

Applicative (2A)

Copyright (c) 2016 - 2017 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 OpenOffice.

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.

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

$f\ x\ y$

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

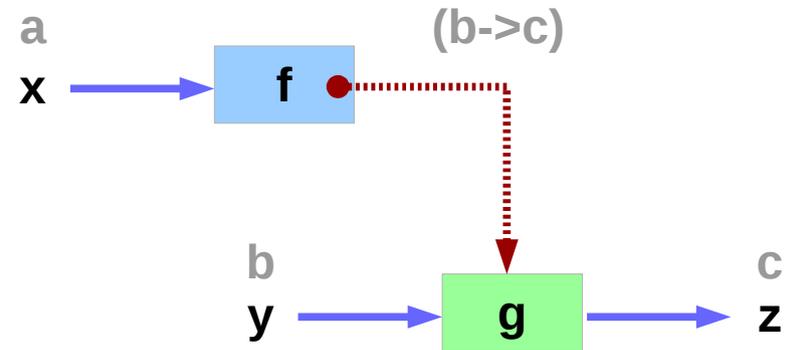
$(f\ x)\ y$

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

$g\ y$

$g :: b \rightarrow c$

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



<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**.

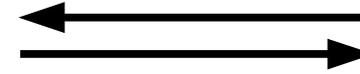
all functions are considered **curried**

all functions take **just one argument**

the curried form

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

currying



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

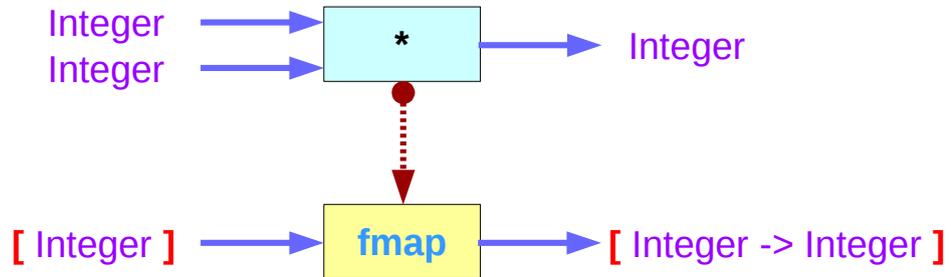
uncurrying

$f \ x \ y$

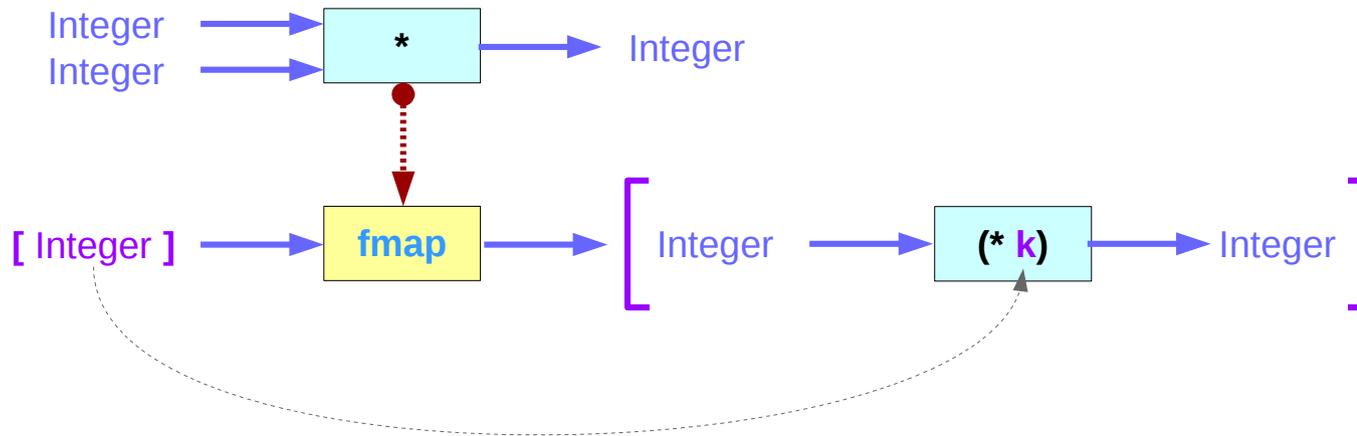
$g \ (x,y)$

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

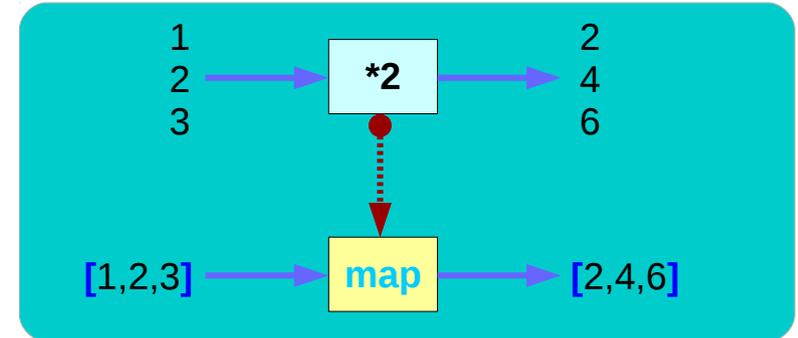
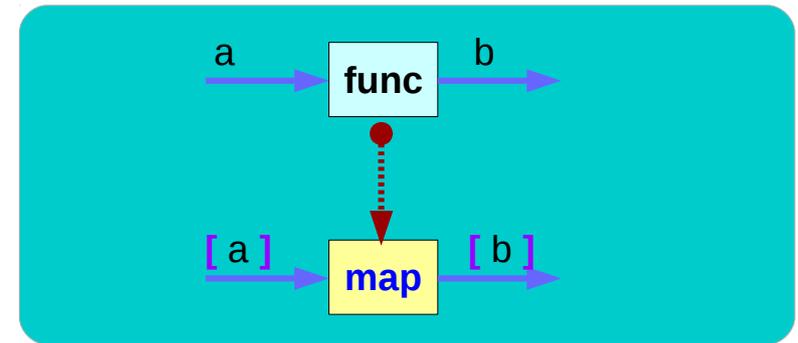
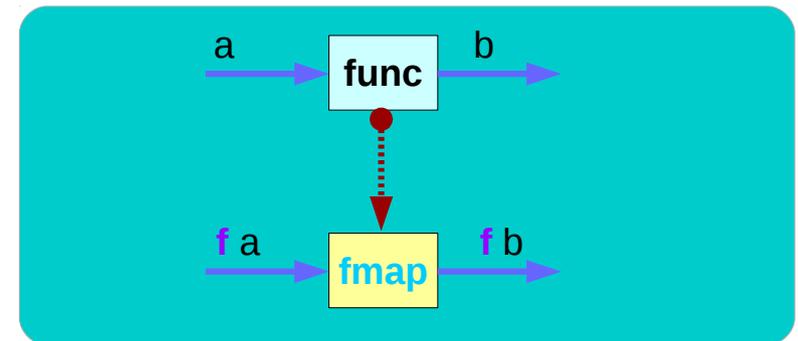
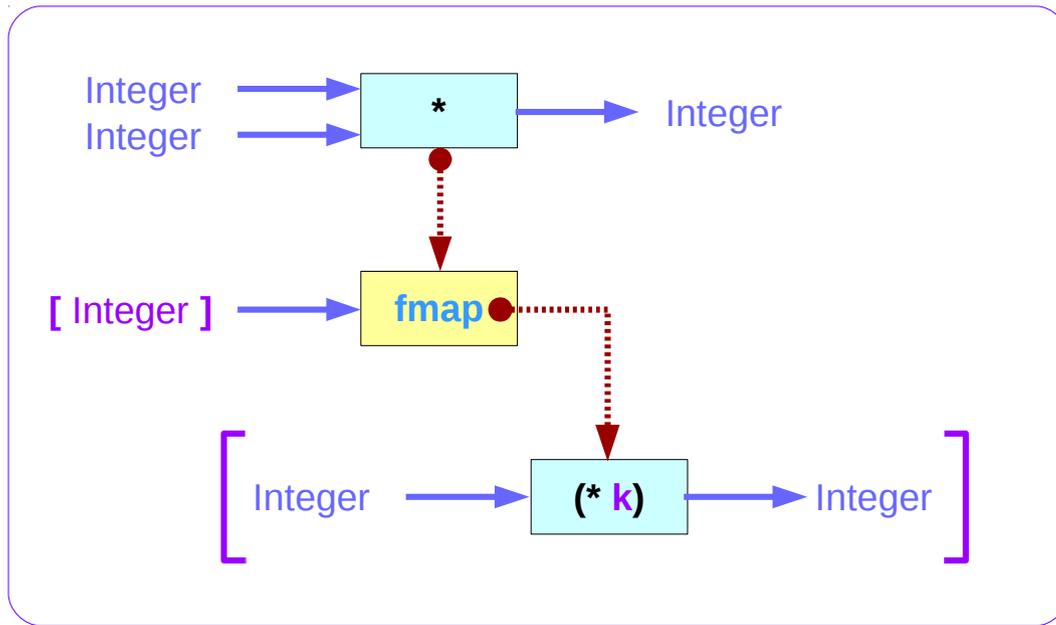
Mapping functions over the Functor [] (1)



Mapping functions over the Functor [] (2)



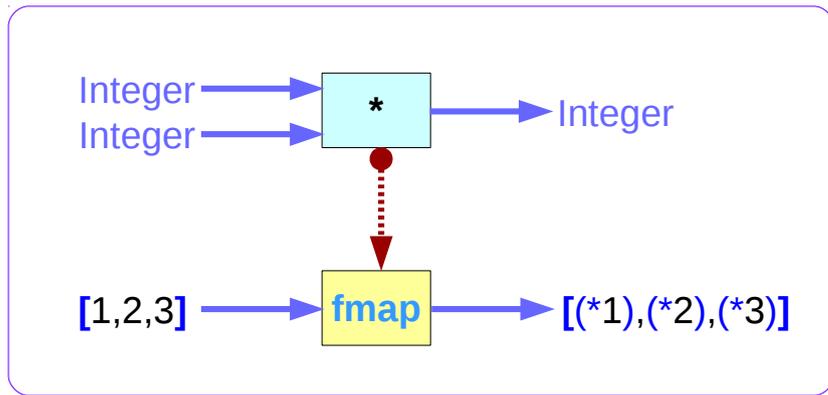
Mapping functions over the Functor [] (3)



Functor →

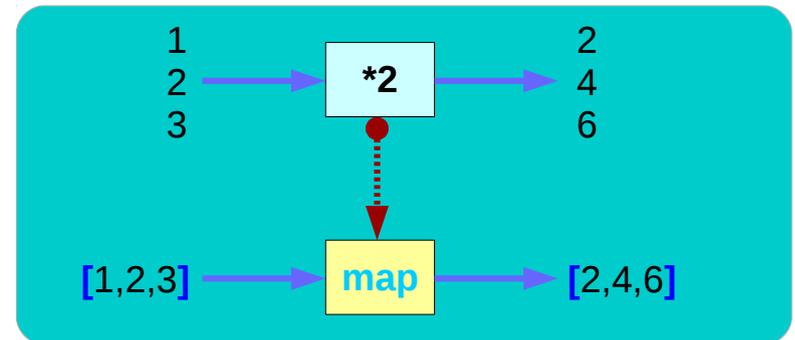
<http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

Mapping functions over the Functor [] (4)



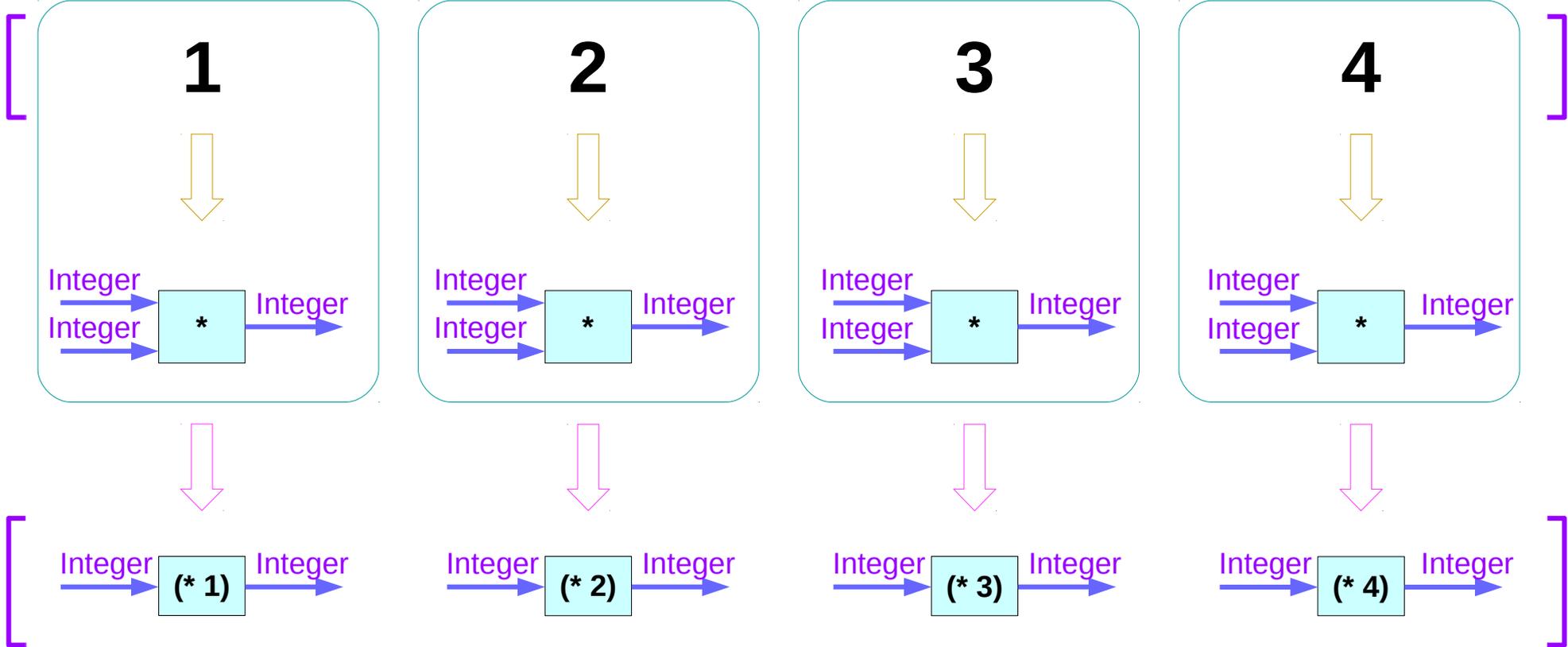
↑
Applicative

Functor →



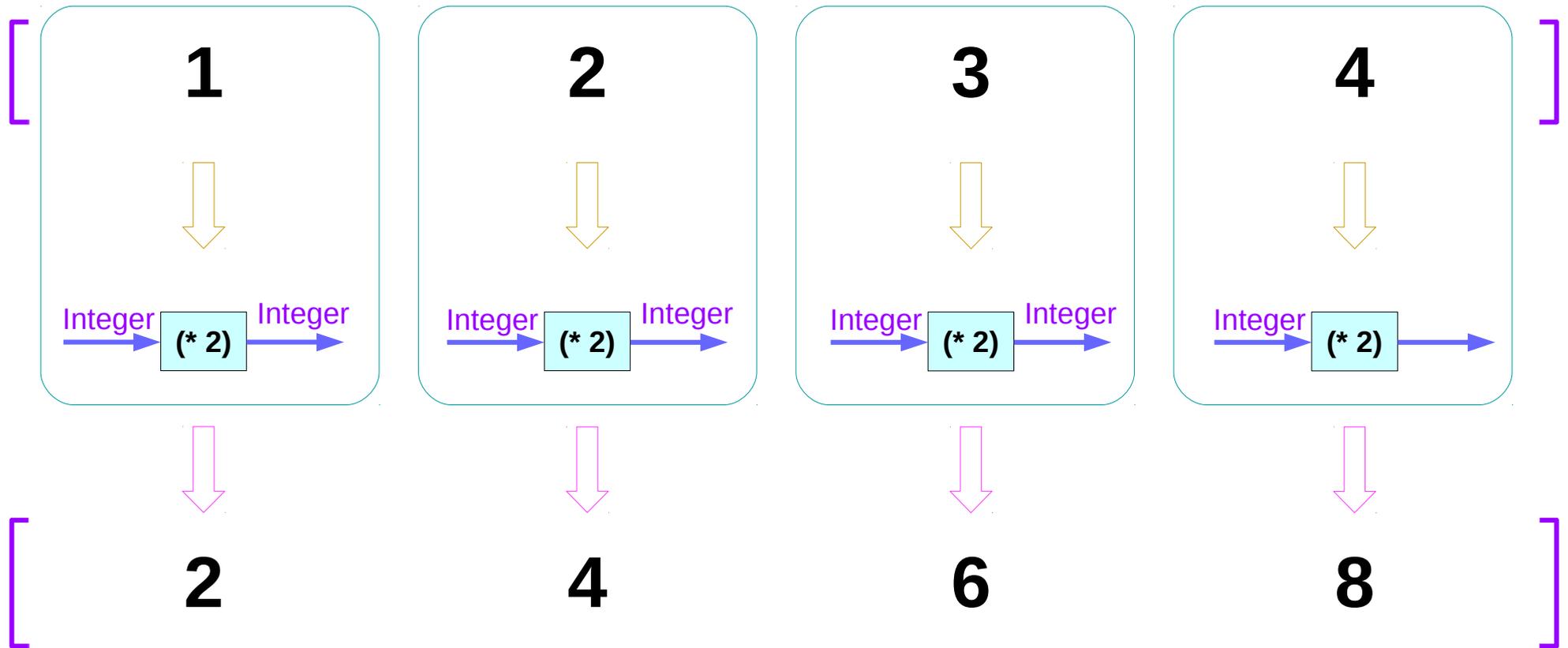
<http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

Applicative : Mapping functions



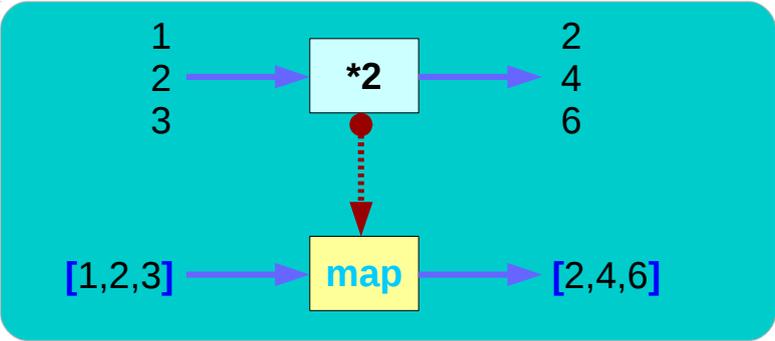
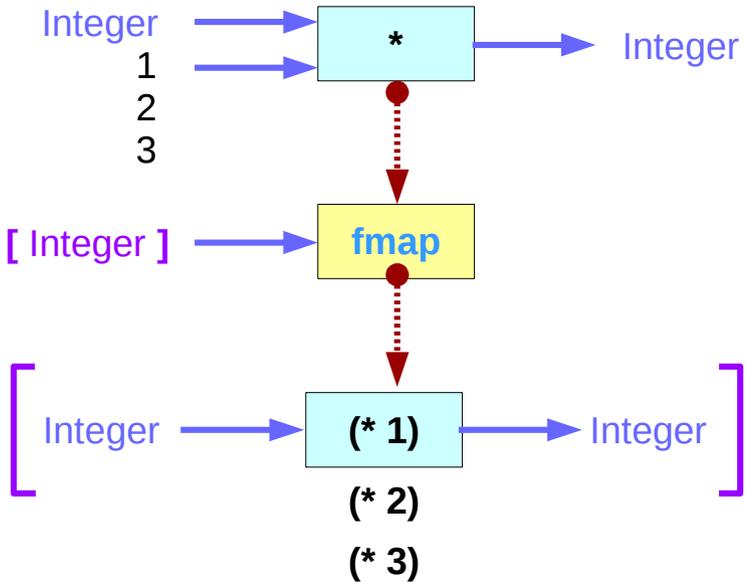
A list of functions

Functor : Mapping values



A list of integers

Applicatives vs. Functors



Double applications of `fmap` (1)

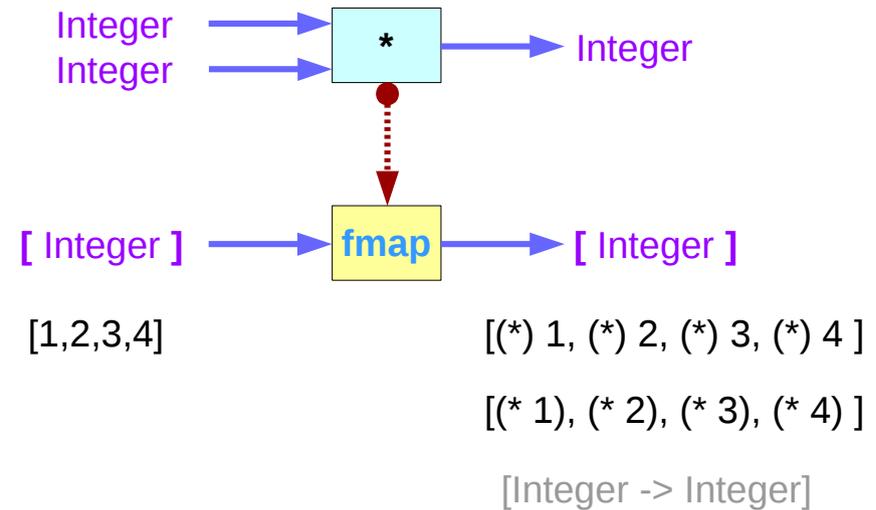
```
ghci> let a = fmap (*) [1,2,3,4]
```

```
ghci> :t a
```

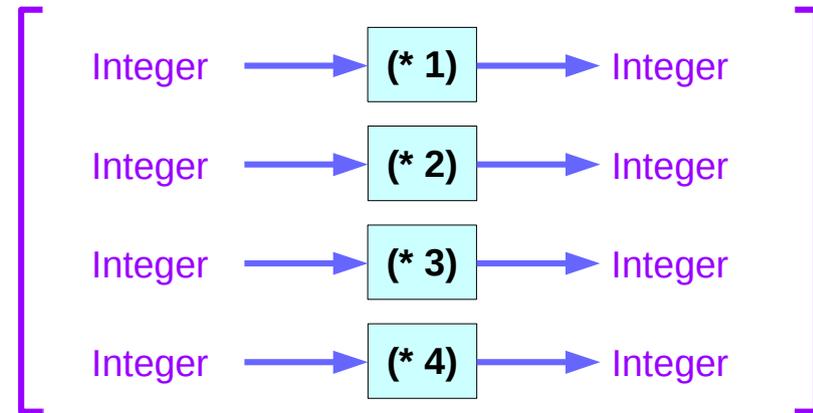
```
a :: [Integer -> Integer]
```

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

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



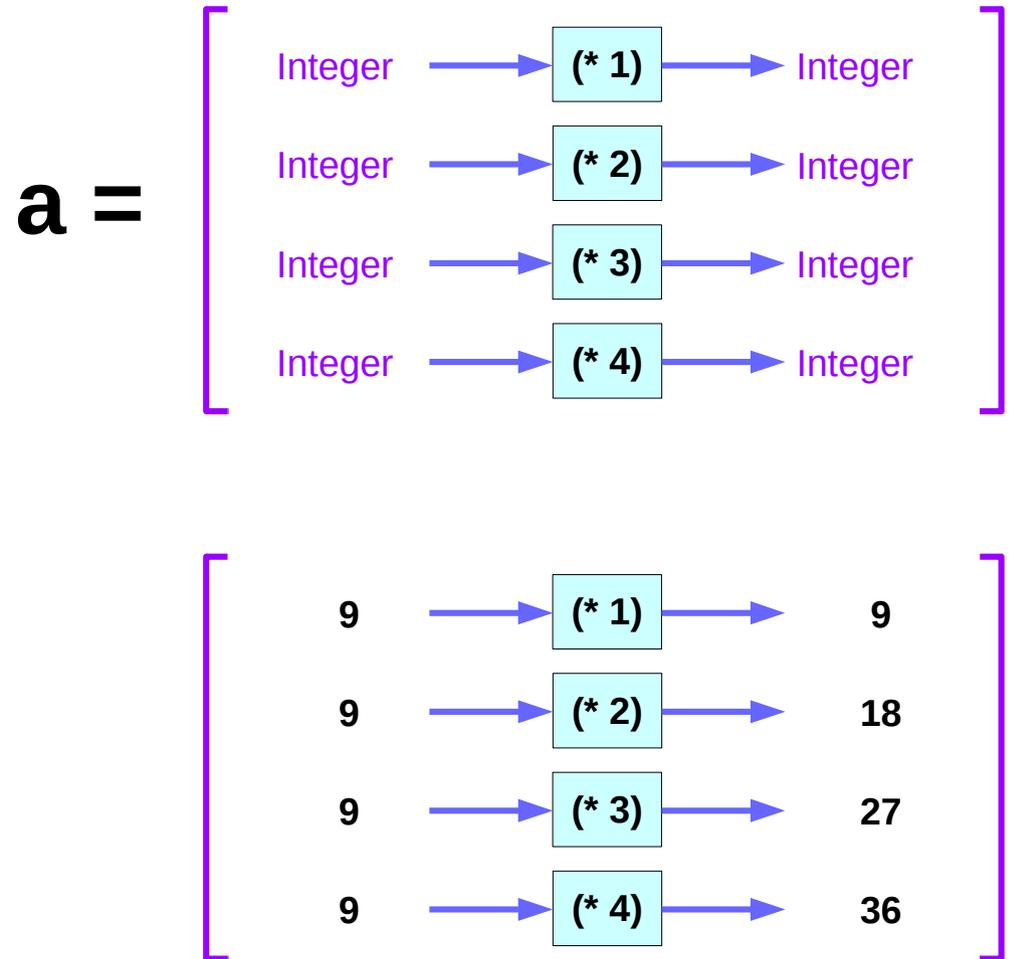
a =



<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]
```

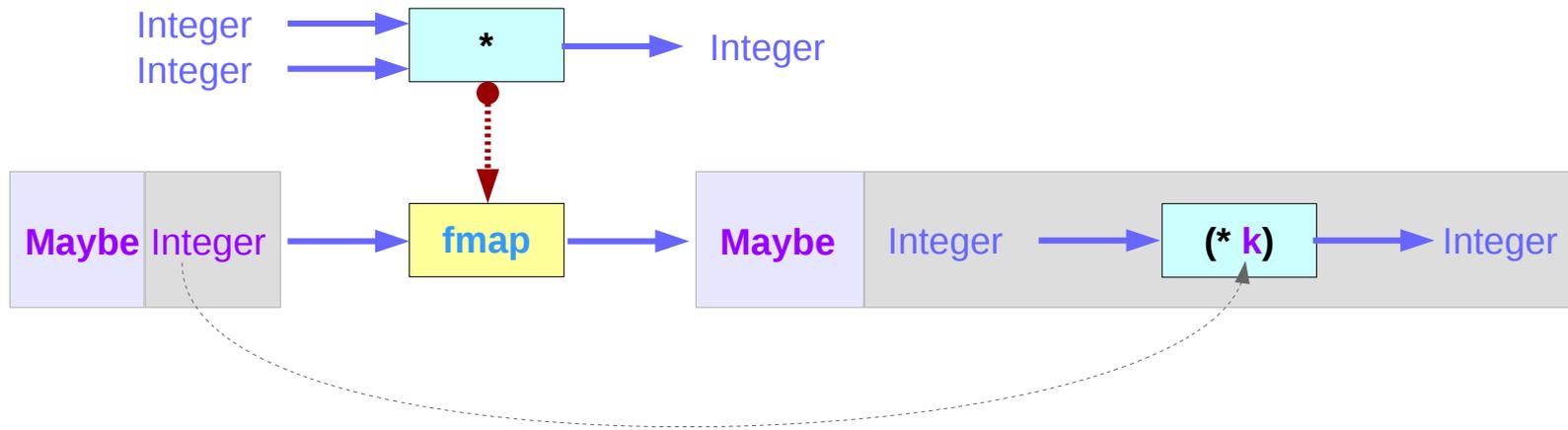
```
[(* 1), (* 2), (* 3), (* 4)]
```

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

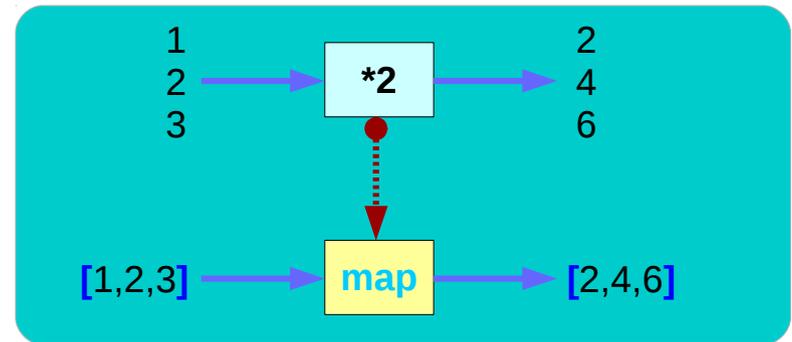
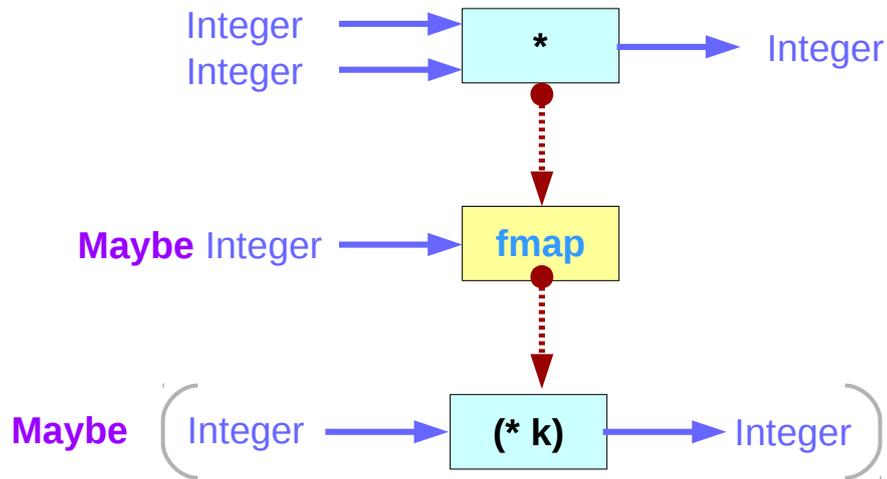
```
[9,18,27,36]
```

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

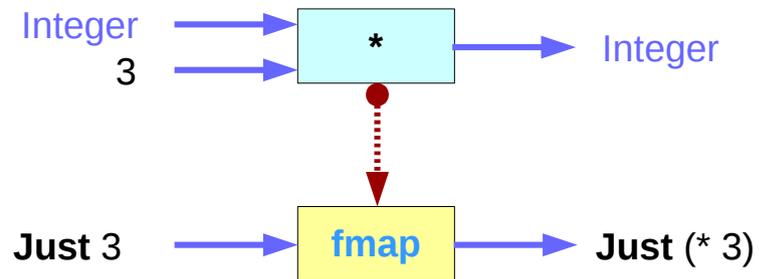
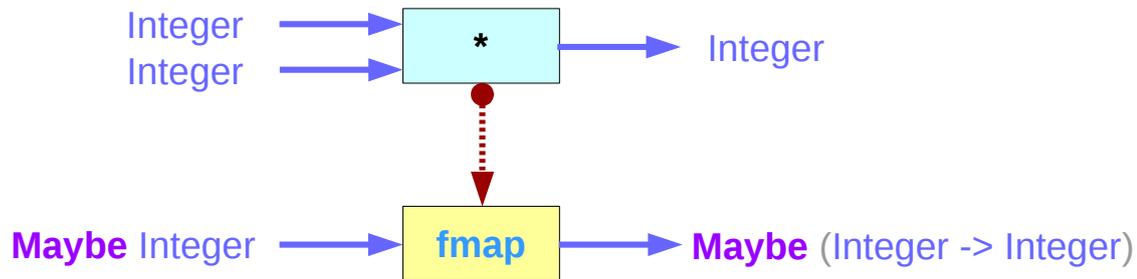
Mapping functions over the Functor Maybe (1)



Mapping functions over the Functor Maybe (2)

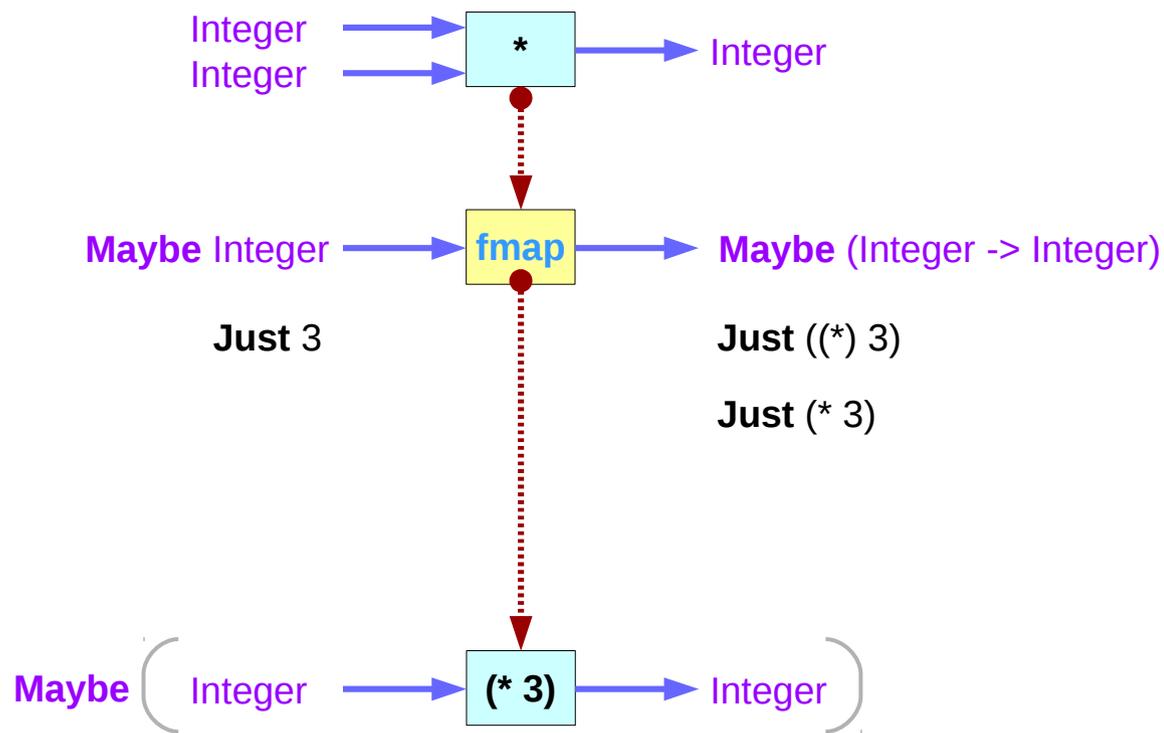


Mapping functions over the Functor Maybe (3)



Function wrapped in Just

`fmap (*) (Just 3)`



function wrapped in a `Just`

`Just (* 3)`

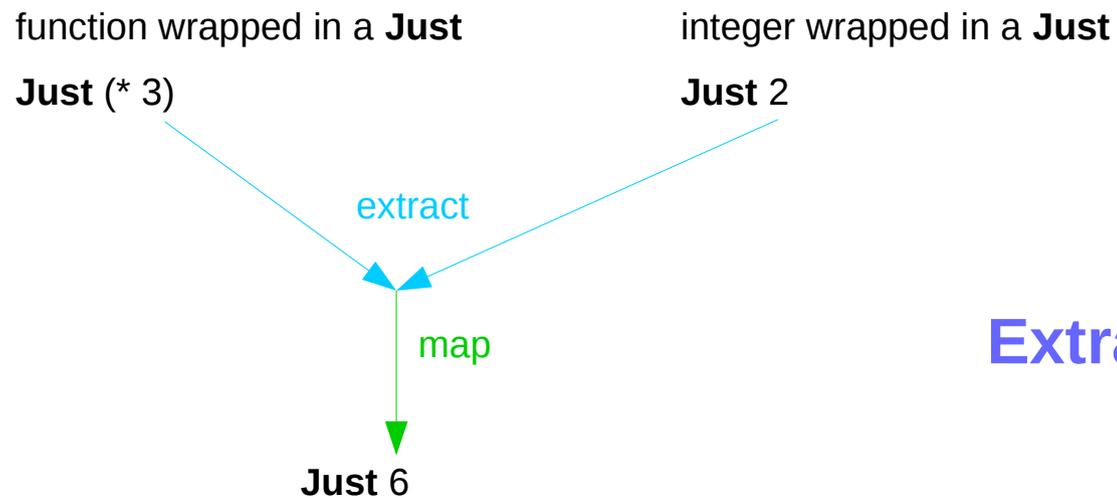
integer wrapped in a `Just`

`Just 2`

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

<*> Application of a function

Just (* 3) <*> Just 2



Extracting and **Mapping**

Just 6

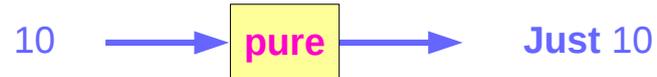
<http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

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**

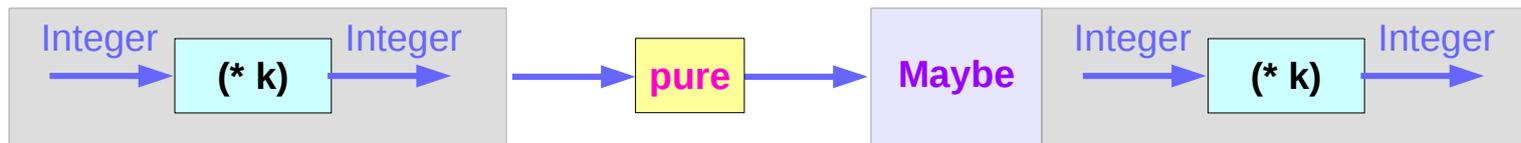
to wrap a **function**

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

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

(+3) → **pure** → **Just** (+3)

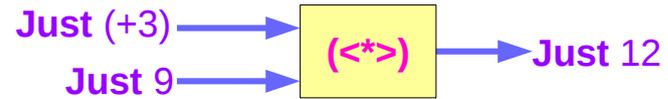
(++"haha") → **pure** → **Just** (++"haha")



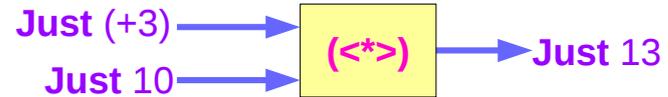
<http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

Applicative Functor Apply <*> Examples

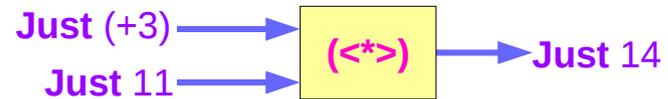
```
ghci> Just (+3) <*> Just 9
Just 12
```



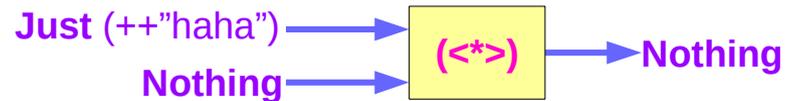
```
ghci> pure (+3) <*> Just 10
Just 13
```



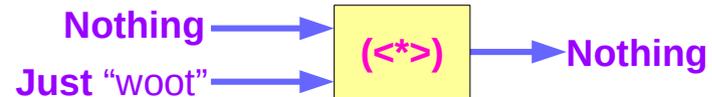
```
ghci> pure (+3) <*> Just 11
Just 12
```



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



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



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

The Applicative Typeclass

Functor

lifts a “normal” function
to a function on computational contexts

fmap of **Functors** cannot
apply a function which is itself in a context
to a value in a context.

fmap method

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

f (a -> b)

Applicative

(<*>) (variously pronounced as “**apply**”, “**app**”, or “**splat**”)
pure, for **embedding** values in a default, “effect free” **context**.

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

App <*>

<*> App

the type of <*> is similar to the type of (\$) but with everything enclosed in an functor f.

<*> is just function application within a computational context.

The type of <*> is also similar to the type of fmap; the only difference is that the first parameter of <*> is

f (a -> b), a function in a context f, instead of a “normal” function (a -> b).

(\$) is just function application: **func \$ x = func x**

pure (+3) <*> Just 11  **Just 14**

Just (+3) <*> Just 11  **Just 14**

fmap (*2) (Just 200)  **Just 400**

Just (*2) <*> (Just 200)  **Just 400**

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

fmap V.S. <*>

fmap (*2) (Just 200) → Just 400

Just (+3) <*> (Just 200) → Just 203

fmap func Fval

(a -> b)

(*2) :: Integer -> Integer

Ffunc <*> Fval

f (a -> b)

Just (+3) :: Maybe (Integer -> Integer)

the first parameter of (<*>) is

f (a -> b), a function in a context f,

instead of a “normal” function **(a -> b)**.

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

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

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

pure

pure takes a value of any type **a**, and returns a context/container of type **f a**.

to create a “default” **container**
or “effect free” **context**.

the behavior of **pure** is quite constrained by the laws that must be satisfied in conjunction with (**<*>**).

Usually, for a given implementation of (**<*>**) there is only one possible implementation of **pure**.

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

The definition of Applicative

```
class (Functor f) => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

The class has a two methods :

pure brings arbitrary values into the functor

(<*>) takes a function wrapped in a functor **f**
and a value wrapped in a functor **f**
and returns the result of the application
which is also wrapped in a functor **f**

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

The Maybe instance of Applicative

```
instance Applicative Maybe where
```

```
  pure          = Just
```

```
  (Just f) <*> (Just x) = Just (f x)
```

```
  _          <*> _      = Nothing
```

`pure` wraps the value with `Just`;

`<*>` applies

the function wrapped in `Just`

to the value wrapped in `Just` if both exist,

and results in `Nothing` otherwise.

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

The Applicative Typeclass

```
class (Functor f) => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

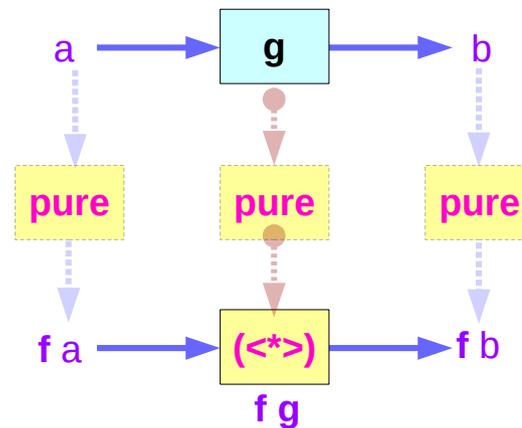
f : Functor, Applicative

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  (Just f) <*> something = fmap f something
```

f : function in a context



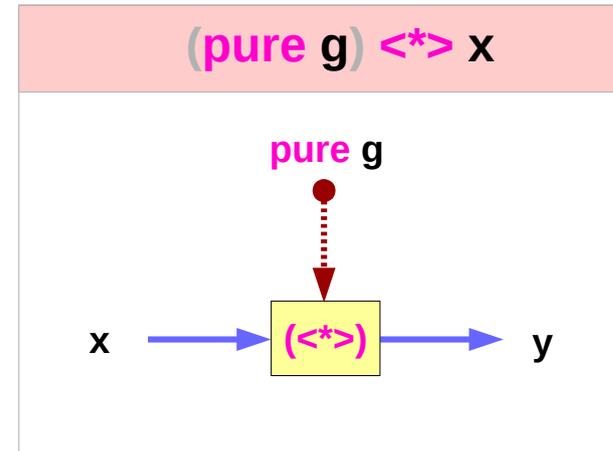
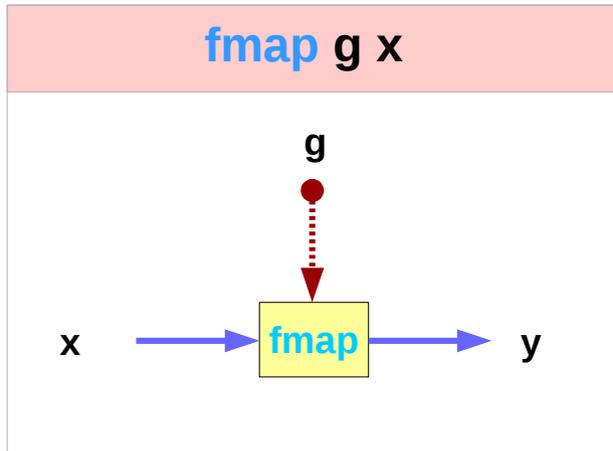
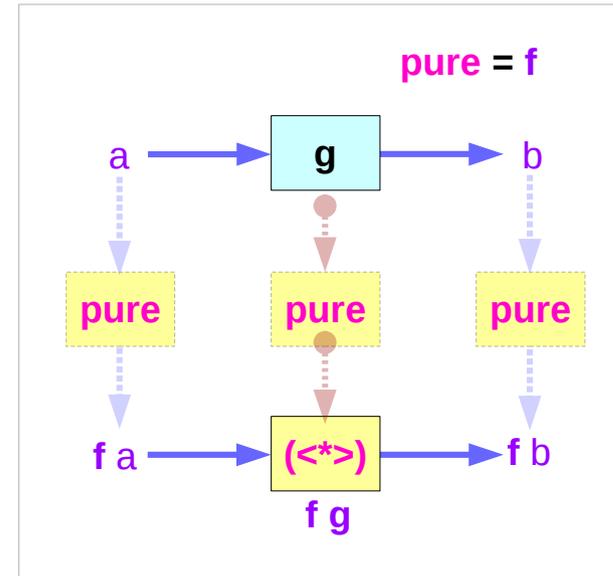
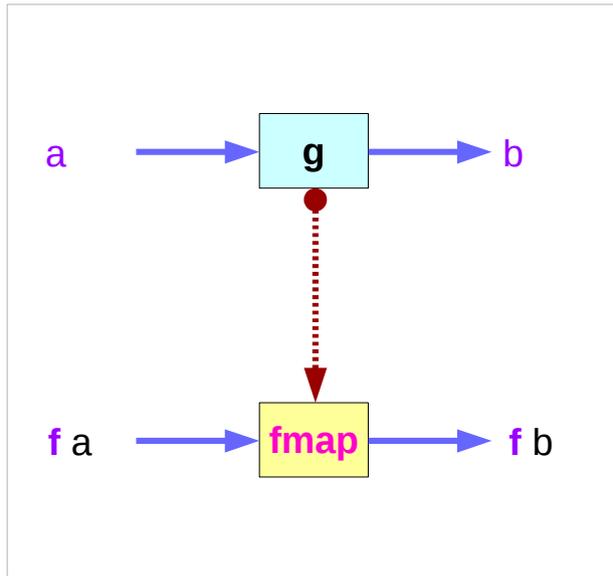
(Functor f) => Applicative f



(Functor f) => Applicative f

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

$$\text{fmap } g \ x = (\text{pure } g) \langle * \rangle x$$



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

Left Associative <*>

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

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

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

```
Just 8
```

```
ghci> pure (+) <*> Just 3 <*> Nothing  
Nothing
```

```
ghci> pure (+) <*> Nothing <*> Just 5  
Nothing
```

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

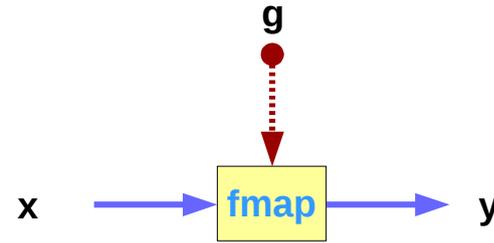
Infix Operator $\langle \$ \rangle$

pure f $\langle * \rangle$ x $\langle * \rangle$ y $\langle * \rangle$ z

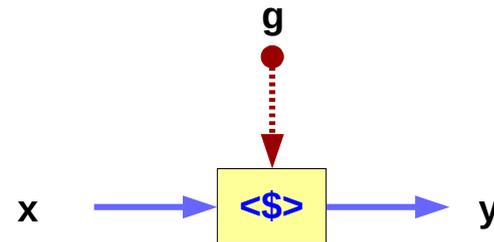
fmap f x $\langle * \rangle$ y $\langle * \rangle$ z

f $\langle \$ \rangle$ x $\langle * \rangle$ y $\langle * \rangle$ z

fmap g x



g $\langle \$ \rangle$ x



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

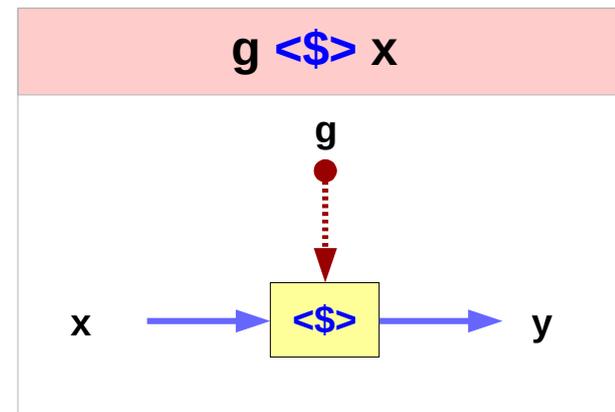
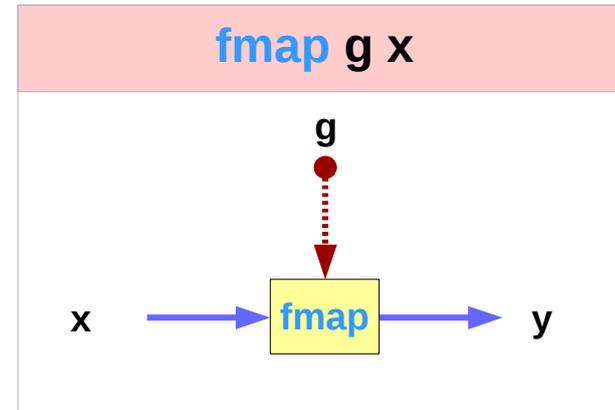
Infix Operator $\langle \$ \rangle$: not a class method

```
class (Functor f) => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

not a class method

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
f <$> x = fmap f x
```

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  (Just f) <*> something = fmap f something
```



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

The Applicative Typeclass

Applicative is a superclass of **Monad**.

every **Monad** is also a **Functor** and an **Applicative**

fmap, **pure**, **(<*>)** can all be used with **monads**.

a **Monad** instance also requires

Functor and **Applicative** instances.

the types and roles of **return** and **(>>)**

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

(*> v.s. >>) and (pure v.s. return)

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

(>>) :: **Monad** m => m a -> m b -> m b

pure :: **Applicative** f => a -> f a

return :: **Monad** m => a -> m a

the constraint changes from **Applicative** to **Monad**.

(*>) in **Applicative**

pure in **Applicative**

(>>) in **Monad**

return in **Monad**

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

The Applicative Laws

The identity law: $\text{pure id} \langle * \rangle v = v$

Homomorphism: $\text{pure f} \langle * \rangle \text{pure x} = \text{pure (f x)}$

Interchange: $u \langle * \rangle \text{pure y} = \text{pure (\$ y)} \langle * \rangle u$

Composition: $u \langle * \rangle (v \langle * \rangle w) = \text{pure (.)} \langle * \rangle u \langle * \rangle v \langle * \rangle w$

The Identity Law

The identity law

pure id <*> v = v

pure to inject values into the functor
in a default, featureless way,
so that the result is as close as possible to the plain value.

applying the **pure id** morphism does nothing,
exactly like with the plain **id** function.

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

The Homomorphism Law

The homomorphism law

$\text{pure } f \text{ <*> pure } x = \text{pure } (f \ x)$

applying a "**pure**" function to a "**pure**" value is the same as applying the function to the value in the normal way and then using **pure** on the result.
means **pure** preserves function application.

applying a non-effectful function **f** to a non-effectful argument **x** in an effectful context **pure** is the same as just **applying** the function **f** to the argument **x** and then injecting the result (**f x**) into the context with **pure**.

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

The Interchange Law

The interchange law

$$u \langle * \rangle \text{pure } y = \text{pure } (\$ y) \langle * \rangle u$$

applying a morphism u to a "pure" value $\text{pure } y$ is the same as applying $\text{pure } (\$ y)$ to the morphism u

$(\$ y)$ is the function that supplies y as argument to another function
– the higher order functions

when evaluating the application of an effectful function u to a pure argument $\text{pure } y$, the order in which we evaluate the function u and its argument $\text{pure } y$ doesn't matter.

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

The Composition Law

The composition law

$$\text{pure } (.) \langle * \rangle u \langle * \rangle v \langle * \rangle w = u \langle * \rangle (v \langle * \rangle w)$$

pure (.) composes morphisms similarly
to how (.) composes functions:

$$(f . g) x = f (g x)$$

$$\begin{aligned} & \text{pure } (.) \langle * \rangle \text{pure } f \langle * \rangle \text{pure } g \langle * \rangle \text{pure } x \\ &= \text{pure } f \langle * \rangle (\text{pure } g \langle * \rangle \text{pure } x) \end{aligned}$$

$$\begin{aligned} u &= \text{pure } f \\ v &= \text{pure } g \\ w &= \text{pure } x \end{aligned}$$

applying the composed morphism **pure** (.) $\langle * \rangle$ **u** $\langle * \rangle$ **v** to **w**
gives the same result as applying **u** $\langle * \rangle$ **v** to **w**

$$\begin{aligned} & u \\ & \langle * \rangle \\ & (v \langle * \rangle w) \end{aligned}$$

it is expressing a sort of associativity property of ($\langle * \rangle$).

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

The Applicative Typeclass

a bonus law about the relation between `fmap` and `(<*>)`:

```
fmap f x = pure f <*> x           -- fmap
```

Applying a "pure" function with `(<*>)`
is equivalent to using `fmap`.

This law is a consequence of the other ones,
so you need not bother with proving it
when writing instances of `Applicative`.

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

Sequencing of Effects (1)

```
>>> [(2*), (3*)] <*> [4, 5]      [(2*)] <*> [4, 5], [(3*)] <*> [4, 5]
[8, 10, 12, 15]
>>> [4, 5] <***> [(2*), (3*)]    [4] <***> [(2*), (3*)], [5] <***> [(2*), (3*)]
[8, 12, 10, 15]
```

```
Prelude> Just 2 *> Just 3
Just 3
Prelude> Just 3 *> Just 2
Just 2
Prelude> Just 2 *> Nothing
Nothing
Prelude> Nothing *> Just 2
Nothing
```

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

Sequencing of Effects

```
Prelude> (print "foo" *> pure 2) *> (print "bar" *> pure 3)
```

```
"foo"
```

```
"bar"
```

```
3
```

```
Prelude> (print "bar" *> pure 3) *> (print "foo" *> pure 2)
```

```
"bar"
```

```
"foo"
```

```
2
```

```
Prelude> (print "foo" *> pure 2) <*> (print "bar" *> pure 3)
```

```
"foo"
```

```
"bar"
```

```
2
```

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

Sequencing of Effects

By the way, if you hear about commutative monads in Haskell, the concept involved is the same, only specialised to Monad.

Commutativity (or the lack thereof) affects other functions which are derived from (`<*>`) as well. (`*>`) is a clear example:

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

(`*>`) combines effects while preserving only the values of its second argument.

For monads, it is equivalent to (`>>`).

Here is a demonstration of it using Maybe, which is commutative:

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

Sequencing of Effects

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

from `Control.Applicative` is not `flip (<*>)`.

That means it provides a way of inverting the sequencing:

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

Sequencing of Effects

The convention in Haskell is to always implement (`<*>`) and other applicative operators using left-to-right sequencing. Even though this convention helps reducing confusion, it also means appearances sometimes are misleading. For instance, the (`<*`) function is not flip (`*>`), as it sequences effects from left to right just like (`*>`):

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

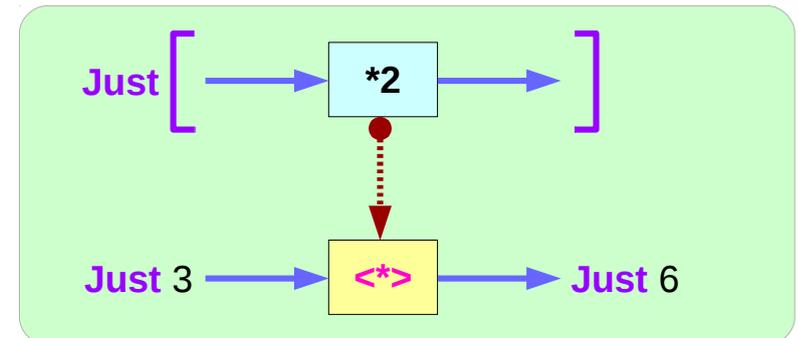
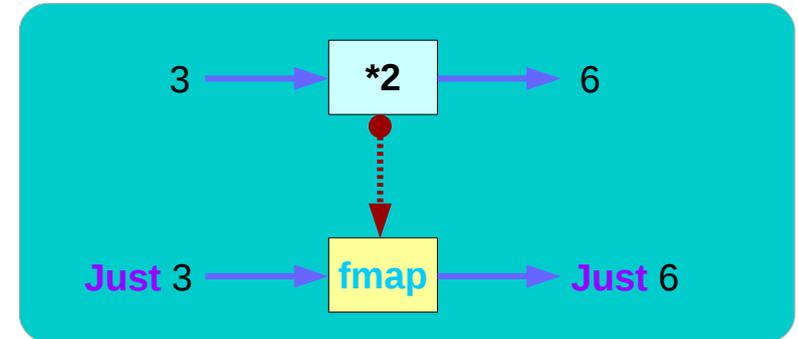
Sequencing of Effects

Functor, Applicative, Monad.

Three closely related functor type classes;

the characteristic methods of the three classes

```
fmap :: Functor    f => (a -> b) -> f a    -> f b
(<*>) :: Applicative f => f (a -> b) -> f a    -> f b
(>>=) :: Monad     m => m a      -> (a -> m b) -> m b
```



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

Sequencing of Effects

replace fmap by its infix synonym, (<\$>);

replace (>=>) by its flipped version, (= <<);

```
fmap :: Functor      f => (a -> b) -> f a      -> f b
(<*>) :: Applicative f => f (a -> b) -> f a      -> f b
(>=>) :: Monad       m => m a      -> (a -> m b) -> m b

(<$>) :: Functor t    => (a -> b)  -> (t a -> t b)
(<*>) :: Applicative t => t (a -> b) -> (t a -> t b)
(= <<) :: Monad t     => (a -> t b) -> (t a -> t b)
```

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

Sequencing of Effects

`fmap`, `(<*>)` and `(=<<)` are all mapping functions over **Functors**.

The differences between them are in what is being mapped over in each case:

`(<$>)` :: **Functor t** => **(a -> b)** -> **(t a -> t b)**

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

`(=<<)` :: **Monad t** => **(a -> t b)** -> **(t a -> t b)**

`fmap` maps arbitrary functions over functors.

`(<*>)` maps **t (a -> b)** morphisms over (applicative) functors.

`(=<<)` maps **a -> t b** functions over (monadic) functors.

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

Sequencing of Effects

The day-to-day differences in uses of Functor, Applicative and Monad follow from what the types of those three mapping functions allow you to do. As you move from `fmap` to `(<*>)` and then to `(>>=)`, you gain in power, versatility and control, at the cost of guarantees about the results. We will now slide along this scale. While doing so, we will use the contrasting terms `values` and `context` to refer to plain values within a functor and to whatever surrounds them, respectively.

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

Sequencing of Effects

The type of `fmap` ensures that it is impossible to use it to change the context, no matter which function it is given.

In $(a \rightarrow b) \rightarrow t \ a \rightarrow t \ b$, the $(a \rightarrow b)$ function has nothing to do with the t context of the $t \ a$ functorial value, and so applying it cannot affect the context.

For that reason, if you do `fmap f xs` on some list `xs` the number of elements of the list will never change.

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

References

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