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.