Applicative (2A)

Young Won Lim
3/2/18

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

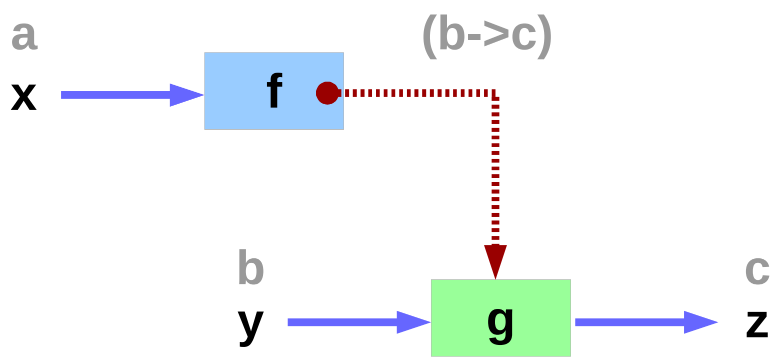
Currying

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 -> b -> c

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

f :: a -> b -> c



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

Curry & Uncurry

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

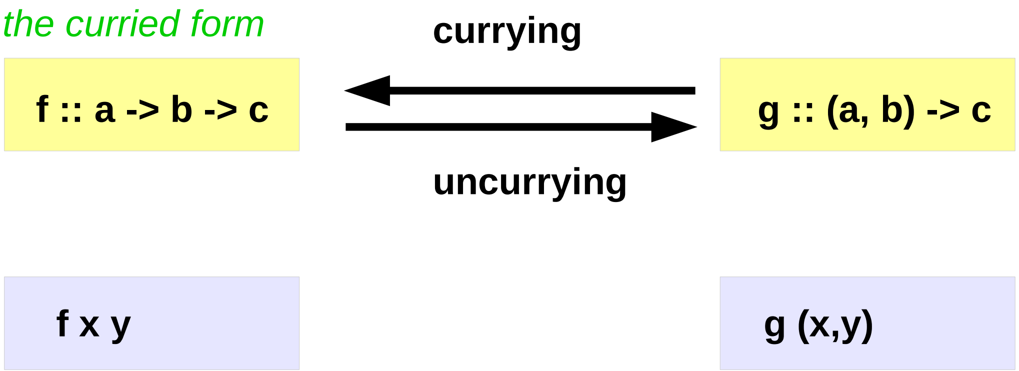
f = curry g
g = uncurry f

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



the curried form

f :: a -> b -> c

currying

uncurrying

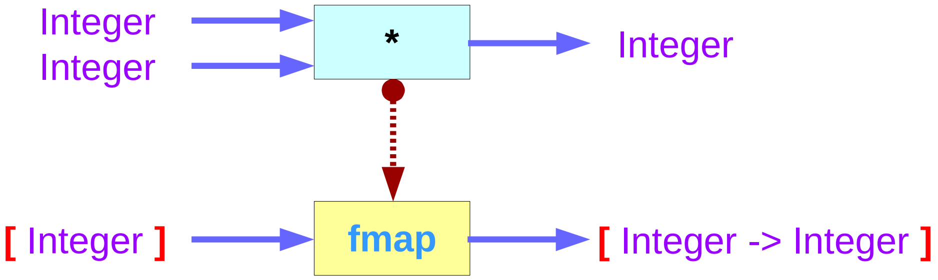
g :: (a, b) -> c

f x y

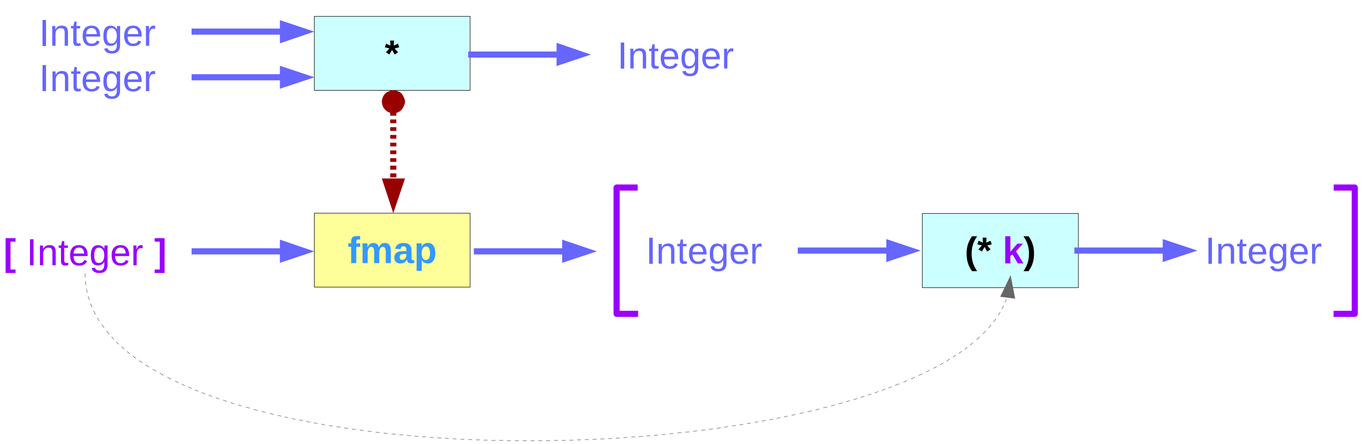
g (x,y)

https://wiki.haskell.org/Currying

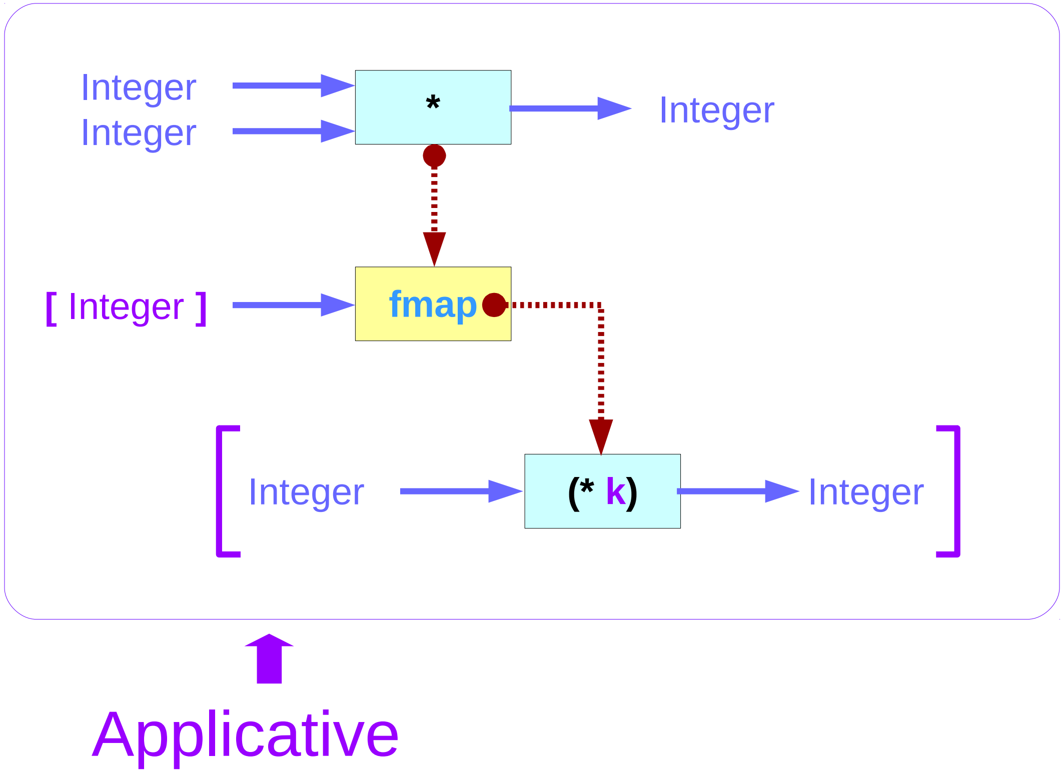
Mapping functions over the Functor [ ] (1)

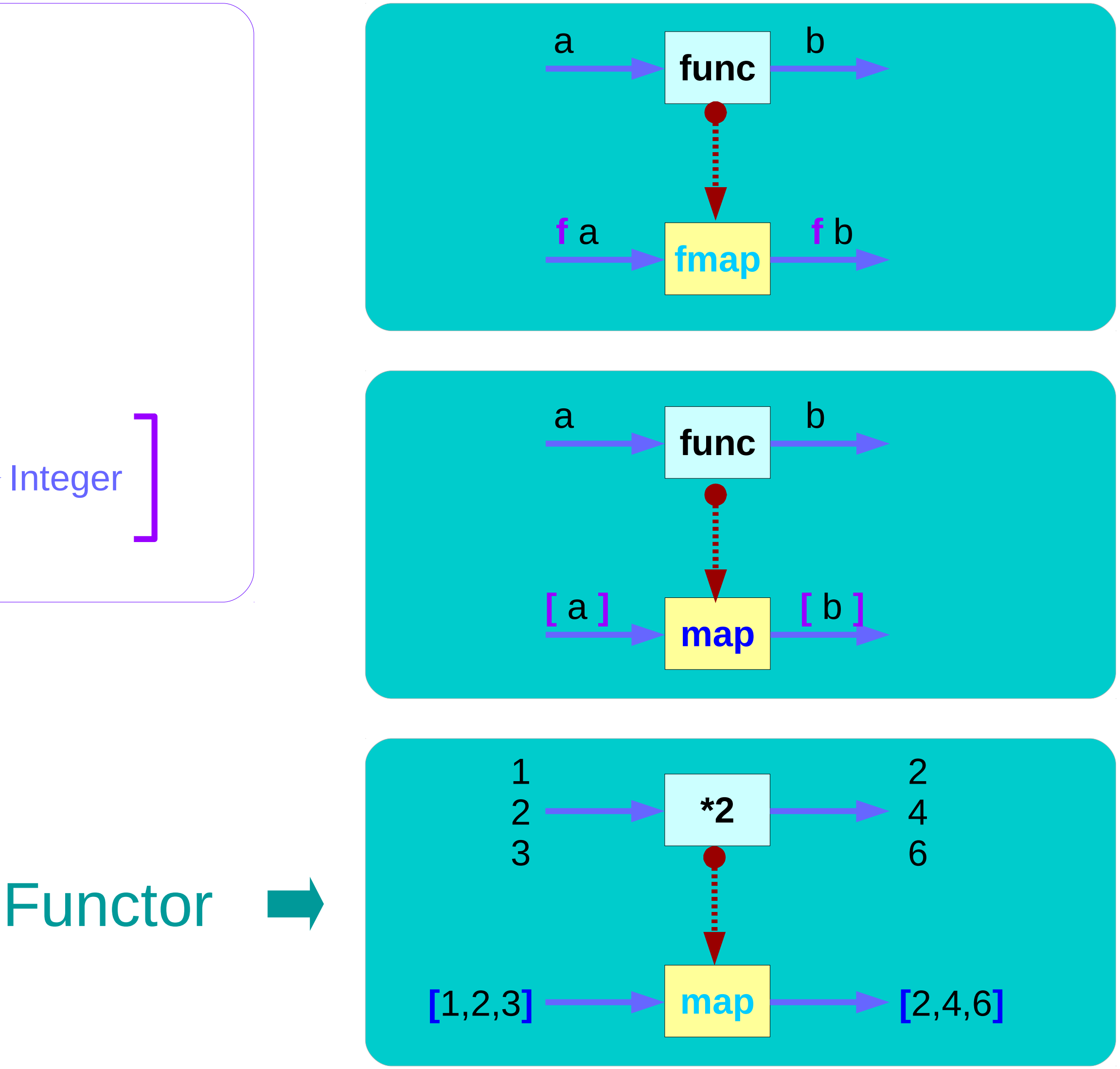


Mapping functions over the Functor [ ] (2)



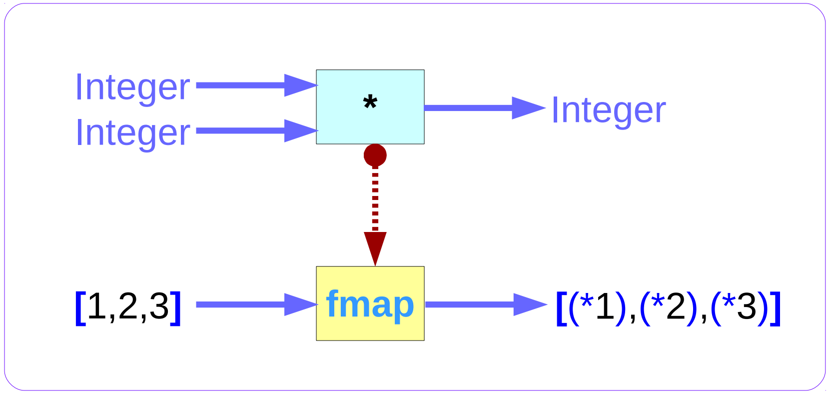
Mapping functions over the Functor [ ] (3)





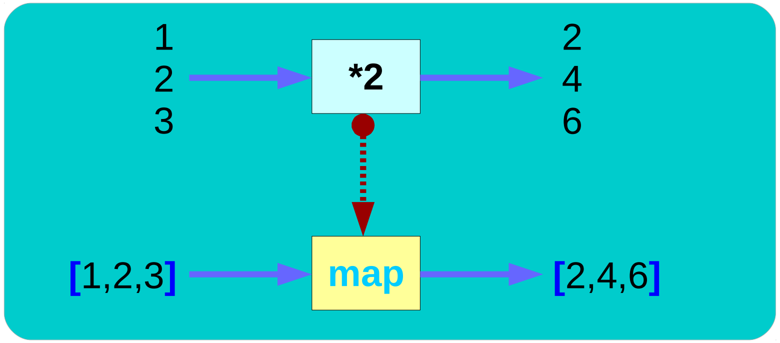
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Mapping functions over the Functor [ ] (4)



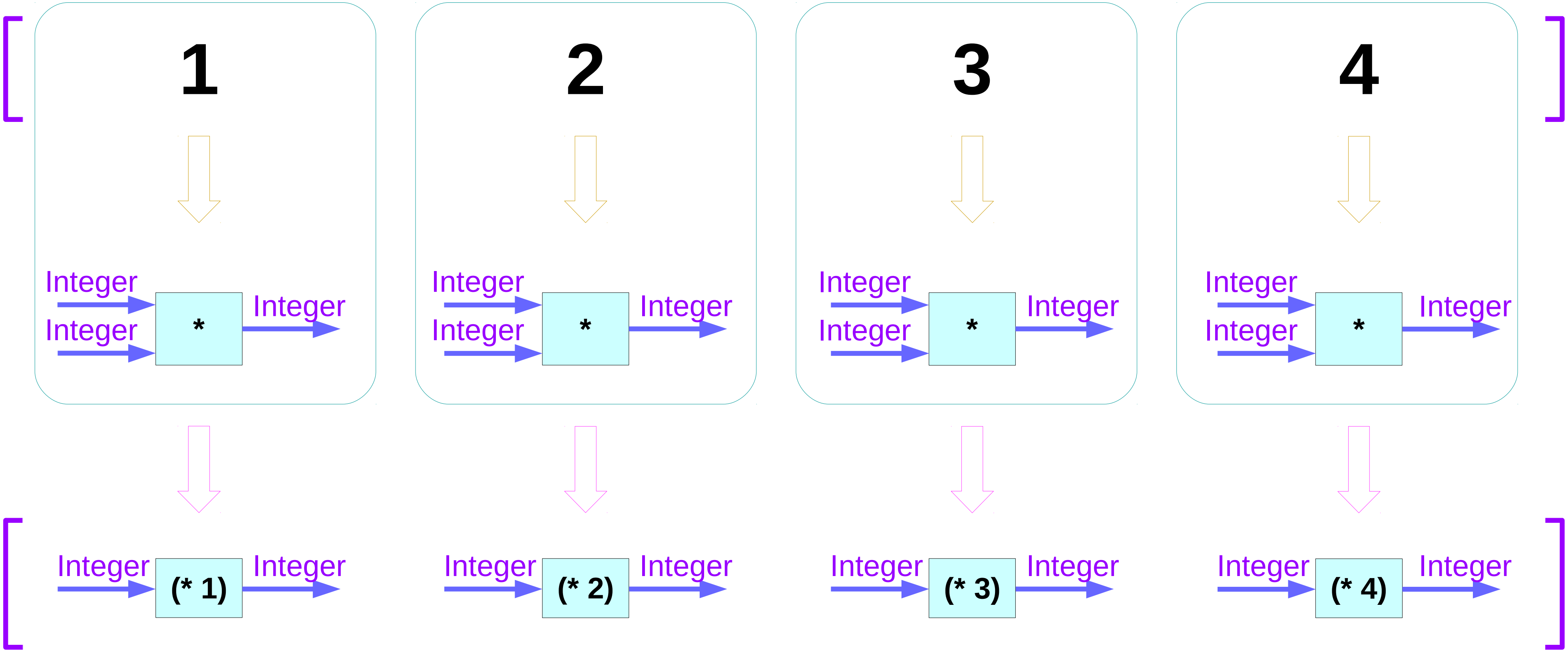
Applicative

Functor



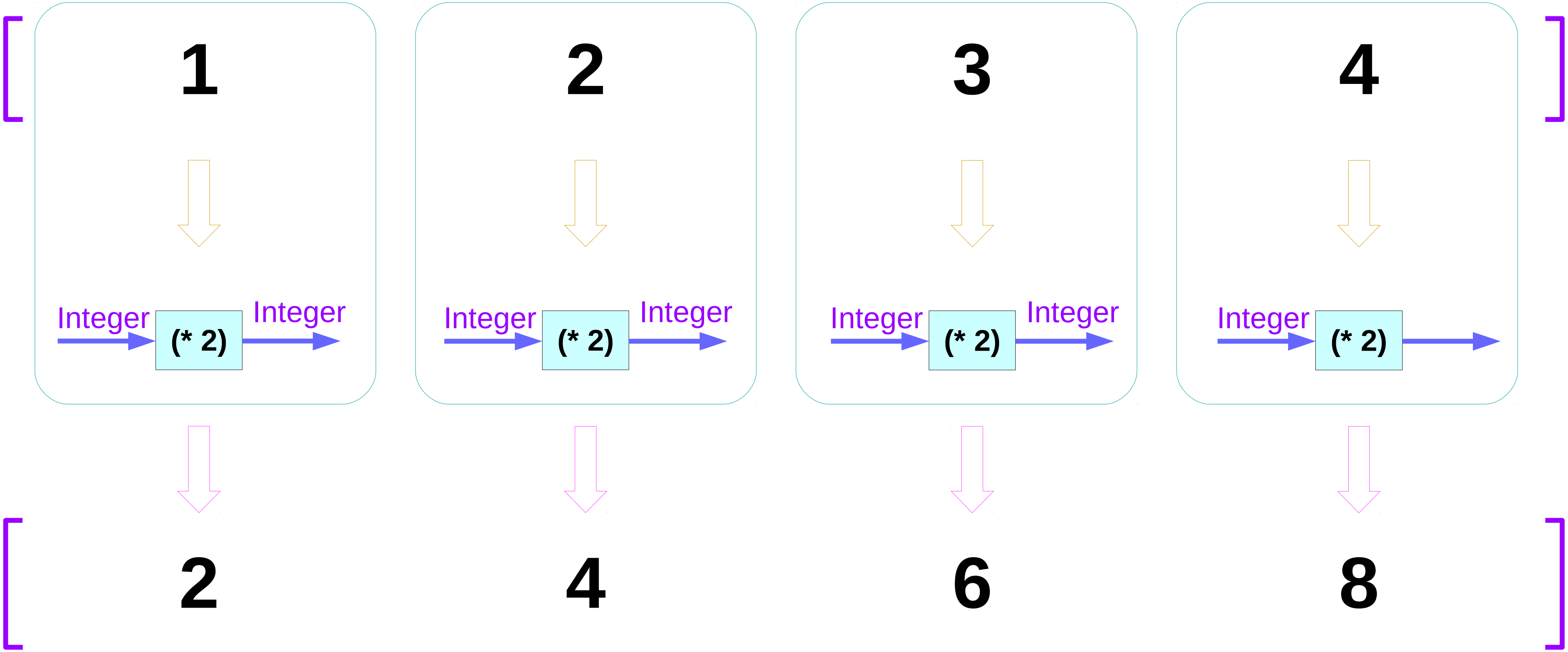
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Applicative : Mapping functions



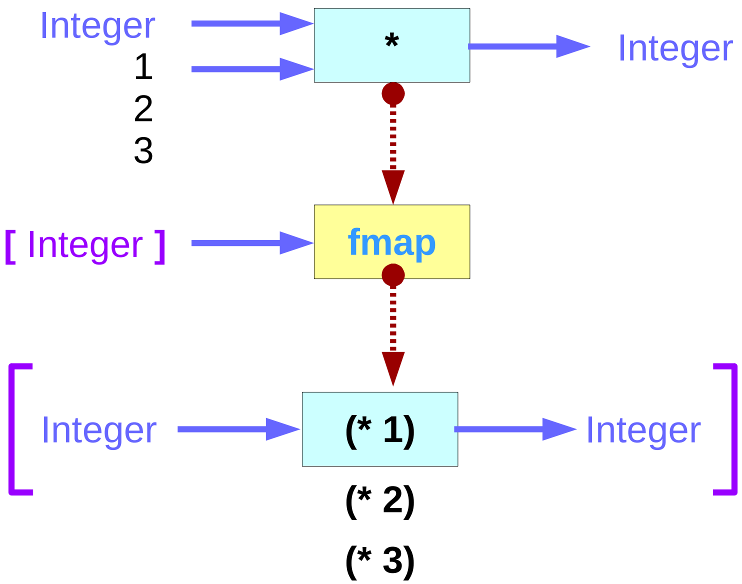
A list of functions

Functor : Mapping values



A list of integers

Applicatives vs. Functors



Integer

1

2

3

*

Integer

[Integer]

fmap

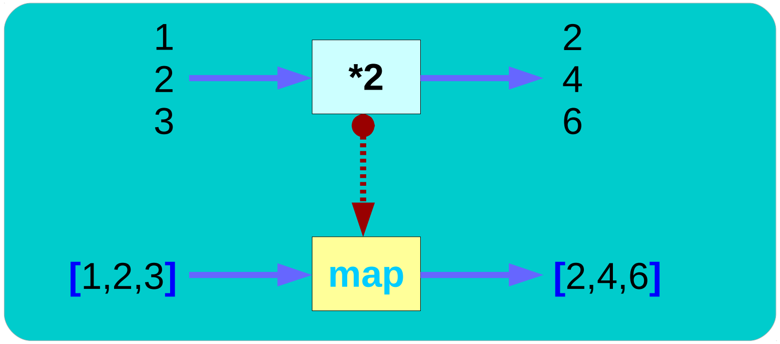
[Integer

(* 1)

(* 2)

(* 3)

Integer]



1

2

3

*2

2

4

6

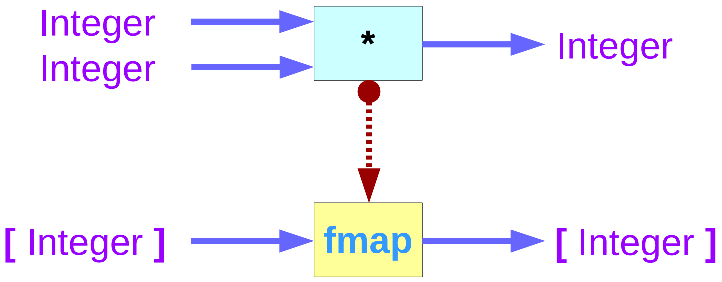
[1,2,3]

map

[2,4,6]

Double applications of **fmap** (1)

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



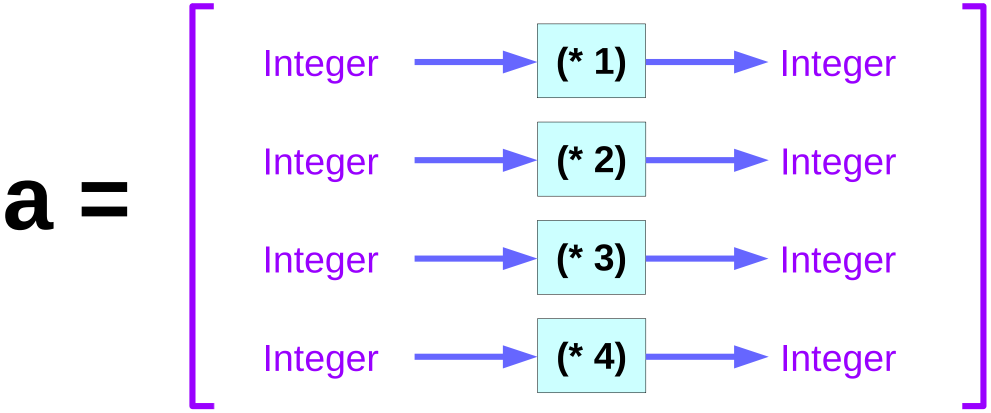
[1,2,3,4]

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

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

[Integer -> Integer]

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

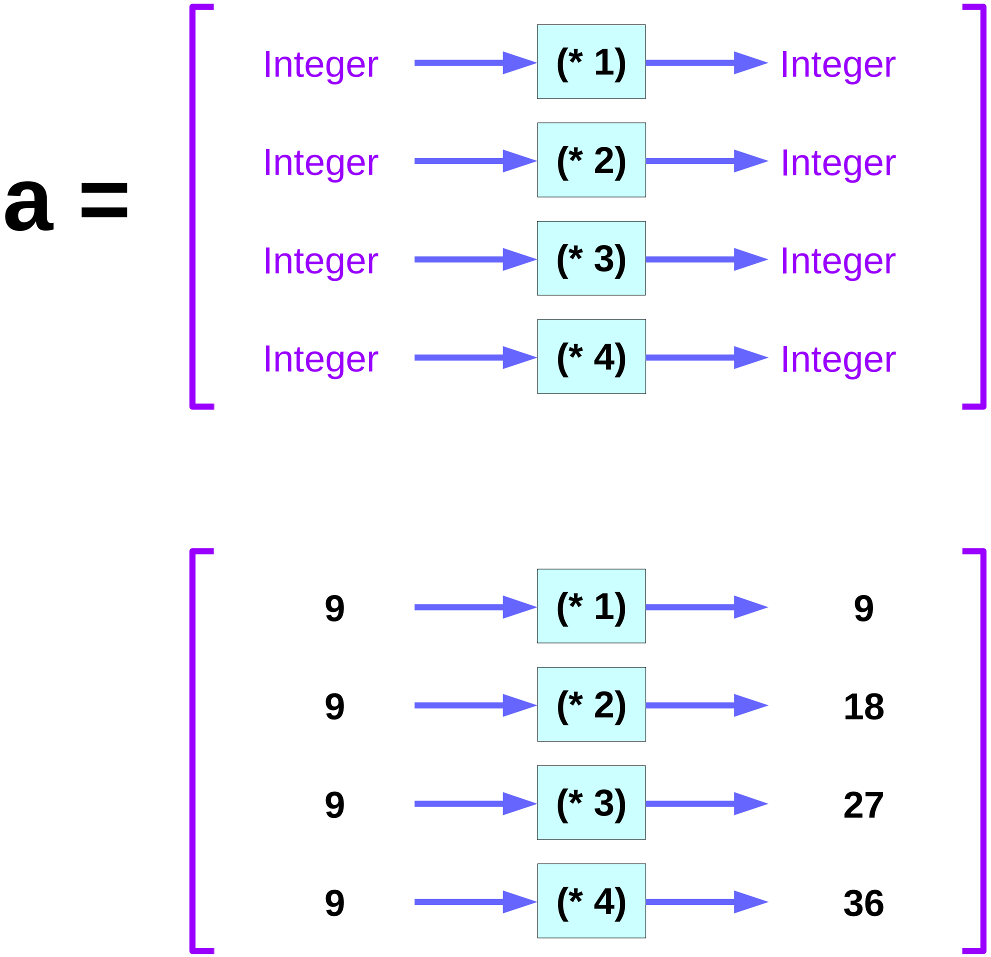


```
ghci> fmap (\f -> f 9) a
[9,18,27,36]
```

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

Double applications of fmap (2)

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



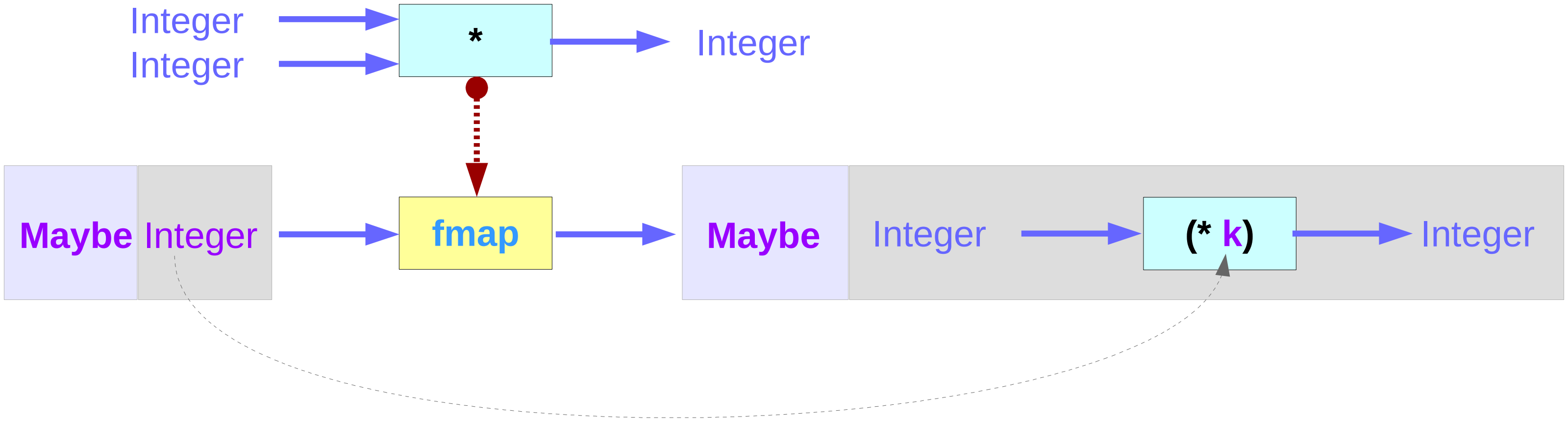
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Applications of fmap

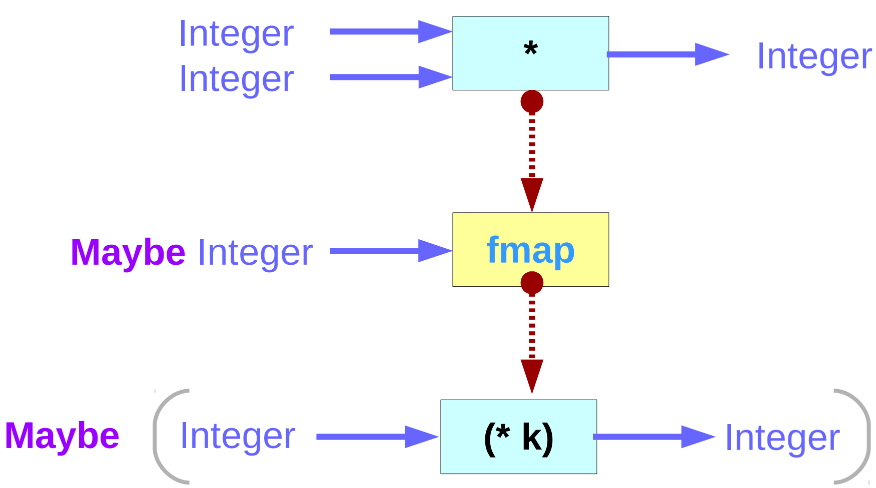
```
fmap (*) [1, 2, 3, 4]

[(*) 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)



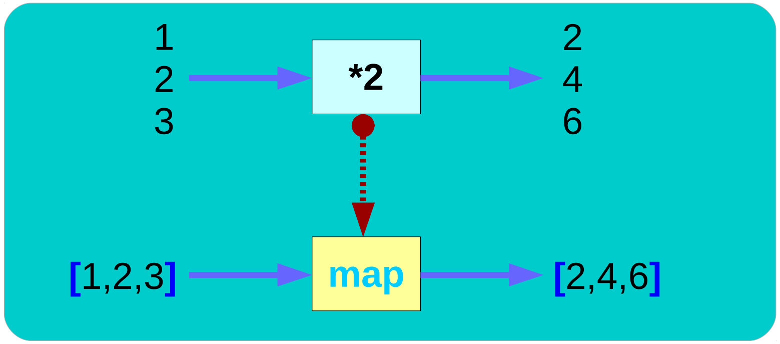
Mapping functions over the Functor Maybe (2)



Integer
Integer
*
Integer

Maybe Integer
fmap

Maybe
Integer
(* k)
Integer



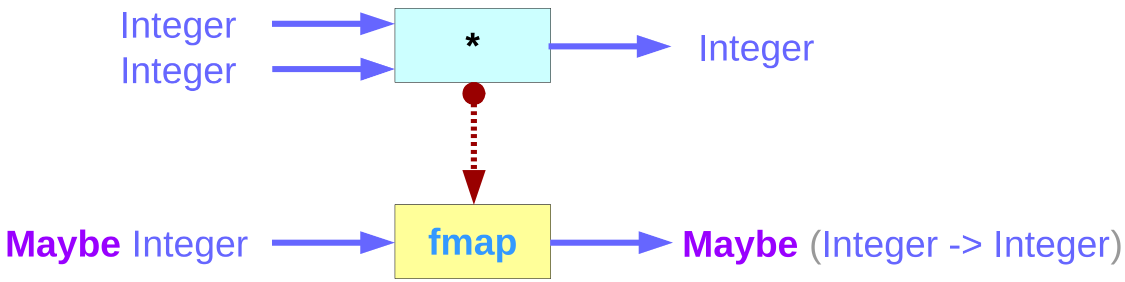
1
2
3
*2
2
4
6

[1,2,3]
map
[2,4,6]

Maybe Integer

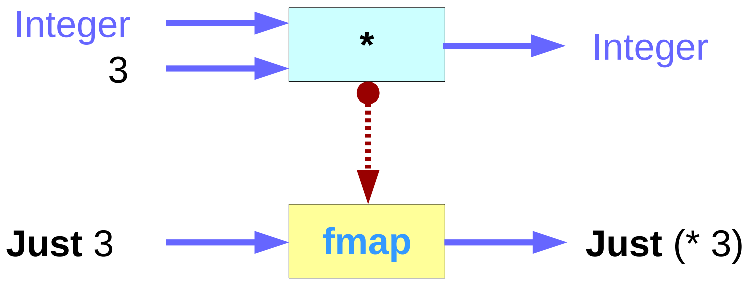
Maybe Integer
(* k)
Integer

Mapping functions over the Functor Maybe (3)



Integer
Integer
*
Integer

Maybe Integer
fmap
Maybe (Integer -> Integer)

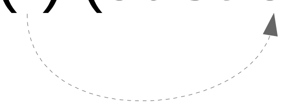


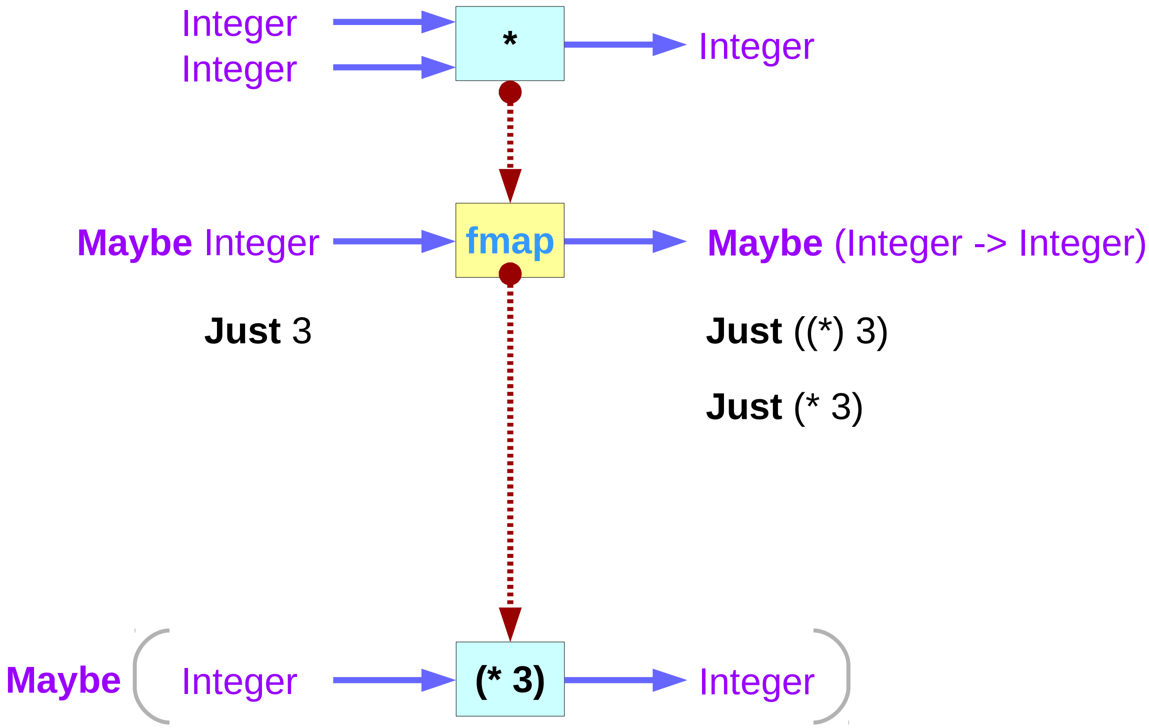
Integer
3
*
Integer

Just 3
fmap
Just (* 3)

Function wrapped in Just

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





function wrapped in a **Just**

Just (* 3)

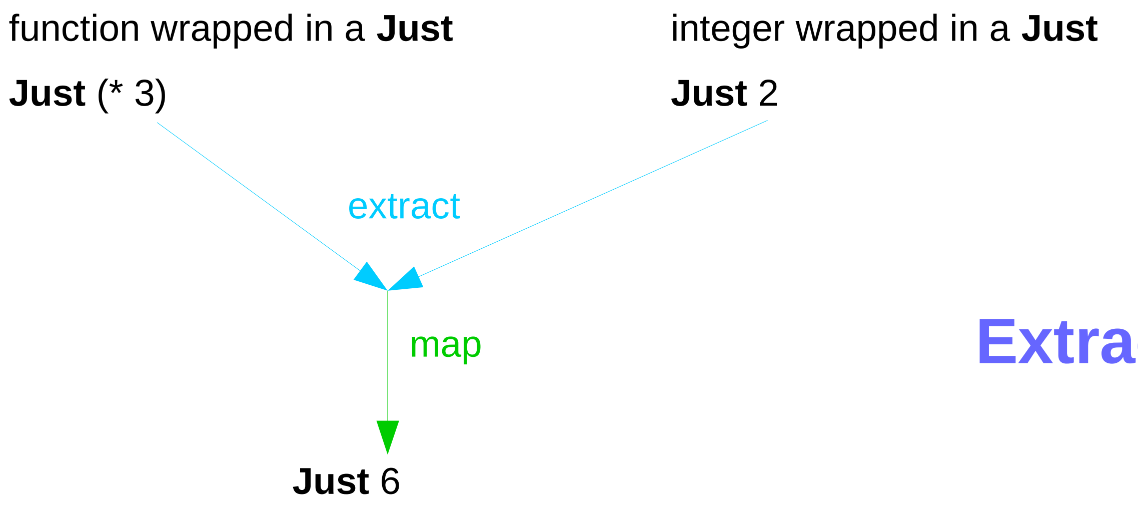
integer wrapped in a **Just**

Just 2

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

<*> Application of a function

Just (* 3) <*> Just 2



function wrapped in a **Just**

Just (* 3)

integer wrapped in a **Just**

Just 2

Just 6

Extracting and Mapping

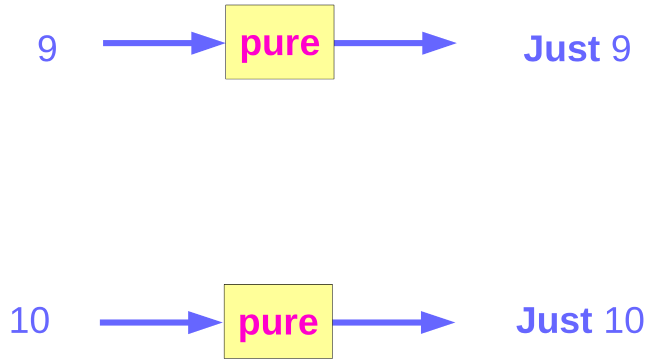
Just 6

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

Default Container Function **Pure**

pure 9 = **Just** 9

pure 10 = **Just** 10



to wrap an **integer**

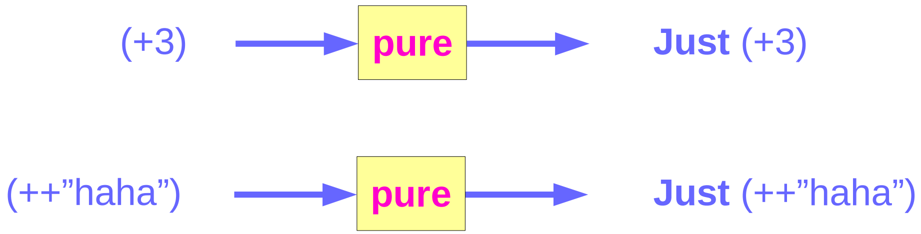
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

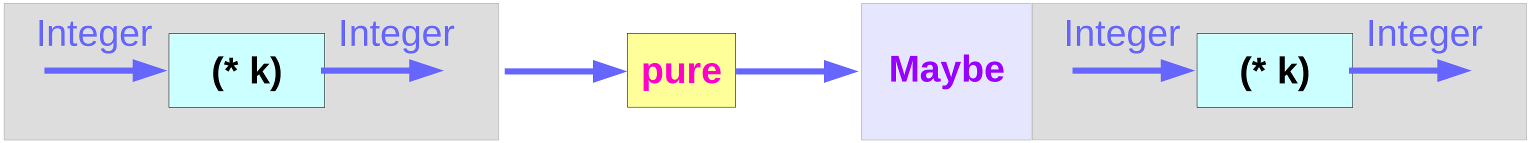
Default Container Function **Pure**

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

to wrap a **function**

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





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

Applicative Functor Apply <*> Examples

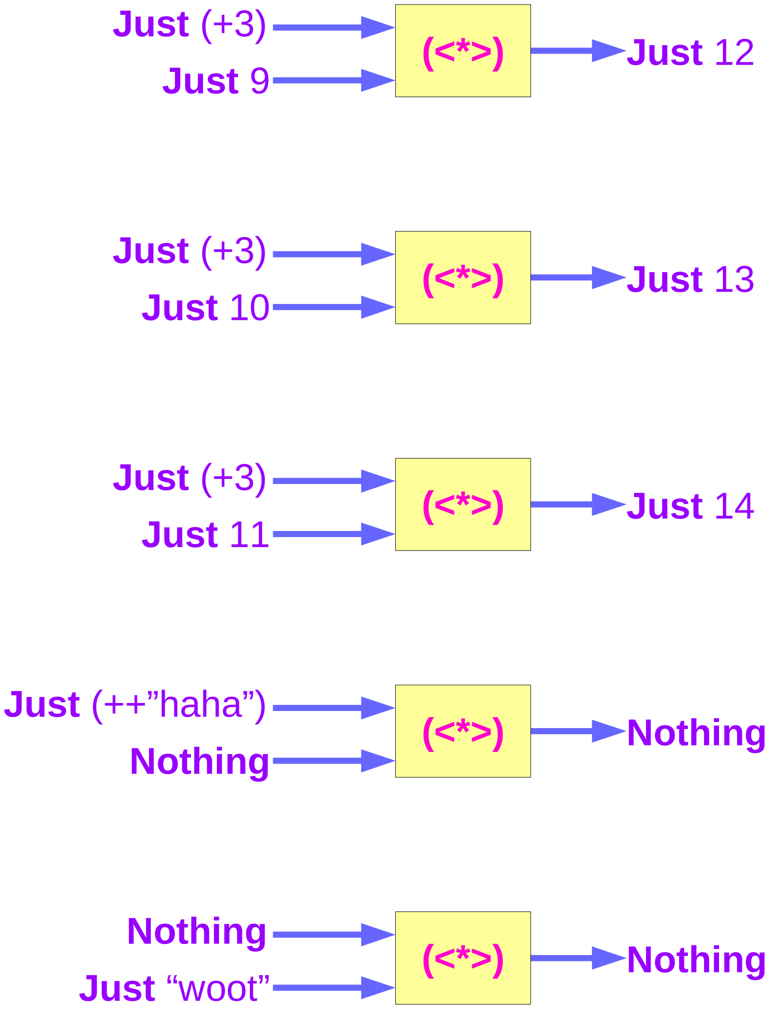
```
ghci> Just (+3) <*> Just 9
Just 12
```

```
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 method

maps functions of the base type (such as Integer)

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

fmap of **Functors** cannot

apply a function which is itself in a context

f (a -> b)

to a value in a context.

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**

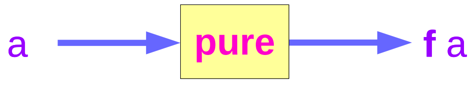
to the value wrapped in **Just** if both exist,

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
```

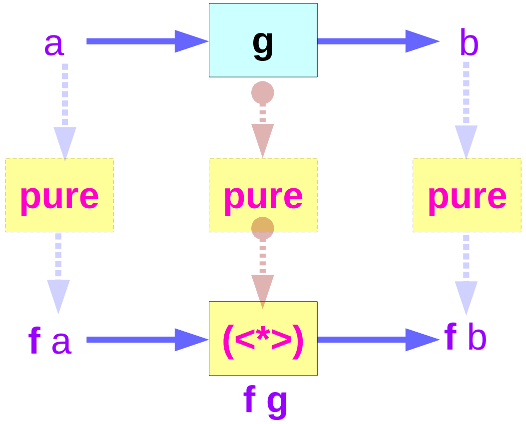


(**Functor f**) => **Applicative f**

f : Functor, Applicative

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

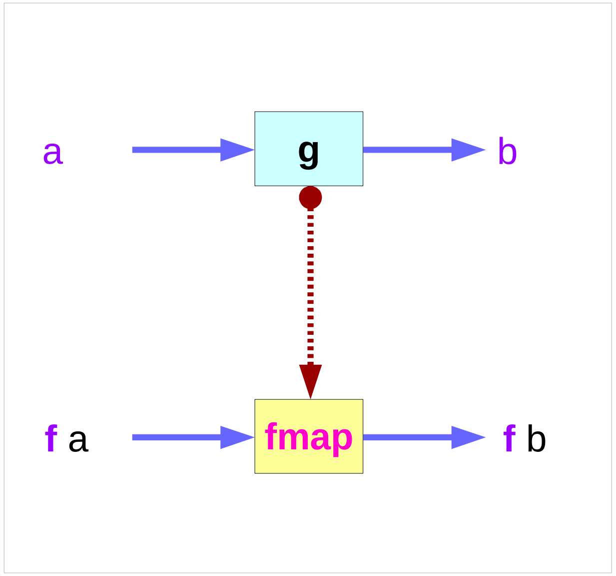
f : function in a context

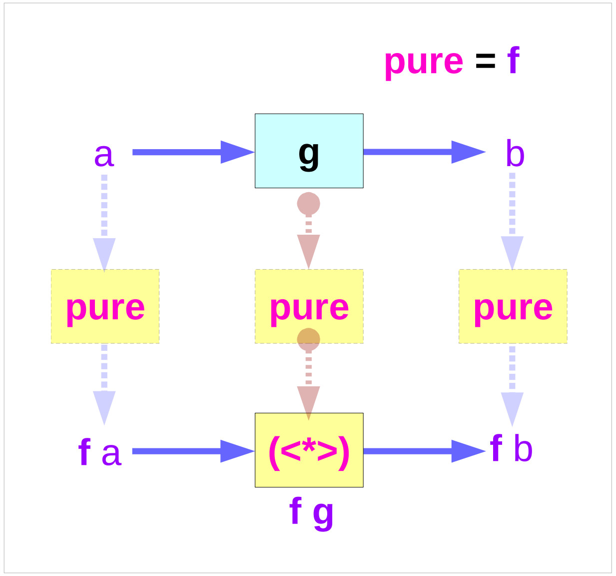


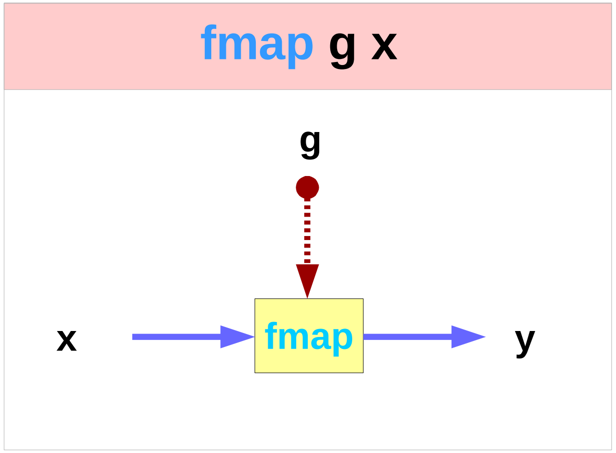
(**Functor f**) => **Applicative f**

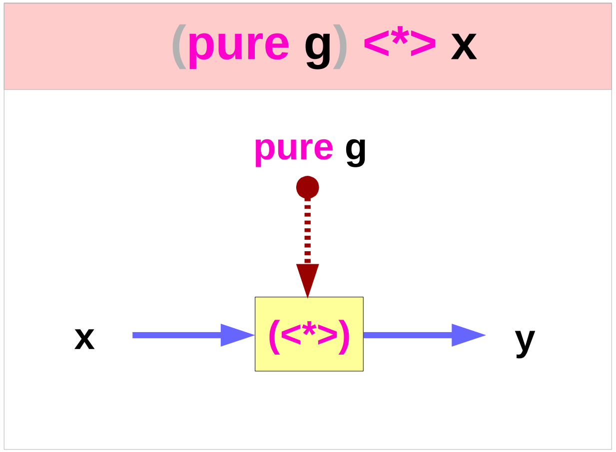
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

fmap g x = (pure g) <*> x





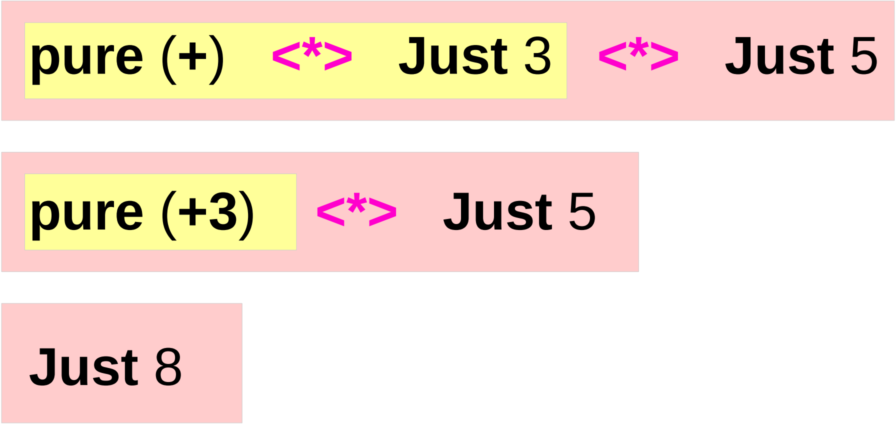




http://learnyouahaskell.com/functors-applicative-functors-and-monoids

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

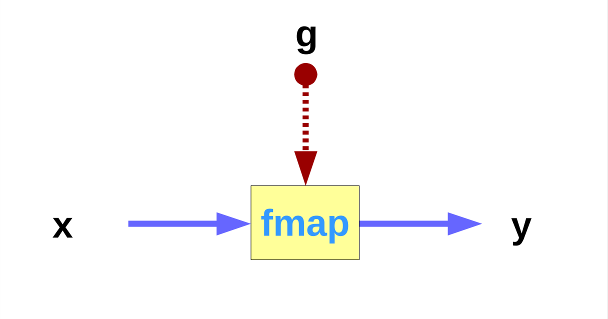
Infix Operator <$>

pure f <*> x <*> y <*> z

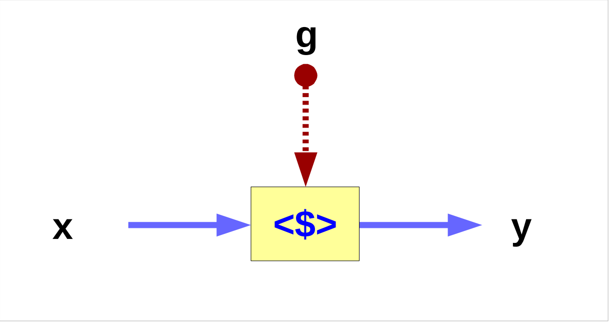
fmap f x <*> y <*> z

f <$> x <*> y <*> z

fmap g x



g <$> x



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

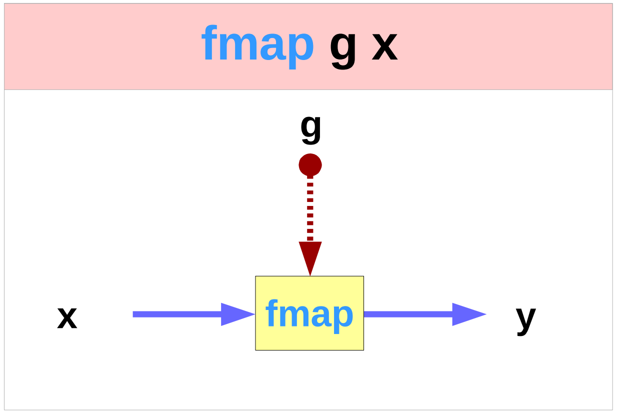
Infix Operator <$> : not a class method

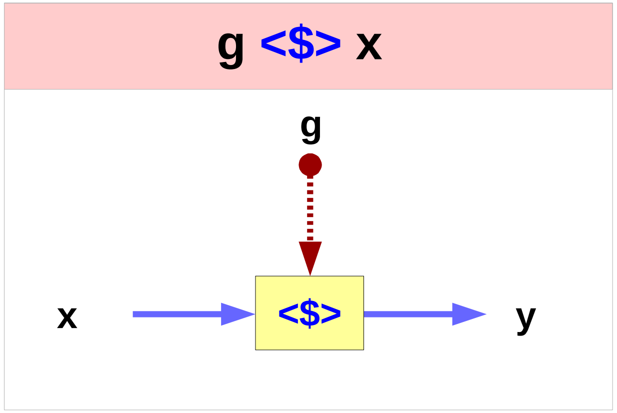
```
class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

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 **monad**s.

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**	**(>>)** in **Monad**
pure in **Applicative**	**return** in **Monad**

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

The Applicative Laws

The identity law: `pure id <*> v = v`

Homomorphism: `pure f <*> pure x = pure (f x)`

Interchange: `u <*> pure y = pure ($ y) <*> u`

Composition: `u <*> (v <*> w) = pure (.) <*> u <*> v <*> 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 **pure f <*> pure x = 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 <*> pure y = pure ($ y) <*> u**

applying a morphism **u** to a "**pure**" value **pure y**

is the same as applying **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 **pure y**,

the order in which we evaluate

the function **u** and its argument **pure y** doesn't matter.

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

The Composition Law

The composition law pure (.) <*> u <*> v <*> w = u <*> (v <*> w)

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

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

pure (.) <*> pure f <*> pure g <*> pure x
= pure f <*> (pure g <*> pure x)

u = pure f
v = pure g
w = pure x

applying the composed morphism pure (.) <*> u <*> v to **w**
gives the same result as applying **u** **u**
to the result of applying **v** to **w** (v <*> w)

it is expressing a sort of associativity property of (<*>).

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**.

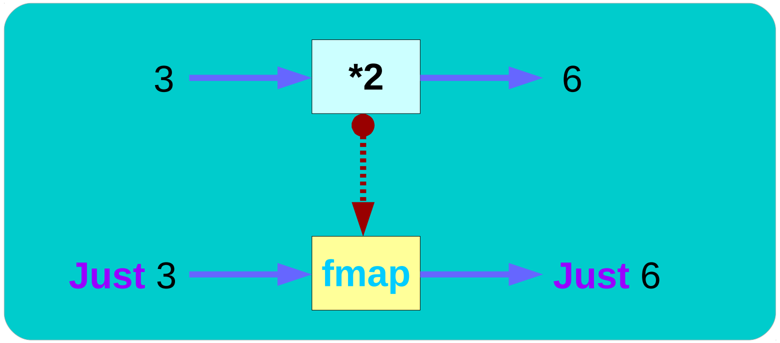
Three closely related functor type classes;

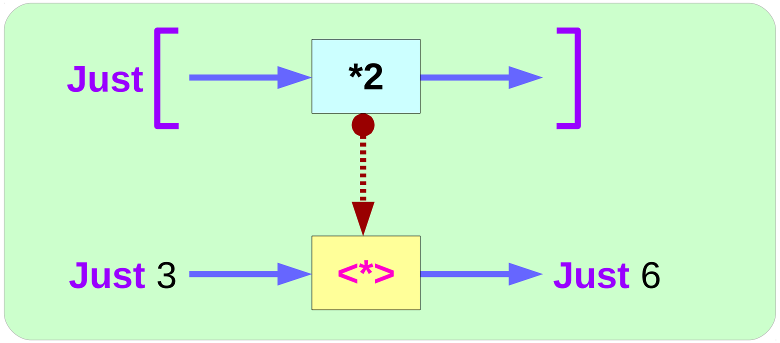
the characteristic methods of the three classes

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 -> b) -> t a -> t b, the (a -> 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

Sequencing of Effects

Functor map <$>

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

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

($>) :: Functor f => f a -> b -> f b

The <$> operator is just a synonym for the fmap function from the Functor typeclass. This function generalizes the map function for lists to many other data types, such as Maybe, IO, and Map.

https://haskell-lang.org/tutorial/operators

Sequencing of Effects

Functor map <$>

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

The <$> operator is just a synonym for the fmap function from the Functor typeclass. This function generalizes the map function for lists to many other data types, such as Maybe, IO, and Map.

https://haskell-lang.org/tutorial/operators

Sequencing of Effects

```
#!/usr/bin/env stack
-- stack --resolver ghc-7.10.3 runghc
import Data.Monoid ((<>))

main :: IO ()
main = do
    putStrLn "Enter your year of birth"
    year <- read <$> getLine
    let age :: Int
        age = 2020 - year
    putStrLn $ "Age in 2020: " <> show age
```

https://haskell-lang.org/tutorial/operators

Sequencing of Effects

In addition, there are two additional operators provided which replace a value inside a Functor instead of applying a function. This can be both more convenient in some cases, as well as for some Functors be more efficient. In terms of definition:

value <$ functor = const value <$> functor

functor $> value = const value <$> functor

x <$ y = y $> x

x $> y = y <$ x

https://haskell-lang.org/tutorial/operators

Sequencing of Effects

Applicative function application <*>

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

Commonly seen with <$>, <*> is an operator that applies a wrapped function to a wrapped value.
It is part of the Applicative typeclass, and is very often seen in code like the following:

```
foo <$> bar <*> baz
```

https://haskell-lang.org/tutorial/operators

Sequencing of Effects

For cases when you're dealing with a Monad, this is equivalent to:

```
do x <- bar
   y <- baz
   return (foo x y)
```

Other common examples including parsers and serialization libraries. Here's an example you might see using the aeson package:

```
data Person = Person { name :: Text, age :: Int } deriving Show
```

```
-- We expect a JSON object, so we fail at any non-Object value.
instance FromJSON Person where
    parseJSON (Object v) = Person <$> v .: "name" <*> v .: "age"
    parseJSON _ = empty
```

https://haskell-lang.org/tutorial/operators

Sequencing of Effects

To go along with this, we have two helper operators that are less frequently used:

*> ignores the value from the first argument. It can be defined as:

```
a1 *> a2 = (id <$ a1) <*> a2
```

Or in do-notation:

```
a1 *> a2 = do
    _ <- a1
    a2
```

For Monads, this is completely equivalent to >>.

https://haskell-lang.org/tutorial/operators

Sequencing of Effects

<* is the same thing in reverse: perform the first action then the second, but only take the value from the first action. Again, definitions in terms of <*> and do-notation:

```
(<*) = liftA2 const
```

```
a1 <* a2 = do
    res <- a1
    _ <- a2
    return res
```

https://haskell-lang.org/tutorial/operators

References

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