Chapter 18 : Monad : Applicative functors

*) Monads can be thought of as an another way of applying functions over structure, with a couple of additional features.

*) defn: class Applicative m ⇒ Monad m where

```
(>>=) :: m a → (a → m b) → m b

(>>) :: m a → m b → m b

return :: a → m a
```

*) Applicative m: Monad is stronger than Applicative & Functor, and we can derive Applicative & Functor in terms of Monad. Just as we can derive Functor in terms of Applicative.

↓

This means that we can write fmap using monadic operations.

↓

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

a → b

[H]

↓ example

```
>> fmap (+1) [1..3]
   [2, 3, 4]
>> [1..3] >>= return . (+1)
   [2, 3, 4]
```

a → m a

|

↓

This happens to be a law. It is important to understand this chain of dependency:

Functor → Applicative → Monad.

i.e. when we implement an instance of Monad for a type, we necessarily have an Applicative and a Functor as well.

*) Core operations : three core ops, although only (>>=) needs to be defined for a minimally complete Monad instance.

(>>=) :: m a → (a → m b) → m b

(>>) :: m a → m b → m b

return :: a → m a : Same as pure. Takes a value & return it inside a structure, whether the structure is list, Just or IO

return. (+1)

↓

>> : Mr. Pointy (Some → sequencing operator)

sequences two actions while discarding any resulting value of the first action.

↓

>>= : bind operator, makes the Monad special.

(*) The novel part of Monad : Conventionally when we use monads, we use the bind fn. Sometimes directly, and sometimes indirectly via do syntax.

↓

```
<$> :: Functor f
    => (a → b) → f a → f b

<*> :: Applicative f
    => f (a → b) → f a → f b

>>= :: Monad f
    => f a → (a → f b) → f b
```

↓

The idea of mapping a fn. over a value while bypassing its surrounding structure is not unique to Monad.

↓

Can be demonstrated by fmapping a fn. of type $(a \to m\ b)$, to make it more like >>=.

↓

```
fmap :: Functor f
     => (a → f b) → f a → f (f b)
```

↓ example

```
>> let andOne x = [x, 1]
>> andOne 10
[10, 1]

>> :t fmap andOne [4, 5, 6]
fmap andOne [4, 5, 6] :: Num t => [[t]]

>> fmap andOne [4, 5, 6]
[[4, 1], [5, 1], [6, 1]]
```

↓

was clear from the type signature that we would reach

here, i.e. is an extra layer of structure. Now what if want to remove the extra layer of structure or to discard it in favour of one layer of that structure.

↓

can be done for lists using concat.

↓

```
concat $ fmap and one [4, 5, 6]
[4, 1, 5, 1, 6, 1]
```

↓

a less general type on concat:

```
concat :: [[a]] → [a]
```

↓

Monad in a sense is a generalisation of concat. The unique part of Monad is the following func.

```
import Control.Monad (join)

join :: Monad m ⇒ m (ma) → ma
```

↓

we can inject more structure with the std. fmap if needed, however the ability to flatten those two layers of structure into one, makes Monad special. And it's by putting join & mapping fn. together that we get the bind, also known as >>=

→ Exercise: bind in terms of fmap & join

```
bind :: Monad m => (a -> mb) -> ma -> mb
bind g x = join $ fmap g x

fmap :: (a -> b) -> fa -> fb.

        (a -> fb) -> fa -> f (fb)

join :: f (fb) -> f b
```

---

*) What monad is not ?

a) Impure. Monadic fns. are pure fns. IO is an abstract datatype, that allows for impure, or effectful, actions and it has a monad instance. But there's nothing impure about Monads.

b) An embedded lang. for imperative programming

↓

while monads are used for sequencing actions in a way that looks like imperative programming, there are also commutative monads that donot order actions.

c) A value. The typeclass describes a specific relationship between elements in a domain and define operations over them, same way we refer to monoids, functors or applicatives for that manner.

d) About strictness: Monadic ops of bind and return

are nonstrict. Some ops can be made strict within a specific instance.

*) Monad also lifts : includes some lift fns that are same as the ones we already seen in Applicative. Only present for compatability reasons, and were used before Applicatives were discovered.

↓

```
liftA :: Applicative f
      => (a -> b) -> f a -> f b

liftM :: Monad m
      => (a1 -> r) -> m a1 -> m r
```

↓

similarly for liftA2 & liftM2, same for liftA3 & liftM3.

↓

zipWith is liftA2 or liftM2 specialized to lists.

↓

```
>> :t zipWith
zipWith :: (a -> b -> c)
        -> [a] -> [b] -> [c]

>> zipWith (+) [3,4] [5,6]
[8,10]

>> liftA2 (+) [3,4] [5,6]
[8,9,9,10]
```

Different results, although same types, is because of the different monoids being used.

→ Do Syntax and monads: do syntax as introduced in modules chapter was being used in the context of IO as syntatic sugar that allowed us to easily sequence actions by feeding the result of one action as the i/p value to the next.

↓

```
(*>) :: Applicative f => fa -> fb -> fb

(>>) :: monad m => ma -> mb -> mb
```

(should in all cases do the same thing, just the underlying constraints are different)

↓

```
>> putStrLn "Hello, " >> putStrLn "World!"
Hello,
World!
```

↓ do disugared using (>>) & (*>)

```
sequencing :: IO ()
sequencing = do
  putStrLn "blah"
  putStrLn "another-thing"
```

```
sequencing' :: IO ()
sequencing' =
  putStrLn "blah" >>
  putStrLn "ano"
```

↓ variable binding ex

```
binding :: IO ()
binding = do
  name <- getLine
  putStrLn name
```

binds it to a variable

```
binding' :: IO ()
binding' = do
  getLine >>= putStrLn
```

passes it directly to putStrLn (fn).

*) when fmap alone isn't enough : fmap putStrLn over getLine won't do anything.

putStrLn <$> getLine

↓

This does look for an input, but does not print it upon entering. i.e. putStrLn doesn't successfully map over getLine.

↓

```
>> : t  putStrLn <$> getLine
putStrLn <$> getLine :: IO (IO ())
```

↓

Now getLine :: IO String, whereas

putStrLn :: String → IO ()

↓

in terms of fmap :

<$> :: Functor f ⇒ (a → b) → f a → f b

Now, putStrLn :: String → IO ()

a → b

(a → f b) → f a → f (f b)

↓

gets specialized to IO ()

↓

This situation is similar to the situation when we used fmap to map a fn. with type a → m b instead of a → b,

↓

(String → IO ()) → IO String → IO (IO ())

effects that would be performed if putStrLn

was evaluated

represents the effects getline must perform to get us a string that the user typed in.

The unit that putStrLn returns.

*) we are able to evaluate IO actions multiple times.

*) we can refer to, compose & map over effectful computations. without performing them. example:

```
>> let printOne = putStrLn "1"
>> let printTwo = putStrLn "2"
>> let twoActions = (printOne, printTwo)
>> :t twoActions
twoActions :: (IO (), IO ())
```

*) To solve the issue with getline & putStrLn, we can use join

```
join $ putStrLn <$> getline
```

↓

monadic actions are pure, and the sequencing operations we use here are ways of nesting lambdas.

↓

Sometimes it is valuable to suspend or otherwise perform an I/O action until some determination is made, so types like IO (IO ()) aren't necessarily invalid.

*) Back to desugaring do syntax

↓

```
bindingAndSequencing :: IO()
bindingAndSequencing = do
    putStrLn "name pls:"
    name <- getLine
    putStrLn ("y helo thar: " ++ name)
```

String ← getLine → IO String

part of do syntax

```
bindingAndSequencing' :: IO()
bindingAndSequencing' =
    putStrLn "name pls:" >>
    getLine >>
    \name -> putStrLn ("y halo thar: " ++ name)
```

>> → sequencing

won't be able to use '←' → do avoids

*) As the nesting intensifies, do syntax can make things a bit cleaner and easier to read.

→ Examples of Monad use

*) list

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

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

return :: a -> [a]
```

↓

can generate more list (or an empty list) inside of our mapped fn, examples.

↓

```
twiceWhenEven :: [Integer] -> [Integer]
twiceWhenEven = do
  x <- xs
  if even x
  then [x*x, x*x]
  else [x*x]
```

→ binds individual values out of the list i/p, like for list comprehension

*) Maybe monad

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

↓

```
data Cow = Cow {
    name :: String
  , age :: Int
  , weight :: Int
} deriving (Eq, Show)
```

```
noEmpty :: String -> Maybe String
noNegative :: Int -> Maybe Int
weightCheck :: Cow -> Maybe Cow
```

```
mkSphericalCow :: String -> Int -> Int -> Maybe Cow
mkSphericalCow name' age' weight' =
  case noEmpty name' of
    Nothing -> Nothing
    Just nammy ->
      case noNegative age' of
        Nothing -> Nothing
        Just aggy ->
          case noNegative weight' of
```

Nothing → Nothing

Just weighty →

weightCheck (Cow

nammy aggy weighty)

↓ cleaning up using 'do'

```
mkSpecialCow name' age' weight' = do
    nammy ← noEmpty name'
    aggy ← noNegative age'
    weight ← noNegative weight'
    weightCheck (Cow nammy aggy weight)
```

↓ >>=

```
mkSpecialCow' name' age' weight' =
    noEmpty name' >>=
      \nammy →
        noNegative age' >>=
          \agey →
            noNegative weight' >>=
              \weighty →
                weightCheck (Cow nammy agey weighty)
```

produces Maybe String but nammy gets bound to String.

m x >>= k = k x

String → Maybe String

String

↓

we can't do the above using Applicative. because our weightCheck fn. depends on the prior existence of a Cow value and returns more monadic structure in its return type Maybe Cow. The prior existence of Cow is also dependent on other arguments to the fn. the sequential processing

of which yields a cow.

```
doSomething = do
  a ← f
  b ← g
  c ← h
  pure (a, b, c)
```

⇒ This can be rewritten using Applicative

f (a → b) → f a → f b

↓ but

```
doSomething = do
  a ← fn
  b ← g a
  c ← h b
  pure (a, b, c)
```

→ g and h are producing monadic structure that can only be obtained by depending on values generated from monadic structure.

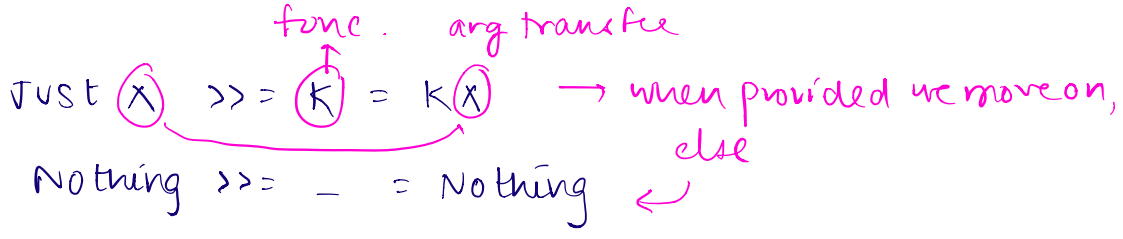
```
f :: Integer → Maybe Integer
f 0 = Nothing
f n = Just n

g :: Integer → Maybe Integer
g i
  | even i = Just (i+1)
  | otherwise = Nothing

h :: Integer → Maybe String
h i = Just ("10191" ++ show i)

doSomething' n = do
```

```
a ← fn
b ← g a
c ← h b
pure (a, b, c)
```

*) With the Maybe applicative, each Maybe computation fails or succeeds independently of each other. Lifting funcs. that are also Just or Nothing over Maybe values.

*) With the Maybe monad, computations contributing to the final computation can choose to return Nothing based on previous computations.

*) On providing a Nothing value on mkSphericalCow', we get Nothing straightaway due to the following defn:



*) Nothing never even gets used as indicated above and this can be demonstrated using:

```
>> Nothing >>= undefined
Nothing
```

*) We use Maybe monad & applicative because they make the computation convenient to code, and more readable as well.

→ Either:

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

```
(>>=) :: Either e a
      -> (a -> Either e b)
      -> Either e b
```

```
return :: Monad m => a -> m a

return ::           a -> Either e a
```

part of the structure

```
module EitherMonad where

type Founded = Int
type Coders = Int

data SoftwareShop =
    Shop {
        founded :: Founded
      , programmers :: Coders
    } deriving (Eq, Show)

data FoundedErr = -- type of associated error
    NegativeYearsFounded
  | TooManyYearsFounded
  | NegativeCoders Coders
```

```
  | TooManyCoders Coders
  | TooManyCodersForYears FoundedCoders
  deriving (Eq, Show)

validateFounded :: Int -> Either FoundedError Founded
validateCoders :: Int -> Either FoundedError Coders
mkSoftware :: Int -> Int -> Either FoundedErrors Coders
mkSoftware years coders = do
    founded <- validateFounded years
  programmers <- validateCoders coders
    if programmers > div founded 10
      then Left $
                TooManyCodersForYears FoundedProgrammers
      else Right $ Shop Founded programmers
```

*) so there is no monad for validation. Applicative & monad instances must have the same behaviour. expressed in the form:

```
import Control.Monad (ap)
(<*>) == ap
```

(way of saying that the Applicative apply for a type must not change behaviour if derived from the Monad instance's bind operation)

```
<*> :: Applicative f
    => f (a -> b) -> fa -> fb
```

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

↓

Deriving applicative from the stronger instance :

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

ap m m' = do
    x <- m
    x' <- m'
    return (x x')
```

↓

We can't have a Monad instance of validation that accumulates errors like the Applicative does. Instead any Monad instance for validation would be identical to Either's Monad instance.

Exercise : Either Monad

```
data Sum a b =
    First a
  | Second b
  deriving (Show)
```

`f (a -> b) -> f a -> f b`

```
instance Functor (Sum a) where
  fmap f (Second b) = Second (f b)
  fmap _ (First a) = First a
```

```
instance Applicative (Sum a) where
  pure b = Second b.
  (Second f) <*> (Second b) = Second (f b)
  _          <*> (First a)  = First a
  (First a)  <*> _          = First a
```

→ won't be applicative values or functions, would be monoidal values.

```
instance Monad (Sum a) where
  return = pure
  Second b >>= f = f b
  First a  >>= _ = First a
```

→ Monad laws :

*) Identity laws → i) `m >>= return = m` (right identity)

ii) `return x >>= f = f x` (left identity)

like pure, return shouldn't change any of the behaviour of the rest of the fn. It is only there to put things into structure when we need to, and the existence of the structure should not affect the computation.

*) Associativity laws → The grouping of the fns. should not have any impact on the final result.

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

↓

Keeping in mind that ">>=" allows the result value of

one fn. to be passed as i/p to the next, when we reassociate the fns, we need to apply f so that g has an i/p value of type m a to start the whole thing off.

→ Application & Composition

*) with Monad the composition looks less neat at first.

```
mcomp :: Monad m
      ⇒ (b → mc)
      → (a → mb)
      → (a → mc)
mcomp f g a = f (g a)
```

→ will give an error, couldn't match b with mb.

↓

f expects b, but g is passing a type of mb to f.

m (mb) → mb

↙

Applying a fn. in presence of some context, structure:

fmap.

↓

that will further give us m (mc) at the end

↓

to resolve that we use join (m(mc) → mc)

↓

```
mcomp f g a = join (f <$> (g a))
```

↓ joining fmap & join & using >>=

```
mcomp f g a = (g a) >>= f
```

*) nothing special needs to be written to make monadic functions compose (as long as the monadic contexts are the same Monad)

↓

Kleisli composition

↓

```
(.) :: (b → c) → (a → b) → a → c
```

```
(>>=) :: Monad m
      ⇒ m a → (a → m b) → m b
```

↓

Kicking Kleisli composition off the ground, flip some arguments around to make types work:

↓

```
(>=>) :: Monad m
      ⇒ (a → m b) → (b → m c) → a → m c
```

Kleisli comp. op.

↓

function composition with monadic structure hanging off the fones we're composing.

↓

```
import Control.Monad ((>=>))

sayHi :: String → IO String

sayHi greeting = do
    putStrLn greeting
```

getline

```
readM :: Read a => String -> IO a
readM = return . read
```

→ provides monadic structure after being bound over the output of sayHi

```
getAge :: String -> IO Int
getAge = sayHi (>=>) readM

askForAge :: IO Int
askForAge = getAge "Age ?"
```

(a → mb)

sayHi :: String → IO String

The String i/p from the user that sayHi returns.

↳ IO sayHi performs to get i/p.

(b → mc)

readM :: String → IO a

The string that read expects, and which sayHi will produce

return/pure perform no IO values.

(a → mc)

getAge :: String → IO a → int that is finally returned on composition

↕

greeting for the user

Combined IO action which performs all effects necessary to produce the final result.