Function Functor (2A)

Young Won Lim

Copyright (c) 2016 - 2018 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

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

This document was produced by using LibreOffice.

Based on

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

http://learnyouahaskell.com/functors-applicative-functors-and-monoids

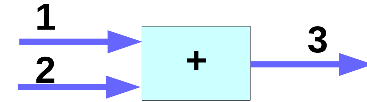
Haskell in 5 steps
https://wiki.haskell.org/Haskell_in_5_steps

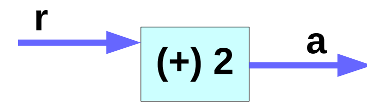
Prefix vs Infix Functions

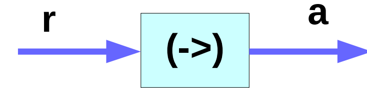
(+) 1 **2**	-- prefix function	(->) r **a**	-- prefix function
(*) 3 **4**	-- prefix function		
1 + **2**	-- infix function	r -> **a**	-- infix function
3 * **4**	-- infix function		

1, 2, 3, 4 : values

r, a : types



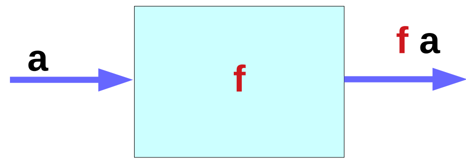


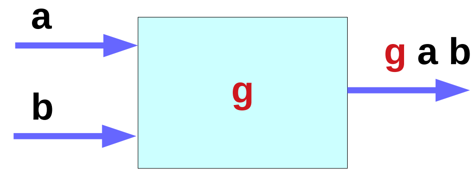


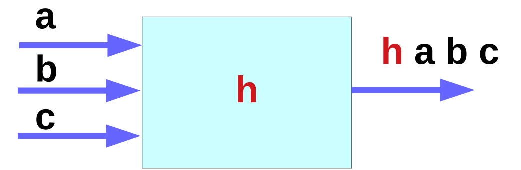
2 + 3 =	**(2 +) 3 =**	**(+) 2 3**
r -> a =	**(r ->) a** =	**(->) r a**
infix	partial app	prefix

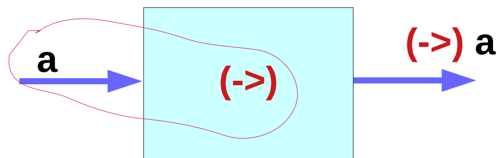
https://www.mjoldfield.com/atelier/2014/07/monads-fn.html

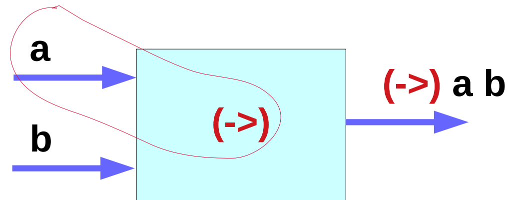
Using Prefix Function (->)

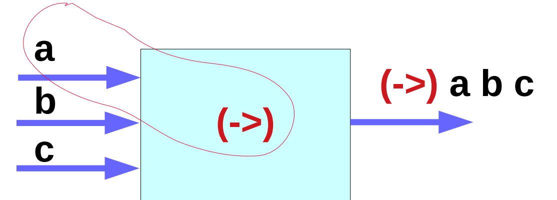


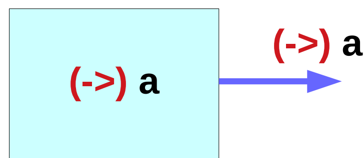


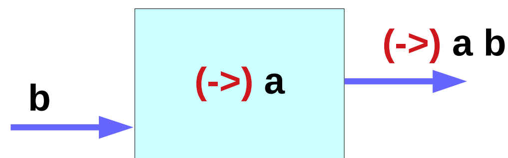


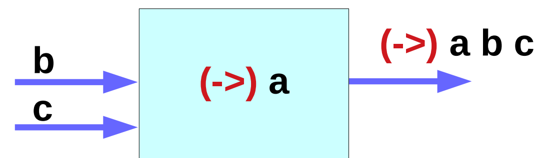




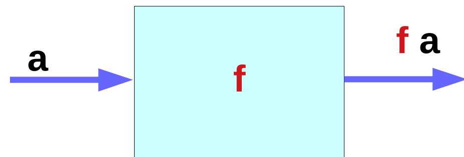




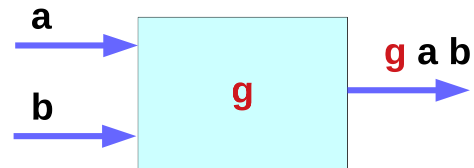




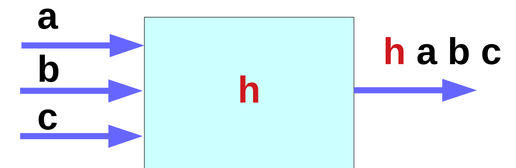
Using Partially Applied Function (a ->)



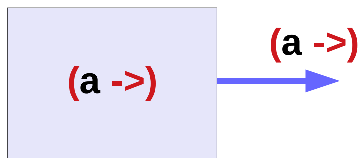
f :: a -> b



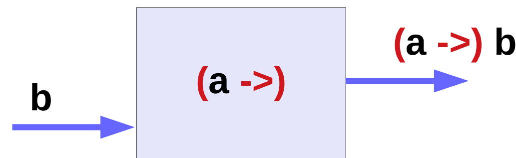
g :: a -> b -> c



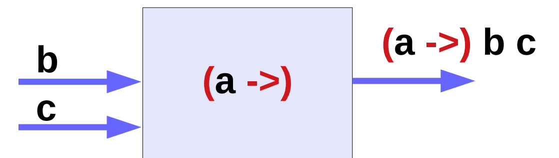
h :: a -> b -> c -> d



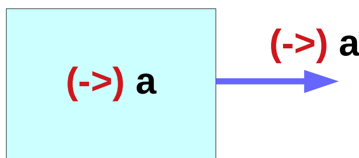
(a ->) :: b

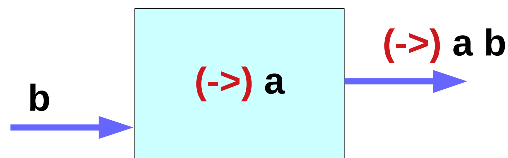


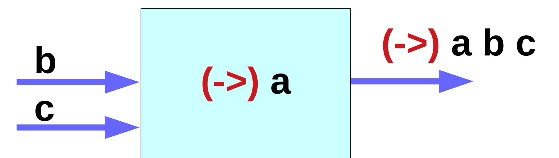
(a ->) :: b -> c



(a ->) :: b -> c -> d







((->) r) Functor Values

```
instance Functor ((->) r) where
    fmap f g =  (\x -> f (g x))
```

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

```
data Maybe a = Just a | Nothing
```

((->) r) Functor values

((->) r) x

((+) 1) 2

((*) 33) 2

Maybe Functor values

Just 2

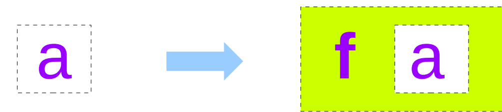
Nothing

http://learnyouahaskell.com/functors-applicative-functors-and-monoids

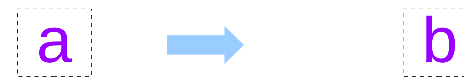
((->) r) Functor : a type constructor

```
instance Functor ((->) r) where
      fmap f g =  (\x -> f (g x))
```

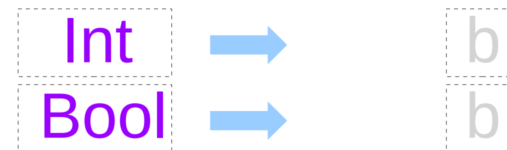
A **functor** is a **type constructor**:
a **function** from **types** to **types**.



The **arrow ->** is also a **type constructor**
but it takes two **type arguments**.
(**a -> b** : the type of a function from a to b)



partial application to **type constructors**
(only one fixed type is provided :
like (r -> ?) - not allowed though)

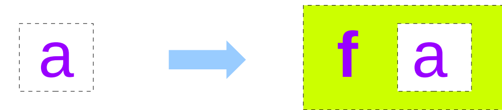


https://www.quora.com/What-is-in-Haskell-How-can-this-be-a-functor-and-a-monad-What-does-it-actually-do-and-mean

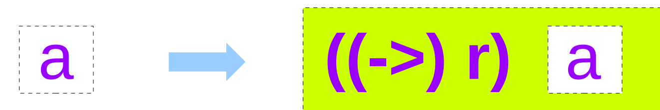
((->) r) Functor : parameterized

```
instance Functor ((->) r) where
    fmap f g =  (\x -> f (g x))
```

type constructor

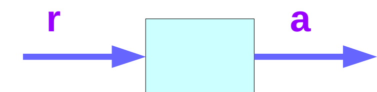


prefix function (->) only with the first argument
((->) r) define a **type** of a **function**
from fixed **r** type to another **type**.



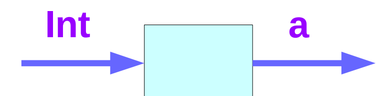
the **parameterized type constructor**

((->) r) a
((->) r a)
r -> a

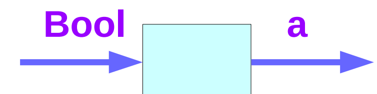


The parameter type **a** is the **return type**
of the **function** that takes **r** as an argument.

((->) Int)
Functors



((->) Bool)
Functors



https://www.quora.com/What-is-in-Haskell-How-can-this-be-a-functor-and-a-monad-What-does-it-actually-do-and-mean

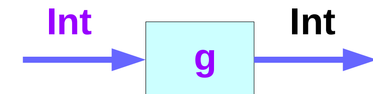
((->) r) Functor Values g

```
instance Functor ((->) r) where
    fmap f g =  (\x -> f (g x))
```

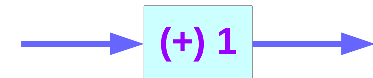
A Functor value is
a function g

((->) Int) Int – a Functor value type

((->) Int)
Functors



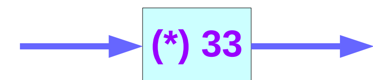
g1



((+) 1) – a Function value (a function)

((*) 33) – a Function value (a function)

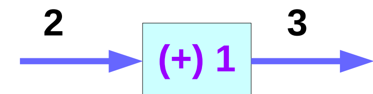
g2



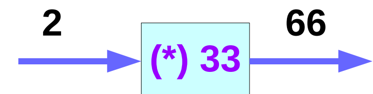
((+) 1) 2 → 3 result of running

((*) 33) 2 → 66 result of running

g1 x



g2 x



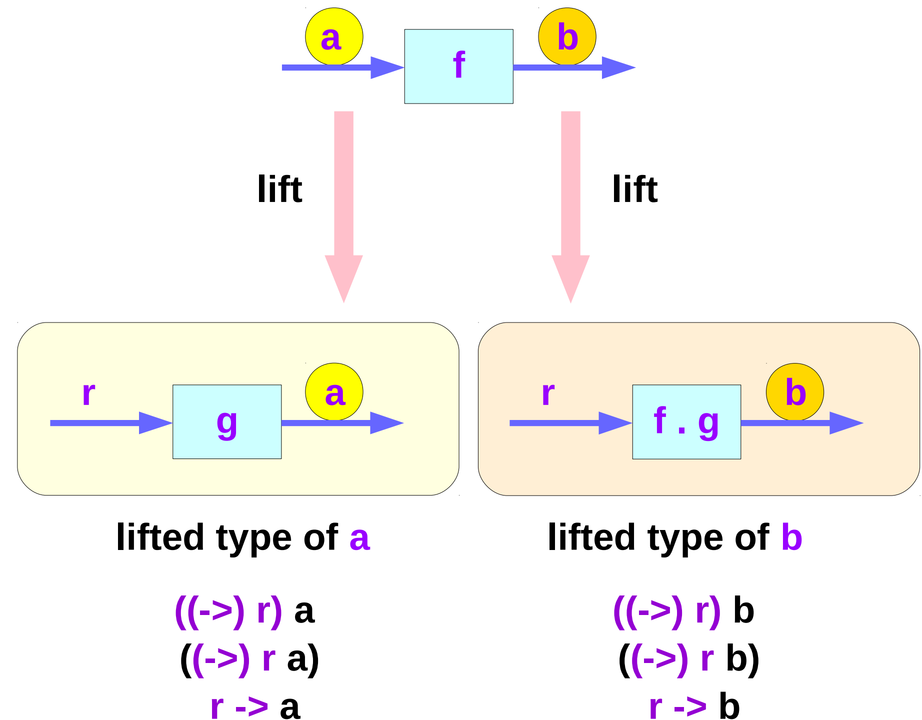
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

((->) r) Functor – function f to be lifted

```
instance Functor ((->) r) where
    fmap f g =  (\x -> f (g x))
```

fmap takes an arbitrary **function f :: a -> b**
and lifts it to a **new function**
that operates between **lifted types**.

apply the **functor**'s **type constructor ((->) r)**,
type a ….. **r -> a**
type b …. **r -> b**



https://www.quora.com/What-is-in-Haskell-How-can-this-be-a-functor-and-a-monad-What-does-it-actually-do-and-mean

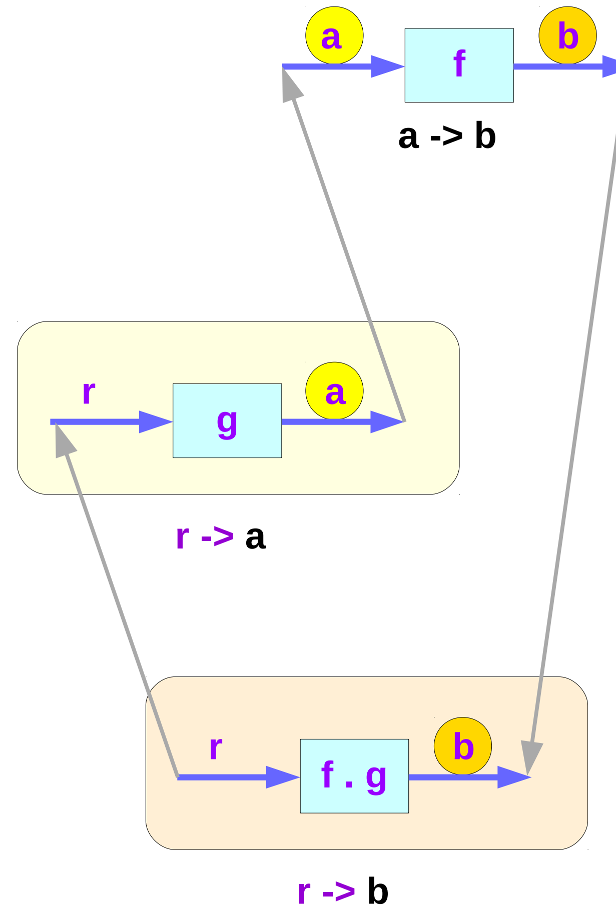
((->) r) Functor – fmap type signature

```
instance Functor ((->) r) where
    fmap f g =  (\x -> f (g x))
```

the type signature of **fmap** :
fmap :: (a->b) -> (r->a) -> (r->b)

a **function g :: r->a** and must produce
a **function r->b**

There is only one way to do it:
go from **r** to **a** using **g** and then from **a** to **b** using **f**.



https://www.quora.com/What-is-in-Haskell-How-can-this-be-a-functor-and-a-monad-What-does-it-actually-do-and-mean

((->) r) Functor Values

```
instance Functor ((->) r) where
    fmap f g =  (\x -> f (g x))
```

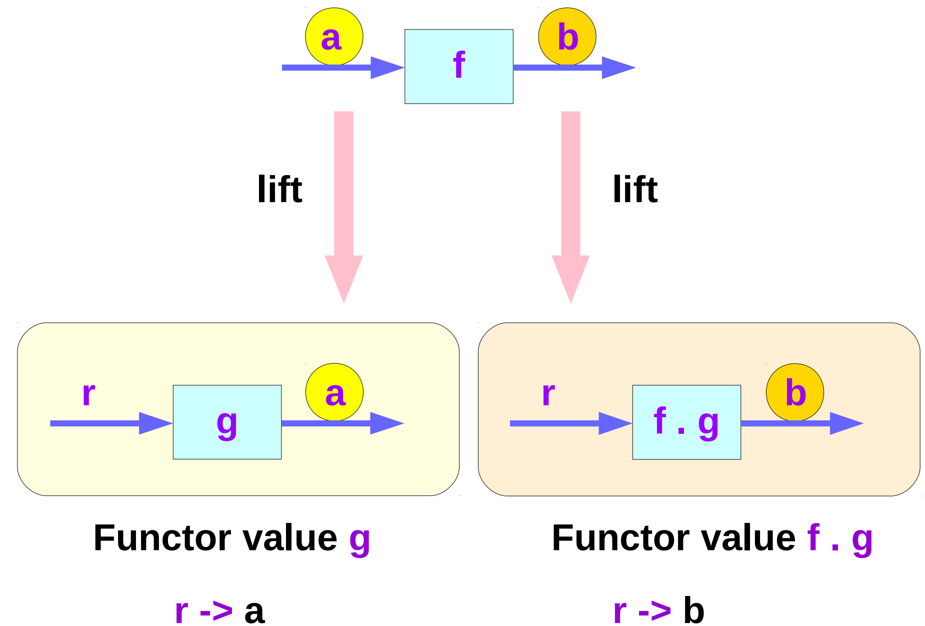
Functor ((->) r) values :

g :: r -> a

f . g :: r -> b

Given

f :: a -> b



https://www.quora.com/What-is-in-Haskell-How-can-this-be-a-functor-and-a-monad-What-does-it-actually-do-and-mean

((->) r) Functor – fmap implementation

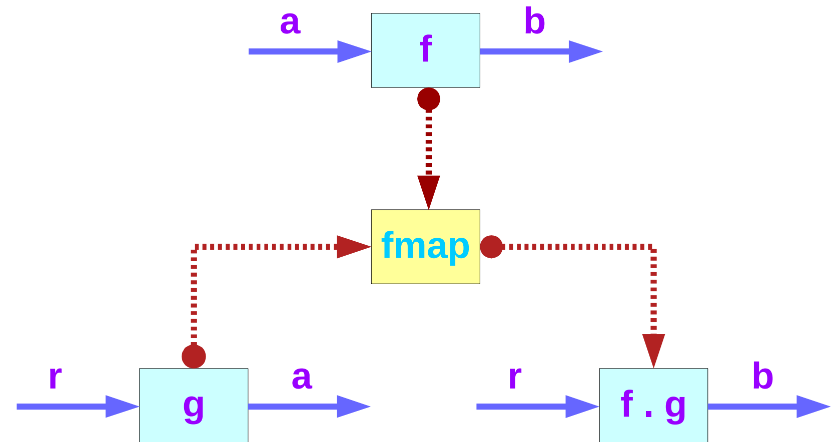
```
instance Functor ((->) r) where
     fmap f g =  (\x -> f (g x))
```

fmap :: (a->b) -> (r->a) -> (r→b)

fmap f g = f . g

fmap f g = (.) f g

fmap   = (.)



f :: a -> b
g :: r -> a
f . g :: r -> b

composition of the two function: **f . g**
fmap f g = f . g

https://www.quora.com/What-is-in-Haskell-How-can-this-be-a-functor-and-a-monad-What-does-it-actually-do-and-mean

(->) r Functor Instance Example

```
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x))
```

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

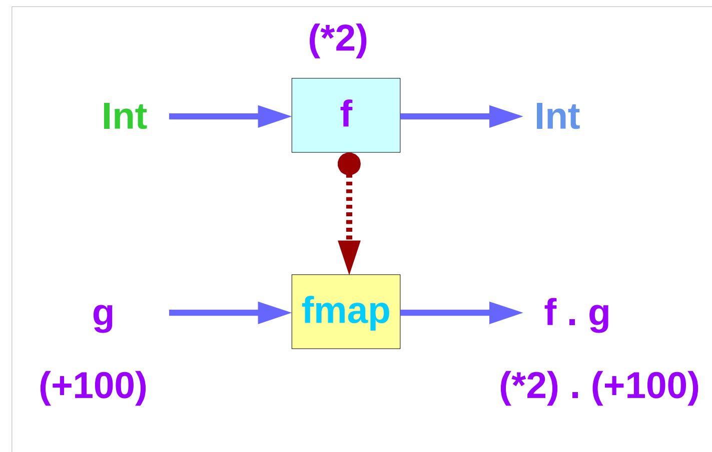
```
instance Functor fr where …
```

```
fmap :: (a -> b) -> ((->) r a) -> ((->) r b)
fmap :: (a -> b) -> (r -> a)    -> (r -> b)
```

```
fmap :: (a -> b) -> fr a -> fr b
```

```
f :: a -> b
g :: r -> a
f . g :: r -> b
```

```
f :: Int -> Int
g :: Int -> Int
f . g :: Int -> Int

Functor ((->) Int)
g = (+100)
f = (*2)
```



http://learnyouahaskell.com/functors-applicative-functors-and-monoids

(->) r Functor

Function (->) r as a **Functor fr**

The function type **r -> a**
can be rewritten as **(->) r a**

When we look at it as **(->) r a**,
we can see **(->)** in a different view point,

because we can consider **(->)** a **type constructor**
that takes <u>two</u> **type parameters r a**, just like **Either**.

f :: **a -> b**
g :: **r -> a**

```
fmap :: (a -> b) -> fr a        -> fr b         fr
fmap :: (a -> b) -> ((->) r a)  -> ((->) r b)   (->) r
fmap :: (a -> b) -> (r  -> a)   -> (r  -> b)    r  ->
```

http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Partial Application (r ->)

a **type constructor** has to take exactly one **type parameter**
to make an **instance** of **Functor**.

thus **(->)** cannot make an **instance** of **Functor**
but **(->)** can make a partial application in the form of **(r ->)**

If the syntax allowed for **type constructors**
to be partially applied with **sections**
(->) r can be written as **(r ->)**

like we can partially apply **+** by doing **(2+)**,
that is the same as **(+) 2**

Now **functions** are **functors**

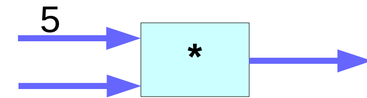
look at the implementation in **Control.Monad.Instances**

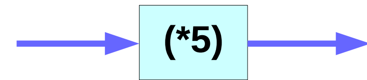
```
fmap :: (a -> b) -> ((->) r a)   -> ((->) r b)
fmap :: (a -> b) -> (r    -> a)  -> (r   -> b)
```

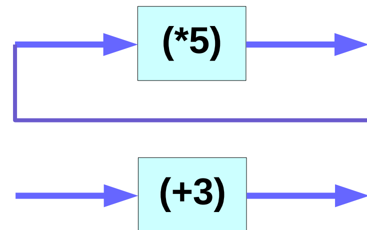
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

((->) r) Prefix Function

```
let f = (*5)
let g = (+3)
(fmap f g) 8

(*5) ((+3) 8) = (*5) 11 = 55


let f = (+) <$> (*2) <*> (+10)
f 3

(*2) 3 = 6
(+10) 3 = 13
(+) 6 13 = 19
```







(fmap (*5) (+3))

http://learnyouahaskell.com/for-a-few-monads-more#reader

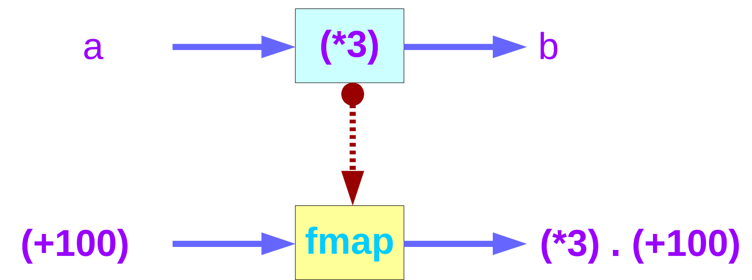
(->) r Functor Examples

```
instance Functor ((->) r) where
    fmap f g =  (\x -> f (g x))
```

```
instance Functor ((->) r) where
    fmap = ( . )
    fmap f g = ( f . g )
```

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

```
ghci> :t fmap (*3) (+100)
fmap (*3) (+100) :: (Num a) => a -> a
ghci> fmap (*3) (+100) 1
303
ghci> (*3) `fmap` (+100) $ 1
303
ghci> (*3) . (+100) $ 1
303
ghci> fmap (show . (*3)) (+100) 1
"303"
```



http://learnyouahaskell.com/functors-applicative-functors-and-monoids

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf