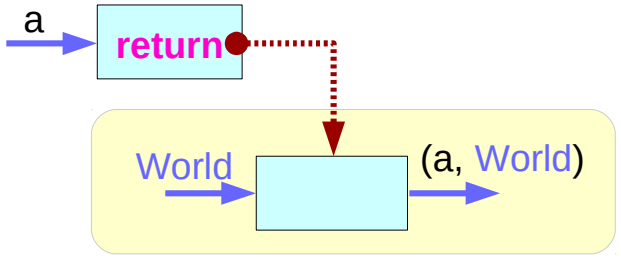


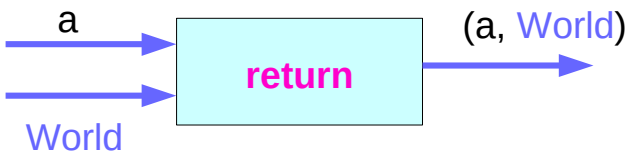
.

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

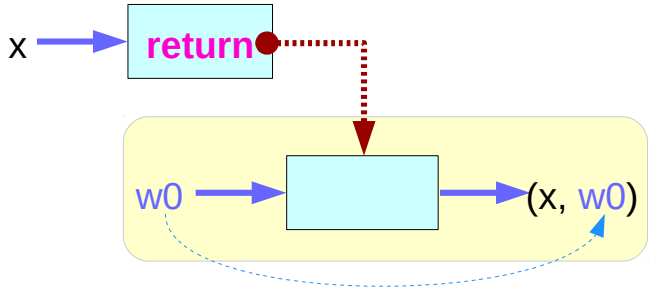
return method and partial application

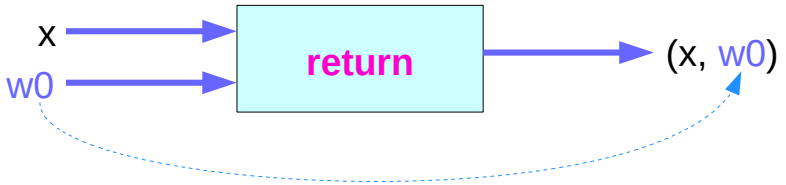
return a :: a -> IO a ⬅ **Types** ➡ **return a World :: (a, World)**





let (x, w0) = return x w0 ⬅ **Values** ➡ **let (x, w0) = return x w0**





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

iox >>= f

the expression (**ioX >>= f**) has type World -> (t, World)

a function that takes a World, called w0,

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

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

which again is a function that takes a World

and returns a **y** and a new, updated World.

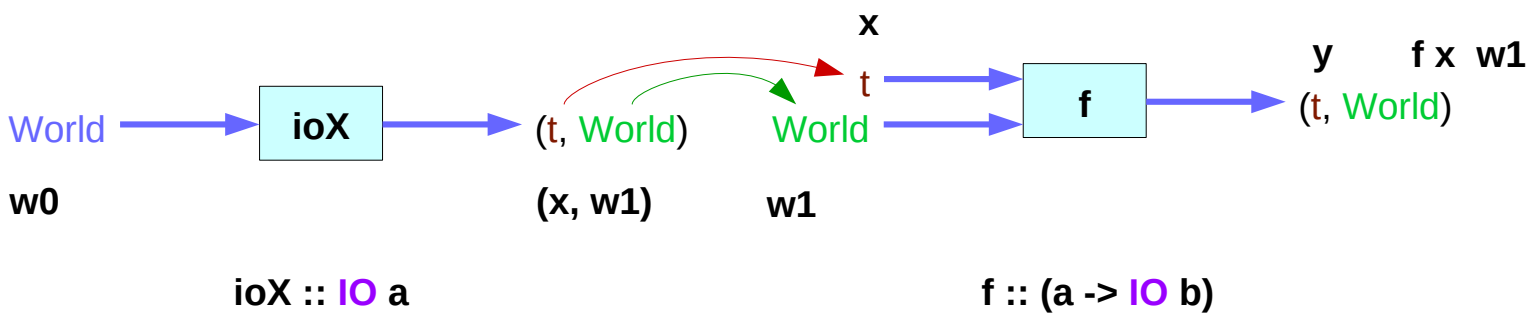
We give it the World we got back from getting **x** out of its monad,

and the thing it gives back to us is the **y** with a final version of the World

.

the implementation of bind

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



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

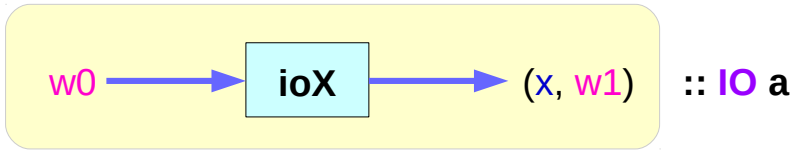
ioX and f types

```
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)
```

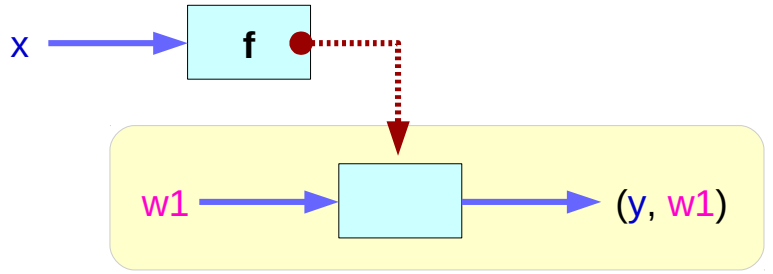
ioX >>= f :: IO a -> (a -> IO b) -> IO b

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



:: IO a

ioX :: IO a



:: IO b

f :: (a -> IO b)

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

ioX w0 and f x w1

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

ioX :: IO a w0 **::** World x **:: a**

f :: a -> IO b w1 **::** World

ioX w0 **::** (a**,** World) ➡ (x, w1) **::** (a**,** World)

f x :: IO b

f x w1 **::** (b**,** World) ➡ (y, w1) **::** (b**,** World)

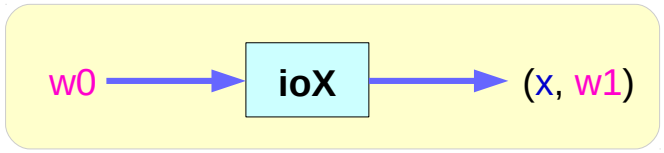
f :: a -> IO b

f x :: IO b

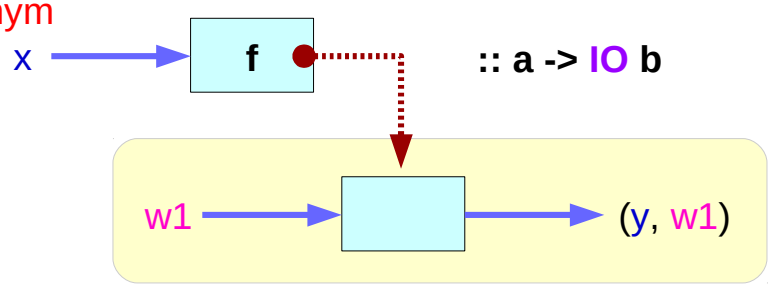
f :: a -> World -> (b World)

f x w1 **::** (**b** World**)**

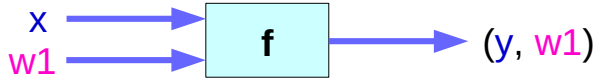
type IO t = World -> (**t**, World) type synonym



ioX w0



f x w1



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

Binding variables

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

ioX :: IO a
f :: a -> IO b

w0 :: World

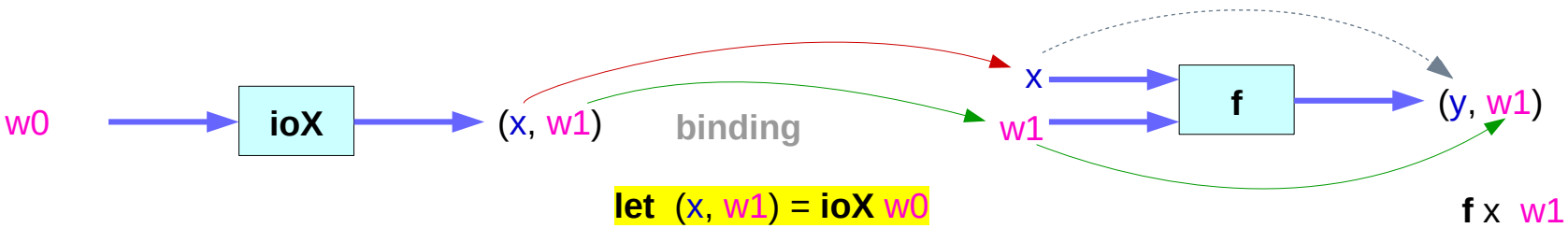
x :: a
w1 :: World

internal variables

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

ioX – monadic value

f – monad returning function



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

Steps of **ioX >>= f** (1. state update, 2. result)

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

type **IO a -> (a -> IO b) -> IO b**

first, execute the **action**

execute **ioX**

State updated

w0 → **w1**

result extracted

x is the **result**

then, compute the new **result**

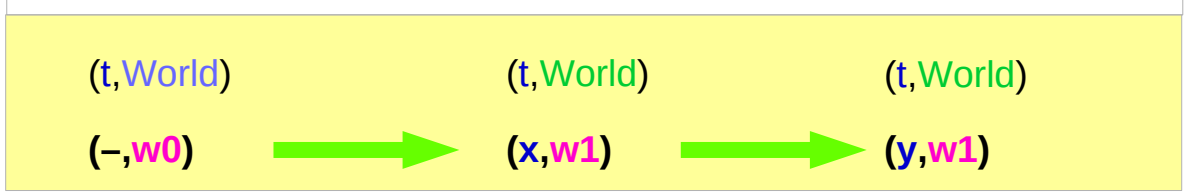
using **f x**

no **State transition**

w1 is remained

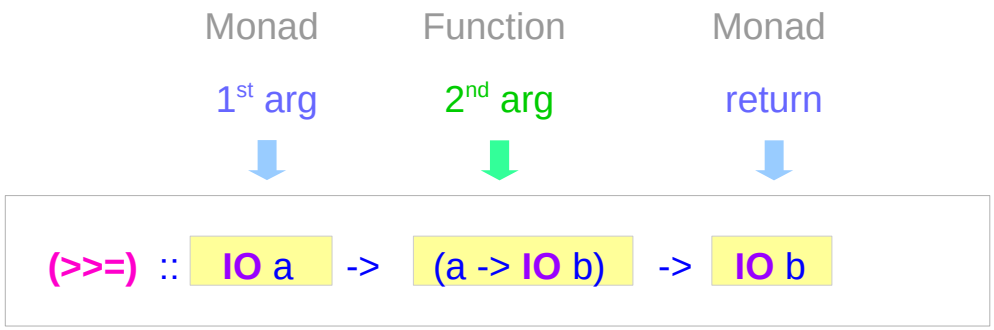
result computed **y = (f x)**

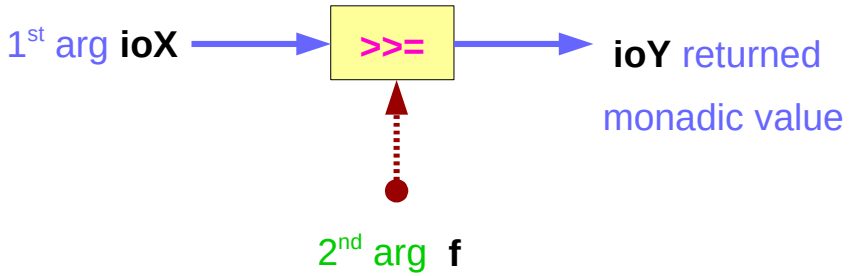
the implementation of bind



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

(>>=) operator type signature

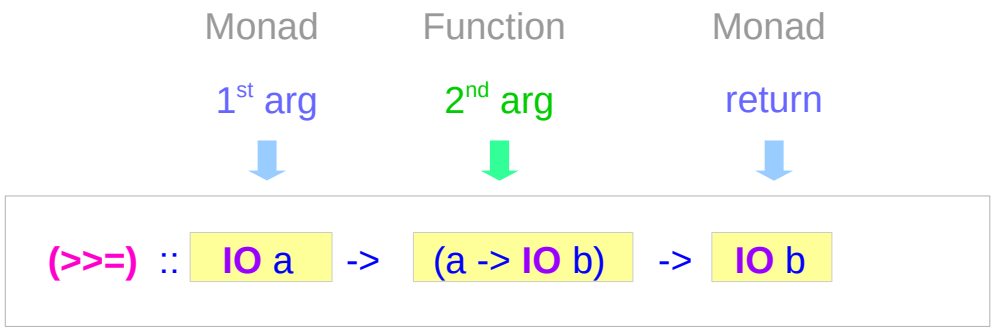


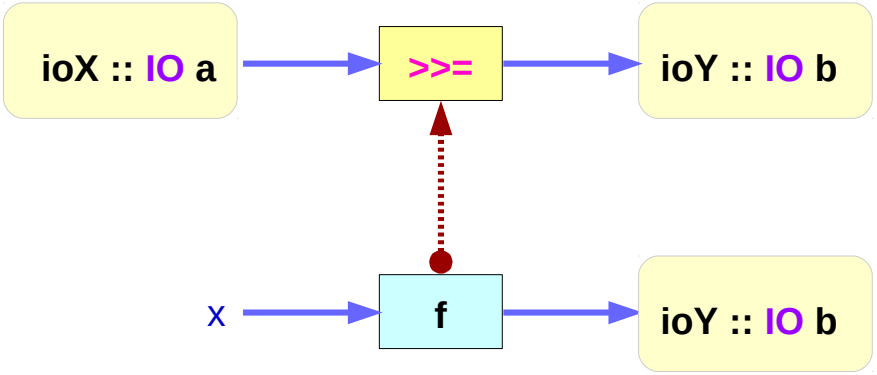


ioX :: IO a

ioY :: IO b

(>>=) operator type diagram

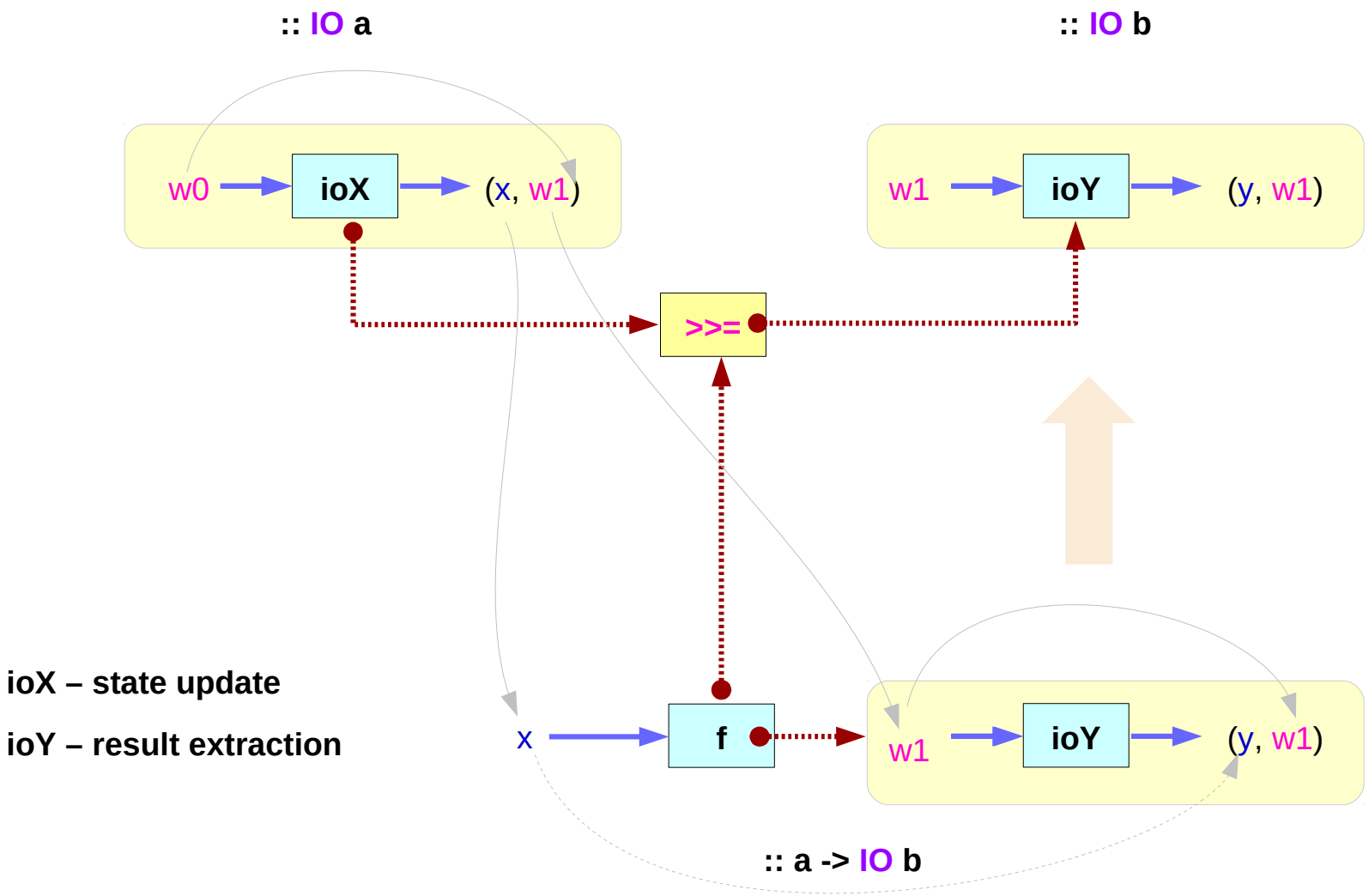




IoX – state update

IoY – result extraction

(>>=) operator threads

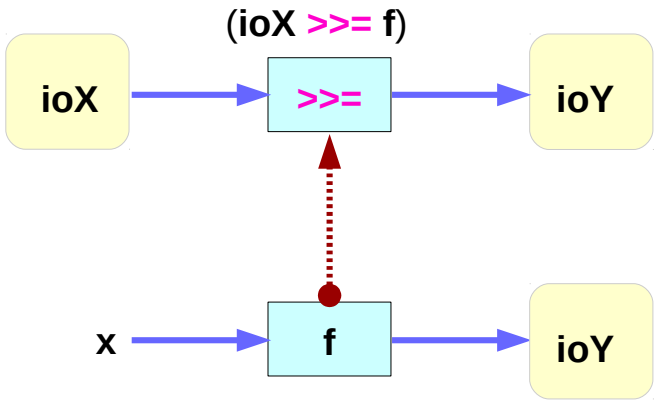


(>>=) operator summary

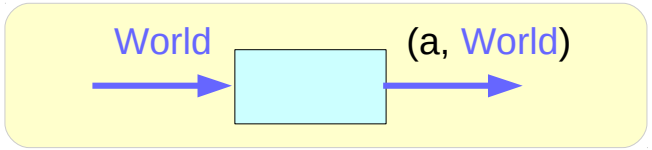
the expression (**ioX >>= f**) has
type **IO a -> (a -> IO b) -> IO b**

ioX :: **IO a** has a **function** type of **World -> (a, World)**
a **function** that takes **w0 :: World**,
returns **x :: a** and the new, updated **w1 :: World**

x and **w1** get passed to **f**, resulting in another **IO** monad,
which again is a **function** that takes **w1 :: World**
and returns **y** computed from **x** and the same **w1 :: World**



ioX :: IO a **f :: a -> IO a** **ioY :: IO b**

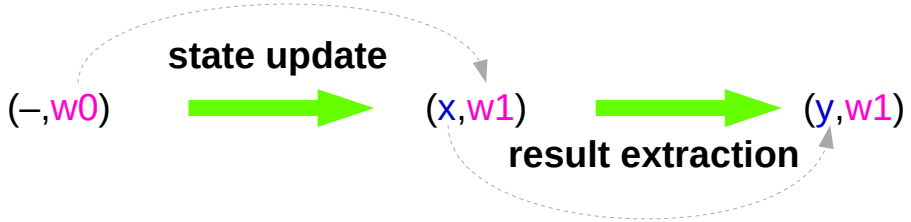


ioX :: IO a

ioY :: IO b

ioX – state update

ioY – result extraction



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

(>>=) operator binding

We give the **IOx** the **w0** — **w0 :: World**

we got back the updated **w1** — **w1 :: World**

and **x** out of its monad — **x :: a**

the **f** is given with

the **x** with — **x :: a**

the updated **w1** — **w1 :: World**

.

The final **IO** Monad

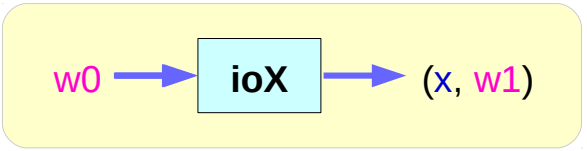
takes **w1** — **w1 :: World**

returns **w1** — **w1 :: World**

and **y** out of its monad — **y :: a**

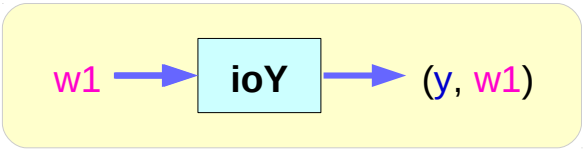
the expression (**ioX >>= f**) has

type **IO a -> (a -> IO b) -> IO b**



let (x, w1) = ioX w0

bind variables



let (y, w1) = ioY w0

bind variables

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

Implementation of IO t

It is impossible

to store the extra copies of the contents of your hard drive

that each of the Worlds contains

given World → updated World

type IO a = RealWorld -> (a, RealWorld)

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

IO Monad in GHC

Which World was given initially?

Which World was updated?

In **GHC**, a **main** must be defined somewhere with type **IO ()**

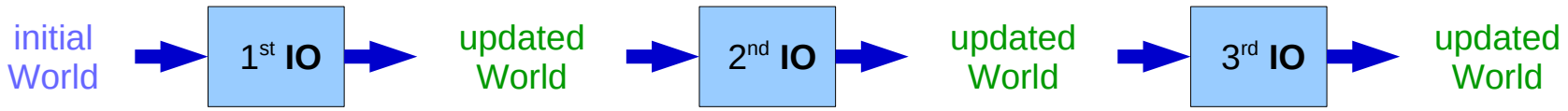
a program execution starts from the **main**

- the initial World is contained in the **main** to start everything off
- the **main** passes the updated World from each **IO**
- to the next **IO** as its initial World

an **IO** that is not reachable from **main** will never be executed

an initial / updated World is not passed to such an **IO**

The modification of the World



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

IO Monad in GHCI

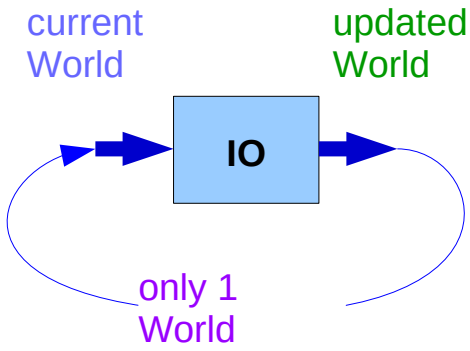
when using **GHCI**,
everything is wrapped in **an implicit IO**,
since the results get printed out to the screen.

there's only 1 World in existence at any given moment.

each **IO** takes that one and only World, consumes it,
and gives back a single new updated World.

consequently, there's no way to accidentally run out of Worlds,
or have multiple ones running around.

the implementation of bind



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

GHCI

Every time a new **command** is given to **GHCI**,

GHCI passes the current World to **IO**,

GHCI gets the *result* of the command back,

GHCI request to display the *result* (**executing actions**)

(which updates the World by modifying

- the contents of the screen or
- the list of defined variables or
- the list of loaded modules or whatever),

GHCI saves the new World to process the next command.

the implementation of bind

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

IO Monad

```
type    IO a  =   s  ->  (a, s)

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

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

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

Threading the state

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


do x <- doSomething
   y <- doSomethingElse
   return (x + y)

\s ->
let (x, s')  = doSomething s
   (y, s'') = doSomethingElse s' in
(x + y, s'')
```

creating data dependecies

s → s' → s''

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
   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/

realWorld# – a reference to the real world

realWorld# is a value of type **State# RealWorld**
which is a **token** that acts as a **reference** to **the real world**.
(it is of **size** 0 and does not occupy any space
on the **stack** or **heap**.)

State# RealWorld values represent
the entire external **runtime state** of the program.
The "real world", as it were.

The **main** value in your program
receives a **State# RealWorld** **value**
that is threaded through the **IO actions** that compose it.

https://stackoverflow.com/questions/32672814/where-is-the-realworld-defined

State# RealWorld

The **primitive State# RealWorld**

RealWorld corresponds to the **s** parameter of our **State** monad

Actually, it's two **primitives**, the **type constructor State#**,
and the magic type **RealWorld** which doesn't have a **# suffix**

This is because **ST** monad also uses
a **type constructor** and a **type parameter** framework

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

RealWorld value

You can treat **State# RealWorld** as a **type**
that represents a very **magical value**:
the value of the entire real world.

only the **main** function can receive a **real world value**,
and it then gets threaded through **sequence** of **IO actions**

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
```

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

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

References

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

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