

Applicatives Overview (2A)

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

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.

$f :: a \rightarrow b \rightarrow c$

$f\ x\ y$

$f :: a \rightarrow b \rightarrow c$

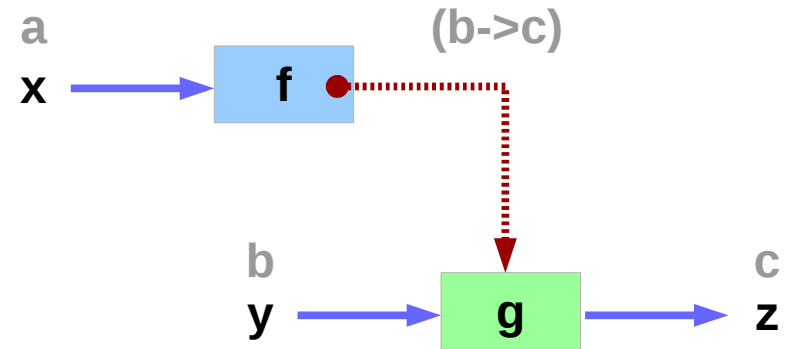
$(f\ x)\ y$

$f :: a \rightarrow (b \rightarrow c)$

$g\ y$

$g :: b \rightarrow c$

$f :: a \rightarrow b \rightarrow c$



<https://wiki.haskell.org/Currying>

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

Curry & Uncurry

$f :: a \rightarrow b \rightarrow c$ the curried form of $g :: (a, b) \rightarrow c$

$f = \text{curry } g$

$g = \text{uncurry } f$

$f \ x \ y = g \ (x,y)$

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

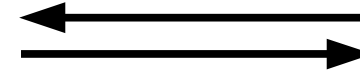
all functions are considered **curried**

all functions take **just one argument**

the curried form

$f :: a \rightarrow b \rightarrow c$

currying



$g :: (a, b) \rightarrow c$

uncurrying

$f \ x \ y$

$g \ (x,y)$

<https://wiki.haskell.org/Currying>

The Functor Typeclass

class Functor f where

fmap :: (a -> b) -> f a -> f b

If a type is an instance of Functor,
you can use fmap to apply a function to values in it.

fmap promotes functions to act on functorial values.

To ensure fmap works sanely, any instance of Functor
must comply with the following two laws:

fmap id = id -- 1st functor law

fmap (g . f) = fmap g . fmap f -- 2nd functor law

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

Maybe - a 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

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
- for some Functors, extracting the value might not be possible (just think about IO).

Partial application

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

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

Partial Application

```
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

<*> Operator

`f (a -> b) -> f a -> f b`

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

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

`Prelude> :t (<*>)`

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

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

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

<*> Operator

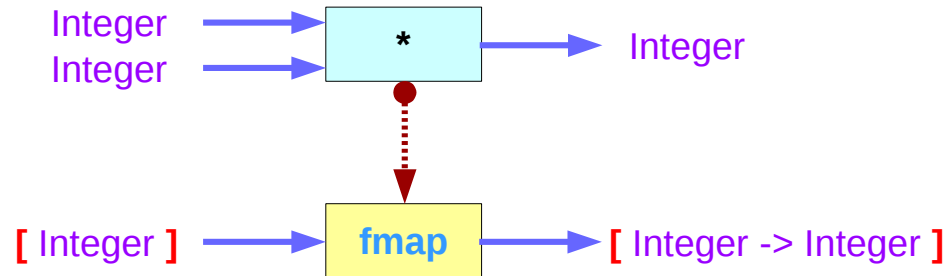
(<*>) is one of the methods of Applicative
the type class of applicative functors -
functors that support function application within their contexts.

Expressions such as (+) <\$> Just 2 <*> Just 3
are said to be written in applicative style,
which is as close as we can get to regular function application
while working with a functor.

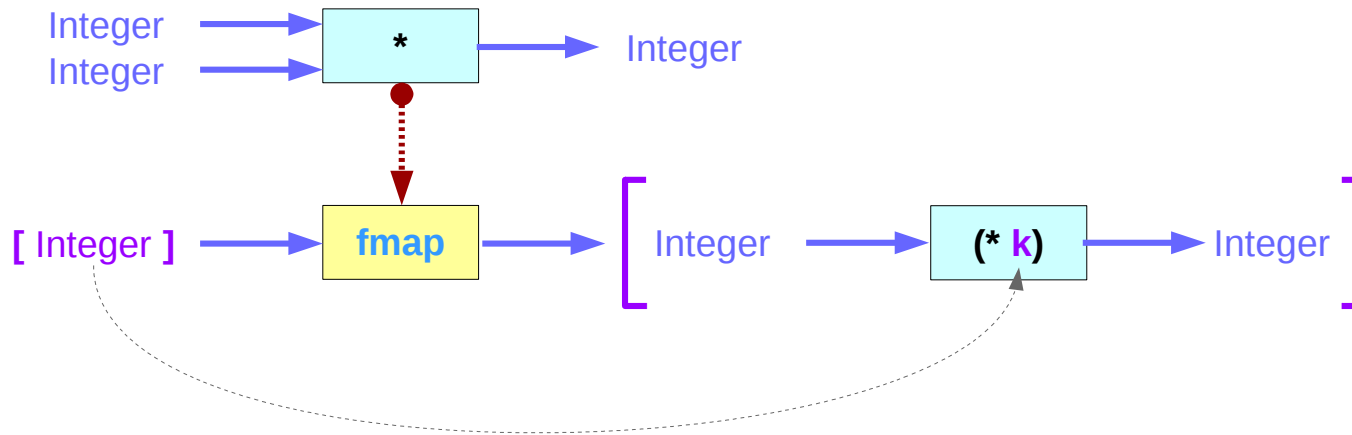
If you pretend for a moment the (<\$>), (<*>) and Just aren't there,
our example looks just like (+) 2 3.

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

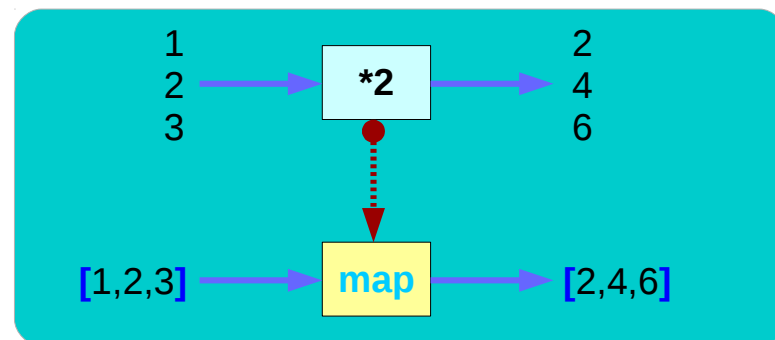
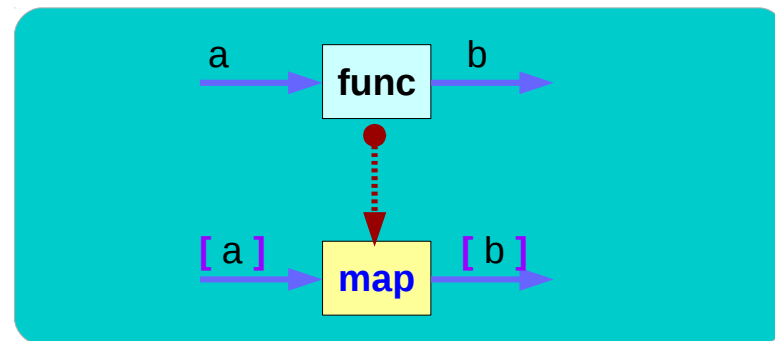
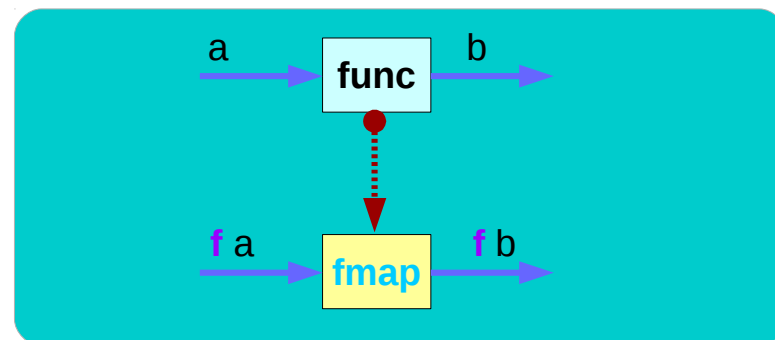
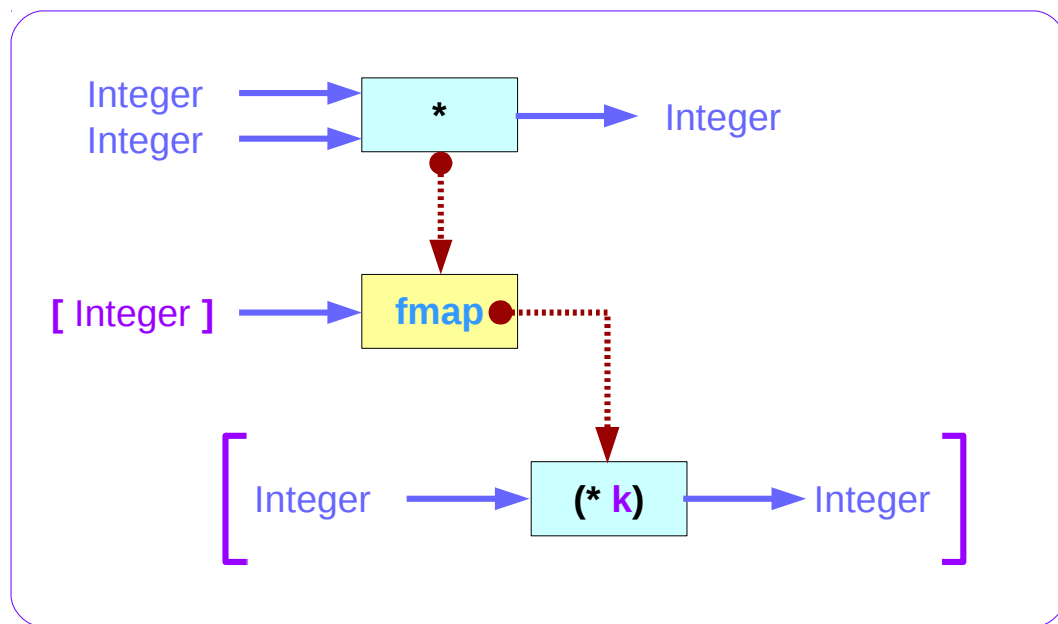
Mapping functions over the Functor [] (1)



Mapping functions over the Functor [] (2)



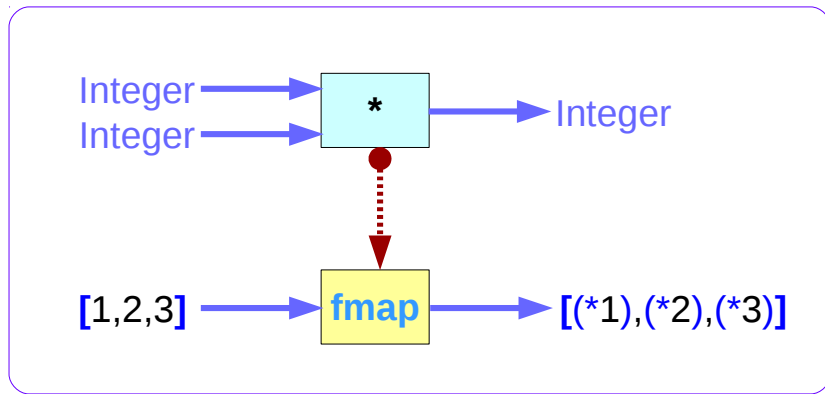
Mapping functions over the Functor [] (3)



Functor →

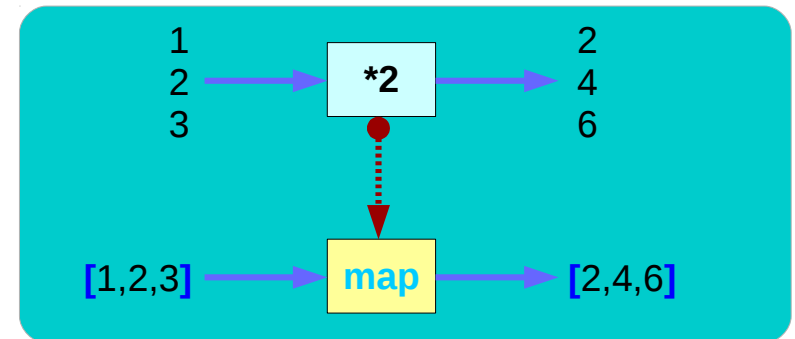
<http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

Mapping functions over the Functor [] (4)



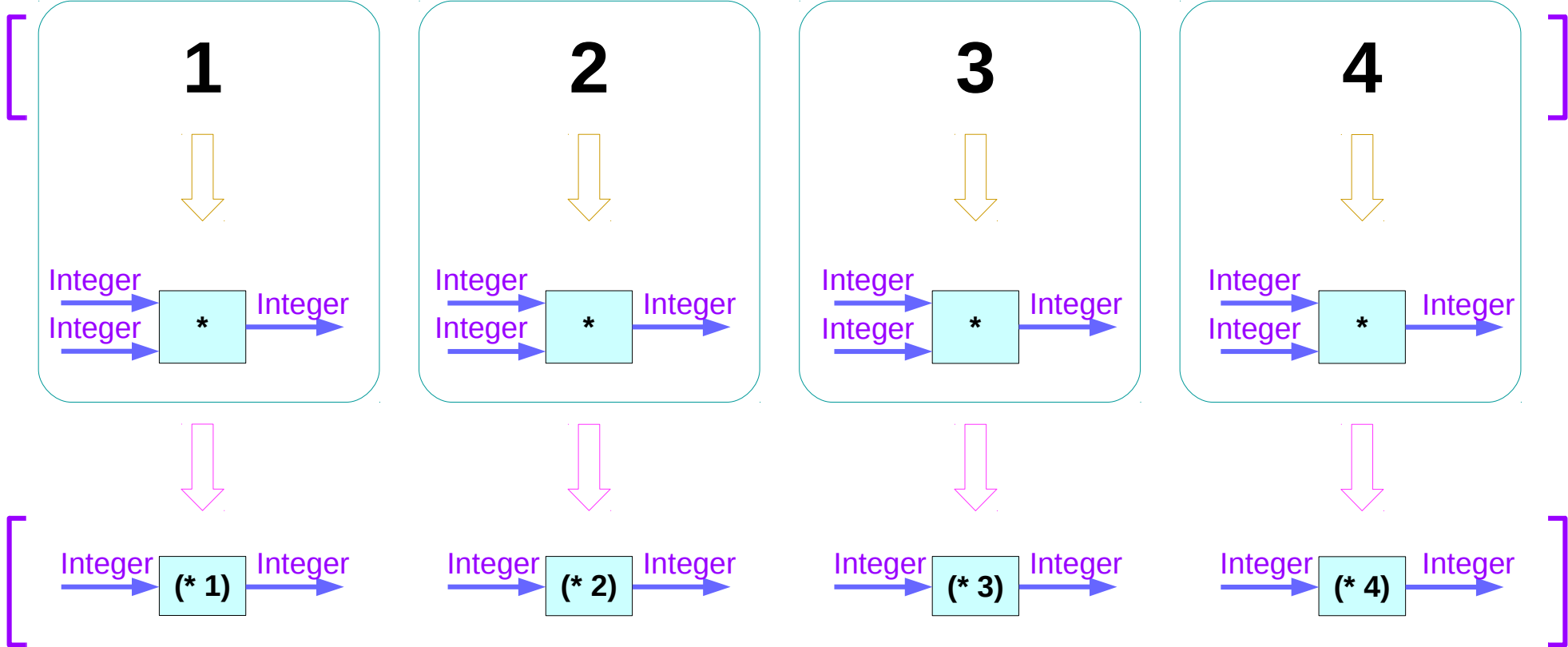
↑
Applicative

Functor →



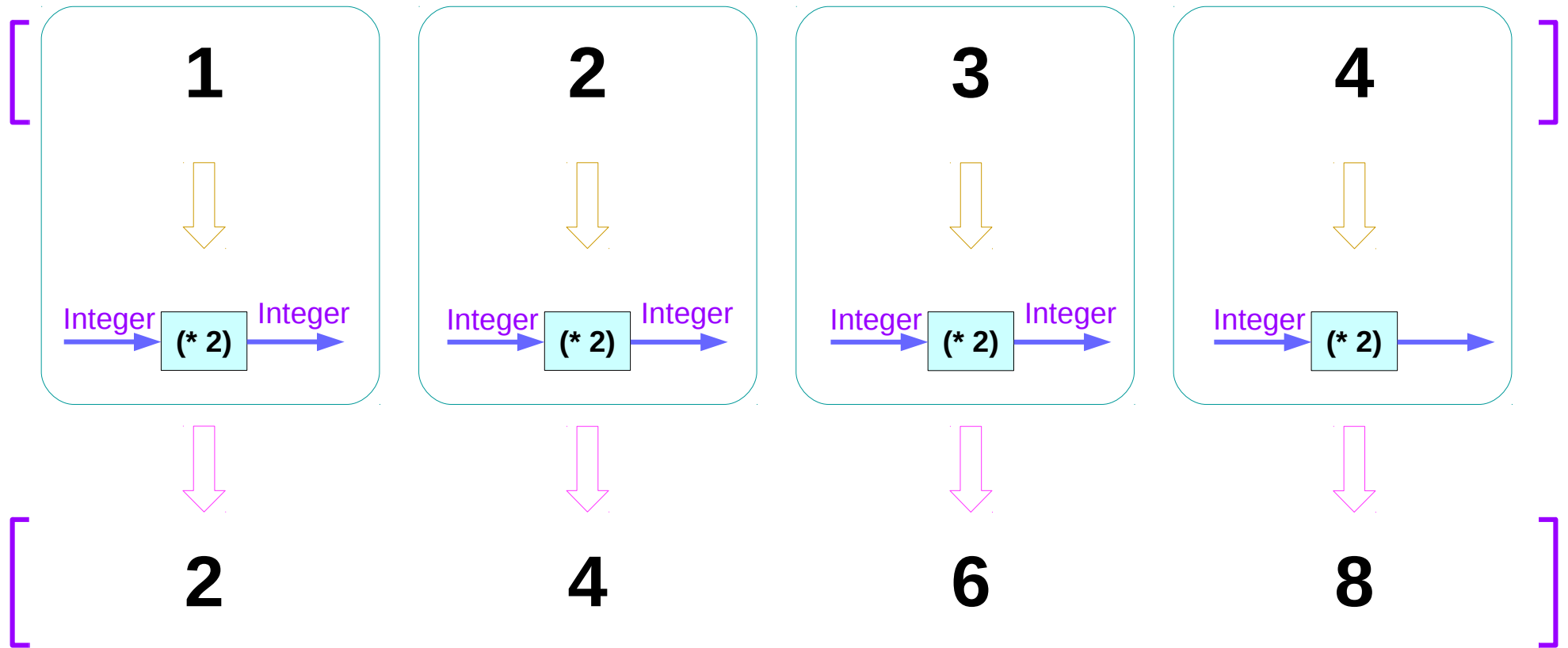
<http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

Applicative : Mapping functions



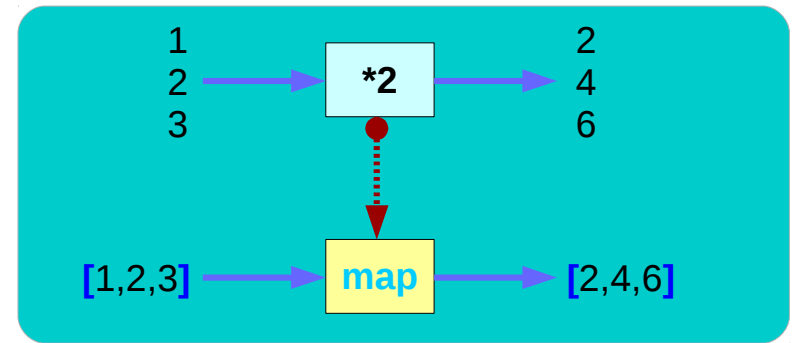
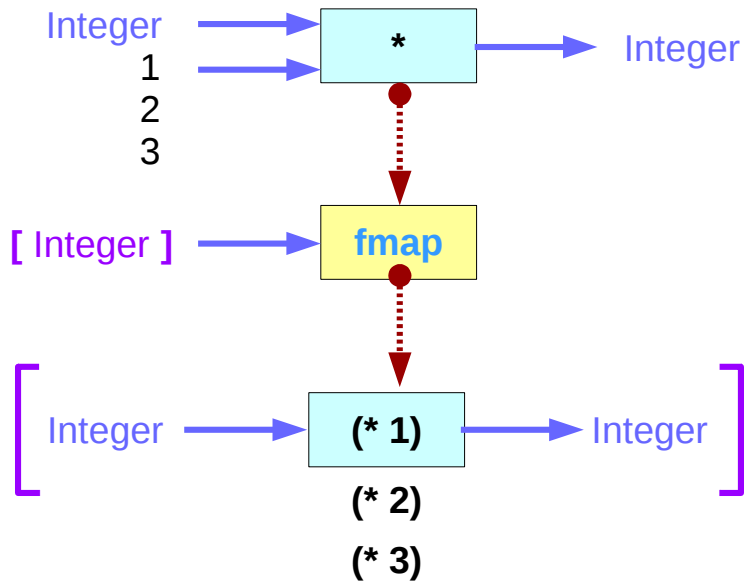
A list of functions

Functor : Mapping values



A list of integers

Applicatives vs. Functors



Double applications of `fmap` (1)

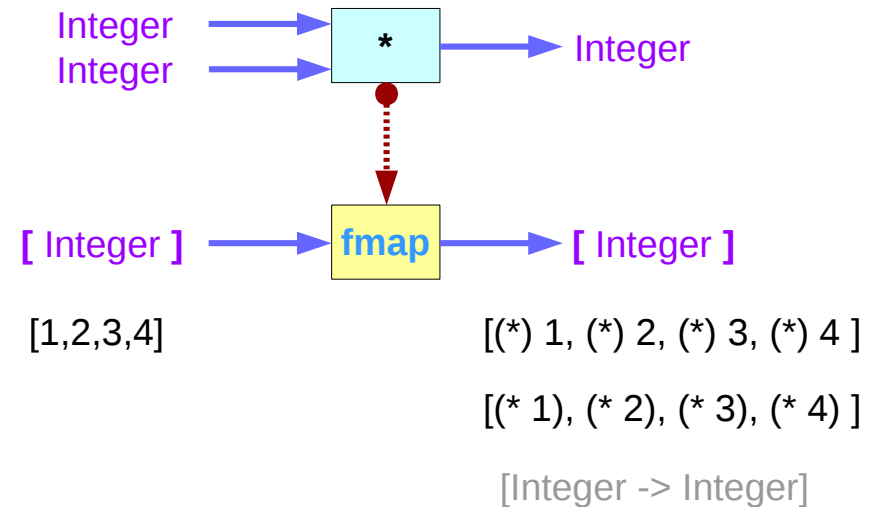
```
ghci> let a = fmap (*) [1,2,3,4]
```

```
ghci> :t a
```

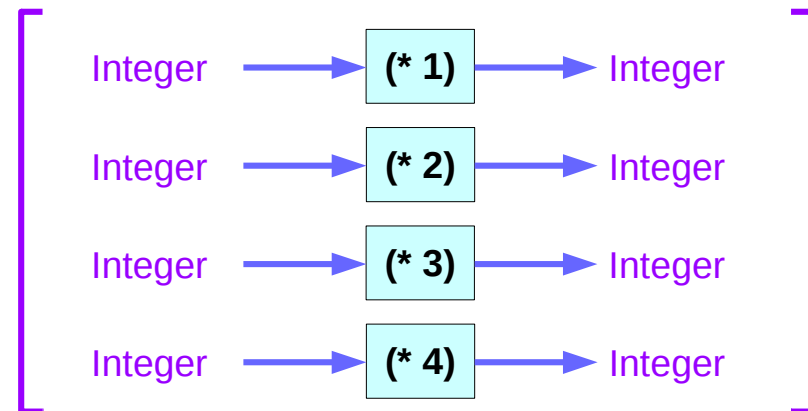
```
a :: [Integer -> Integer]
```

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

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



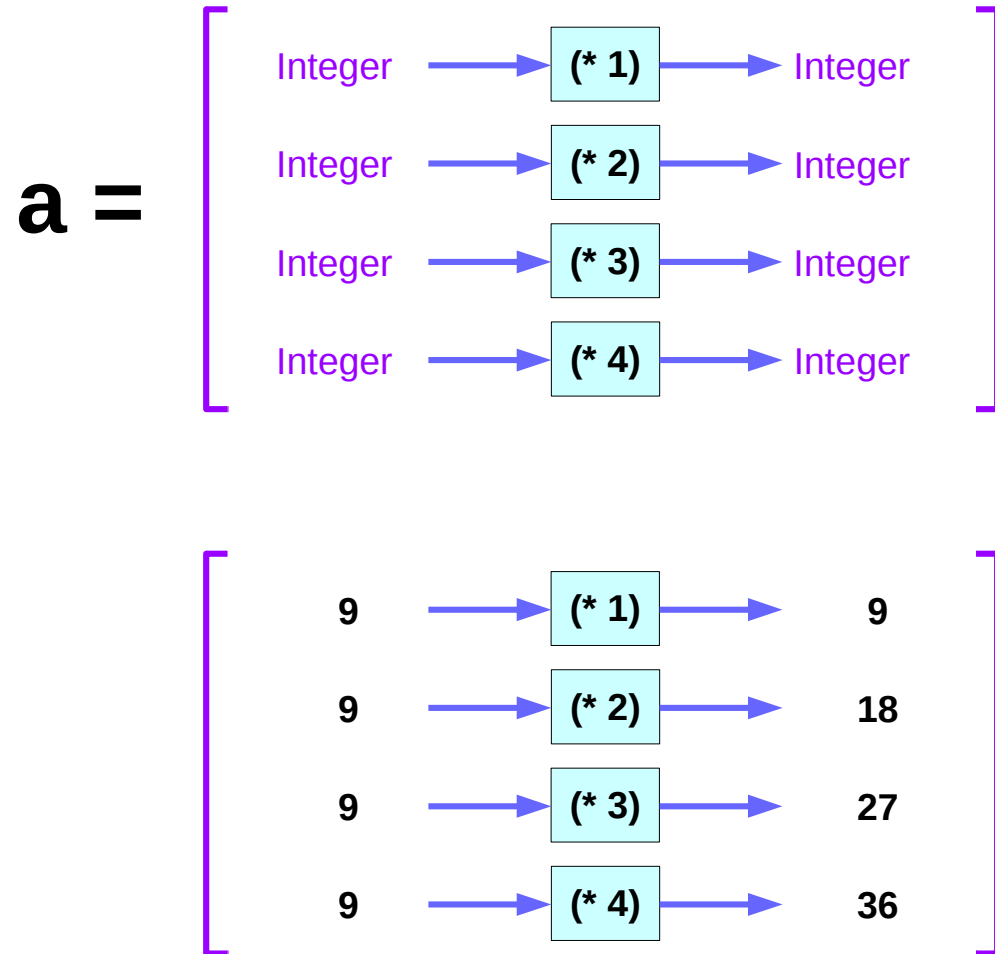
a =



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

Double applications of `fmap` (2)

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



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

Applications of `fmap`

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

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

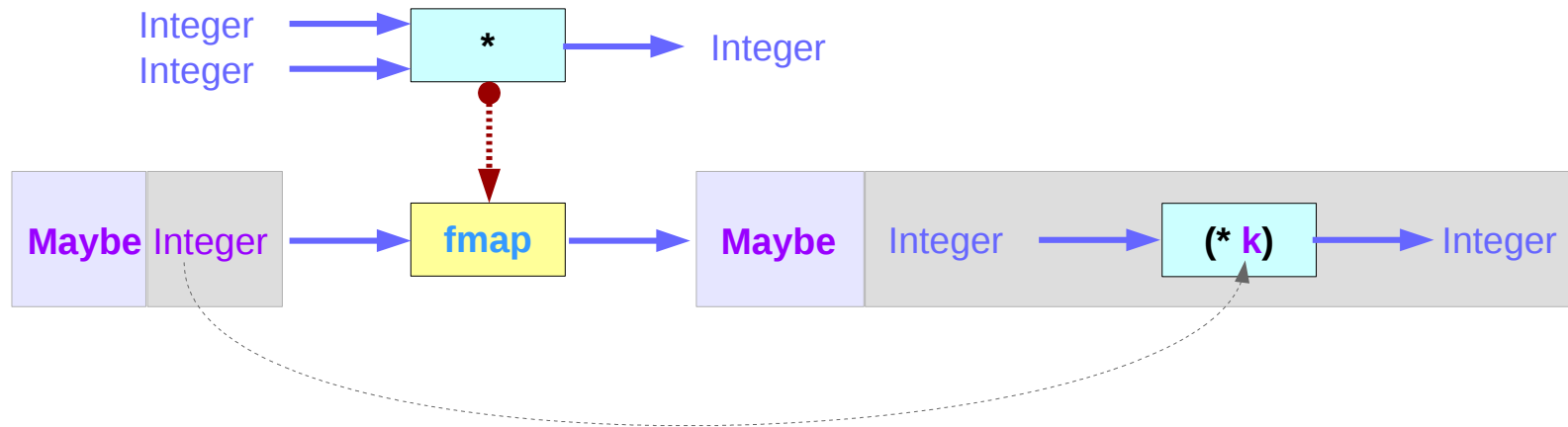
```
[(* 1), (* 2), (* 3), (* 4)]
```

```
fmap (\f -> f 9) [(* 1), (* 2), (* 3), (* 4)]
```

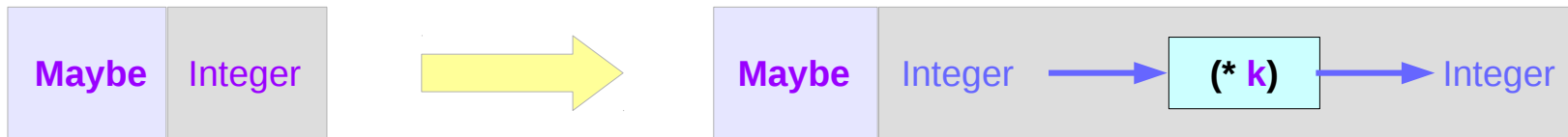
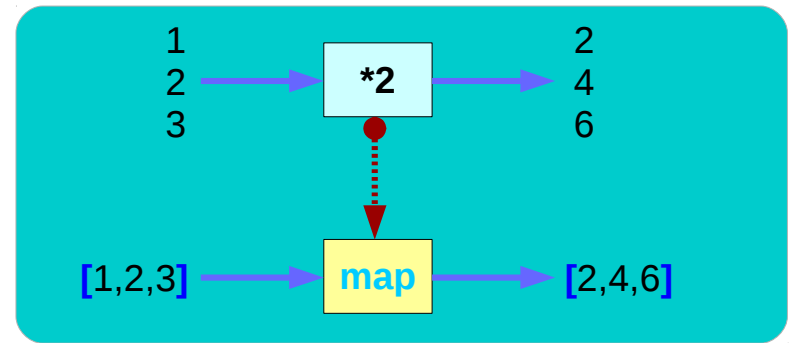
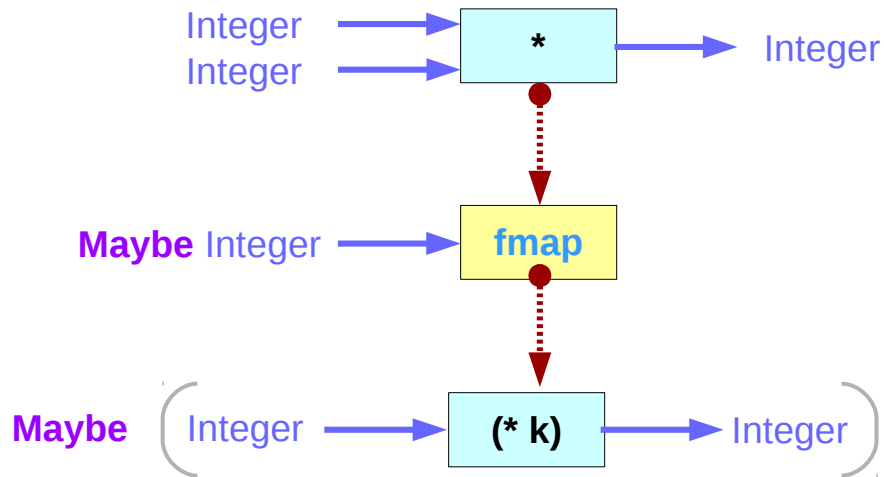
```
[9,18,27,36]
```

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

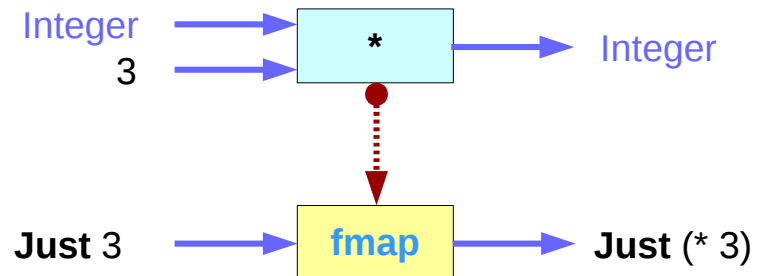
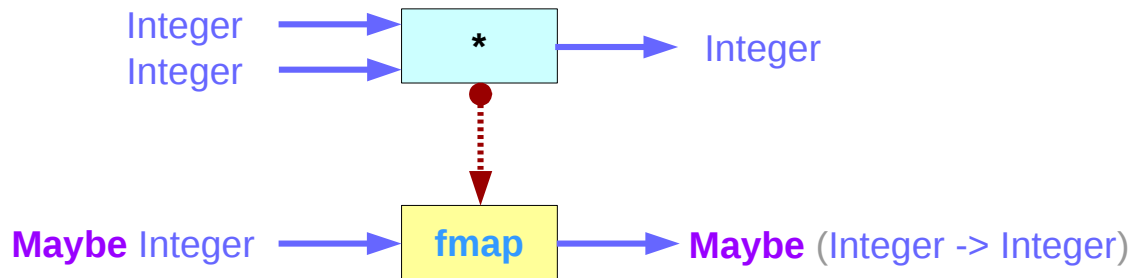
Mapping functions over the Functor Maybe (1)



Mapping functions over the Functor Maybe (2)

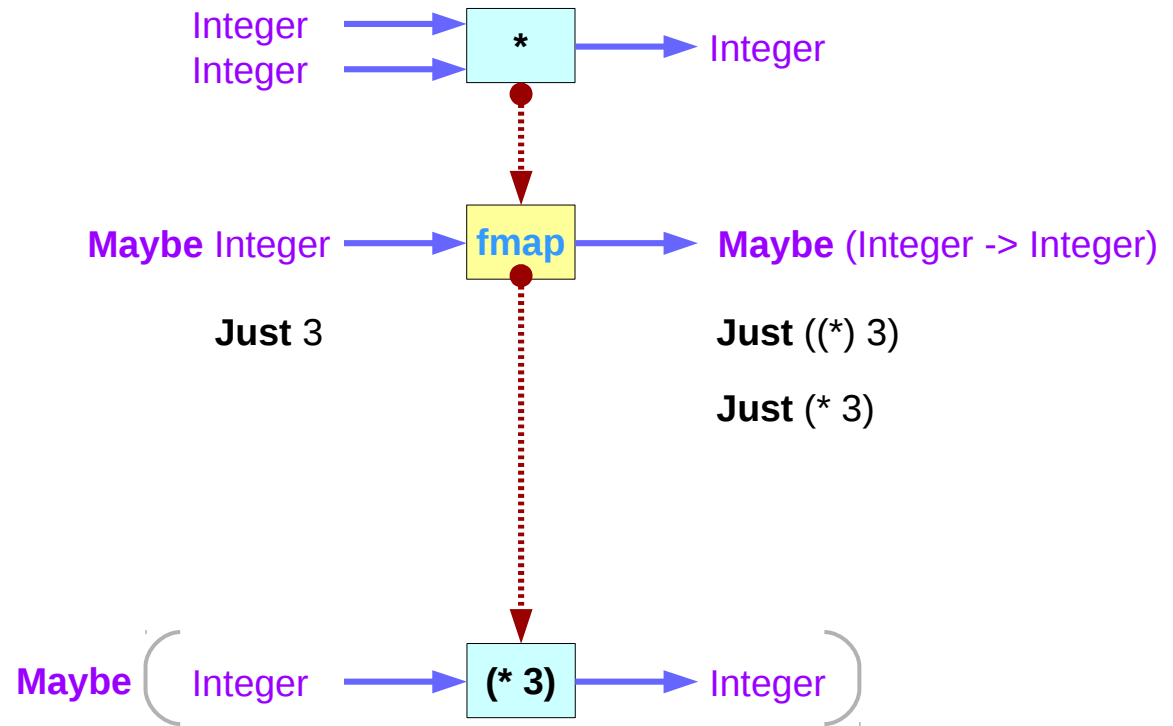


Mapping functions over the Functor Maybe (3)



Function wrapped in Just

`fmap (*) (Just 3)`



function wrapped in a `Just`

`Just (* 3)`

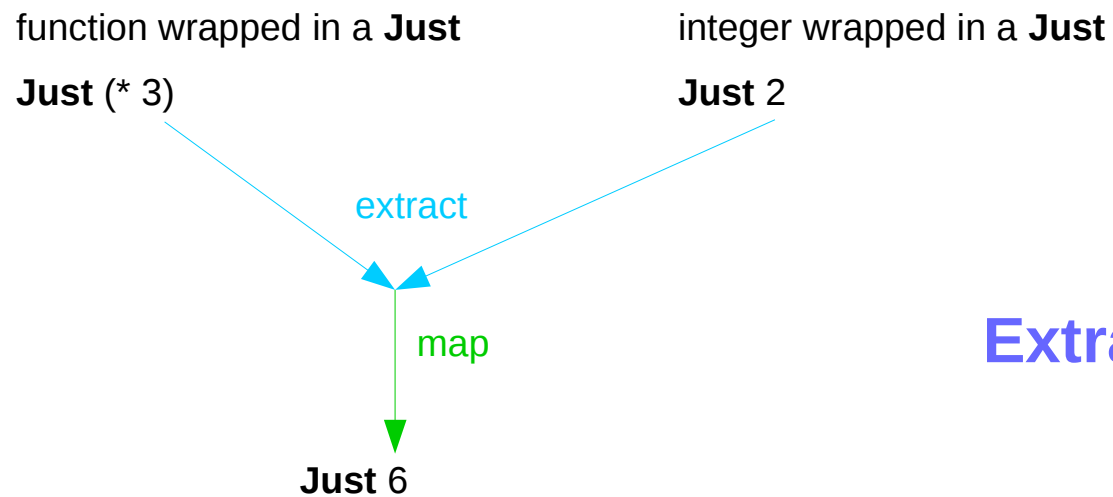
integer wrapped in a `Just`

`Just 2`

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

<*> Application of a function

Just (* 3) <*> Just 2



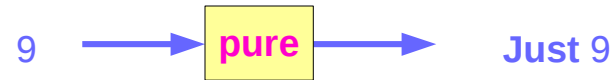
Extracting and **Mapping**

Just 6

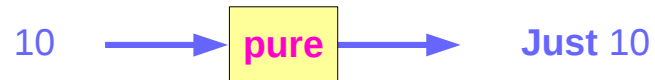
<http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

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**

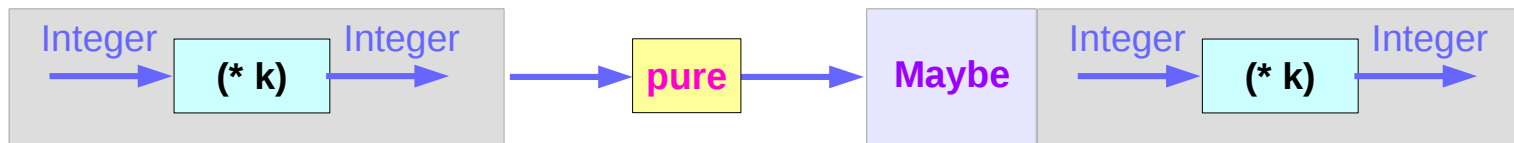
to wrap a **function**

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

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

(+3) → **pure** → **Just** (+3)

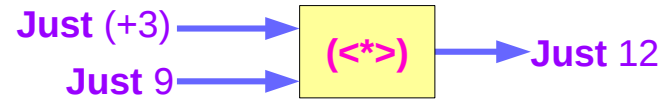
(++"haha") → **pure** → **Just** (++"haha")



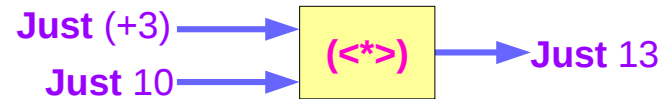
<http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

Applicative Functor Apply <*> Examples

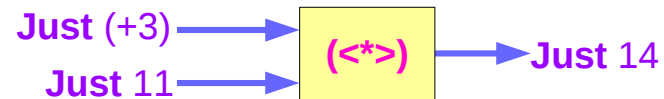
```
ghci> Just (+3) <*> Just 9
Just 12
```



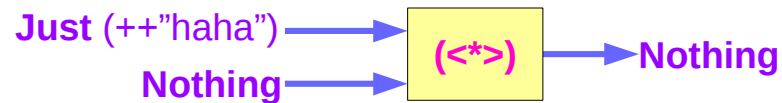
```
ghci> pure (+3) <*> Just 10
Just 13
```



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



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



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



<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>