Monad P1 : Overview (2A)

Young Won Lim
3/9/19

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.

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

Monad – a parameterized type

a **monad** is a parameterized type **m**

Maybe is not a concrete type

Maybe Int is a concrete type

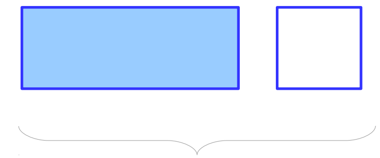
```
class Monad m where ...
```

```
instance Monad Maybe where ...
```

m a

Maybe a

single parameter



Monadic type

Maybe Int

Maybe Float

IO Float

IO ()

…

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

A notion of computations

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

a **statement** in an imperative language **M**

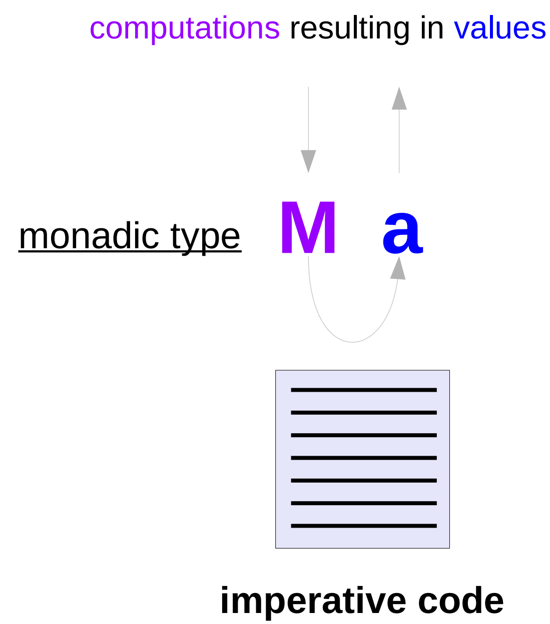
that returns a value of type **a** as its **result**;

this **statement** describes what **effects** are possible.

executing this **statement** returns the **result**

which is like executing a **function**

effects + result



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.

a **statement** describes which **effects** are possible.

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

In the case of **Maybe**,

the **semantics** allow us to express failures

when a statement fails to produce a result,

allowing the following statements to be **skipped**

an immediate abort

a valueless return in the middle of a computation.

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

A value of type **M a**

mx :: **M a**

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

an execution of a function
computations that result in values

a in **M a** shows what type of **value**
is produced by the operation

M a represent a parameterized **Monad** type

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

the type **M a**	a monadic value **mx**
⇕	⇕
an imperative language **M**	a statement in **M** returning a type **a** value
function definition	function application, execution, a return value

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

A **Monad** type

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 type** 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.

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

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

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

collection of methods
to be implemented

Rules + Data

Rules

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

Monad methods

a **monad** is a parameterized type **m**

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

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

to sequence **m a** type values.

the **do** notation can be used

generally, the **(>>=)** bind operator is used

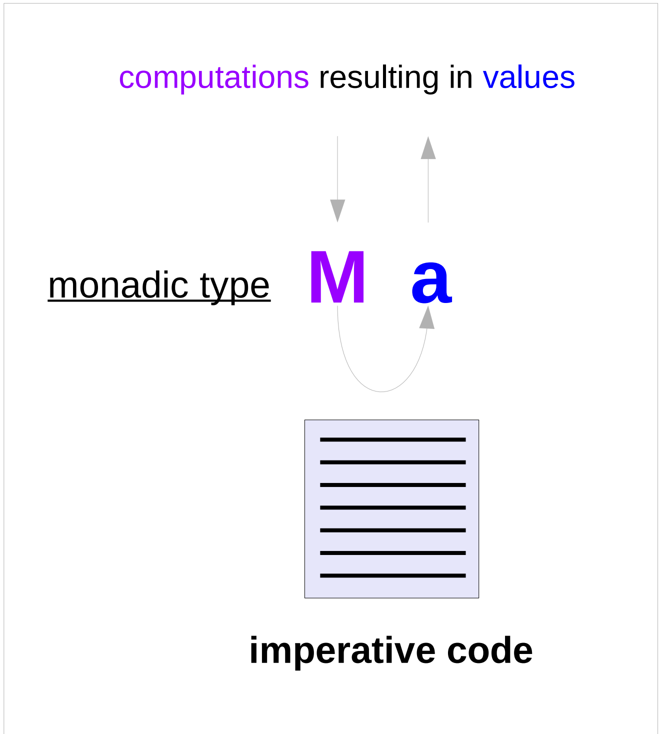
m a represent

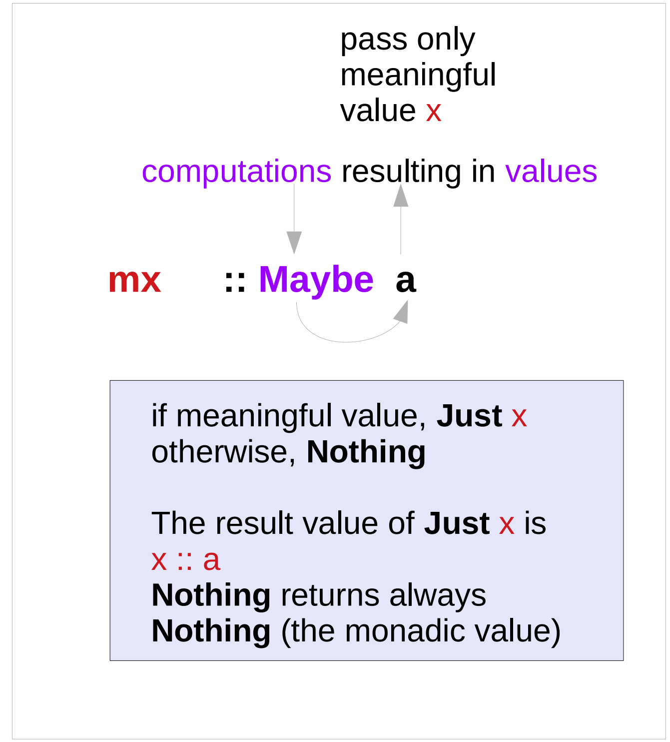
a parameterized **Monad** type

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

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

Maybe Monad – an action and its result





semantics

effects

mx has two forms

Just x

Nothing

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

Maybe Monad Instance

```
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 -> Maybe b
```

method type signatures

return method definition

>>= method definition

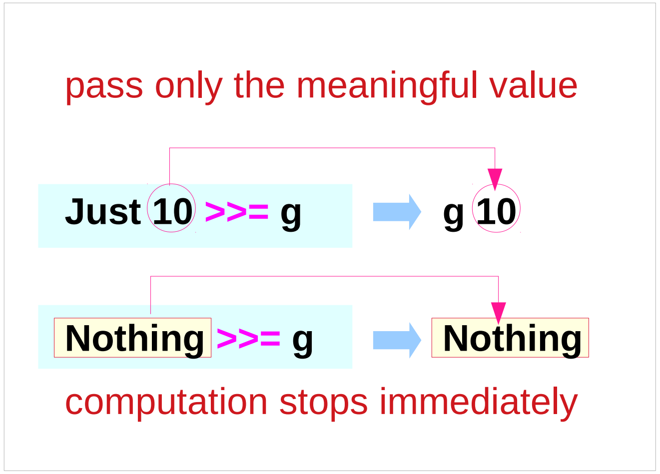
m a

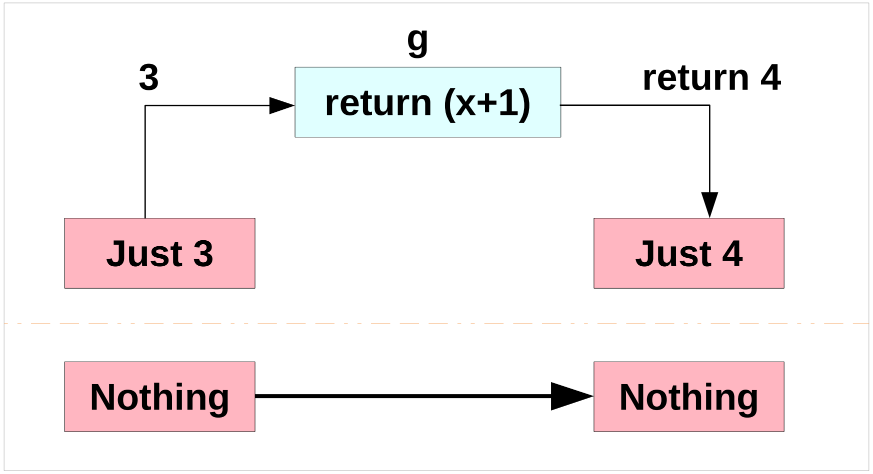
Maybe a

a parameterized type

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

Maybe Monad – the bind operator (**>>=**)





g :: a -> m b

g :: Int -> Maybe Int
g = \x -> return (x+1)

g x = return (x+1)

a general function **g** can return **Nothing** depending on its input **x** (eg. divide by zero)

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

Maybe Monad – (>>=) type signature

```
(Just x) >>= f   =   f x

Assume
(Just x) :: Maybe Int

f :: Int -> Maybe Int
f = \x -> return x+1

f x = return x+1          -- Just (x+1) :: Maybe Int
                          -- Nothing :: Maybe Int

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

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

Maybe Monad – the assignment operator (**<-**)

```
dt1 = do { x <- Just 3;
           if x == 3 then return 33;
                     else return 44;}
```

Just 3 ➡ **Just 33**

x = 3

```
dt2 = do { x  <- Just 4;
           if x == 3 then return 33;
                     else return 44;}
```

Just 4 ➡ **Just 44**

x = 4

```
dt3 = do { x  <- Nothing;
           if x == 3 then return 33;
                     else return 44;}
```

Nothing ➡ **Nothing**

No assignment to x

After evaluating the monadic value,
only the result 33 is assigned to x

Only a meaningful number
is assigned to x

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

Maybe Person type

A **value** of the type **Maybe Person**,
is interpreted as a **statement** in an imperative language
that **returns** a **Person** as the **result**, or **fails**.

father p, which is a function application,
has also the type **Maybe Person**

```
p          :: Person
father p   :: Maybe Person
mother q   :: Maybe Person
```

```
father :: Person -> Maybe Person
mother :: Person -> Maybe Person
```

father p = { **Just q** / **Nothing** }

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

Maybe (Person, Person) type

```
bothGrandfathers :: Person -> Maybe (Person, Person)

bothGrandfathers p =
  father p >>=
      (\dad -> father dad >>=
         (\gf1 -> mother p >>=
            (\mom -> father mom >>=
               (\gf2 -> return (gf1,gf2) ))))
```

```
bothGrandfathers p = do {
      dad    <- father p;
      gf1    <- father dad;
      mom  <- mother p;
      gf2    <- father mom;
      return (gf1, gf2);
    }
```

p :: Person

father p :: Maybe Person

mother q :: Maybe Person

dad :: Person

gf1 :: Person

mom :: Person

gf2 :: Person

(gf1, gf2) :: Maybe (Person, Person)

gf1 is only used in the final return

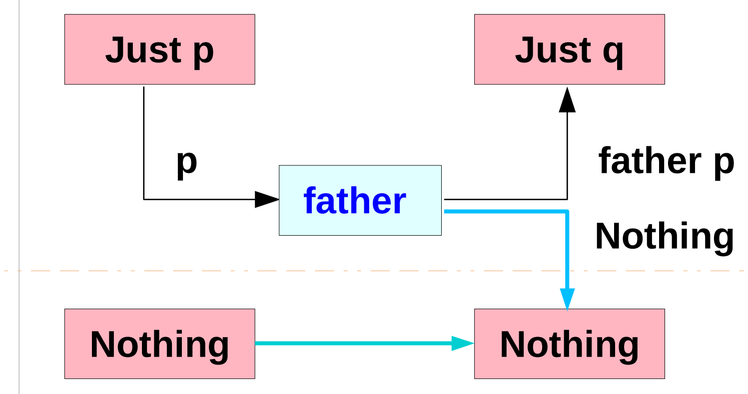
https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

Fail to return result exception

Sequencing operator **>>=** and **do** **bock** look like
an **imperative** programming code
but they support **exceptions** : **Nothing**

father and **mother** are **functions**
that might **fail** to produce results,
raising an **exception** instead; **Nothing**

when any exception happens,
the whole code will fail, i.e.
terminate with an exception
(evaluate to **Nothing**).



p :: Person

father p :: Maybe Person

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

Maybe Monad – the value for failure

The **Maybe** monad provides

a simple model of computations that can fail,

a **value** of type **Maybe a** is

either **Nothing** (failure) or

the form **Just x** for some **x** of type **a** (success)

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

List Monad – the value for failure

The **list** monad generalizes this notion,

by permitting multiple results in the case of success.

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 methods

```
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 :: [a]

f :: a -> [b]

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

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

List Monad bind operator example 1

xs >>= **f** applies the function **f**
to each of the *results* in the list **xs** = [x1, x2, x3]

f x1 = [y1, y2]
f x3 = [y5, y6]
f x2 = [y3, y4]

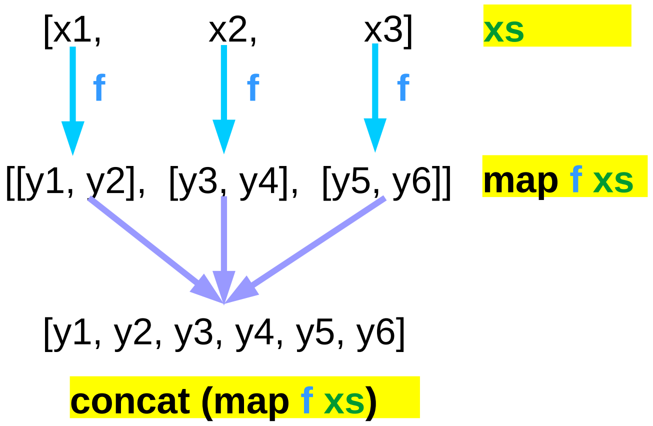
to give a *nested* *list* of *results*,

[[y1,y2], [y3,y4], [y5,y6]] **map f xs**

which is then concatenated
to give a *single list* of *results*.

[y1, y2, y3, y4, y5, y6] **concat (map f xs)**

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



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

List Monad bind operator example 2

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

f :: Int -> [ Int ]
f = \n -> [1 .. n]

[1, 2, 3] >>= \n -> [1..n]

[1,1,2,1,2,3]
```

```
f 1 = [1]
f 2 = [1, 2]
f 3 = [1, 2, 3]

[[1], [1,2], [1,2,3]]
```

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

Monad sequencing operators >> and >>=

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** that the argument and function use

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

Contexts of >> and >>=

Just 10 >> g ⇒ **g**

no value passing

g cannot be a function

g can be **Maybe** monad value

g :: Maybe Int

Just 10 >>= f ⇒ **f 10**

10 is passed to the function

f has an argument

f :: Int -> Maybe Int

Just 10 :: Maybe Int

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

Monad **sequencing** operators and **do** statements

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

an implementation of the **semicolon**

>>
do ;

```
Just 3 >> Just 4
```

```
do Just 3 ; Just 4
```

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

an implementation of the **semicolon** (;) and

assignment (<-) of the result

of a previous computational step.

>>=
do <- ;

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

```
do x <- Just 3 ;    return (x + 3)
```

x

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

Bind operator (>>=) and the function application (let)

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

x <- foo; return (x + 3)

x

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

foo :: m a
return :: a -> m b

>>= and **return** are substantial

the operation depends on the particular **Monad m**

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

let x = foo in (x + 3)

x

foo & (\x -> id (x + 3))

foo :: a

arg & func   func arg

v & f = f v

reverse function application &
nothing to do with 'AND'

& and **id** are trivial;
id is the identity function
just returns its parameter
unmodified

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

Bind operator (>>=) and the semantics of Maybe (1)

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

Bind operator (>>=) and the semantics of Maybe (2)

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

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

	semantics
In the particular case of **Maybe**,	
if **foo** fails to produce a result,	**Nothing**
the second step will be skipped and	
the whole combined computation will also fail immediately.	**Nothing**

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

Function application using & and id

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

let x = foo in (x + 3) **foo & (\x -> id (x + 3))**

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, x ← foo

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

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

Reverse Function Application &

(&) :: a -> (a -> b) -> b

arg function

& is just like $ only backwards.

foo $ bar $ baz $ bin

semantically equivalent to:

bin & baz & bar & foo

& is useful because the order in which functions are applied to their arguments read left to right instead of the **reverse** (which is the case for $).

This is closer to how English is read so it can improve code clarity.

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

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

1) return

2) bind (>>=)

3) then (>>)

4) fail

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

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 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
                 m = new_s,
                 s  =  a
                 (k a) new_s
                 (k s) m
```

```
case expression of
        pattern -> result
        pattern -> result
        pattern -> result
          ...
```

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

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 (methods)

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

Monad Minimal Definition

A minimal definition of **monad**

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

The function and operator

- are methods of the **Monad** type class
- have types (type signatures)

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

are required to obey three laws

m = { **Maybe**, **IO**, **ST**, **State s** }

a **type constructor m**

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

Monad Laws

every **instance** of the Monad type class must obey

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

m :: **M a** monadic value of type **M a**

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

x :: **a**

f :: a -> **M** b

f x :: **M** b

return :: a -> **M** a

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

m :: **M** a

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

Monad Laws Examples (1)

```
m >>= return    = m           -- right unit
return x >>= f  = f x         -- left unit
```

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

right unit

```
(m  >>=  return)      =  m
(Just 3 >>=  return)      =  Just 3
```

left unit

```
((return  x)  >>=  f)    =  f x
((return  3)  >>=  (\x -> return (x+1)))    =  return 4 = Just 4
```

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

Monad Laws Examples (2)

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

f =  (\x -> return (x+1))
g = (\x -> return (2*x))

m = Just 3

(Just 3 >>= f) >>= g      =   Just 3>>= (\x -> f x >>= g)
```

```
m :: M a
x :: a
f :: a -> M b
g :: b -> M c
```

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

Monad Laws Examples (3)

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

((Just 3) >>= f)
((Just 3)  >>=  (\x -> return (x+1)) )  = Just 4
((Just 3) >>= f) >>= g
((Just 4)  >>=  (\x -> return (2*x)) )  =  Just 8

(\x -> f x >>= g)
((\x -> return (x+1))  >>=  (\x -> return (2*x)) ) = (\x -> return (2*(x+1)))
((Just 3)  >>=  (\x -> return (2*(x+1))) )  =  Just 8
```

```
m :: M a
x :: a
f :: a -> M b
g :: b -> M c
```

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

fmap and a functor M

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

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**

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

join and a functor **M**

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

as the computation going *deeper* into the monad,

nothing is being taken "out" of the monad

with successive steps being *collapsed*

into a single layer of the monad.

M (…(M (M a))…)

↓ successive applications of **join**

M a

https://stackoverflow.com/questions/15016339/haskell-computation-in-a-monad-meaning

The associativity and identity of join

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

   M ( M (M a))  ⟹  M (M a)  ⟹  M a
   M ( M (M a))  ⟹  M (M a)  ⟹  M a

return :: a -> M a

   join (return x) = return x
```

```
import Control.Monad
join (Just (Just 10))
⟹ Just 10
join (Just (Just (Just 10)))
⟹ Just (Just 10)
```

it doesn't matter when **join** is applied,

as long as *the nesting order is preserved*

(a form of *associativity*)

the *monadic layer* introduced by **return** does *nothing*

(an *identity* value for **join**).

https://stackoverflow.com/questions/15016339/haskell-computation-in-a-monad-meaning

Function application, packaging, flattening

fmap <u>applies</u> a **function** <u>to</u> a **value** <u>in a container</u>

return <u>packages</u> a **value** <u>in a container</u>

join <u>flattens</u> a container <u>in containers</u>

applying

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

return :: a -> M a *packaging*

join :: M (M a) -> M a *flattening*

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

Three orthogonal functions and >>=

```
fmap :: (a -> b) -> (m a -> m b)
return :: a -> m a
join :: m (m a) -> m a
```

starting with arguments of type **m a** and **a -> m b**,

use **fmap** to get the type of **m (m b)**,

```
fmap ::    (a ->   b) -> (m a -> m   b )
                  ⬇                   ⬇
           (a -> m b) -> (m a -> m (m b))
```

join to *flatten* the nested "layers" to get just **m b**.

```
join . fmap ::  (a -> m b) -> (m a -> m   b )
```

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

must figure out the followings

Assumption: **m a** is a parameterized Monad type

https://stackoverflow.com/questions/15016339/haskell-computation-in-a-monad-meaning

>>= by join.fmap

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

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

```
join.fmap :: (a -> m b) -> m a -> m b

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

must figure out the followings

Assumption: **m** is a monadic value of **M a** type

Assumption: **m a** is a parameterized Monad type

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

join.fmap vs concat.map

(>>=) in terms of join and fmap

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

```
instance Monad [] where
  -- return :: a -> [a]
  return m  =  [m]
  -- (>>=)  :: [a] -> (a -> [b]) -> [b]
  m >>= g  =  concat (map g m)
```

must figure out the followings

Assumption: **m** is a monadic value of **M a** type

Assumption: **m** is a value of **a** type

Assumption: **m** is a monadic value of [**a**] type

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

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

fmap & join by >>= & return

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

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

fmap (*3) (Just 10) = Just 10 >>= return . (* 3) ➡ Just 30

join (Just (Just 10)) = Just (Just 10)) >>= id ➡ Just 10

must figure out the followings

Assumption: **x** is a monadic value of **M a** type

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

Monad's lifting capability

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

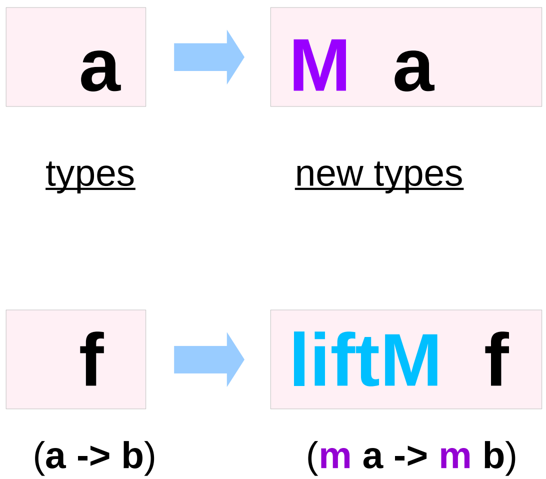
Monads

- **map** types to new types
- that represent "computations that result in values"

liftM (like **fmap**)

- can **lift** regular functions into **Monad** types

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



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

liftM function over monadic values

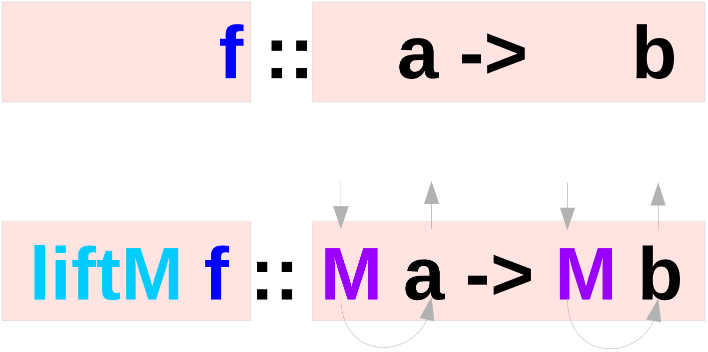
Control.Monad defines **liftM**

liftM transform a <u>regular function</u>

into a "computations that results in the value

obtained by evaluating the function."

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



computations that results in the value obtained by evaluating the function

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

liftM function & fmap

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

liftM is merely

fmap implemented with (>>=) and return

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

liftM and fmap are therefore interchangeable.

Assumption:

x is a monadic value of m a type

f :: a -> b

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

(>>=) & fmap comparisons

```
fmap f xs  =    xs >>= (return . f)
```

- xs is a monadic value of **m a** type
- **f :: a -> b**

```
xs >>= f  =  concat (map f xs)
```

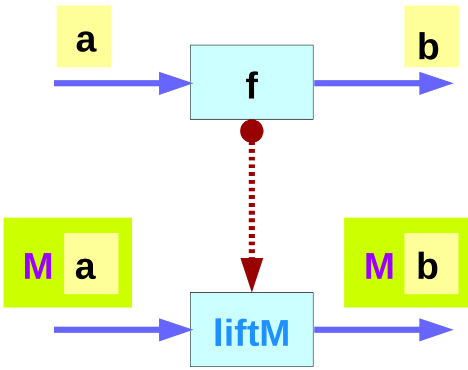
- xs is a monadic value of **[a]** type
- **f :: a -> [b]**

```
xs >>= f  =  join  (fmap f xs)
```

- xs is a monadic value of **m a** type
- **f :: a -> m b**

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

liftM – function lifting



type lifting

a → M a

function lifting

f :: a -> b

liftM f :: M a -> M b

return – type lifting

The function **return** <u>lifts</u> a plain *value* **a** to **M a**

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

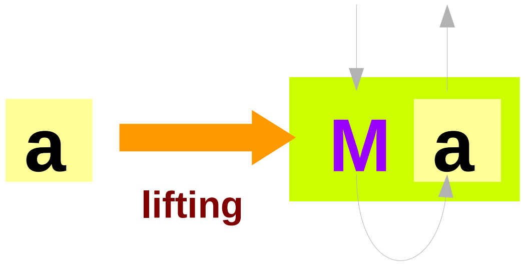
This is ensured by **Monad Laws**,

```
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

ap Function

Control.Monad defines **ap** function

application

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)

application

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

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

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

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

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

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

[7] https://en.wikibooks.org/wiki/Haskell/Understanding_monads

[8] https://en.wikibooks.org/wiki/Haskell/Understanding_monads/State

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

[10] https://stackoverflow.com/questions/15016339/haskell-computation-in-a-monad-meaning

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