Chapter 16 : functors

*) core concept remains the same about abstracting a common pattern & making certain laws regarding the same.

*) functor : pattern of mapping over a structure (remember fmap)

→ what is a functor? way to apply a func. over or around some structure, that we don't want to alter. Implemented using typeclass, just like Monoid.

↓

The defn:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

- class → to begin defn. of a typeclass
- Functor → name of the typeclass we intend to define
- f → reference to a type, with functorial structure
- f (in f a) → functor that takes an argument. i.e. f is a type that has an instance of the functor typeclass.

→ fmap : does the same thing with lists as map does, but generalises behaviour over different types with functorial structures. example

```
>> fmap (+1) (Just 1)
Just 2

>> fmap (10/) (4,5)
(4, 2.0) ?

>> let rca = Right "Chris Allen"
>> fmap (++ ", Esq.") rca
```

```
type E e = Either e
type C e = Constant e
type I = Identity
-- Functor f =>
fmap :: (a -> b) ->   f a ->   f b
     :: (a -> b) ->  [ ] a ->  [ ] b
     :: (a -> b) -> Maybe a -> Maybe b
     :: (a -> b) ->   E e a ->   E e b
     :: (a -> b) ->  (e,) a ->  (e,) b
     :: (a -> b) ->     I a ->     I b
     :: (a -> b) ->   C e a ->   C e b
```

Right "Chris Allen, Esq."

→ Let's talk about f, baby : f in the typeclass defn. of functor must be the same 'f' throughout an entire defn, and it must refer to the type that implements the typeclass.

) f must have the kind (→ *). This can be reasoned from the type signature of the fmap operation, wherein f is applied to a to yield a concrete type.

*) Shining star come into view :

```
:K (→)
(→) :: (* → * → *)
```

(get a concrete value when we apply it to two arguments i.e. i/p type & o/p type)

Example : class Else where

```
e :: b → f (g a b c)
```

b :: *

f :: * → *

g a b c is an arg, which will be a concrete value

g :: * → * → * → *

req three applns to yield a concrete type

>> class Impish v where

```
impossibleKind :: v → v a
```

→ won't work since GHC will notice that v sometimes has

arguments, sometimes not.

*) Just as GHC has type inference, it also has kind inference and just as it does with types, it can not only infer the kinds but also validate that they're consistent & make sense.

*) Functor is fn. applⁿ.

↓

functor is a typeclass for function applⁿ. "over" or "through" some structure f that we want to leave untouched.

infix operator for fmap : <$>

↓

special version of $

↓

can be verified using :t

(will become * else)

↑

(* → *)

↑

*) data FixMe Pls a = FixMe
| Pls a
deriving (Eq, Show)

→ doesn't have the type arg.

instance Functor FixMe Pls where
fmap _ FixMe = FixMe
fmap f (Pls a) = Pls (f a)

→ fn. applied over f inside the structure

a → b f a f b → lines up perfectly

*) Typeclasses and constructor classes : The facility of being able to use typeclasses with higher-kind types gives us the means of talking about the contents of types independently from the type that structures those

contents. Hence we have something like fmap that allows us to alter the contents of a value without altering the structure, around the value (a list, or JUST)

→ functor laws : 2 basic laws : i) Identity

fmap id == id

↓

If we fmap the identity fn., it should return the same value as entrusted by identity. The outer structure shouldn't be changed. i.e.

fmap id "Hi Julie"

"Hi Julie"

ii) composition: fmap (f . g) = (fmap f . fmap g)

↓

The answer of mapping a fn. comp. should be same as when they were composed after individual mapped fn. example:

```
>> fmap ((+1) . (*2)) [1..5]
[3, 5, 7, 9, 11]

>> fmap (+1) . fmap (*2) $ [1..5]
[3, 5, 7, 9, 11]
```

*) Structure preservation: functions must be structure preserving.

→ The Good, the Bad & the ugly :

·) law abiding & law-breaking functor instances.

*) Example for identity

```
data WhoCares a =
    ItDoesnt
  | Matter a
  | WhatThisIsCalled
  deriving (Eq, Show)
```

↓ law abiding instance

```
instance Functor WhoCares where
  fmap _ ItDoesnt = ItDoesnt
  fmap _ WhatThisIsCalled = WhatThisIsCalled
  fmap f (Matter a) = Matter (f a)
```

↓ law breaking instance (breaks identity law)

```
instance Functor WhoCares where
  fmap _ ItDoesnt = WhatThisIsCalled
  fmap _ WhatThisIsCalled = ItDoesnt
  fmap f (Matter a) = Matter (f a)
```

*) Example for composition

```
data CountingBirds a = Heisenberg Int a
                       deriving (Eq, Show)

instance Functor CountingBird where
  fmap f (Heisenberg n a) =
    Heisenberg (n+1) fa
```

fmap :: Functor f ⇒ (a → b) → f a → f b

doesn't match up well.

↓

```
>> let u = "Uncle"
>> let oneWhoKnocks = Heisenberg 0 u
>> fmap (++ "Jesse") oneWhoKnocks
Heisenberg 1 "Uncle Jessie"
>> let f = ((++ "Jesse"). (++ "lol"))
>> fmap f oneWhoKnocks.
Heisenberg 1 "Uncle Lol Jesse"
```

↓

composition of fmap

↓

```
>> let j = (++ "Jesse")
>> let l = (++ "lol")
>> fmap j. fmap l $ oneWhoKnocks
Heisenberg 2 "Uncle lol Jesse"
```

↓

can be rectified easily be replacing (n+1) with n.

Better to think of anything not a part of the final type argument of our f in functor as being part of the structure, that the fns. being lifted should be oblivious to.

→ Commonly used functors

```
-- lms :: List (Maybe (String))
>> let ave = Just "ave"
>> let n = Nothing
>> let w = Just "wohoo"
>> let lms = [ave, n, w]
>> replaceWithP lms
'p'
>> :t replaceWithP lms
replaceWithP lms :: Char
```

-- List (Maybe (String)) → Char --

↓

-- o/p of replaceWithP is always same --

↓

```
>> fmap replaceWithP lms
"ppp"
```

-- fmap leaves the list structure intact around our list

```
>> :t fmap replaceWith lms
fmap replaceWithP lms :: [Char]
```

↓

Here replaceWithP's i/p is Maybe (String)

o/p is Char, ∴

fmap replaceWithP lms accomplishes

List [Maybe (String)) → List Char ~ String

↓

moving on

>> (fmap . fmap) replaceWithP lms.

fmap (fmap replaceWithP) lms.

fmap (fmap replaceWithP) [Just "Ave", Nothing, Just "woho"]

[fmap replaceWithP Just "Ave",
fmap replaceWithP Nothing,
fmap replaceWithP Just "woho"] fmap f (X

[Just "p", Nothing, Just "p"]

*) type checking for fmap . fmap

fmap fmap

(.) :: (b → c) → (a → b) → a → c

fmap :: Functor f ⇒ (m → n) → (f m → f n)

fmap :: Functor g ⇒ (x → y) → (g x → g y)

a

(.) :: ((m → n) → f m → f n) → (a → (m → n)) →
(a → (f m → f n))

b c

a → x → y

(m → n) → (g x → g y)

((g x → g y) → (f m → f n)) → (x → y → g x → g y)

→ ((x → y) → f m → f n)

$$((gx \to gy) \to (f(gx) \to f(gy)) \to ((x \to y) \to (gx \to gy))$$
$$\to (C(x \to y) \to f(gx) \to f(gy))$$

$$\boxed{(x \to y) \to f(gx) \to f(gy)}$$

$$\downarrow$$

$$(\text{Functor } f, \text{functor } g) \Rightarrow (x \to y) \to f(gx) \to f(gy)$$

*) fmap can be thought of taking $a \to b$ & bringing it one level up. to $f\,a \to fb$. i.e. mapping once lifts it once, and according behaviour.

$$a \to b$$
$$\downarrow$$
$$fa \to fb$$
$$\downarrow$$
$$g(fa) \to g(fb)$$

(*) Another lift

>> let trip fmap = fmap . fmap . fmap

>> trip fmap replacewith? lms.

[fmap . fmap Just "are",

fmap . fmap Nothing,

fmap . fmap Just "wohoo"]

lifts remember

↑

[Just fmap "are", Nothing fmap, Just fmap "wohoo"]

[Just "ppp", Nothing, Just "ppppp"]

*) let ha = Just ["Ha", "Ha"]

>> let lmls = [ha, Nothing, Just []]

>> (fmap . fmap) replaceWithP lmls. (2 levels)

[fmap replaceWithP ha, fmap replaceWithP Nothing,

fmap replaceWithP Just []]

↓

[Just 'p', Nothing, Just 'p']

>> let tripfmap = fmap . fmap . fmap

>> tripfmap replaceWithP lmls. (another level)

[fmap . fmap replaceWithP ha,

fmap . fmap replaceWithP Nothing,

fmap . fmap . replaceWithP Just []]

↓

[Just fmap . replaceWithP ["Ha", "Ha"],

Nothing, Just fmap replaceWithP []]

[Just ['p', 'p'], Nothing, Just ""]

>> tripfmap . fmap replaceWithP lmls. (another level)

[Just ["pp", "pp"], Nothing, Just ""]

*) summarizing the types :

replaceWithP :: [Maybe [Char]] → Char level 0

fmap . replaceWithP :: [Maybe [Char]] → [Char]

appln : level 1 Maybe [Char] → Char

(fmap . fmap) replaceWithP :: [Maybe [Char]] → [Maybe Char] level 0

level 1 : Maybe [Char] → Maybe Char

appln : level 2 : [Char] → Char

(fmap . fmap . fmap) replaceWithP :: [Maybe [Char]] → [Maybe [Char]] level 0

level 1 : Maybe [Char] → Maybe [Char]

level 2 : [Char] → [Char]

appln : level 3 : Char → Char

*) Lifting by removing one level of structure & then applying at final level.

→ Exercises : Heavy Lifting

1. a = fmap (+1) $ read "[1]" :: [Int]

2. b = (fmap . fmap) (++ "lol") (Just ["Hi", "Hello"])

3. c = (*2) . (\x → x - 2)

4. d = ((return '1' ++) . (show) . (\x → [x, 1..3]))

5. e :: IO Integer

   e = let ioi = readIO "1" :: IO Integer

   changed = fmap read $ fmap ("123"++)

   $ fmap show ioi

   allows us to lift over IO to the underlying field, and that is worked with

   in fmap (*3) changed

→ Transforming the unapplied argument

```
data Two a b = Two a b
             deriving ( )

data Or a b = First a
            | Second b
            deriving ( )
```

↓

similar to (,) & Either

↓

kind : * → * → * , which isn't compatible with functor (only structure with * → * are allowed)

↓

The kindedness of the type can be reduced by applying arguments.

↓

So to fix the kind incompatibility for our two f or types, we apply one of the arguments of each type constructor, giving us kind * → *.

↓

```
instance Functor (Two a) where
  fmap f (Two a b) = Two a (f b)
```

a → untouchable i.e. part of the structure

↓

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

→ Ignoring possibilities : The functor instances for either & maybe are handy for times you intend to ignore the left cases, which are typically our error or failure cases.

↓

Maybe :

```
incIfJust :: Num a => Maybe a -> Maybe a
incIfJust (Just x) = Just $ x + 1
incIfJust Nothing = Nothing
```

```
showIfJust :: Show a => Maybe a -> Maybe String
showIfJust (Just s) = Just $ show s
showIfJust Nothing  = Nothing
```

↓ using fmap

```
incMaybe :: Num a => Maybe a -> Maybe a
incMaybe m = fmap (+1) m

showMaybe :: Show a => Maybe a -> Maybe String
showMaybe m = fmap (show) m
```

↓

fmap has no reason to concern itself with the Nothing - there's no value there for it operate on, so all seems to be working well.

↓ another layer of abstraction, writing them without name arguments

```
incMaybe'' :: Num a => Maybe a -> Maybe a
incMaybe'' = fmap (+1)
```

(similarly for showMaybe)

↓

these don't have to specific to maybe, fmap acts over all functor instances.

↓

```
:t fmap (+1)
fmap (+1) :: (Functor f, Num a) => f a -> f a
```

↓

with that they can be written as more generic functions.

↓

```
liftedInc :: (Functor f, Num b)
          => fb → fb
liftedInc = fmap (+1)
```

*) making these fns polymorphic in the type of functorial makes them more reusable

↓

```
>> liftedInc [1..5]
[2, 3, 4, 5, 6]
```

```
data Possibly a = LolNope
                | Yeppers a
                deriving ()

instance Functor Possibly where
  fmap _ LolNope = LolNope
  fmap f (Yeppers a) = Yeppers (f a)
```

---

*) Either :

```
incIfRight :: Num a => Either e a → Either e a
incIfRight (Right n) = Right $ n+1
incIfRight (Left e) = Left e
```

↓

simplifying using fmap.

```
incIfRight :: Num a => Either e a -> Either e a
incIfRight n = fmap (+1) n
```

↓

eta-contract to drop obvious argument

↓

```
incIfRight :: Num a => Either e a -> Either e a
incIfRight' = fmap (+1)
```

↓

removing specification to Either to generalise to functor instances

↓

```
liftedInc :: (Num a, Functor f) => f a -> f a
liftedInc = fmap (+1)
```

Exercise: short exercise

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

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

→ A somewhat suprising functor : the behaviour around the const or Constant functor instance is suprising.

↓

const function (returns first element)

↓

const datatype has a similar concept in mind.

↓

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

(b → phantom type)

↓

kind :: * → * → * (not a valid functor)

↓

Const a ✓

Const x

Const a b x

↓

```
instance Functor (Const a) where
    fmap _ (Constant v) = (Constant v)
```

↓

operation independent to the fn. in question

↓

```
fmap (const 2) (Constant 3)
Constant { getConstant = 3 }
```

→ More Structures, More functors : At times, the structure of our types may require that we also have a functor instance for an intermediate type. example

```
data Wrap f a =          -- a is an argument to f
  Wrap (f a)
  deriving (Eq, Show)
```

↓

```
Instance Functor (Wrap f) where
  fmap f (Wrap fa) = Wrap (f fa)
```

↓

won't work, because there's this f that we're not hopping over, and a (the value fmap should be applying to is an argument to functor type)

↓

```
Instance Functor Wrap f where      -- f needs to be a functor
  fmap f (Wrap fa) = Wrap (fmap f fa)
```

adding a constraint

↓

```
instance Functor f => Functor (Wrap f) where
  fmap f (Wrap fa) = Wrap (fmap f fa)
```

→ IO functor : IO type is an abstract datatype i.e. there are no data constructors that we're allowed to pattern match on, so the typeclasses IO provides are the only

way, we get to work with type IO a.

↓

one of the simplest provided is functor

↓

```
getInt :: IO Int
getInt = fmap read getLine
```

IO String

fmap lifts read over the IO type.

↓

```
>> fmap (const ()) getInt
10
```

→ no result → ghci doesn't print any value of the type IO ()

↓

```
>> fmap (+1) getInt
10
11
>> fmap (++ " and me too!") getLine
hello
"hello and me too!"
```

↓ can be accomplished using do syntax

```
meTooIsm :: IO String
meTooIsm = do
  input ← getLine
  return (input ++ " and me too!")
```

→ what if we want to do something different? What if we wanted to transform only the structure and leave the type argument to that structure or type constructor alone. Through this we arrive at natural transformations.

↓

nat :: (f → g) → fa → ga : not possible since we can't have higher-kinded types as argument types to the fn. type

↓

modest change to fix it

↓

{-# LANGUAGE RankNTypes #-}

type Nat f g = forall a. fa → ga

(doing opposite of what a functor does i.e. transforming the structure, preserving the values as they were.

↓

quantification of a in the right hand side of the declaration allows us to obligate all funcs of this type to be oblivious to the contents of the structures f & g in much the same way that the identity fn. cannot do anything but return the argument it was given.

↓

Syntatically allows us to avoid talking about a in the type of Nat — which is what we want, we shouldn't

have any specific information about the contents of f and g because we're supposed to be only performing a structural transformation, not a fold.

↓

If we try to elide the 'a' from the type arguments w/o quantifying it separately, we'll get an error, we'll even get an error if we don't turn on RankNTypes pragma. Example.

```
type Nat f g = forall a. f a → g a

maybeToList :: Nat Maybe []

maybetoList Nothing = []

maybeToList (Just a) = [a]
```

(a+1 isn't allowed)

↓ instead of using forall, what if we define a in the type.

type Nat f g a = f a → g a, the a+1 case is allowed i.e. the structure alongwith the values (which was principally supposed to stay same) can also be modified.

→ functors are unique to a datatype : in part because of parametricity, in part because arguments to type constructors are applied in order of definition.

→ Quickchecking functor instances : functor laws :

fmap id = id

fmap (p . q) = (fmap p) . (fmap q)

↓

These can be turned into Quickcheck properties

↓

```
functorIdentity :: (Functor f, Eq (f a))
                => f a -> Bool
functorIdentity x =
  fmap id x = x
```

applied functor

(Just 3)

↓

fa :: *

[1]

[(int)]

↓

```
functorCompose :: (Eq (f c), Functor f)
               => (a -> b)
               -> (b -> c)
               -> f a
               -> Bool
functorCompose f g x =
(fmap g (fmap f x)) == (fmap (g . f) x)
```

↓ providing concrete instances

```
f :: [Int] -> Bool
f x = functorIdentity x
```

↓

quickCheck f

```
>> let c = functorCompose (+1) (*2)
>> let li x = c (x :: [Int])
```

quickCheck li ✓

*) Making QuickCheck generate fns too : Coarbitrary covers the fn. argument type, whereas Arbitrary typeclass is used for the fn. result type.

↓

```
import Test.QuickCheck
import Test.QuickCheck.Function

functorCompose' :: (Eq (f c), Function c)
                => f a
                -> Fun a b
                -> Fun b c
                -> Bool
functorCompose' x (Fun _ f) (Fun _ g) =
  (fmap (g . f) x) == (fmap g . fmap f $ x)
```

→ pattern matching on the fun value that we're asking QuickCheck to generate.

↓

fun type is essentially a product of the weird fn. type and an ordinary Haskell fn. generated from the weirdo.

↓

```
type IntToInt = Fun Int Int
type IntFC = [Int] -> IntToInt -> IntToInt -> Bool
let fc' = functorCompose'
```

quickCheck (fc' :: IntFC)

*) can't print those fun values (using reboseCheck)

Exercises : Instances of Fun

1. newtype Identity a = Identity a

Instance functor (Identity a) where

fmap f (Identity x) = Identity (f x)

2. data Pair a = Pair a a

Instance Functor (Pair) where

fmap f (Pair x x') = Pair (f x) (f x')

3. data Two a b = Two a b

Instance functor (Two a) where

fmap f (Two x y) = Two x (f y)

4. data Three a b c = Three a b c

Instance functor (Three a b) where

fmap f (Three x y z) = Three x y (f z)

5. data Three' a b = Three' a b b

Instance functor (Three' a) where

fmap f (Three' x y z) = Three x (f y) (f z)

8. data Trivial = Trivial