Monad Overview (3B)

Young Won Lim
2/17/18

Based on

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

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.

Based on

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

Monad, Monoid

monad (plural monads)

- An ultimate atom, or simple, unextended point; something ultimate and indivisible.
- (mathematics, computing) A monoid in the category of endofunctors.
- (botany) A single individual (such as a pollen grain) that is free from others, not united in a group.

monoid (plural monoids)

- (mathematics) A **set** which is closed under an associative binary operation, and which contains an element which is an identity for the operation.

class Monad **m** where ...

m a

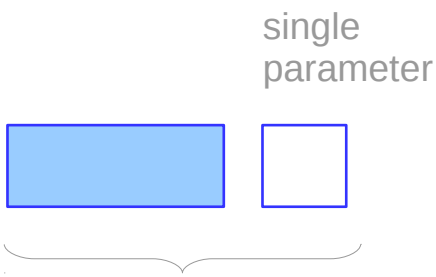
m b

instance Monad **Maybe** where ...

m a

m b

Maybe a



https://en.wiktionary.org/wiki/monad, monoid

Maybe Monad

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



instance Monad Maybe where
   -- return      :: a -> Maybe a
   return x       =  Just x

   -- (>>=)       :: Maybe a -> (a -> Maybe b) -> Maybe b
   Nothing   >>= _ =  Nothing
   (Just x)   >>= f =  f x
```

```
f :: a -> m b
```

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

Maybe Monad

a monad is a parameterized type **m**

that supports **return** and **>>=** functions of the specified types

m must be a parameterized type,
rather than just a type (**Maybe** is not a concrete type)

It is because of this declaration

(Just x) **>>=** f = f x

that the **do** notation can be used to sequence **Maybe** values.

More generally, Haskell supports the use of this notation
with any monadic type.

examples of types that are monadic,
the benefits that result from recognizing and exploiting this fact.

class Monad **m** where ...

m a

m b

instance Monad **Maybe** where ...

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

List Monad

The **Maybe** monad provides a simple model of computations
that can fail,
a value of type Maybe a is either **Nothing** (failure)
the form Just x for some x of type a (success)

The **list** monad generalises this notion,
by permitting multiple results in the case of success.

More precisely, a value of [a] is
either the empty list **[ ]** (failure)
or the form of a non-empty list [x1,x2,...,xn] (success)
for some xi of type a

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

List Monad

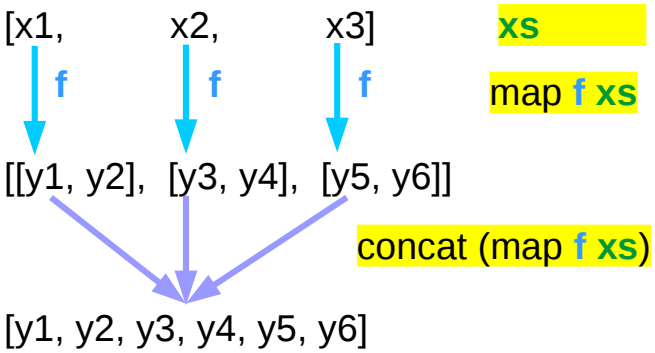
```
instance Monad [] where
  -- return :: a -> [a]
  return x  =  [x]

  -- (>>=)  :: [a] -> (a -> [b]) -> [b]
  xs >>= f  =  concat (map f xs)
```

return converts a value into a *successful* result containing that value

>>= provides a means of *sequencing* computations
that may produce *multiple results*:

xs >>= **f** applies the function **f** to each of the *results* in the list **xs**
to give a *nested list* of *results*,
which is then concatenated to give a *single list* of *results*.

xs :: [a]
f :: a -> [b]
(>>=) :: [a] -> (a -> [b]) -> [b]



(Aside: in this context, [] denotes the list type [a] without its parameter.)

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

A Type **Monad**

Haskell does not have states

but it's type system is powerful enough
to construct the stateful program flow

defining a **Monad** type in Haskell
- similar to defining a **class** in an object oriented language (C++, Java)
- a **Monad** can do much more than a class:

A **Monad** is a **type** that can be used for

- **exception handling**
- **parallel program workflow**
- **a parser generator**

http://www.idryman.org/blog/2014/01/23/yet-another-monad-tutorial/

Types: rules and data

Haskell **types** are the ***rules*** associated with the **data**,
not the actual **data** itself.

Object-Oriented Programming enable us
- to use **classes / interfaces**
  - to define **types**,
  - the **rules** (**methods**) that interacts with the actual **data**. **Rules + Data**

- to use **templates**(c++) or **generics**(java)
  - to define more **abstracted rules** that are more reusable **Rules**

Monad is pretty much like **templates / generic class**.

http://www.idryman.org/blog/2014/01/23/yet-another-monad-tutorial/

Monad Rules

A **type** is just **a set of rules**, or **methods** in Object-Oriented terms

A **Monad** is just yet another type, and
the definition of this type is defined by **four rules**:

1) **bind (>>=)**
2) **then (>>)**
3) **return**
4) **fail**

Rules

http://www.idryman.org/blog/2014/01/23/yet-another-monad-tutorial/

Monad Applications

1. **Exception Handling**

2. **Accumulate States**

3. **IO Monad**

http://www.idryman.org/blog/2014/01/23/yet-another-monad-tutorial/

A value of type **M a**

a **value** of type **M a** is interpreted **mx** :: **M a**

as a **statement** in an imperative language **M**
that returns a value of type **a** as its result;

and the **semantics** of this language
are determined by the **monad M**.

Maybe a
IO a
ST a
State s a

the type of result **a** : **IO** a

the type **M a**

an imperative language **M**

a value **mx**

statements returning a type **a** value

execution, a function, a return value

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

A value of type **M a**

a **value** of type **M a** is interpreted

as a **statement** in an imperative language **M**
that returns a value of type **a** as its result;

and the **semantics** of this language
are determined by the **monad M**.

computations that result in values

an immediate abort
a valueless return in the middle of a computation.

types are the ***rules*** associated with the **data**,
not the actual **data** itself. (*classes in C++)*

Maybe a	an imperative language
IO a	statements
ST a	computations
State s a	rules
	execution
	a function
	return

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

Semantics of a language M

Semantics : what the language **M** allows us to say.

In the case of **Maybe**,

the **semantics** allow us to express failures

when a statement fails to produce a result,

allow statements that are following to be *skipped*

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

Semicolon and Assignment

the **then** operator **(>>)**

an implementation of the **semicolon**

;

The **bind** operator **(>>=)**

an implementation of the **semicolon** and

assignment (binding) of the result

of a previous computational step.

<- ;

; ->

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

A function application and the bind operator

a **let** expression as a **function** application,

```
let x = foo in (x + 3)

foo  &  (\x -> id (x + 3))      -- v & f = f v
```

& and **id** are trivial;

id is the identity function
just returns its parameter unmodified

an **assignment** and **semicolon** as the **bind** operator:

```
x <- foo; return (x + 3)

foo >>= (\x -> return (x + 3))
```

>>= and **return** are substantial.

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

& and id

a **let** expression as a **function** application,

```
let x = foo in (x + 3)

foo  &  (\x -> id (x + 3))      -- v & f = f v
```

The **&** operator combines together two *pure calculations*,
foo and **id (x + 3)**

while creating a new binding for the variable **x** to hold **foo**'s value,

making **x** available to the second computational step: **id (x + 3)**.

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

>>= and return

an **assignment** and **semicolon** as the **bind** operator:

```
x <- foo; return (x + 3)

foo >>= (\x -> return (x + 3))
```

The bind operator **>>=** combines together two computational steps,
foo and **return (x + 3)**,
in a manner particular to the **Monad** M,

while creating a new binding for the variable **x** to hold **foo**'s result,

making **x** available to the next computational step, **return (x + 3)**.

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

>>= and return – Semantics of Maybe Monad

an **assignment** and **semicolon** as the **bind** operator:

```
x <- foo; return (x + 3)

foo >>= (\x -> return (x + 3))
```

In the particular case of **Maybe**, ***semantics***

if **foo** fails to produce a result,

the second step will be skipped and

the whole combined computation will also fail immediately.

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

Monad Class Function >> & >>=

>>= and >> : methods from the Monad *class*

Monad Sequencing Operator

>> is used to **order** the **evaluation** of expressions
within some context;
it makes evaluation of the *right*
depend on the evaluation of the *left*

Monad Sequencing Operator with value passing

>>= **passes** the result of the expression on the *left*
as an argument to the expression on the *right*,
while preserving the context the argument and function use

Just 10 :: Maybe Int

context
semantics
effects

https://www.quora.com/What-do-the-symbols-and-mean-in-haskell

Monad Definition

A **monad** is defined by

a **type constructor m**;
a function **return**;
an operator (**>>=**) "**bind**"

The function and operator are methods of the Monad type class and have types

return :: a -> **m** a

(>>=) :: **m** a -> (a -> **m** b) -> **m** b

are required to obey three laws

m a
- **Maybe a**
- **IO a**
- **ST a**
- **State s a**

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

Monad Laws

every instance of the Monad type class must obey the following three laws:

```
m >>= return     =  m                          -- right unit
return x >>= f   =  f x                        -- left unit
(m >>= f) >>= g  =  m >>= (\x -> f x >>= g)     -- associativity
```

```
(m  >>=  return)     =  m

(return  x)  >>=  f   =  f x

((m >>= f) >>= g)   =  m >>= (\x -> f x >>= g)
```

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

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

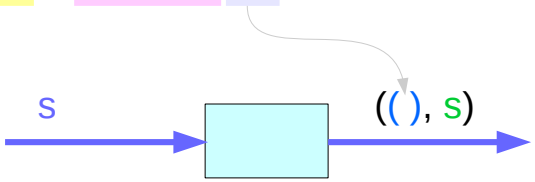
https://www.cs.hmc.edu/~adavidso/monads.pdf

Monadic Operations

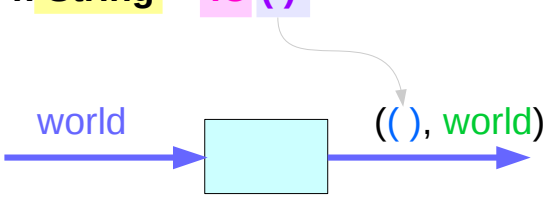
Monadic operations tend to have types which look like

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

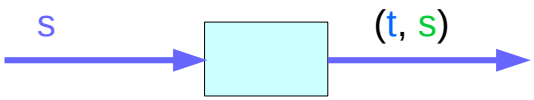
val-out-type

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 ( )

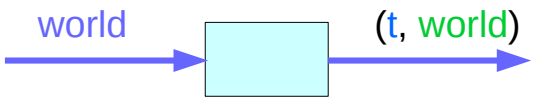
delivers a string to stdout but does not return anything meaningful

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

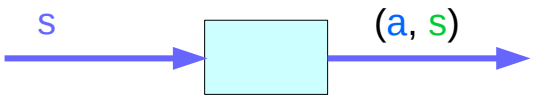



```
newtype State s a = State { runState :: s -> (a, s) }
```

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
```

m a
- Maybe a
- IO a
- ST a
- State s 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

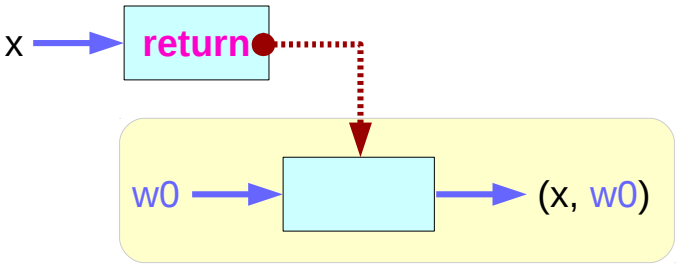
```
instance Monad (State s) where

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

IO Monad - return method

The **return** function takes x
and gives back a function
that takes a w0 :: World
and returns x along with the updated World,

but not modifying the w0 :: World it was given

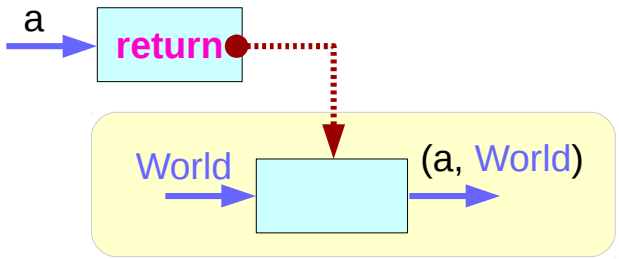


.

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

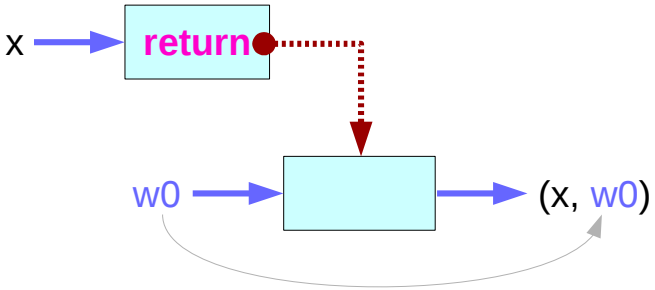
IO Monad - return method type

return a :: a -> IO a

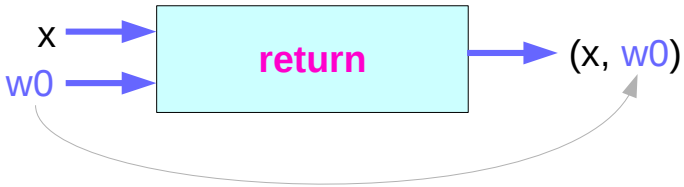


return a World :: (a, World)

let (x, w0) = return x w0



let (x, w0) = return x w0



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

IO Monad - >>= operator

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**,

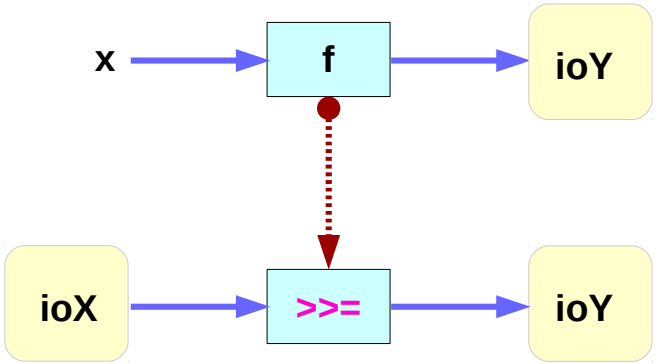
returns **x :: a** and the new, updated **w1 :: World**

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 >>= f)

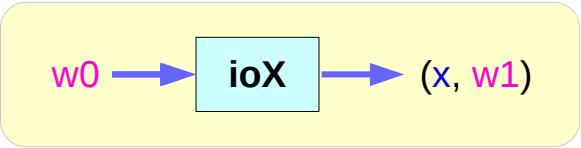


ioX :: IO a **f :: a -> IO a** **ioY :: IO b**

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

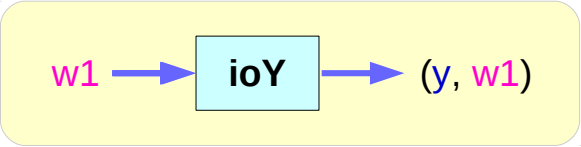
IO Monad - >>= operator binding

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**



let (x, w1) = ioX w0

bind variables



let (y, w1) = ioY w0

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

IO Monad - >>= operator implementation

(t,World)		(t,World)		(t,World)
(–,w0)	→	(x,w1)	→	(y,w1)

first, execute the **action**	then, compute the new **result**
State updated	**No State transition**
Result extracted	**Result computed**

the implementation of bind

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

IO Monad Instance

```
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

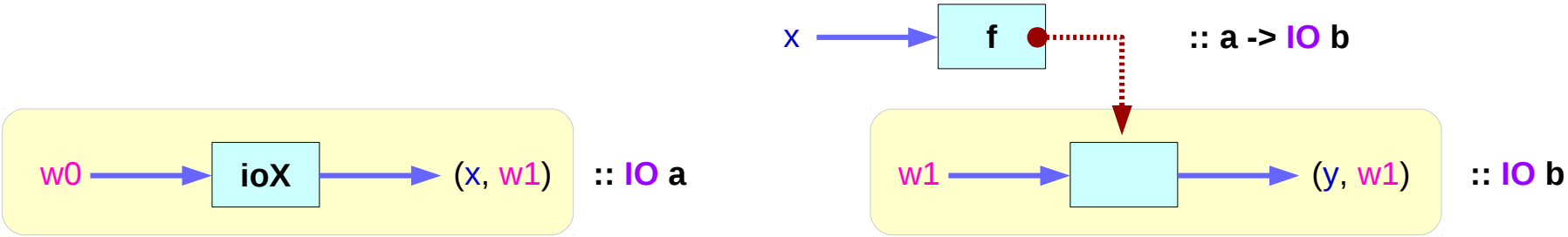
https://www.cs.hmc.edu/~adavidso/monads.pdf

IO Monad – IO a, IO b 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

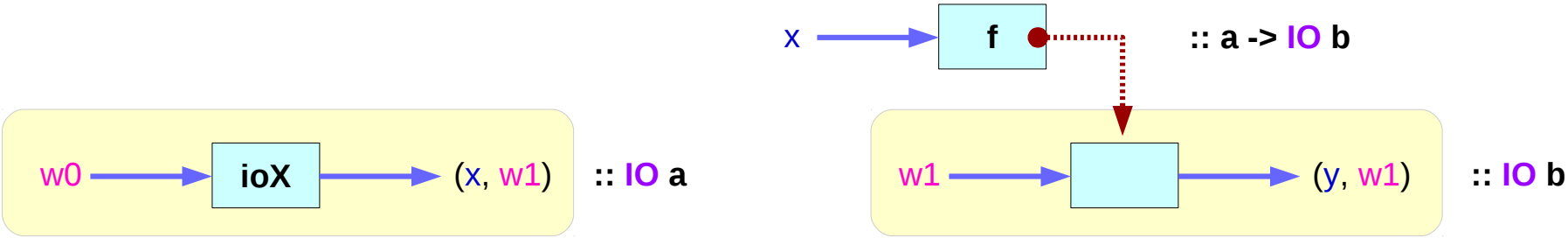
type **IO** t = World -> (t, World) type synonym



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

IO Monad – (a -> IO b) type

```
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 – binding variables

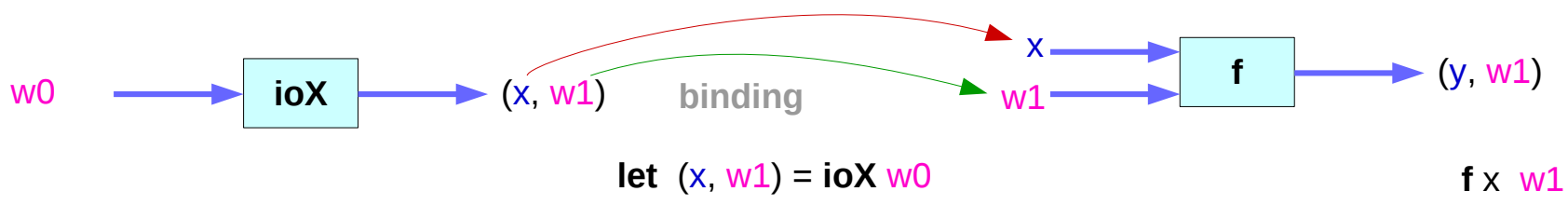
ioX >>= f :: IO a -> (a -> IO b) -> IO b

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

x :: a
w1 :: World

internal variables

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

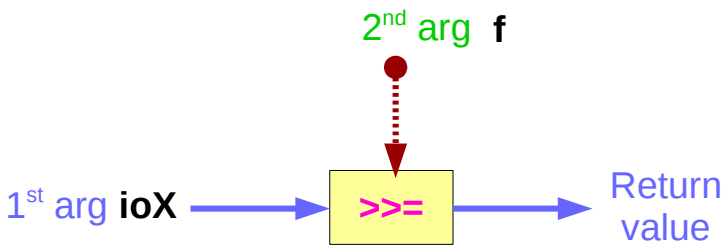


https://www.cs.hmc.edu/~adavidso/monads.pdf

IO Monad - (>>=) operator type

Monad	Function	Monad
1st arg	2nd arg	return

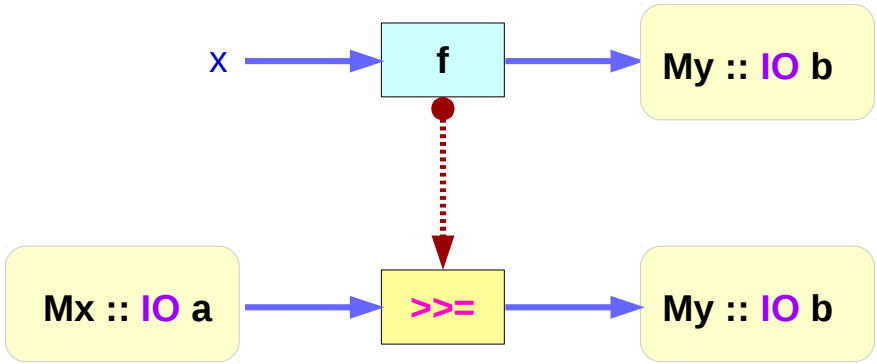
```
(>>=) :: IO a -> (a -> IO b) -> IO b
```



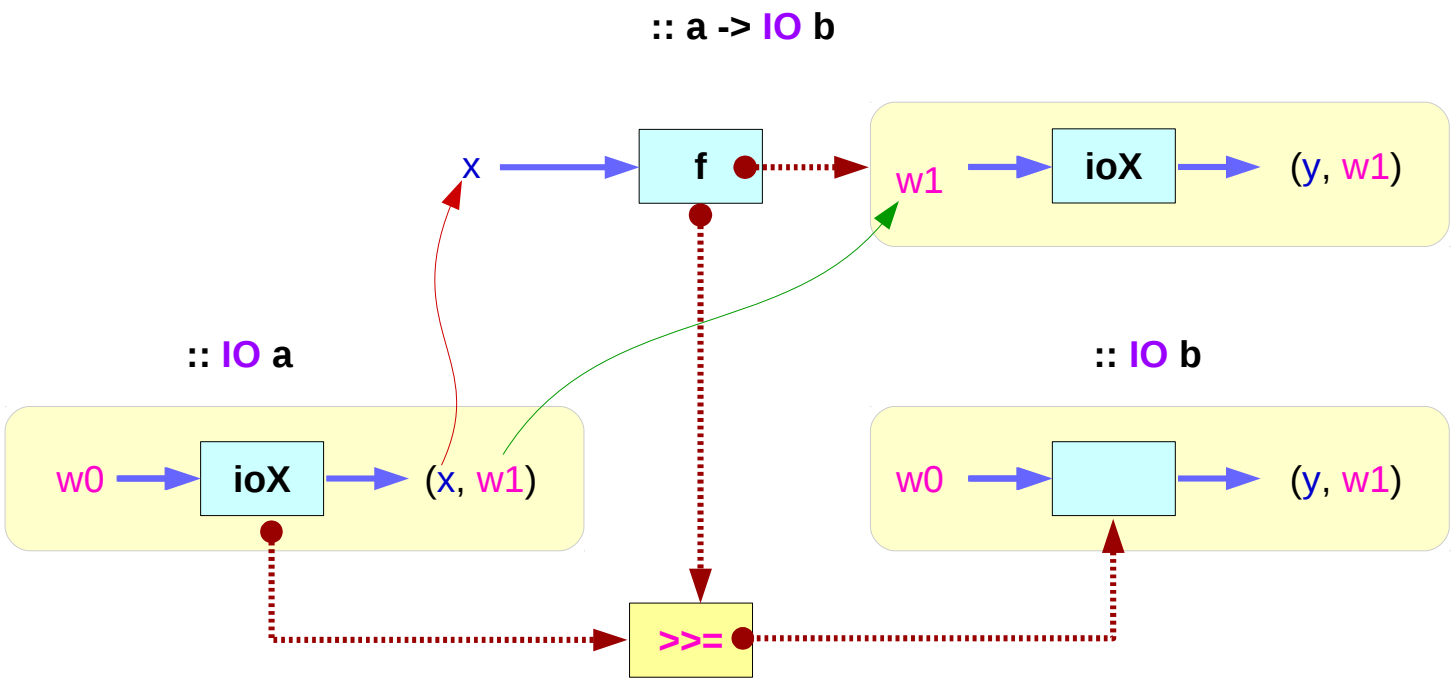
IO Monad - (>>=) operator type

Monad	Function	Monad
1st arg	2nd arg	return

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



IO Monad - (>>=) operator type



IO Monad and ST Monad

```
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)
```

```
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 Monad

```
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

Another Monad Definition

```
fmap   :: (a -> b) -> M a -> M b  -- functor

return :: a -> M a

join   :: M (M a) -> M a
```

the **functors-as-containers** metaphor

a **functor M** – a **container**
M a *contains* a value of type **a**
fmap allows functions to be applied to values in the **container**

fmap applies a function to a value in a container
return packages a value in a container
join flattens a container in containers

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

fmap and join

fmap applies a given function to every element in a container
return packages an element into a container,
join takes a container of containers and flattens it into a single container.

(>>=) in terms of join and fmap

```
m >>= g = join (fmap g m)
```

fmap and join in terms of (>>=) and return

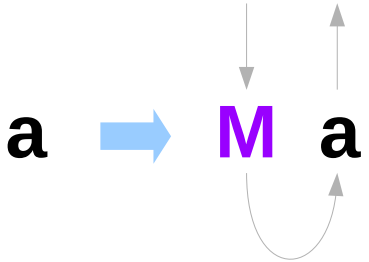
```
fmap f x = x >>= (return . f)
join x   = x >>= id
```

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

Monad – Mapping and Lifting

mapping a new type

Monads map types to new types
that represent "computations that result in values"



lifting function

can lift **functions** into **Monad types**
via a very fmap-like function called **liftM**
that turns a regular function into a
"computation that results in the value
obtained by evaluating the function."

f :: a -> b
liftM f :: M a -> M b

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

Lifting

The function **return** lifts a plain *value* **a** to **M a**

The *statements* in the imperative language **M** when executed, will result in the value **a** without any additional effects particular to **M**.

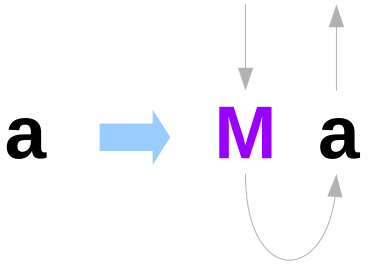
This is ensured by **Monad Laws**,

foo >>= return === foo

```
foo >>= return
foo
```

return x >>= k === k x;

```
return x >>= k
k x;
```



https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

liftM Function

a **Monad** is just a special **Functor** with extra features

Monads

map types to new types

that represent "computations that result in values"

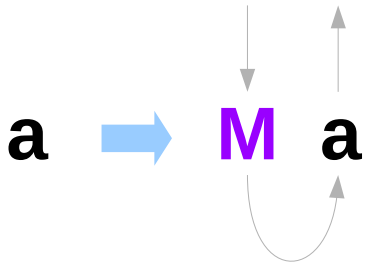
can lift regular functions into **Monad** types

via a **liftM** function (like a **fmap** function)

liftM transform a regular function

into a "computations that results in the value

obtained by evaluating the function."



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

liftM Function

Control.Monad defines **liftM**,
a function with a strangely familiar type signature...

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
```

liftM is merely
fmap implemented with (**>>=**) and **return**

liftM and **fmap** are therefore interchangeable.

```
      f ::    a ->    b
liftM f :: M a -> M b
```

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

ap Function

Another Control.Monad function
with an uncanny type is **ap**:

```
ap :: Monad m => m (a -> b) -> m a -> m b
```

Analogously to the other cases,
ap is a monad-only version of (<*>).

```
M f :: M (a -> b)
ap M f :: M a -> M b
```

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

liftM vs fmap and ap vs <*>

```
liftM :: Monad m =>  (a -> b) -> m a -> m b
fmap ::  Functor f =>   (a -> b) -> f a -> f b
```

```
ap :: Monad m => m (a -> b) -> m a -> m b
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

```
(>>=) ::  Monad m => m a  -> (a -> m b)  -> m b
```

https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

References

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