Functor (1A)

Young Won Lim
7/12/17

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice.

Based on

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Haskell in 5 steps
https://wiki.haskell.org/Haskell_in_5_steps

Typeclasses

Typeclasses are like interfaces

defines some behavior
- comparing for equality
- comparing for ordering
- enumeration

Instances of that typeclass
- types possessing such behavior

Such behavior is defined by
- **function definition**
- **type declaration** to be implemented

a type is an instance of a typeclass implies
the functions defined by the typeclass with that type can be used

No relation with classes in Java or Python

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

A Typeclass Example

the Eq typeclass

defines the functions **==** and ***/=***

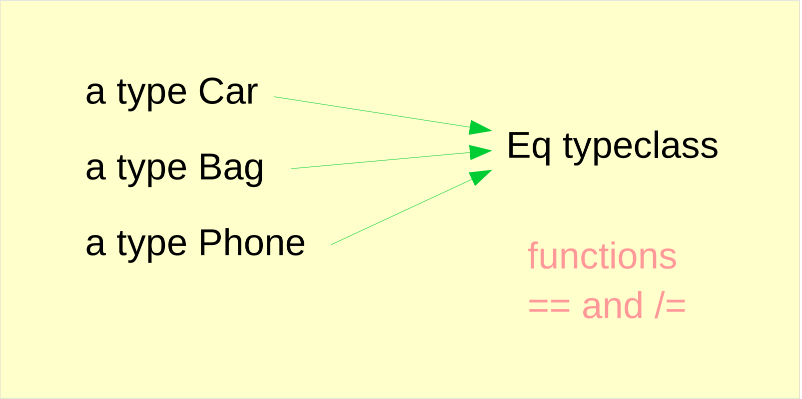
a type Car

comparing two cars c1 and c2 with the equality function ==

The Car type is an instance of Eq typeclass

Instances : various types

Typeclass : a group or a class of these similar types



http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Eq Typeclass Example

```
class Eq a where
    (==) :: a -> a -> Bool          - a type declaration
    (/=) :: a -> a -> Bool          - a type declaration
    x == y = not (x /= y)           - a function definition
    x /= y = not (x == y)           - a function definition


data TrafficLight = Red | Yellow | Green

instance Eq TrafficLight where
    Red    == Red     = True
    Green  == Green   = True
    Yellow == Yellow  = True
    _ == _            = False
```

```
ghci> Red == Red
True
ghci> Red == Yellow
False
ghci> Red `elem` [Red, Yellow, Green]
True
```

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Show Typeclass Example

```
class Show  a where
   show :: a  -> String            - a type declaration
   * * *
```

```
data TrafficLight = Red | Yellow | Green

instance Show TrafficLight where
    show Red = "Red light"
    show Yellow = "Yellow light"
    show Green = "Green light"
```

```
ghci> [Red, Yellow, Green]
[Red light,Yellow light,Green light]
```

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Show Typeclass Example

```
class (Eq a) => Num a where
  ...

class Num a where
  ...
```

class constraint on a class declaration
only we state that our type a must be an instance of **Eq**

an instance of **Eq**
before being an instance of **Num**

When defining the required function bodies
in the class declaration or
in instance declarations,

we can safely use == because a is a part of **Eq**

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Show Typeclass Example

class constraints in class declarations

to make a typeclass a subclass of another typeclass

class constraints in instance declarations

to express requirements about the contents of some type.

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Show Typeclass Example

the **a** : a concrete type

Maybe : not a concrete type
: a type constructor that takes one parameter
produces a concrete type.

Maybe a : a concrete type

```
instance (Eq m) => Eq (Maybe m) where
    Just x    ==  Just y          =     x == y
    Nothing   ==  Nothing         =     True
          _ == _                  =     False
```

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Functor typeclass

the Functor typeclass is basically for things that can be mapped over

ex) mapping over lists

the list type is part of the Functor typeclass

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Functor typeclass

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

The Functor typeclass

defines one function, **fmap**,

no default implementation

the type variable **f**

not a concrete type (a concrete type can hold a value)

a type constructor taking one type parameter

Maybe Int : a concrete type

Maybe : a type constructor that takes one type as the parameter

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Functor typeclass

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

fmap takes

- a function from one type to another (a -> b)
- a **functor f** applied with one type
- returns a **functor f** applied with another type.

```
map :: (a -> b) -> [a] -> [b]
```

map takes

- a function from one type to another
- a list of one type
- returns a list of another type

map is just a fmap that works only on lists. Here's how the list is an instance of the Functor typeclass.

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Functor typeclass

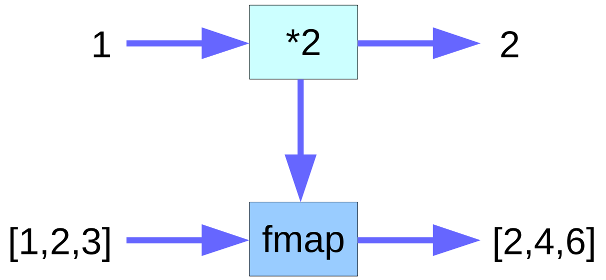
```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

  map :: (a -> b) -> [a] -> [b]
```

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

ghci> fmap (*2) [1..3]
[2,4,6]
ghci> map (*2) [1..3]
[2,4,6]

map is just a fmap that works only on lists



Here's how the list is an instance of the Functor typeclass.

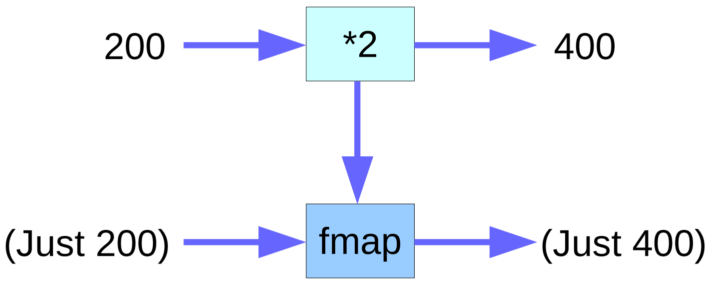
```
instance Functor [ ] where
  fmap = map
```

f : a type constructor that takes one type
[ ] : a type constructor that takes one type
[a] : a concrete type ([Int], [String] or [[String]])

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Functor typeclass

```
ghci> fmap (*2) (Just 200)
Just 400
ghci> fmap (*2) Nothing
Nothing
```




```
ghci> fmap (++ " HEY GUYS IM INSIDE THE JUST") (Just "Something serious.")
Just "Something serious. HEY GUYS IM INSIDE THE JUST"
ghci> fmap (++ " HEY GUYS IM INSIDE THE JUST") Nothing
Nothing
```

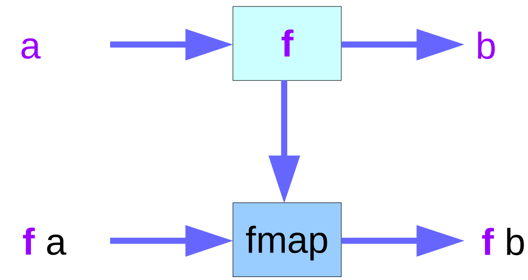
http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

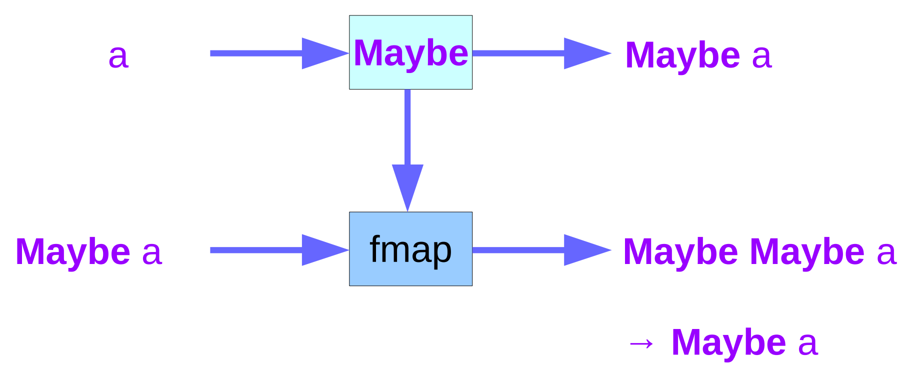
Functor typeclass

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

instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap f Nothing = Nothing
```

If an empty value of **Nothing**, then just return a **Nothing**.
If a single value packed up in a **Just**,
then we apply the function on the *contents* of the **Just**.





http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Maybe as a functor

Functor typeclass:

- transforming one type to another
- transforming operations of one type to those of another

Maybe a is an instance of a **functor** type class

Functor provides **fmap** method

maps functions of the base type (such as Integer)
to *functions* of the lifted type (such as Maybe Integer).

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

Maybe as a functor

A *function* **f** transformed with **fmap**
can work on a **Maybe** value

```
case maybeVal of
  Nothing  -> Nothing        -- there is nothing, so just return Nothing
  Just val -> Just (f val)   -- there is a value, so apply the function to it

    father :: Person -> Maybe Person
    mother :: Person -> Maybe Person

       f :: Int             -> Int
fmap  f :: Maybe Integer -> Maybe Integer
```

a **Maybe** **Integer** value: **m_x**

fmap f m_x

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

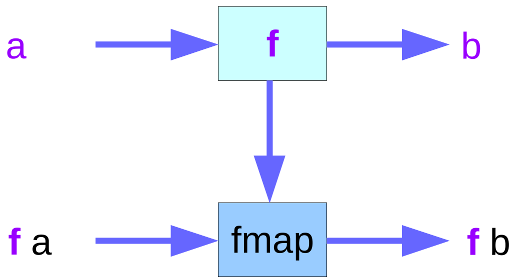
Maybe as a functor

In fact, you could apply a whole chain of
lifted Integer -> **Integer** functions to **Maybe Integer** values
and only have to worry about explicitly checking for **Nothing** once when you're finished.

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

Functor typeclass

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



The Functor typeclass

defines one function, **fmap**,

no default implementation

the type variable **f**

not a concrete type (a concrete type can hold a value)

a type constructor taking one type parameter

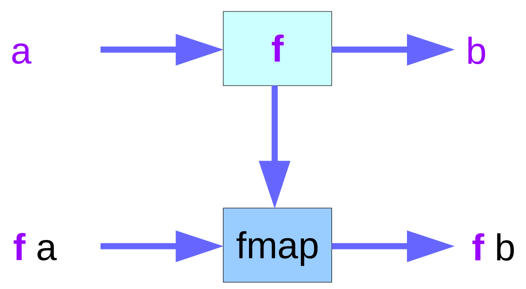
Maybe Int : a concrete type

Maybe : a type constructor that takes one type as the parameter

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Functor typeclass

```
instance Functor IO where
    fmap f action = do
        result <- action
        return (f result)
```



http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

Functor typeclass

```
putStr "Hello"  >>
putStr " "      >>
putStr "world!" >>
putStr "\n"
```

```
do { putStr "Hello"
   ; putStr " "
   ; putStr "world!"
   ; putStr "\n" }
```

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

Functor typeclass

```
do { action1            -- by monad laws equivalent to:  do { action1
   ; action2            --                                  ; do { action2
   ; action3 }          --                                       ; action3 } }
```

```
action1 >>
do { action2
   ; action3 }
```

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

Functor typeclass

The bind operator (>>=)

passes a value,

(the result of an action or function),

downstream in the binding sequence.

do notation assigns a variable name

to the passed value

using the **<-**

```
do { x1 <- action1
   ; x2 <- action2
   ; mk_action3 x1 x2 }
```

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

Functor typeclass

```
do { x1 <- action1
   ; x2 <- action2
   ; mk_action3 x1 x2 }
```

```
action1 >>= (\ x1 -> action2 >>= (\ x2 -> mk_action3 x1 x2 ))
```

```
action1
  >>=
    (\ x1 -> action2
       >>=
         (\ x2 -> mk_action3 x1 x2 ))
```

```
action1 >>= (\ x1 ->
  action2 >>= (\ x2 ->
    mk_action3 x1 x2 ))
```

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

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf
[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf