Chapter 17 : Applicative : monoidal functors

Apply
<*>

*) Applicative typeclass allows for fn. appln. lifted over structure. But with applicative the fn. applied is also embedded in some structure.

→ Defining Applicative

every type that has Applicative instance must have a functor instance

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

↓

*) pure lifts something into functorial (applicative) structure. Bare minimum bit of structure or structural identity.

) <> : referred to as "apply" or "ap"

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

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

f → point of difference

*) Along with core functions, Control.Applicative library provides fns. like liftA, liftA2 & liftA3.

```
liftA  :: Applicative f => (a -> b) -> f a -> f b
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
```

liftA3 :: Applicative f ⇒ (a → b → c → d) →

fa → fb → fc → f d

Similar to fmap but with more than 1 arg to the function being lifted & mapped.

→ functor vs Applicative

*) applicatives are monoidal functors.

*) fmap :: (a → b) → fa → fb

<*> :: f(a → b) → fa → fb

) fmap f x = pure f <> x

↓

lifts the function's structure to the level of x

↓

can be thought of as a means of embedding a value of any type in the structure we're working with.

↓

```
>> pure 1 :: [Int]
[1]
>> pure 1 : Maybe Int
Just 1
>> pure 1 : Either a Int
Right 1
```

left side handled differently than the right, in a manner similar to that of a functor

↓

left part is a part of the structure, hence can't be touched.

→ Applicative functors are monoidal functors

```
($) :: (a → b) → a → b
<$> :: (a → b) → f a → f b
<*> :: f (a → b) → f a → f b
```

→ acts as a nice proxy for ordinary fn. apphn in its type

↓

function is now also embedded in a functorial structure

lifting our fn. over the structure, f wrapped around our value f then applying the fn. to the value

---

*) monoidal part in monoidal functor

in `f (a → b) → f a → f b`, the two args are:

i) `f (a → b)`

ii) `f a`

↓

we can't ignore the 'f' here & just apply the fn. to 'a' to get 'b' & leaving the 'f' untouched like we used to do with fmap, since there we didn't have a structure associated to a fn. But here we do & we need to unite them to get fb & not just b wrapped around by an untouched f structure. i.e. both the structure & the values inside get modified but the type

of the functor remains same. This is where monoid comes in. Applying an operation over values of the same type & giving an o/p of the same type. The monoidal behaviour occurs over the function structure & functorial over the values.

↓

Provided f is a type with a monoid instance.

↓

<$>	f	f	f
$	a → b	a	b
<*>	f (a→b)	f a	f b

↓

In another words, we're enriching the function application with the very structure, we were previously mapping over with the functor. Examples

↓

```
>> [(*2), (*3)] <*> [4, 5]

[2*4, 2*5, 3*4, 3*5]
[8, 10, 12, 15]
```

↓

Although the actual appln. of each (a→b) to a value of type a is quite ordinary, we now have a list of functions rather than a single fn.

↓

But list of fns. can be applied to only values wrapped in list structure.

More examples :

i) Just (*2) <*> Just 2

Just 4

ii) Just (*2) <*> Nothing

Nothing

iii) Nothing <*> Just 2

Nothing

with the applicative going a structure to a fn., the fn. might not even be provided in cases where the structure is a list or maybe, etc

*) Show me the monoids

```
>> fmap (+1) ("blah", 0)
("blah", 1)
```

The two-tuple demonstrates the monoid in applicative to us.

```
instance Monoid a => Applicative ((,) a)
instance (Monoid a, Monoid b) => Monoid (a, b)
```

In applicative instance of tuple, we don't need a monoidal for the b because we're using the fn. appln. to produce the b. But for the first type we still need the monoidal behaviour because we have two values

and need to somehow turn that into one value of the same type.

↓

("Woo", (+1)) <*> ("Hoo!", 0)

("Woo Hoo!", 1)

↓ result of monoidal appn.

↳ fn. applied from b value on left to b value on right

↓

>> (Sum 2, (+1)) <*> (Sum 0, 0)

(Sum {getSum = 23}, 1)

↓

>> (All True, (+1)) <*> (All False, 0)

(All {getAll = False}, 1)

↓

Doesn't really matter what Monoid, but we need some way of combining or choosing our values.

Tuple monoid & applicative instances:

```
instance (Monoid a, Monoid b) => Monoid (a,b) where
  mempty = (mempty, mempty)
  mappend (a, b) (a', b') = (a `mappend` a',
                             b `mappend` b')
```

```
instance (Monoid a) => Applicative ((,) a) where
  pure x = (mempty, x)   : providing structure to fn
  (u, f) <*> (v, x) = ((u `mappend` v), (f x))
```

→ Maybe Monoid & Applicative : Monoid & Applicative instances aren't required or guaranteed to have the same monoid of structure, and the functorial part may change the way it behaves.

↓

```
instance Monoid a => Monoid (Maybe a) where
  mempty = Nothing
  mappend m Nothing = m
  mappend Nothing m = m
  mappend (Just a) (Just a') = Just (mappend a a')

instance Applicative (Maybe a) where
  pure f = Just f
  Nothing <*> _ = Nothing
  _ <*> Nothing = Nothing
  Just f <*> Just a = Just (f a)
```

→ Applicatives in use

→ List Applicative

```
(<*>) :: f (a -> b) -> f a -> f b
```

```
(<*>) :: [] (a -> b) -> [] a -> [] b
```

```
pure :: a -> f a
pure :: a -> [] a
```

↓

can be done in ghci using `:set -XTypeApplications`

```
>> :t (<*>) @[]
(<*>) @[] :: [a -> b] -> [a] -> [b]
>> :t (pure) @[]
pure @[] :: a -> [a]
```

↓

With the list applicative, we're mapping a plurality of fns. over a plurality of values:

```
>> [(+1), (*2)] <*> [2, 4]
[3, 5, 4, 8]
```

structure remains same

types are as defined by the fn, but the type signature says nothing about how the structure should play or behave, till the time it is of the same type i.e. even the structure's value is changing.

↓

Now, each fn. value mapped to all values in the second list, and returns one list. The part that it doesn't return two lists or a nested list or some other configuration in which both structures are preserved is the monoidal part.

↓

↓

the reason we don't have a list of functions concatenated separately with a list of values is the fn. appln part.

↓

tuple constructor with list Applicative : infix operator for fmap to map tuple constructor over the first list embedding an unapplied function into some structure, returning a list of partially applied fns. The applicative operator will then be used to get the final structure

↓

```
>> (,) <$> [1,2] <*> [3,4]
[(1,3),(2,3),(1,4),(2,4)]
```

↓

liftA2 gives us another way to look at this

```
liftA2 (,) [1,2] [3,4]
```

↓

```
>> (+) <$> [1,2] <*> [3,5]
liftA2 (+) [1,2] [3,5]
```

```
>> :t lookup
lookup :: Eq a => a -> [(a,b)] -> Maybe b
>> let l = lookup 3 [(3,"hello")]
>> l
Just "hello"
```

```
>> fmap length $ l
  5
>> let c (x:xs) = toupper x : xs
>> fmap c $ l
  Just "Hello"
```

Nothing, if key not present

↑

↓

So lookup searches inside a list of tuples to find the first tuple that matches the argument & returns the paired value wrapped inside a Maybe context.

↓

Now that we have values wrapped in Maybe context, perhaps we'd like to apply some fns. to them. This is where we want applicative operations.

```haskell
import Control.Applicative

f x =
  lookup x [ (3, "hello")
           , (4, "julie")
           , (5, "kbai")]
g y =
  lookup y [ (7, "sup?")
           , (8, "chris")
           , (9, "aloha")]

h z =
  lookup z [(2, 3), (5, 6), (7, 8)]
m x =
  lookup x [(4, 10), (8, 13), (1, 9001)]
```

↓

look things up and add them together

↓

```
>> f 3
Just "hello"
```

```
>> g 8
Just "chris"

>> (++) <$> f 3 <*> g 7
Just ("hello"++) <*> Just ("sup?")
Just ("hellosup?")

>> (+) <$> h 5 <*> m1
(+) <$> Just 6 <*> Just 9001
Just (6+) <*> Just 9001
Just (9007)

>> (+) 2 <$> h 5 <*> m 6
(+) <$> Just 6 <*> Nothing
Just (6+) <*> Nothing
Nothing
```

↓ doing the same things with liftA2

```
>> liftA2 (+) h5 m6
Nothing

>> liftA2 (+) h5 m1
9007
```

↓

Your applicative context can sometimes also be IO.

Example. `(++) <$> getLine <*> getLine`

`(,) <$> getLine <*> getLine`

→ Exercises : Lookups

pure
<$>
<*>

1. added :: Maybe Integer

added = (+3) <$> (lookup 3 $ zip [1,2,3] [4,5,6])

or

(+) <$> Just 3 <*> (lookup 3 $ zip [1,2,3] [4,5,6])

2. y :: Maybe Integer

y = lookup 3 $ zip [1,2,3] [4,5,6]

Just 6

z :: Maybe Integer

z = lookup 2 $ zip [1,2,3] [4,5,6]

Just 5

tupled :: Maybe (Integer, Integer)

tupled = (,) y z

= (,) <$> y <*> z

3. x :: Maybe Int

x = elemIndex 3 [1,2,3,4,5]

Just 2

y :: Maybe Int

y = elemIndex 4 [1..5]

Just 3

max' :: Int → Int → Int

```
max' = max

maxed :: Maybe Int
maxed = max' <$> x <*> y
```

---

*) Identity : way to introduce structure without changing the semantics of what we're doing.

```
type Id = Identity

(<*>) :: f (a -> b) -> f a -> f b
(<*>) :: Id (a -> b) -> Id a -> Id b

pure :: f a
pure :: Id a
```

↓

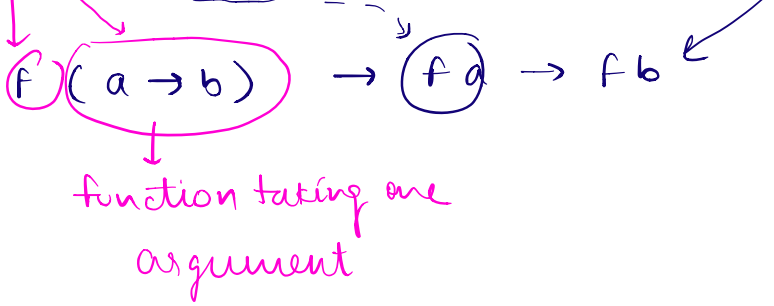
```
>> let xs = [1..3]
>> let xs' = [9, 9, 9]
>> const <$> xs <*> xs'
   [const 1 , const 2 , const 3] <*> [9,9,9]
   [1, 1, 1, 2, 2, 2, 3, 3, 3]
>> mkId = Identity
>> const <$> mkIdentity xs <*> mkIdentity xs'

const <$> Id [1,2,3] <*> Id [9 9 9]
```

Id (const [1,2,3]) <*> (Id [999])

Id const [1,2,3] [999]

Id [1,2,3]



↓

Having this extra bit of structure around our values lifts the const fn, from mapping over the lists to mapping over the Identity.

→ Exercise : Identity Instance

Identity a <*> Identity

Applicative instance for Identity

fmap

```
newtype Identity a = Identity a
                     deriving (...)

Instance Functor Identity where
    fmap f (Identity a) = Identity f a

Instance Applicative Identity where
    pure = Identity
    Identity f <*> Identity a = Identity (f a)
```

→ Constant : similar to Identity, except this not only provides structure, but it also acts like the const func. sort of throws away function application.

↓

```
type C = Constant
(<*>) :: f (a → b) → f a → f b
(<*>) :: C (a → b) → C a → C b
```

```
pure :: a → f a
pure :: a → Ca          constant sum 1
```

↓

```
>> let f = Constant (Sum 1)
>> let g = Constant (Sum 2)
>> f <*> g
Constant {getConstant = Sum {getSum = 3}}
```

b part is the non structural part where function could have been stored, but that is phantom

↓

can't do anything because it can only hold one value. function doesn't exist, and the b is a ghost. Monoid operation is performed between the terms inside. Value is part of the structure, and structure modifications are done through monoids.

Exercise : Constant Instance

```
newtype Constant a b = Constant {getConstant :: a}
  deriving()

Instance Functor (Constant a) where
  fmap _ (Constant {getConstant = a}) =
    Constant {getConstant = a}

Instance Monoid a ⇒ Applicative (Constant a) where
  pure b = Constant {getConstant = mempty}
  (Constant {getConstant = a}) <*>
    (Constant {getConstant = b}) =
      Constant {getConstant = a <> b}
```

→ Maybe Applicative : When f is Maybe, we are even taking into account the possibility that the fn. itself might not exist.

↓

Using Maybe Applicative

↓

```
validateLength :: Int → String → Maybe String
validateLength maxLen s =
    if (length s) > maxLen
    then Nothing
    else Just s

newtype Name = Name String deriving (Eq)
newtype Address = Address String deriving (Eq)

mkName :: String → Maybe Name
mkName s =
    fmap Name $ validateLength 25 s

mkAddress :: String → Maybe Address
mkAddress a =
    fmap Address $ validateLength 100 a
```

↓

Smart constructor for a person

↓

```
data Person = Person Name Address
              deriving ()

mkPerson :: String → String → Maybe Person

mkPerson n a =
  case mkName n of
    Nothing → Nothing
    Just n' →
      case mkAddress a of
        Nothing → Nothing
        Just a' →
          Just $ Person n' a'
```

↓

we leveraged fmap in the single case of mkName & mkAddress, but when we do the same with mkPerson, we run into issues:

```
fmap (fmap Person (mkName "Babe"))
     (mkAddress "old macdonald's")
```

↓

The problem here is that our (a → b) is hiding inside Maybe once it was lifted, and that does not match the agreed upon type of fmap. fmap accepts only fns.

↓

We need to be able to map a function embedded in our f. This is where Applicative helps us.

since <*> :: Functor f ⇒ f (a → b) →
f a → f b

↓

```
>> let s = "old macdonald's"
>> let addy = mkAddress s
>> let n = "Babe"
>> let b = mkName n
>> Person <$> b <*> addy
```

↓

<$> used to lift Person over the structure, but then it hides behind the structure & hence then we had to employ <*> to apply our embedded fn.

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

noEmpty :: String → Maybe String
noEmpty "" = Nothing
noEmpty s = Just s
```

```
noNegative :: Int -> Maybe Int
noNegative n
  | n >= 0    = Just n
  | otherwise = Nothing

cowFromString :: String -> Int -> Int -> Maybe Cow
cowFromString name' age' weight' =
  Cow <$> (noEmpty name')
      <*> (noNegative age')
      <*> (noNegative weight')
```

(a → b → c → d) →
f a → f b → f c → f d

↓ can also use liftA3

Here Cow :: * → * → * → * (a → b → c → d)

f :: Maybe
a :: String
b :: Int
c :: Int
d :: Cow

↓

```
liftA3 Cow (noEmpty name')
           (noNegative age')
           (noNegative weight')
```

↓

Basic motivation behind Applicative: I want to do something kinda like an fmap, but my func. is embedded in the functorial structure too, not only the value I want

to apply my fn. to.

↓

with Maybe, we're just saying that there exists a possibility that we don't have a function at all

→ Exercise: fixer upper

1. const <$> Just "Hello" <*> (pure "World" :: Maybe String)

2. (,,,) Just 90

<*> Just 10 Just "Tirenese" [1,2,3]

(,,,) <$> Just 90 <*> Just 10 <*> Just "Tirenese"

<*> (pure [1,2,3] :: Maybe [Int])

→ Applicative Laws : ① Identity

pure id <*> v = v

pure id <*> [1..5] = [1,2,3,4,5]

↓

Similar to the identity for functor,

↓

Comparing

↓

fmap id [1..5]

pure id <*> [1..5]

id [1..5]

all three should return the same answer

pure → embedding our id function

(2) Composition :

```
pure (.) <*> u <*> v <*> w =

u <*> (v <*> w)
```

↓

States that the result of composing our functions first and then applying them and the result of applying the functions first and then composing them should be the same. example

```
pure (.) <*> [(+1)] <*> [(*2)] <*> [1..3]

[(+1)] <*> ([(*2)] <*> [1..3])
```

(3) Homomorphism : a structure preserving map b/w two algebraic structures.

↓

The effect of applying a fn. that is embedded in some structure to a value that is embedded in some structure should be same as applying a fn. to a value w/o affecting any outside structure

```
pure f <*> pure x = pure (f x)

pure (+1) <*> pure 1 :: Maybe Int

pure ((+1) 1) :: Maybe Int
```

④ Interchange : u <*> pure y = pure ($ y) <*> u

To the left of <*>, must always be a fn. embedded in some structure, and in the above dyn. u represents a function embedded in some structure. for example.

Just (+2) <*> pure 2

Just 4

pure raises 2 to Maybe type due to the type on the left

for the right side → pure ($ y) <*> u

By sectioning the $ fn. appln. operator with the y, we create an environment in which the y is there, awaiting a fn. to apply to it.

↓ example

pure ($ 2) <*> Just (+2)

>> :t ($ 2)

($ 2) :: Num a ⇒ (a → b) → b

>> Just (+2)

Just (+2) :: Num a ⇒ Maybe (a → a)

→ Quickchecking Applicatives : checkers is a library that provides some nice properties and utilities for Quickcheck.

↓

```
module BadMonoid where

import Data.Monoid
import Test.QuickCheck
```

```
import Test.QuickCheck.Checkers
import Test.QuickCheck.Classes

data Bull = Fools
          | Twoo
          deriving (Eq, Show)

instance Arbitrary Bull where
  arbitrary =
    frequency [(1, Fools)
              , (1, Twoo)]

instance Monoid Bull where
  mempty = Fools
  mappend _ _ = Fools

instance EqProp Bull where (=-=) = eq

main :: IO()
main = quickBatch (monoid Twoo)
```

↓

don't have to define the Monoid laws as Quickcheck properties ourselves as they are bundled into a TestBatch called monoid.

↓

we need to define EqProp for our custom datatype. This is straightforward because checkers exports a fn. called eq. which reuses the pre-existing Eq instance for the datatype.

↓

we're sending a value of one type to monoid so that

it knows which Arbitrary instance to use to get the random values. Note: This value isn't the only one being used.

The type of the TestBatch which validates App. instances is a bit gnarly

```
applicative
:: ( Show a, Show (m a), Show (m (a -> b))
   , Show (m (b -> c)), Applicative m
   , CoArbitrary a, EqProp (m a)
   , EqProp (m b), EqProp (m c)
   , Arbitrary a, Arbitrary b
   , Arbitrary (m a)
   , Arbitrary (m (a -> b))
   , Arbitrary (m (b -> c)))
   => m (a, b, c) -> TestBatch
```

Testing a pre-existing Applicative instance like List or Maybe.

*) A trick to manage a An like this. We know that it's going to want Arbitrary instances for the Applicative structure, functions (from a → b, b → c) embedded in that structure, and also that it wants EqProp instances. That's all well & good, but we can ignore that.

m (a, b, c) → TestBatch is the thing we care about. We could pass an actual value giving us our Applicative structure & three values which could be of different type, but don't have to be. We can even pass a bottom with a type assigned to let it know what to randomly generate for validating the Applicative instance

```
>> let xs = [("b", "w", 1)]
>> quickBatch $ applicative xs
```

identity ✓

composition ✓

homomorphism ✓

interchange ✓

functor ✓

Again it's not running on the value being passed, rather inferring the type from it & then generating values for it accordingly.

*) A bottom example with type assigned

>> type SSI = [(String, String, Int)]

>> let trigger :: SSI

>> trigger = undefined.

>> quickBatch $ applicative trigger

→ ZipList Monoid : Another way to monoidically combine lists, other than concatenation : ZipList Monoid

↓

combines the values of two lists as parallel sequences using a monoid provided by the values themselves to get the job done.

↓

```
[1,2,3] <> [4,5,6]

[ 1 <> 4
, 2 <> 5
, 3 <> 6
]
```

Just for repr. purposes.

↓

to make this work, we need to assert this to either Sum or Product.

↓

1 <> (2 :: Sum Integer)

↓

Sum {getSum = 3}

↓

Ziplist monoid not provided by Prelude, hence need to write one ourselves.

↓

<> <$> <*>

```
module Apl where

import Control.Applicative
import Data.Monoid
import Test.QuickCheck
import Test.QuickCheck.Checkers
import Test.QuickCheck.Classes.
```

has issues right now

```
instance Monoid a => Monoid (ZipList a) where
  mempty = ZipList []
  mappend = liftA2 mappend

instance Arbitrary a => Arbitrary (ZipList a) where
  arbitrary = ZipList <$> arbitrary

instance Arbitrary a => Arbitrary (Sum a) where
  arbitrary = Sum <$> arbitrary

instance Eq a => EqProp (ZipList a) where
  (=-=) = eq
```

↓

```
>> let zl = ZipList [1 :: Sum Int]
>> quickBatch $ monoid zl
```

leftIdentity ✗

rightIdentity ✗

associativity ✓

empty ziplist is the zero but not the identity here, since the mappend here with mempty will give empty list & not the same element sent as i/p.

↓

This is resolved as follows:

↓

```
instance Monoid a
  => Monoid (ZipList a) where
  mempty = pure mempty
  mappend = liftA2 mappend
```

List Applicative Exercise