Monad Overview (2A)

Young Won Lim
1/1/19

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 <u>not</u> a <u>concrete</u> type

Maybe Int is a <u>concrete</u> type

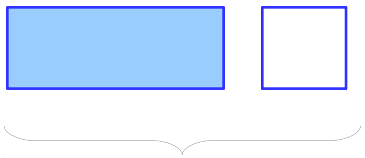
class Monad m where ...

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

Young Won Lim
1/1/19

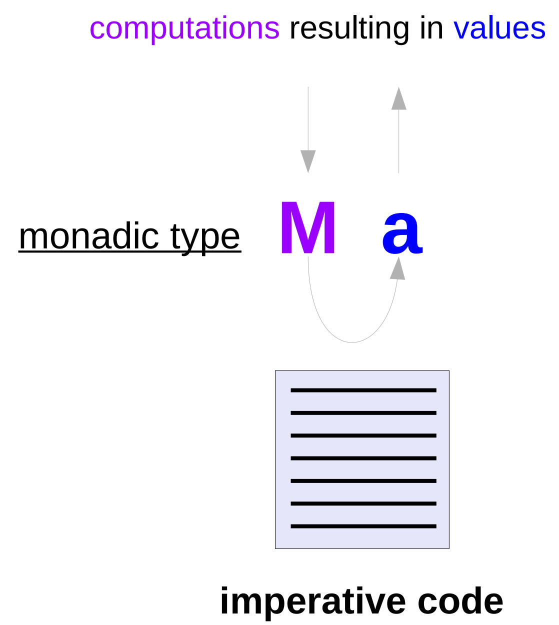
A notion of computations

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**;

a **statement** in an imperative language **M**
describes which **effects** are possible.

executing a **statement** returns the **result**
running 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** in an imperative language **M**

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 statements that are following 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 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** ⇕ an imperative language **M**

function definition

a monadic value **mx** ⇕ a statement in **M** returning a type **a** value

function application, execution, a return value

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

A Type **Monad**

Haskell does not have **states**

but it's type system is powerful enough

to construct the stateful program flow

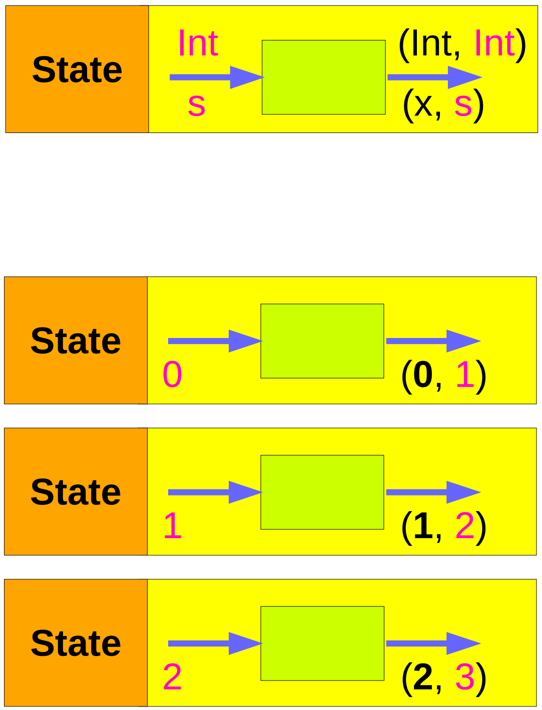
defining a **Monad type** in Haskell

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

A **Monad type** can be used for

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

stateful computations based on **function application**



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 <u>sequence</u> **m a** type values.

the **do** notation can be used

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

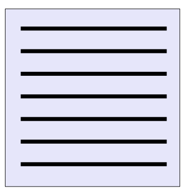
```
tick :: State Int Int
tick = do   n <- get
            put (n+1)
            return n

test  = do  tick                  -- (0,1)
            tick                  -- (1,2)

test  = tick  >>= tick

runState test 0                   -- (4,6)
```

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

Maybe Monad – an action and its result

computations resulting in values

monadic type **M a**



imperative code

pass only
meaningful
value x

computations resulting in values

mx :: Maybe a

if meaningful value, **Just** x
otherwise, **Nothing**

The result value of **Just** x is
x :: a
Nothing returns always
Nothing (the monadic value)

semantics

effects

mx has two forms

Just x

Nothing

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

Maybe Monad Instance

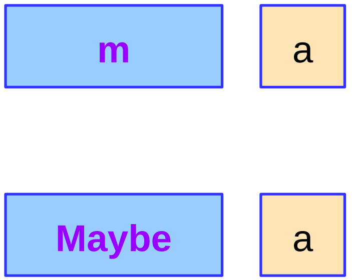
```
class Monad m where

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

method type signatures

```
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
```

return method definition

>>= method definition



a parameterized type

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

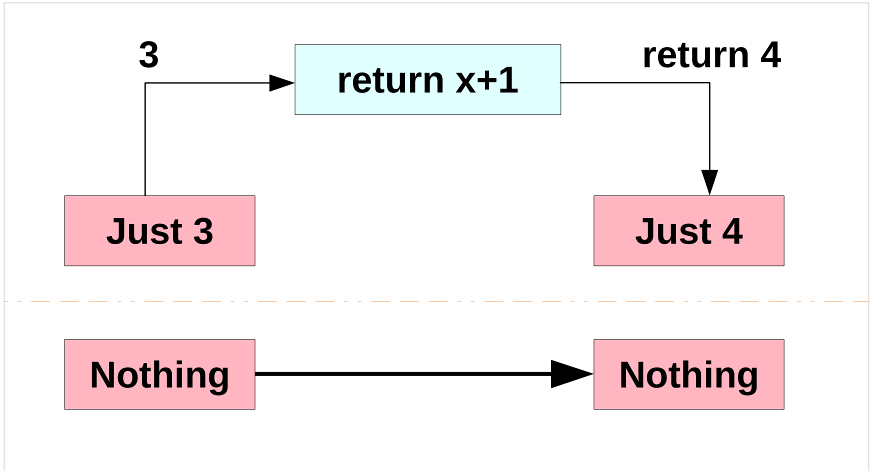
Maybe Monad – the bind operator (>>=)

pass only the meaningful value

Just 10 >>= g ➡ g 10

Nothing >>= g ➡ Nothing

computation stops immediately



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 3) :: Maybe Int

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

f x = return x+1          -- Just (x+1) :: m b

(>>=)  :: 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

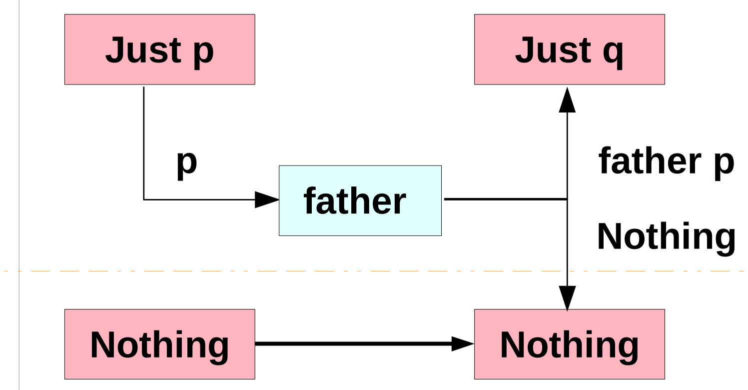
https://en.wikibooks.org/wiki/Haskell/Understanding_monads#cite_note-3

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

xs >>= **f** applies the function **f** to each of the *results* in the list **xs**

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

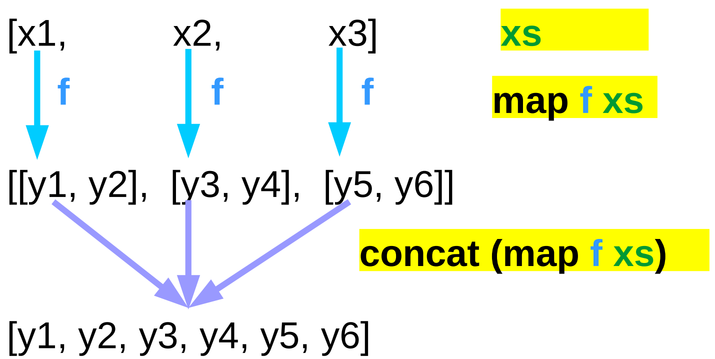
to give a *nested list* of *results*,

[[y1,y2], [y3,y4], [y5,y6]]

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

[y1, y2, y3, y4, y5, y6]

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



xs :: [a]

f :: a -> [b]

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

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

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

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

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

Monad Applications

1. **Exception Handling**	**Maybe a**
2. **Accumulate States**	**State s a**
3. **IO Monad**	**IO a**

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 (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 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

```
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**

x :: **a**

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

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

m :: **M** a

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

f x :: **M** b

f x >>= g :: M c

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

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

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

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

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

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

```
((m >>= f)
((m >>= f) >>= g)
```

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

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

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

then (>>) and **bind (>>=)** operators

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

an implementation of the **semicolon**

<- ;

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

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

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

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

assignment (**<-**) of the **result**

of a previous computational step.

>>= ->

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

x

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

Contexts of >> 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

Just 10 :: Maybe Int

context
semantics
effects

Just 10 >>= f

⇒ **f 10**

10 is passed to the function
f as an argument

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

>>= 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,	**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

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
```

reverse function application &

& 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

Reverse Function Application &

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

& 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

& and id

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

```
let x = foo in (x + 3)        foo  &  (\x -> id (x + 3))      -- v & f = f $ v = fv
```

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

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 – a function form

Monadic operations tend to have types which look like

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

the types of **inputs** to a **monadic operation**

the type of a **return** **value** from a **monadic operation**

function type

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

Monadic Operations – returning a monadic value

Monadic operations tend to have types which look like

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

a monadic operation

= a function

- **inputs**
- **a return value**

– another **function is returned**

– executing this returned function

– returns a **function** as a **value**

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

Monadic Operations – the result of a monadic value

Monadic operations tend to have types which look like

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

effect-monad produces a

result of a type of val-out-type

computations

statement

in the imperative language

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

Monadic Operations – type application

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

type application
cf) function application

the return type is a **type application** like a function application

the function part tells you ………………………… **effect-monad**

which effects are possible

the argument part tells you ……………………… val-out-type

what sort of value is produced by the operation.

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

Monadic Operations – **IO** and **State** Monads

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

Monadic operations using **IO** and **State**

have a **return value**, as well as

performing **side-effects**.

the only point of using these monadic operations is

to perform a **side-effect**,

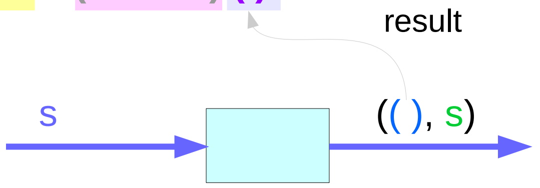
writing to the screen …………. IO Monad
storing some state …………. State Monad

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

Monadic Operation – the result type

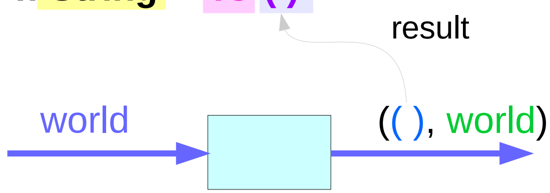
val-in-type-1 **->** ... **->** val-in-type-n **->** **effect-monad** val-out-type

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



the execution result type of the returned function

putStr :: String -> IO ( )



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

Monadic Operations – **put** example

put :: s -> State s ( )

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

the operation is used *only for its effect*;

the *value* delivered is *uninteresting*

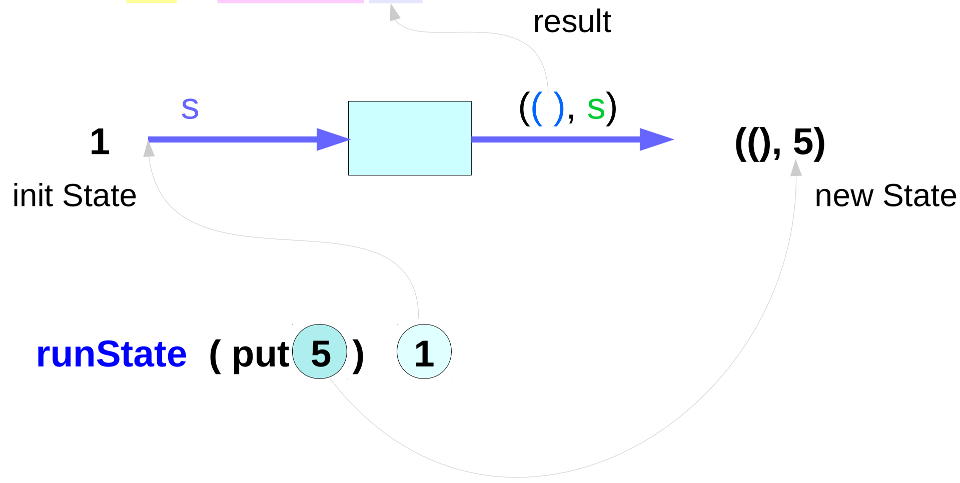
one value input type — **s**

the effect-monad — **State s**

the value output type — **( )**

effect-monad	val-out-type
(State s)	**( )**

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



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

Monadic Operations – **putStr** example

putStr :: String -> IO ( )

delivers a string to **stdout**

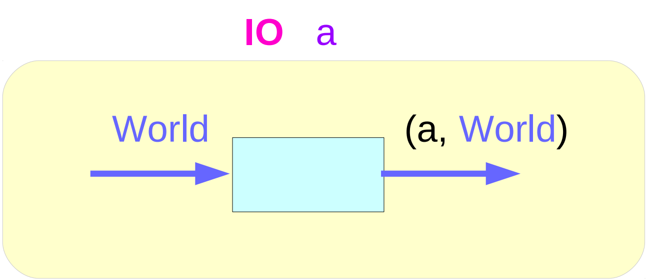
but does not return anything meaningful

one value input type **s**

the **effect-monad** **IO**

the value output type **( )**

effect-monad	val-out-type
IO	( )



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

Monadic Operations – underlying functions

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

a parameterized type

a monad type

effect-monad

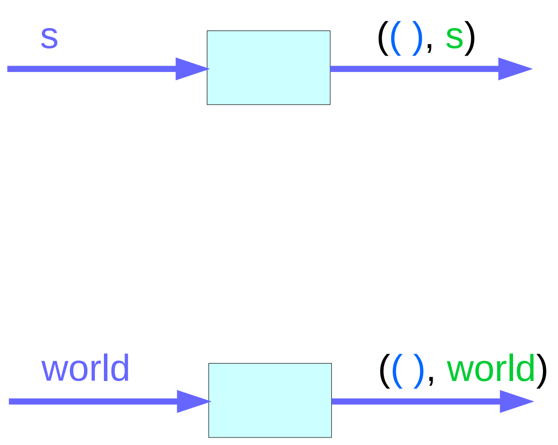
statements in the imperative language

about a function

executable

execution result

val-out-type

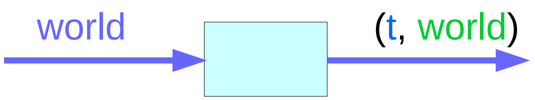


https://stackoverflow.com/questions/16892570/what-is-in-haskell-exactly

IO t and State s a types

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

type synonym

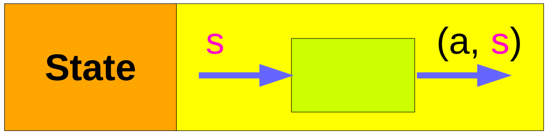



```
newtype State s a = State { runState :: s -> (a, s) }
```

s : the type of the state,

a : the type of the produced result

s -> (a, s) : function type



accessor function

runState :: State s a -> (s -> (s, a))

IO Monad – return method

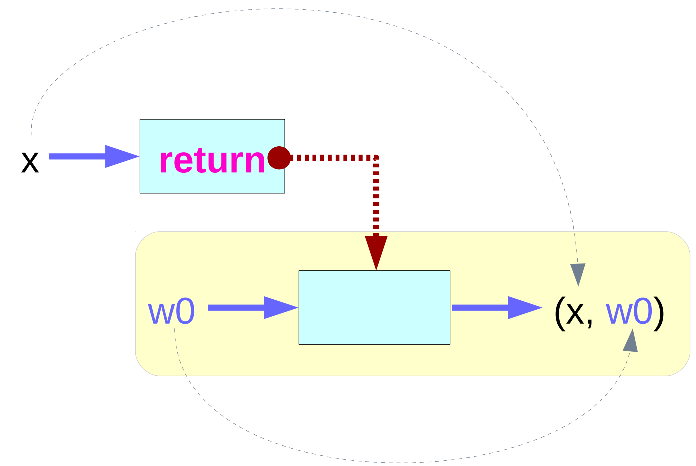
The **return** function takes x

and gives back a function

that takes a w0 :: World

and returns x along with the updated World,

but not modifying the given w0 :: World



.

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

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

Pure functional programs

Why do you need a monad?

Pure functional languages are different from **imperative languages** like C, or Java in that,

- a pure functional program is not necessarily executed in a specific order, one step at a time.
- A Haskell program is more akin to a mathematical function, in which you may solve the "equation" in any number of potential orders.
- it eliminates the possibility of certain kinds of bugs (data dependency, particularly those relating to things like **state**)

https://stackoverflow.com/questions/44965/what-is-a-monad

Execution orders

However, certain problems like **console programming**, and **file i/o**, need things to happen in a particular order, or need to maintain **state**.

One way to deal with this problem is to create

- a kind of **object** that represents the **state** of a computation, and
- a series of **functions** that take a **state object** as input, and return *a new modified* **state object**.

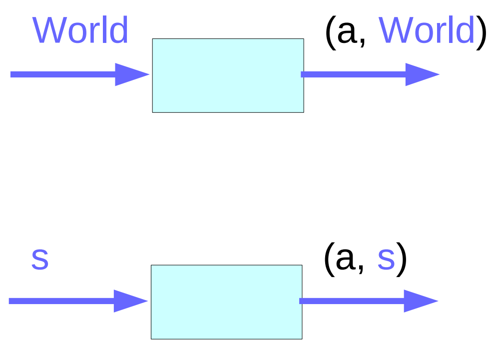
https://stackoverflow.com/questions/44965/what-is-a-monad

A hypothetical state value

a hypothetical **state** value can

represent the **state** of a console screen.

- exact value is not important,
- an array of byte length ascii characters that represents what is currently *visible* on the screen, and
- an array that represents the last line of *input* entered by the user, in pseudocode.
- create some functions that take console **state**, modify it, and return a new console **state**.



https://stackoverflow.com/questions/44965/what-is-a-monad

Nesting style for a particular execution order

consolestate MyConsole = new consolestate;

for a pure functional manner, a possible choice is to nest a lot of function calls inside each other.

```
consolestate FinalConsole =
    print( input( print( myconsole, "Hello, what's your name?" ) ),"hello, %inputbuffer%!" );
```

- this programming keeps the pure functional style
- while forcing changes to the console to happen in a particular order.

https://stackoverflow.com/questions/44965/what-is-a-monad

No-nesting style

- more than just a few operations at a time
- more than nesting functions
- a more convenient way to write it

```
consolestate FinalConsole = myconsole:
                 print("Hello, what's your name?"):
                 input():
                 print("hello, %inputbuffer%!");
```

https://stackoverflow.com/questions/44965/what-is-a-monad

Monad, bind and lift operators

If you have a **type** (such as consolestate) that you define along with a few **functions** designed specifically to operate on that type,

you can make a whole package of **type** definition and related **functions** into a **monad** by defining an **operator** like :

(**bind operator**) automatically feeds **return** values on its left, into **function** parameters on its right,

(**lift operator**) turns normal functions, into functions that work with that specific kind of **bind operator**.

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

f :: a -> b

liftM f :: m a -> m b

https://stackoverflow.com/questions/44965/what-is-a-monad

Bind operator >>=

```
putStrLn "What is your name?"
>>= (\_ -> getLine)
>>= (\name -> putStrLn ("Welcome, " ++ name ++ "!"))
```

The **>>=** operator takes a value (on the left side)

and combines it with a function (on the right side),

to produce a new value.

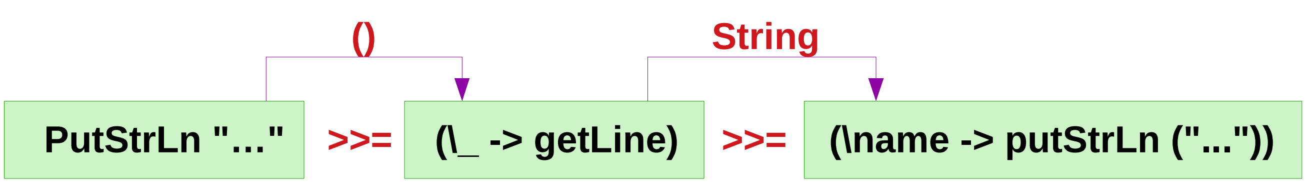
This new value is then taken by the next **>>=** operator

and again combined with a function to produce a new value.

>>= can be viewed as a mini-evaluator.

putStrLn :: String -> IO ()

getLine :: IO String



https://stackoverflow.com/questions/44965/what-is-a-monad

Monadic operation

a **monad**

- is a parameterized type
- is an instance of the **Monad type class**
- defines **>>=** along with a few other operators.
- just a **type** for which the **>>=** operation is defined.

In itself **>>=** is just a cumbersome way of **chaining functions**,
but with the presence of the **do-notation**
which hides the "**plumbing**",
the **monadic operations** turns out to be a very nice
and useful abstraction, useful many places in the language,
and useful for creating your own mini-languages in the language.

https://stackoverflow.com/questions/44965/what-is-a-monad

>>= : an overloaded operator

Note that **>>=** is overloaded for different types,
so every monad has its own implementation of >>=.
(All the operations in the chain have to be of the type
of the same monad though, otherwise the >>= operator won't work.)

The simplest possible implementation of **>>=** just takes the value
on the left and applies it to the function on the right and
returns the result, but as said before,
what makes the whole pattern *useful* is
when there is something extra going on
in the monad's implementation of >>=.

https://stackoverflow.com/questions/44965/what-is-a-monad

Combining functions

in a **do-block**, every operation (basically every line) is
wrapped in a separate anonymous function.

These functions are then combined using the **bind** operator

the **bind** operation combines **functions**,

it can execute them as it sees *fit*:
sequentially, multiple times, in reverse, discard some,
execute some on a separate thread when it feels like it and so on.

https://stackoverflow.com/questions/44965/what-is-a-monad

Various Monad applications (1)

1) The **Failure Monad**:

If each step returns a success/failure indicator,

bind can execute the next step only if the previous one succeeded.

a failing step can abort the whole sequence "automatically",

without any conditional testing from you.

2) The **Error Monad** or **Exception Monad:**

Extending the Failure Monad, you can implement **exceptions**

By your own definition (not being a language feature),

you can customize how they work.

(e.g., can ignore the first two exceptions and

abort when a third exception is thrown.)

https://stackoverflow.com/questions/44965/what-is-a-monad

Various Monad applications (2)

3) The **List Monad:**

each step returns multiple results, and the bind function iterates over them, feeding each one into the next step

No need to write loops all over the place

when dealing with multiple results.

4) The **Reader Monad**

As well as passing a result to the next step,

the bind function pass extra data around as well

This extra data now doesn't appear in your source code,

but it can be still accessed from anywhere,

without a manual passing

https://stackoverflow.com/questions/44965/what-is-a-monad

Various Monad applications (3)

5) The **State Monad** and the **Writer Monad**

the extra data can be replaced.

This allows you to simulate destructive updates

without actually doing destructive updates

you can trivially do things that would be impossible with real destructive updates.

For example, you can undo the last update,

or revert to an older version.

https://stackoverflow.com/questions/44965/what-is-a-monad

Various Monad applications (4)

You can make a monad where calculations can be paused,
so you can pause your program,
go in and tinker with internal state data,
and then resume it.

You can implement continuations as a monad.

https://stackoverflow.com/questions/44965/what-is-a-monad

List Monad Examples

```
[x*2 | x<-[1..4], odd x]

t = do  x <- [1..4]
        if odd x then [x*2]  else []


[1..4] >>= (\x -> if odd x then [x*2] else [])
                1               [2]
                2               [ ]
                3               [6]
                4               [ ]
```

Monads as computation builders
the monad chains operations
in some specific, useful way.

in the **list comprehension** example:

if an operation returns a list,
then the following operations are
performed on ***every item*** in the list.

https://stackoverflow.com/questions/44965/what-is-a-monad

Reader Monad Examples

Reader r a

where **r** is some "**environment**" and

a is some **value** you create from that environment

let r1 = return 5 :: Reader String Int

:t r1

r1 :: Reader String Int

a Reader that takes in a String and returns an Int.

The String is the "environment" of the Reader.

https://blog.ssanj.net/posts/2014-09-23-A-Simple-Reader-Monad-Example.html

Reader Monad Examples

```
Reader r a
let r1 = return 5 :: Reader String Int
r1 :: Reader String Int



(runReader r1) "this is your environment"
5
runReader :: Reader r a -> r -> a
```

So runReader takes in a Reader and an environment (r)
and returns a value (a).

https://blog.ssanj.net/posts/2014-09-23-A-Simple-Reader-Monad-Example.html

Reader Monad Examples

```haskell
import Control.Monad.Reader

tom :: Reader String String
tom = do
    env <- ask
    return (env ++ " This is Tom.")

jerry :: Reader String String
jerry = do
  env <- ask
  return (env ++ " This is Jerry.")
```

```haskell
tomAndJerry :: Reader String String
tomAndJerry = do
    t <- tom
    j <- jerry
    return (t ++ "\n" ++ j)

runJerryRun :: String
runJerryRun = (runReader tomAndJerry)
"Who is this?"

Who is this? This is Tom.
Who is this? This is Jerry.
```

https://blog.ssanj.net/posts/2014-09-23-A-Simple-Reader-Monad-Example.html

I/O Monad Examples

```
do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Welcome, " ++ name ++ "!")
```

getChar :: IO Char

Read a character from the standard input device

getLine :: IO String

Read a line from the standard input device

Monads as computation builders

the monad chains operations

in some specific, useful way.

in the **IO monad** example

the operations are performed sequentially,

but a hidden variable is passed along,

which represents the **state** of the **world**,

allows us to write **I/O code** in a **pure**

functional manner.

https://stackoverflow.com/questions/44965/what-is-a-monad

A Parser Example

```
parseExpr = parseString <|> parseNumber

parseString = do
        char '"'                                    -- \".*\"
        x <- many (noneOf "\"")
        char '"'
        return (StringValue x)

parseNumber = do
    num <- many1 digit
    return (NumberValue (read num))
```

The operations (char, digit, etc) either match or not

the monad manages the **control flow**:

The operations are performed sequentially until a match fails, in which case the monad backtracks to the latest **<|>** and tries the next option.

Again, a way of chaining operations with some additional, useful semantics.

https://stackoverflow.com/questions/44965/what-is-a-monad

Parser – **char**, **digit**

char :: Stream s m Char => Char -> ParsecT s u m Char

char c parses a single character c.

Returns the parsed character (i.e. c).

semiColon = **char** ';'

digit :: Stream s m Char => ParsecT s u m Char

Parses a digit.

Returns the parsed character.

https://stackoverflow.com/questions/44965/what-is-a-monad

Parser – **many**, **many1**, **noneOf**

many :: ReadP a -> ReadP [a]

Parses **zero or more** occurrences of the given parser.

many1 :: ReadP a -> ReadP [a]

Parses **one or more** occurrences of the given parser.

noneOf :: Stream s m Char => [Char] -> ParsecT s u m Char

As the dual of **oneOf**, **noneOf** cs succeeds

if the current character not in the supplied list of characters cs.

Returns the parsed character.

consonant = **noneOf** "aeiou"

https://stackoverflow.com/questions/44965/what-is-a-monad

Parser – **<|>** combinator

(<|>) :: (ParsecT s u m a) -> (ParsecT s u m a) -> (ParsecT s u m a)

This combinator implements choice.

The parser p <|> q first applies p.

If it succeeds, the value of p is returned.

If p fails without consuming any input, parser q is tried.

https://stackoverflow.com/questions/44965/what-is-a-monad

Strictness declaration (1)

strictness declaration

it must be evaluated to what's called "weak normal head form"

when the data structure value is created.

```
data Foo = Foo Int Int !Int !(Maybe Int)
```

```
f = Foo (2+2) (3+3) (4+4) (Just (5+5))
```

The function **f** above, when evaluated, will return a "thunk":

that is, the code to execute to figure out its value.

At that point, a Foo doesn't even exist yet, just the code.

delayed computation

https://stackoverflow.com/questions/993112/what-does-the-exclamation-mark-mean-in-a-haskell-declaration

Strictness declaration (2)

```
data Foo = Foo Int Int !Int !(Maybe Int)
f = Foo (2+2) (3+3) (4+4) (Just (5+5))
```

But at some point someone may try to look inside it

```
case f of
    Foo 0 _ _ _ -> "first arg is zero"
    _           -> "first arge is something else"
```

This is going to execute enough code to do what it needs

So it will create a **Foo** with four parameters

The first parameter, we need to evaluate all the way to 4,

where we realize it doesn't match.

https://stackoverflow.com/questions/993112/what-does-the-exclamation-mark-mean-in-a-haskell-declaration

Strictness declaration (3)

```
data Foo = Foo Int Int !Int !(Maybe Int)
f = Foo (2+2) (3+3) (4+4) (Just (5+5))
```

The second parameter doesn't need to be evaluated, because we're not testing it. Thus, instead of storing the computation Results 6, store the code (3+3) that will turn into a 6 only if someone looks at it.

The third parameter, however, has a **!** in front of it, so is ***strictly evaluated***: (4+4) is executed, and 8 is stored in that memory location.

https://stackoverflow.com/questions/993112/what-does-the-exclamation-mark-mean-in-a-haskell-declaration

Strictness declaration (4)

```
data Foo = Foo Int Int !Int !(Maybe Int)
f = Foo (2+2) (3+3) (4+4) (Just (5+5))
```

The fourth parameter is also ***strictly evaluated***.

we're evaluating not fully, but only to **weak normal head form**.

figure out whether it's **Nothing** or **Just** something,

and store that, but we go no further.

That means that we store not **Just 10** but actually **Just (5+5)**,

leaving the thunk inside unevaluated.

https://stackoverflow.com/questions/993112/what-does-the-exclamation-mark-mean-in-a-haskell-declaration

Async Monad Examples

```
let AsyncHttp(url:string) =
    async {  let req = WebRequest.Create(url)
             let! rsp = req.GetResponseAsync()
             use stream = rsp.GetResponseStream()
             use reader = new System.IO.StreamReader(stream)
             return reader.ReadToEnd() }
```

The **async {}** syntax indicates that the **control flow** in the block is defined by the **async monad**.

GetResponseAsync actually waits for the response on a separate thread, while the main thread returns from the function.

The last three lines are executed on the spawned thread when the response have been received.

In most other languages you would have to explicitly create a separate function for the lines that handle the response.

The **async monad** is able to "split" the block on its own and postpone the execution of the latter half.

https://stackoverflow.com/questions/44965/what-is-a-monad

Functors as containers

```
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**

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

Function application, Packaging, Flattening

fmap applies a **function** to a **value** in a container

return packages a **value** in a container

join flattens a container in containers

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

return :: a -> M a     packaging

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

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

>>= vs. fmap & join

(>>=) 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
```

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

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

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

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

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

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

Monad's lifting capability

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

Monads

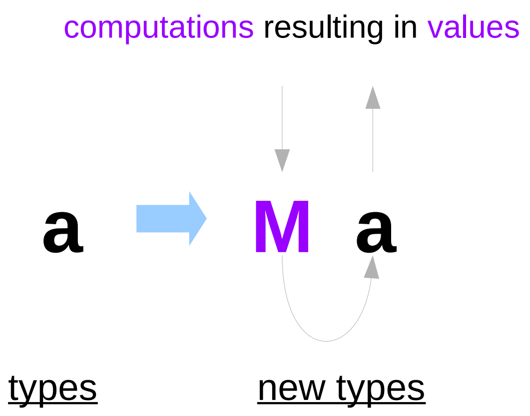
map types to new types

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

Control.Monad defines liftM

liftM transform a regular function

into a "computations that results in the value

obtained by evaluating the function."

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.

f :: a -> b

liftM f :: M a -> M b

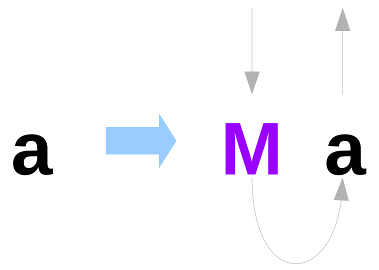
computations that results in the value obtained by evaluating the function

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

Monad – mapping a type and lifting a function

mapping a new type

Monads <u>map</u> types to new types
that represent "computations that result in values"
The function **return** <u>lifts</u> a plain *value* **a** to **M a**



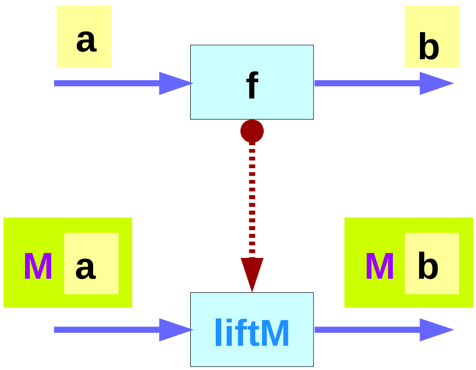
lifting function

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

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

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

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

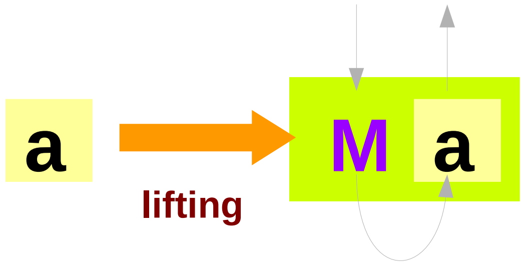
return x >>= k === k x;

foo >>= return

foo

return x >>= k

k x;



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

ap Function

Control.Monad defines **ap** function

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

Three Orthogonal Functions

Thinking of extraction : a slightly misleading intuition.

Nothing is being "extracted" from a monad.

The more *fundamental* definition of a monad

can be stated by three orthogonal functions:

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

return :: a -> m a

join :: m (m a) -> m a

m is a monad.

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

Three Orthogonal Functions and >>=

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

return :: a -> m a

join :: m (m a) -> m a

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

how to implement **(>>=)** with these:

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

your only option is using **fmap** to get something of type **m (m b)**,

(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**.

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

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

Monad Law

join :: m (m a) -> m a

<u>nothing</u> is being taken "out" of the monad

as the computation going <u>*deeper*</u> into the monad,

with successive steps being <u>*collapsed*</u>

into a single layer of the monad.

when **join** (m (m a) -> m a) is applied, it doesn't matter

as long as *the nesting order is preserved* (a form of *associativity*)

that the <u>*monadic*</u> <u>*layer*</u> introduced by **return**

does <u>*nothing*</u> (an *identity* value for **join**).

Left identity	**return a >>= f**	**f a**
Right identity	**m >>= return**	**m**
Associativity	**(m >>= f) >>= g**	**m >>= (\x -> f x >> g)**

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

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

References

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

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