Applicatives Overview (3A)

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.

Young Won Lim
6/6/18

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

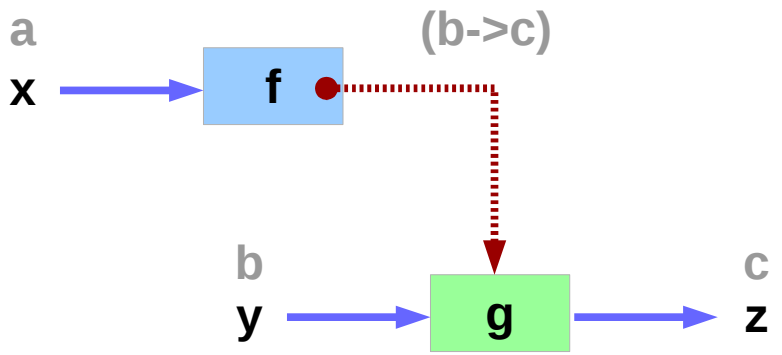
Currying

Currying recursively transforms
a function that takes multiple arguments
into a function that takes just a single argument and
returns another function if any arguments are still needed.

f :: a -> b -> c

f :: a -> b -> c ➡ **f :: a -> (b -> c)**

f x y	**f :: a -> b -> c**
(f x) y	**f :: a -> (b -> c)**
g y	**g :: b -> c**



https://wiki.haskell.org/Currying
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

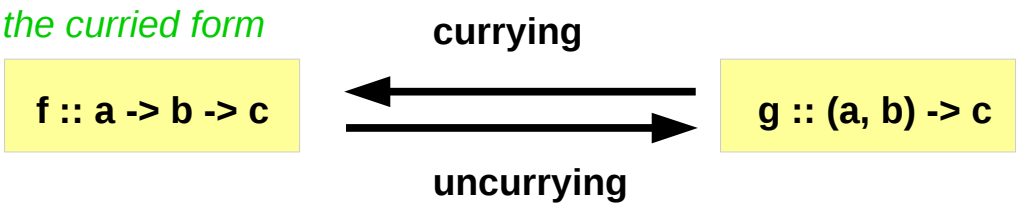
Curry & Uncurry

f :: a -> b -> c is the curried form of **g :: (a, b) -> c**

f = curry g
g = uncurry f

f x y = g (x,y)

the curried form is usually more convenient because it allows partial application.



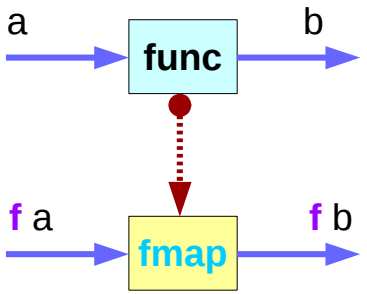
all functions are considered curried

all functions take just one argument

https://wiki.haskell.org/Currying

Functor typeclass – instances

```
class Functor f

instance Functor Maybe
instance Functor [ ]
```




```
function fmap
function func
type constructor  f
```

f is a **type constructor** taking <u>one</u> type parameter

Maybe instance of the **Functor typeclass**
[ ] instance of the **Functor typeclass**

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

The Functor Typeclass

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

If a type **f** is an instance of **Functor**,
fmap can be used to apply
a function of the type (a -> b)
to values of a type **a** in it (**f a**).

fmap promotes functions **(a -> b)**
to act on **functorial values**. **(f a -> f b)**

```
fmap id = id                    -- 1st functor law
fmap (g . h) = fmap g . fmap h  -- 2nd functor law
```

```
g :: b -> c
h :: a -> b
g . h :: a -> c
```

To ensure **fmap** works sanely,
any instance **f** of **Functor**
must comply with the above two laws

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

Examples of fmap over the Maybe Functor instance

```
fmap negate (Just 2)
```

≡

```
negate <$> Just 2
```

Just (-2)

Just (-2)

```
fmap negate Nothing
```

≡

```
negate <$> Nothing
```

Nothing

Nothing

Infix synonym

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

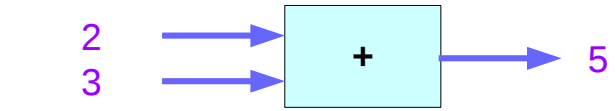
Applying two argument functions

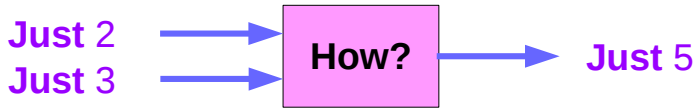
Problem:

to apply a function of two arguments

to functorial values

Ex: to sum **Just 2** and **Just 3**





The brute force approach would be

extracting the values from the **Maybe** wrapper.

- we have to do tedious checks for **Nothing**
- extracting the value might is not possible

  for functors like **IO**

Partial application

- use **fmap** to partially apply (+) to the first argument:

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

Partial Application

Prelude> :t **(+)**

(+) :: Num a => a -> a -> a

Prelude> :t **(+) <$> Just 2**

(+) <$> Just 2 :: Num a => Maybe (a -> a)

Functions wrapped in Maybe

(+) <$> Just 2 ➡

Just (2+)

Prelude> **(<$> Just 3) <$> (+) <$> Just 2** ➡

Just (Just 5)

(<$> Just 3) <$> (+) <$> Just 2

(<$> Just 3) <$> Just (2+)

Just((2+) <$> Just 3)

Just (Just 5)

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

Partial Application

```
Prelude> :t (<$> Just 3)
(<$> Just 3) :: Num a => (a -> b) -> Maybe b

Prelude> (<$> Just 3) (*3)
Just 9
Prelude> (<$> Just 3) (+3)
Just 6
```

```
Prelude> :t (+)
(+) :: Num a => a -> a -> a
Prelude> :t (+) <$> Just 2
(+) <$> Just 2 :: Num a => Maybe (a -> a)
```

```
Prelude> (<$> Just 3) <$> (+) <$> Just 2
Just (Just 5)
```

```
(<$> Just 3) <$> (+) <$> Just 2
(<$> Just 3) <$> Just (2+)
Just( <$> Just 3 (2+) )
Just( (2+) <$> Just 3 )
Just (Just 5)
```

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

<*> Operator

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

(+) <$> Just 2 <*> Just 3

Just (2+) <*> Just 3

Just 5

(+) <$> Just 2 :: f (a -> b)

Just (2+) :: f (a -> b)

Just 3 :: f a

((+) <$> Just 2) <*> Just 3 : f (a -> b) -> f a

Just 5 :: f b

Prelude> **:t (<*>)**

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

(+) <$> Just 2 <*> Just 3

Just 5

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

Applicative Style

(<*>) is one of the methods of **Applicative** the type class of applicative functors - functors that support function application within their contexts.

Expressions **(+) <$> Just 2 <*> Just 3** are said to be written in applicative style, which look like regular function application while working with a functor.

(+) <$> Just 2 <*> Just 3

(+) 2 3

(+) <$> Just 2 <*> Just 3

Just ((+) 2 3)

Just 5

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

<*> Application of a function

Just (+3) <*> Just 2

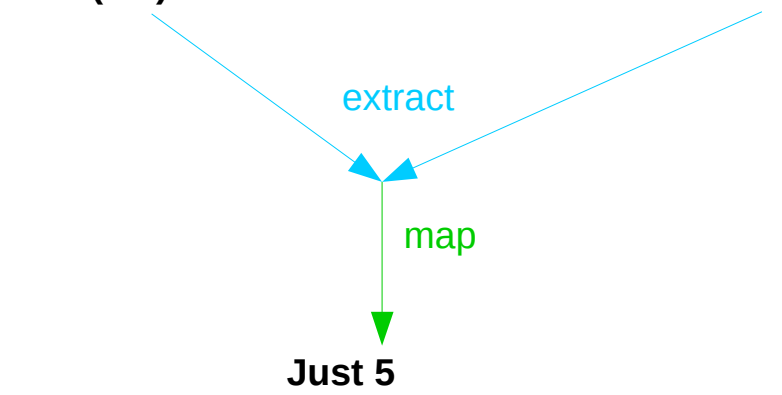
(<*>) :: f (a -> b) -> f a -> f b

function wrapped in a **Just**

Just (+ 3)

integer wrapped in a **Just**

Just 2

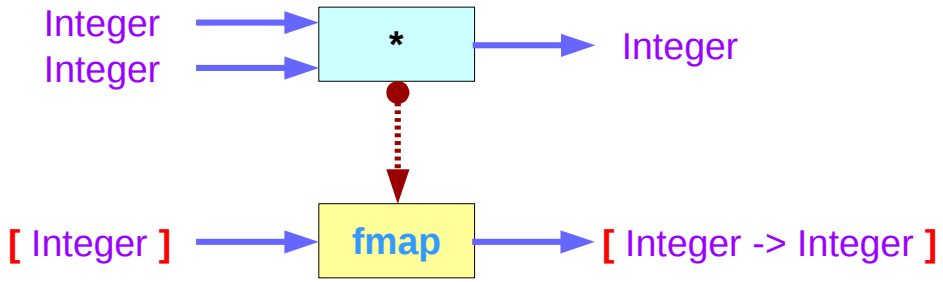


Extracting and **Mapping**

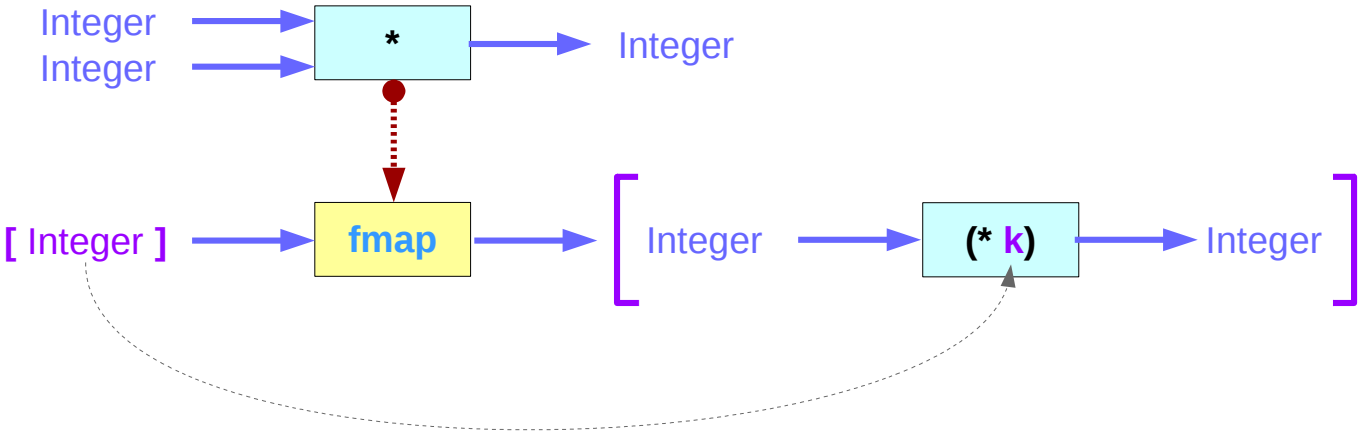
Just 5

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

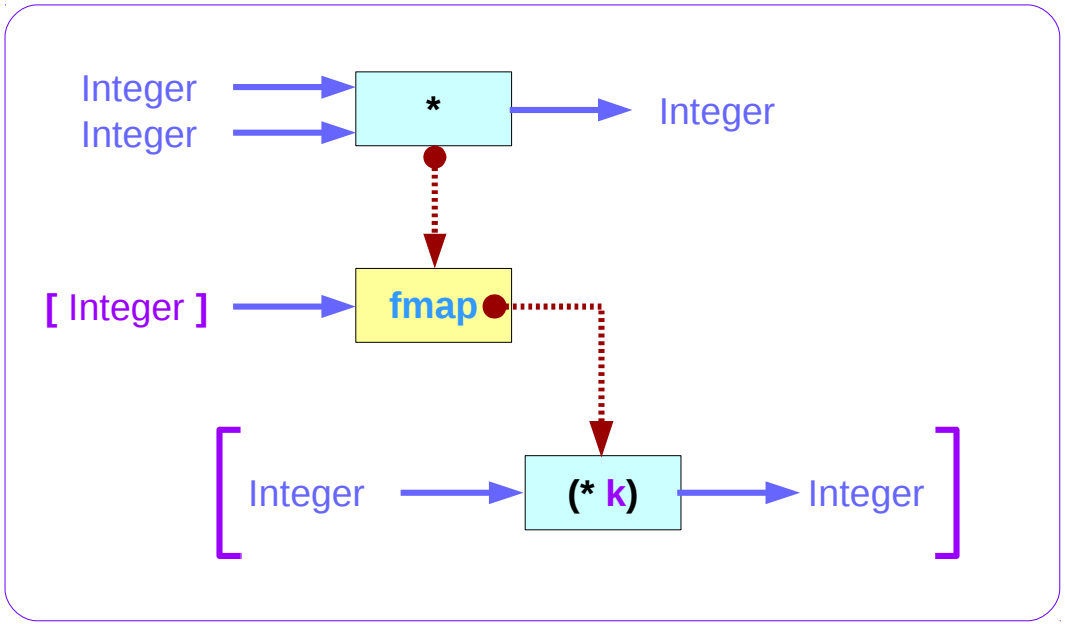
Mapping functions over the Functor [ ] (1)



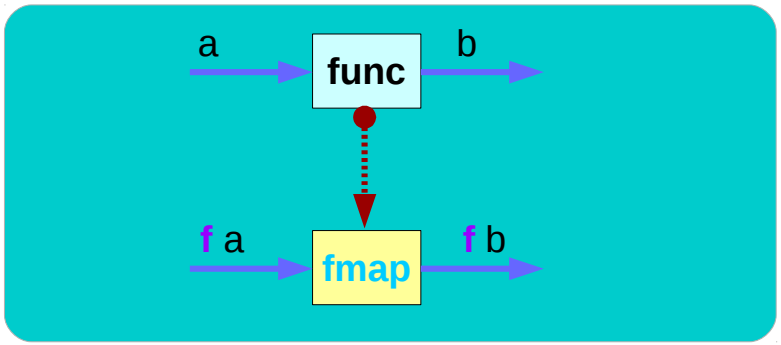
Mapping functions over the Functor [ ] (2)

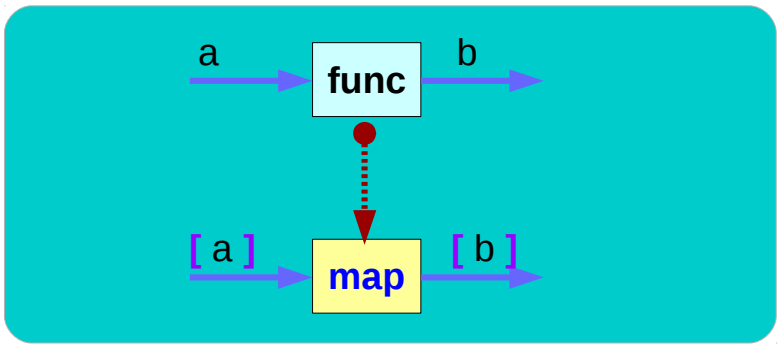


Mapping functions over the Functor [ ] (3)

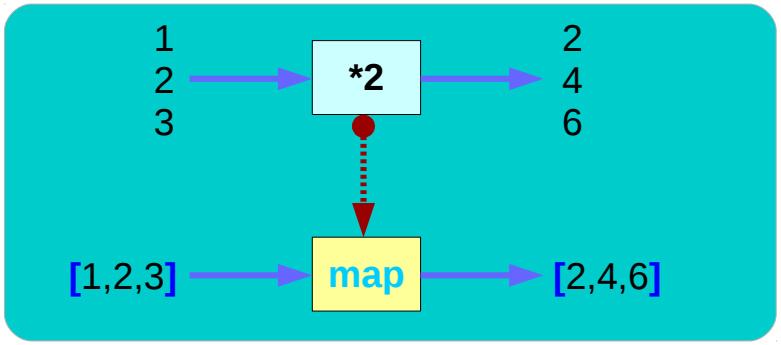


Applicative



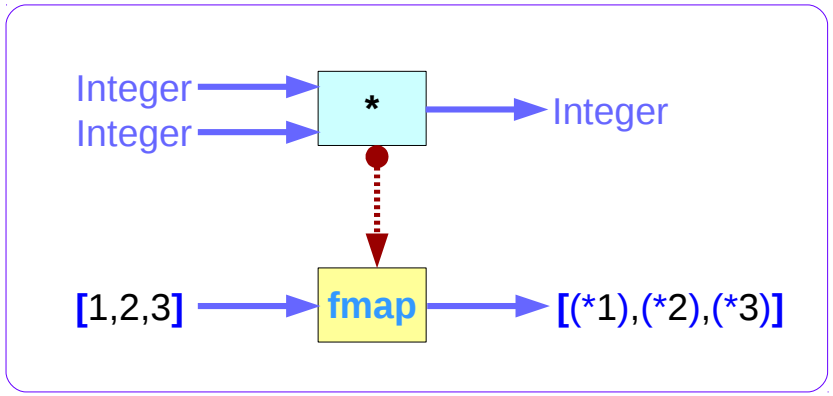


Functor



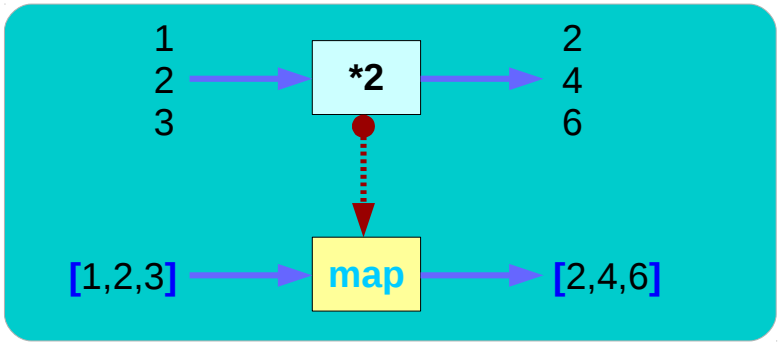
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Mapping functions over the Functor [ ] (4)



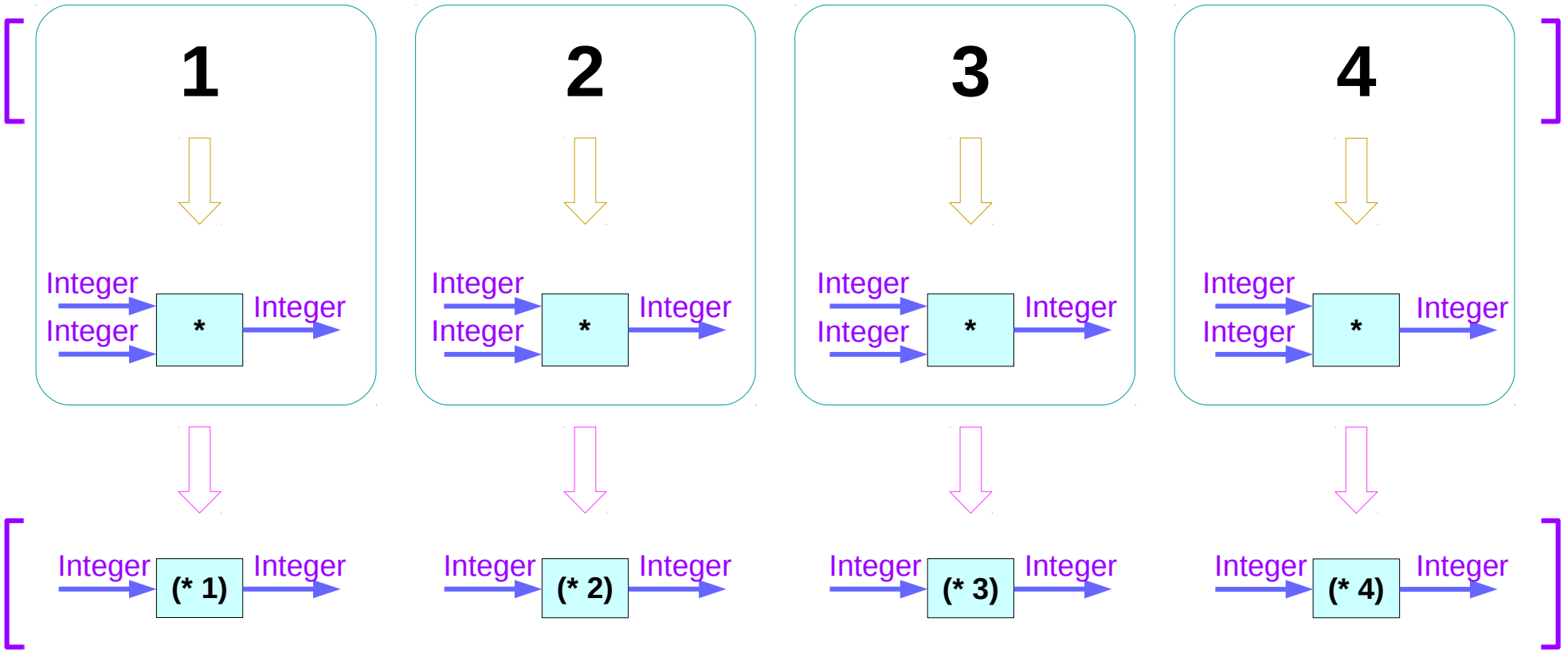
Applicative

Functor



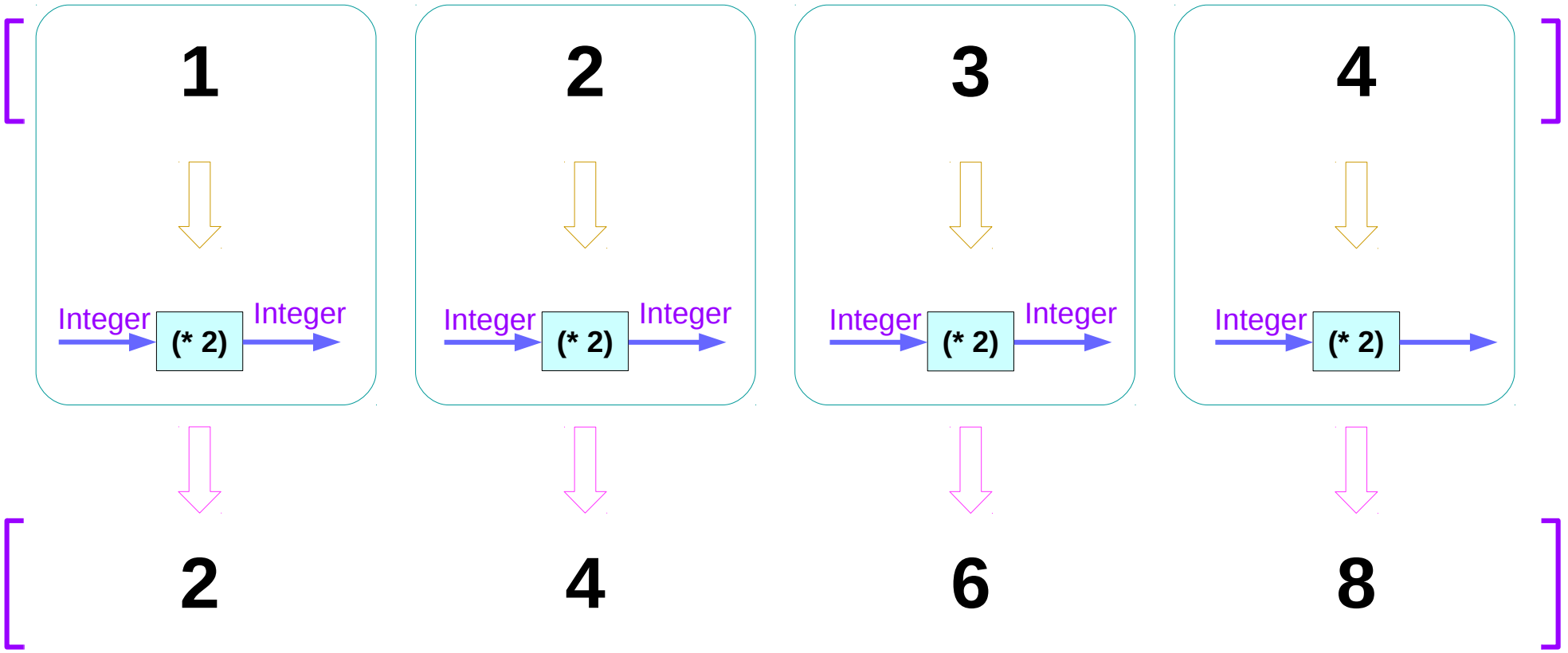
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Applicative : Mapping functions



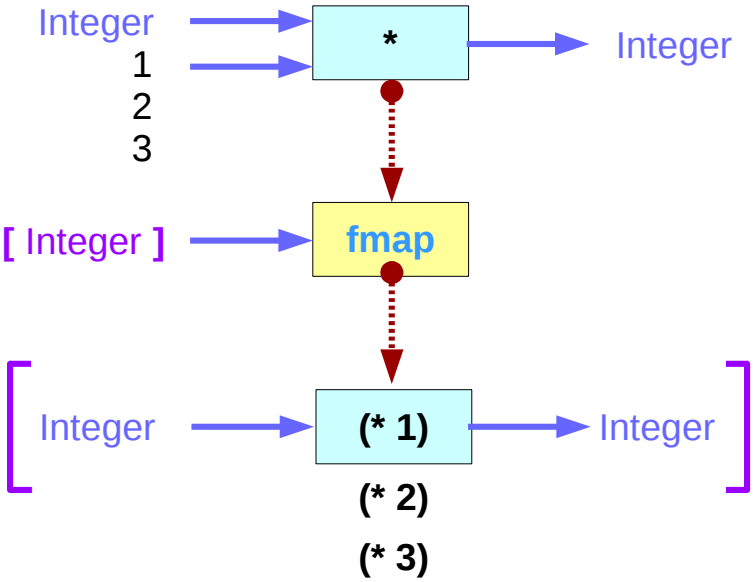
A list of functions

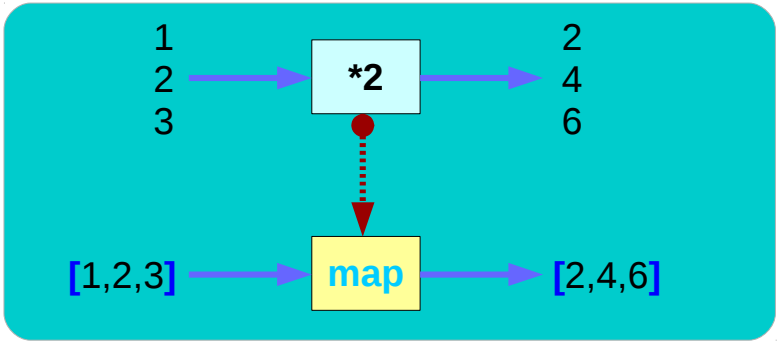
Functor : Mapping values



A list of integers

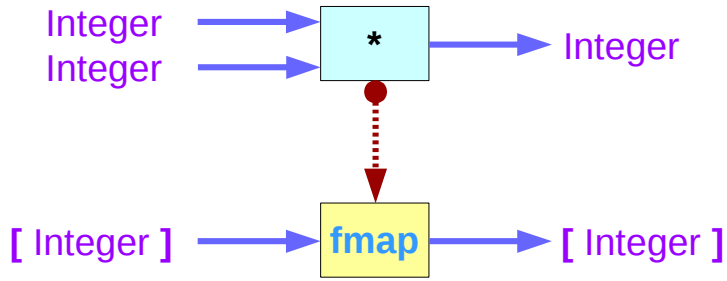
Applicatives vs. Functors





Double applications of fmap (1)

```
ghci> let a = fmap (*) [1,2,3,4]
```



[1,2,3,4]

[(*) 1, (*) 2, (*) 3, (*) 4]

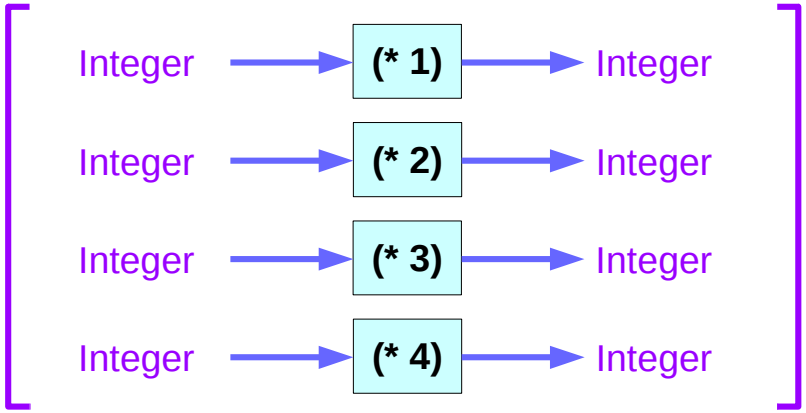
[(* 1), (* 2), (* 3), (* 4)]

[Integer -> Integer]

```
ghci> :t a
a :: [Integer -> Integer]
```

```
ghci> fmap (\f -> f 9) a
[9,18,27,36]
```

a =



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

Double applications of fmap (2)

ghci> let a = **fmap** (*) [1,2,3,4]

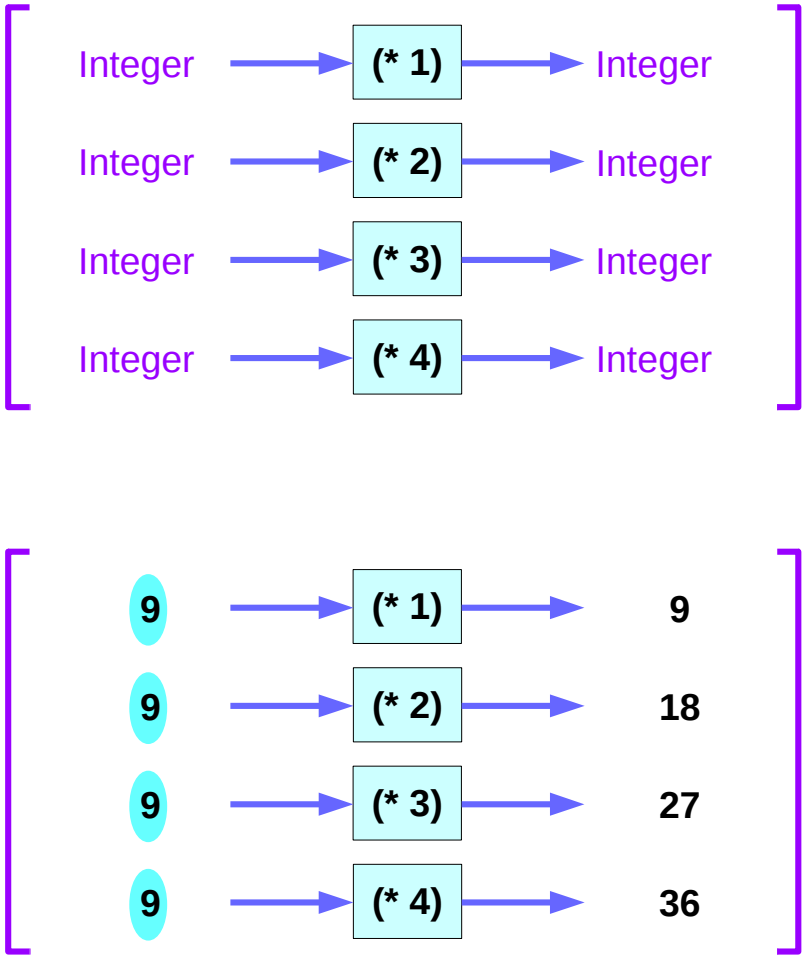
1st fmap

ghci> **fmap** (\f -> f 9) a

[9,18,27,36]

2nd fmap

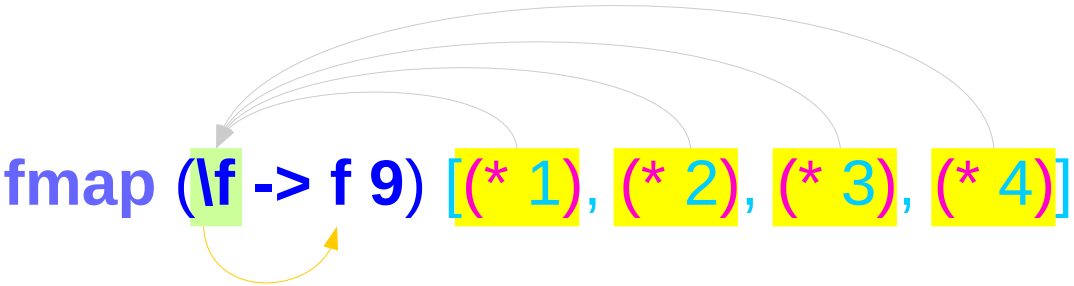
a =



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

Applications of fmap

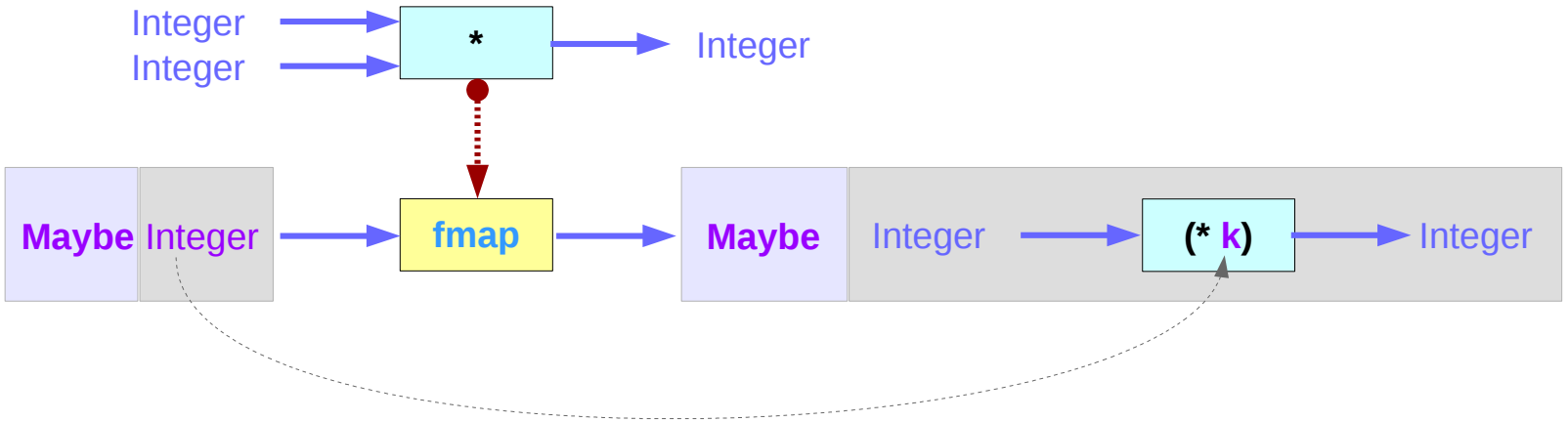
```
fmap (*) [1, 2, 3, 4]

[(*) 1, (*) 2, (*) 3, (*) 4]

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



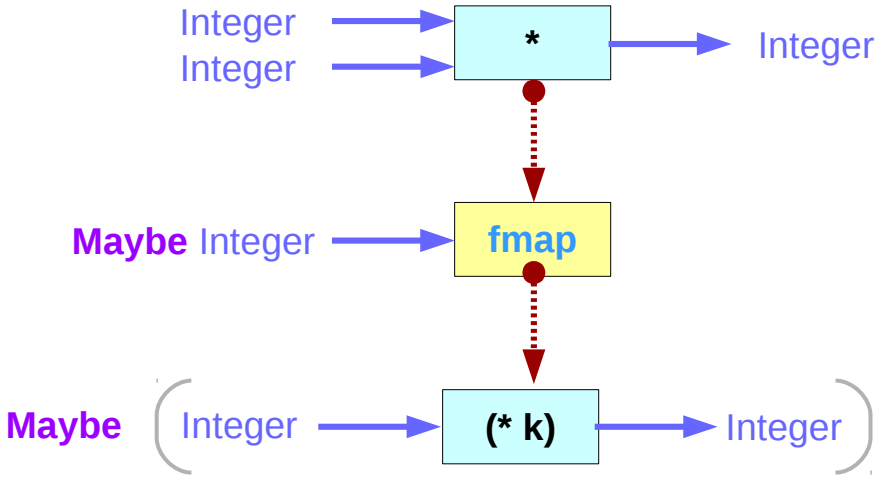

```
[9,18,27,36]
```

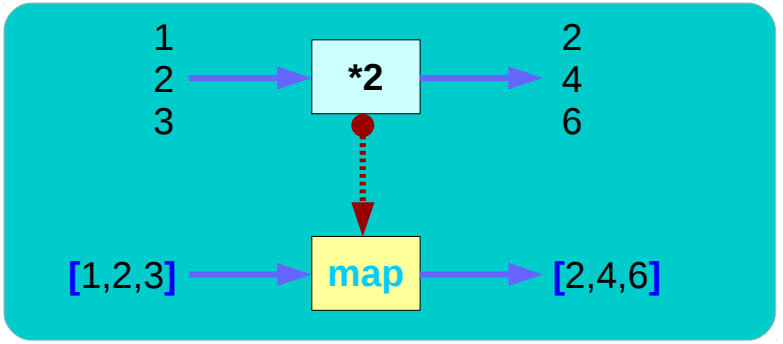
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Mapping functions over the Functor Maybe (1)

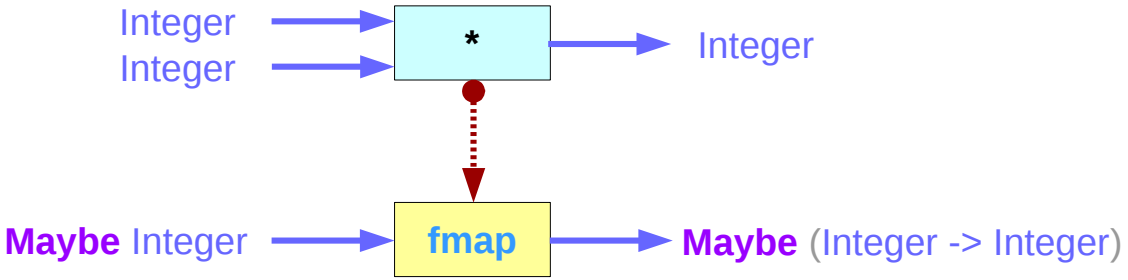


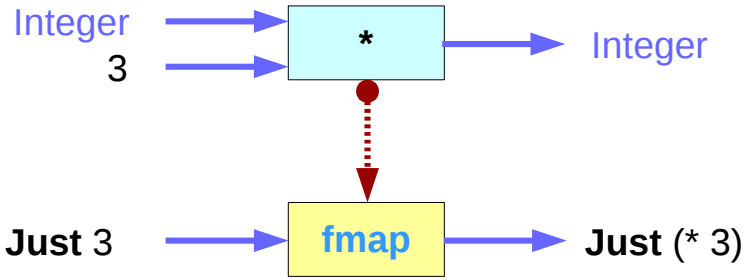
Mapping functions over the Functor Maybe (2)





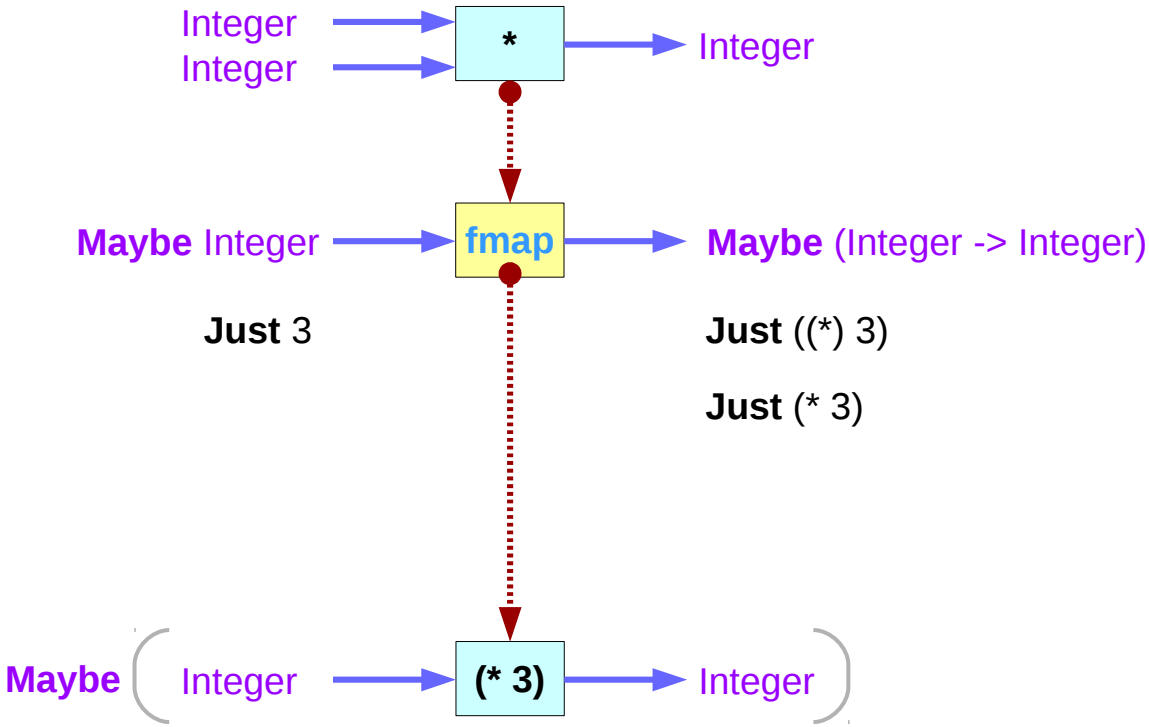
Mapping functions over the Functor Maybe (3)





Function wrapped in Just

fmap (*) (**Just** 3)



function wrapped in a **Just**

Just (* 3)

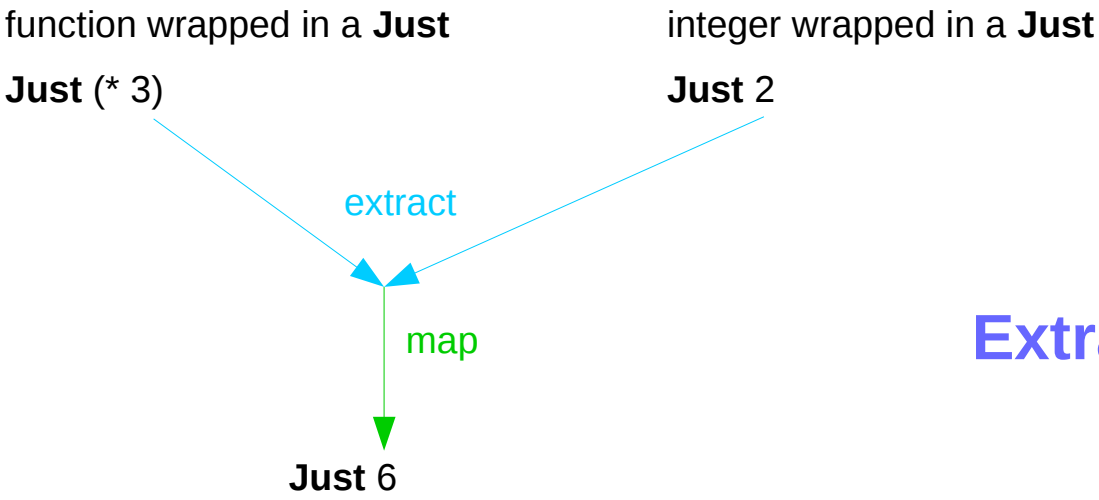
integer wrapped in a **Just**

Just 2

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

<*> Application of a function

Just (* 3) <*> Just 2



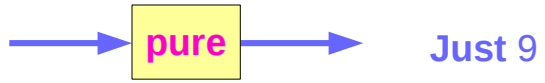
Extracting and **Mapping**

Just 6

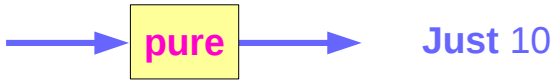
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Default Container Function **Pure**

pure 9 = **Just** 9



pure 10 = **Just** 10



to wrap an **integer**

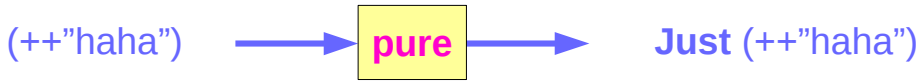
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

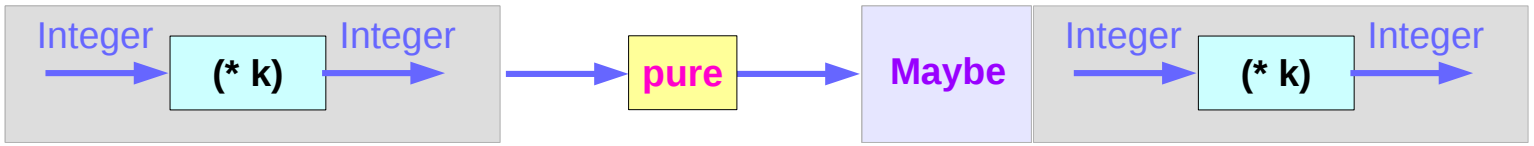
Default Container Function **Pure**

pure (+3) = **Just** (+3)

to wrap a **function**

pure (++"haha") = **Just** (++"haha")

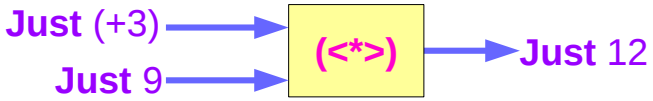




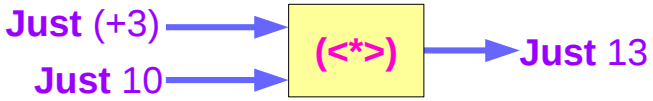
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Applicative Functor Apply <*> Examples (1)

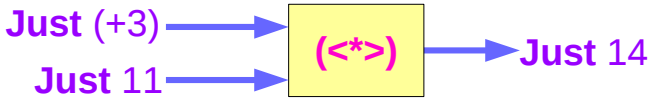
```
Prelude> Just (+3) <*> Just 9
Just 12
```



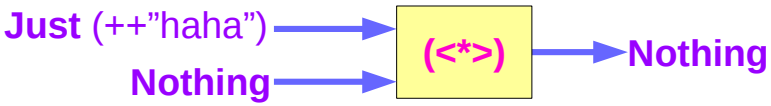

```
Prelude> pure (+3) <*> Just 10
Just 13
```



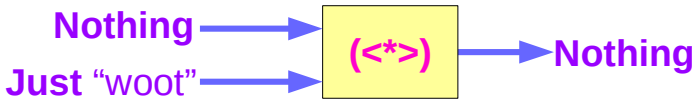

```
Prelude> pure (+3) <*> Just 11
Just 12
```




```
Prelude> Just (++"hahah") <*> Nothing
Nothing
```



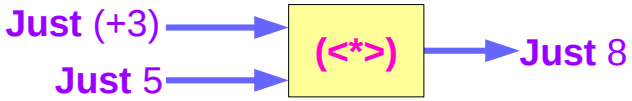

```
Prelude> Nothing <*> Just "woot"
Nothing
```



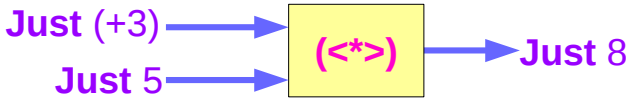
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Applicative Functor Apply <*> Examples (2)

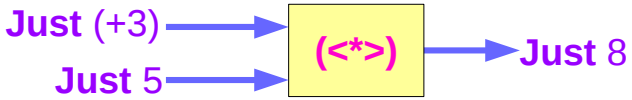
```
Prelude> (+) <$> Just 3 <*> Just 5
Just 8
```




```
Prelude> pure (+) <*> Just 3 <*> Just 5
Just 8
```




```
Prelude> Just (+) <*> Just 3 <*> Just 5
Just 8
```



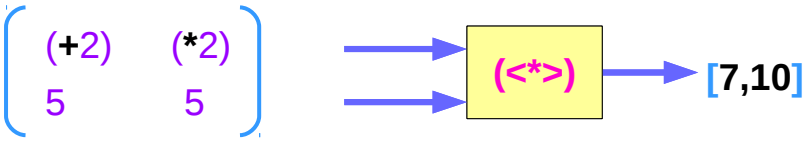

```
Prelude> :t (+) <$> Just 3
(+) <$> Just 3 :: Num a => Maybe (a -> a)
```

```
Prelude> :t Just (+3)
Just (+3) ::Num a => Maybe (a -> a)
```

Applicative Functor Apply <*> Examples (3)

Prelude> [(+), (*)] <*> [2] <*> [5]

[7,10]

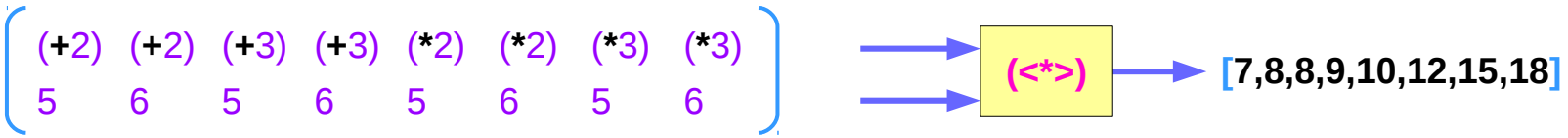


Prelude> [(+), (*)] <*> [2,3] <*> [5]

[7,8,10,15]

Prelude> [(+), (*)] <*> [2,3] <*> [5,6]

[7,8,8,9,10,12,15,18]



References

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