Applicative Methods (3B)

Young Won Lim
5/7/18

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

This document was produced by using LibreOffice.

The definition of Applicative

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

f (a -> b) :: a **function** wrapped in f

f a :: a **value** wrapped in f

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,

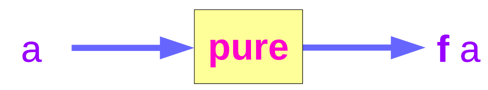
and results in **Nothing** otherwise.

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

An Instance of the Applicative Typeclass

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

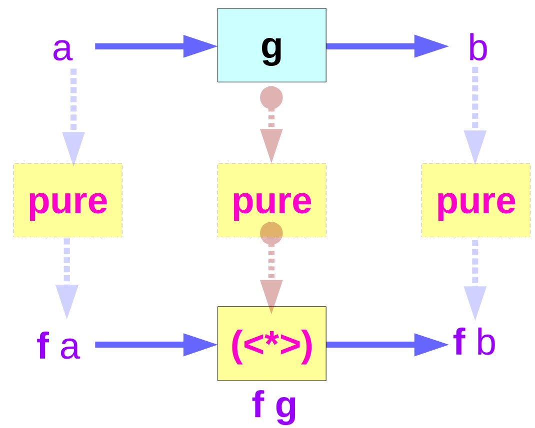
f : Functor, Applicative



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

```
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

Left associative <*>, fmap, and <$>

pure g <*> x <*> y <*> z

fmap g x <*> y <*> z

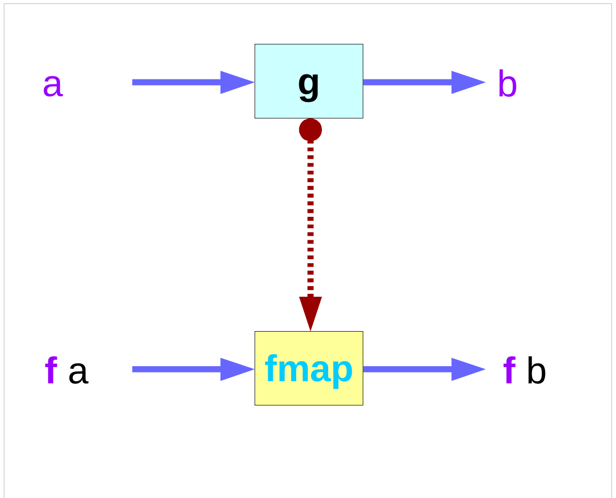
g <$> x <*> y <*> z

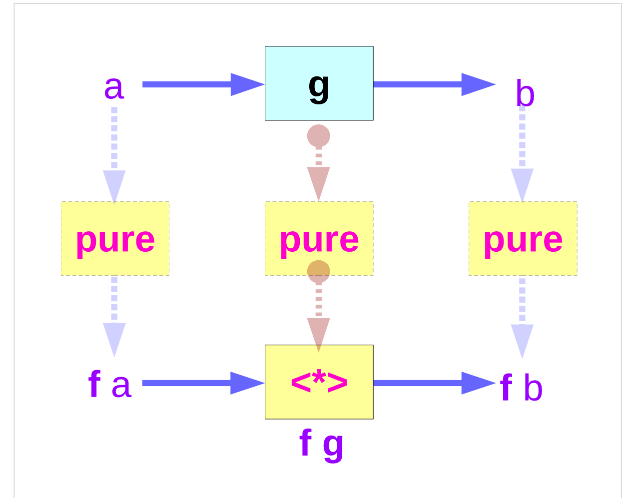
g :: f (a -> b -> c -> d)
x :: f a
y :: f b
z :: f c

infix operator <$>

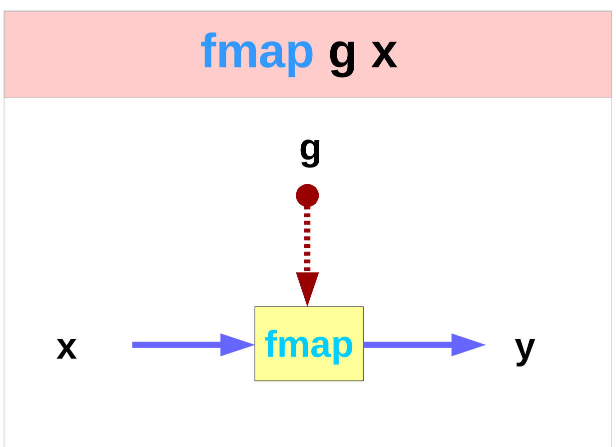
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

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

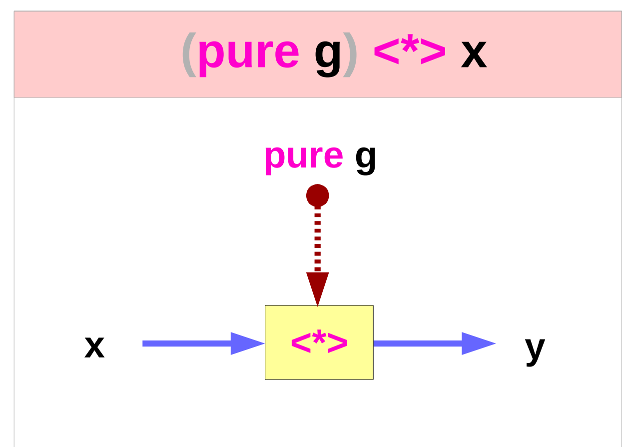




pure = f

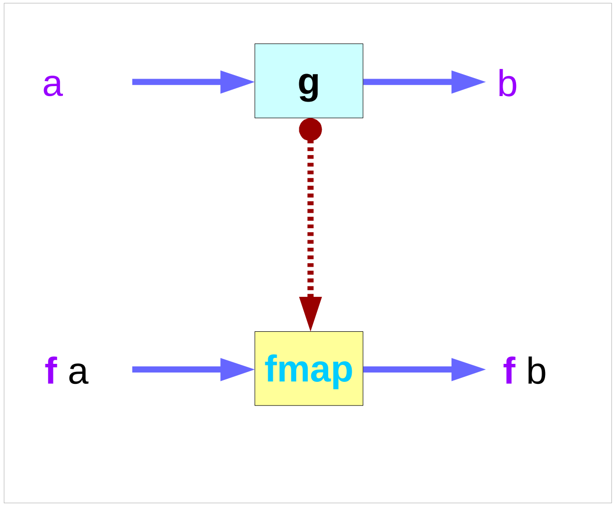


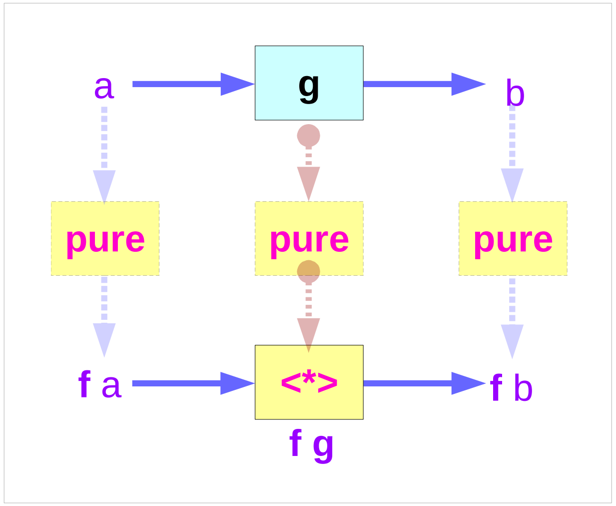
x :: f a
y :: f b



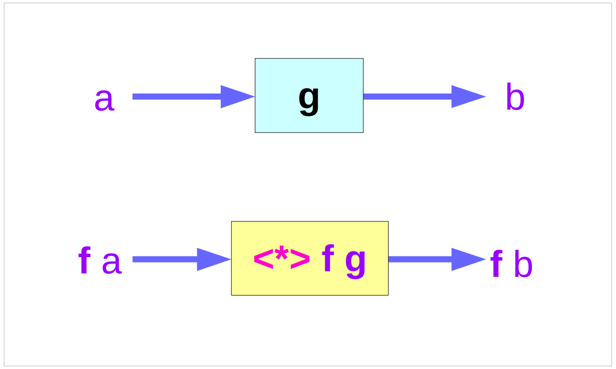
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

f a <*> f g





pure = f



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

Left associative <*> examples

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

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

Just (+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 Operators <*> vs <$> - a type view

h <*> x <*> y

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

h :: **f** (a -> b -> c)
x :: **f** a
h <*> x :: **f** (b -> c)

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

g <$> x <*> y

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

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

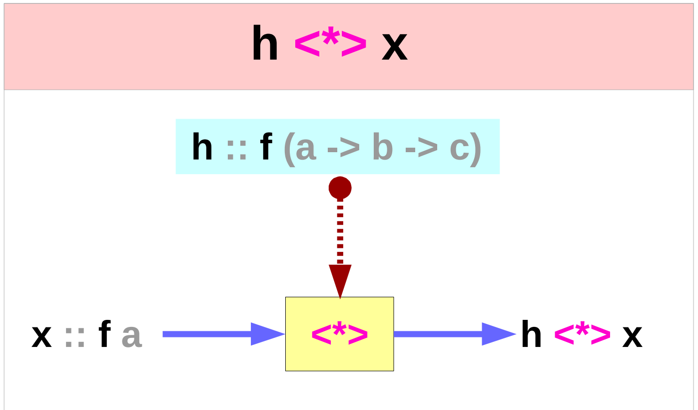
g :: (a -> b -> c)
x :: **f** a
g <$> x :: **f** (b -> c)
y :: **f** b
g <$> x <*> y :: **f** c

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

Infix Operators <*> vs <$> - a curried function view

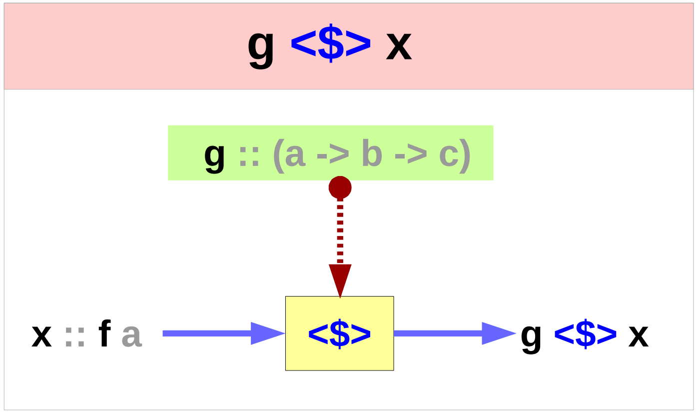
h <*> x <*> y

h :: f (a -> b -> c)
x :: f a
y :: f b



g <$> x <*> y

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

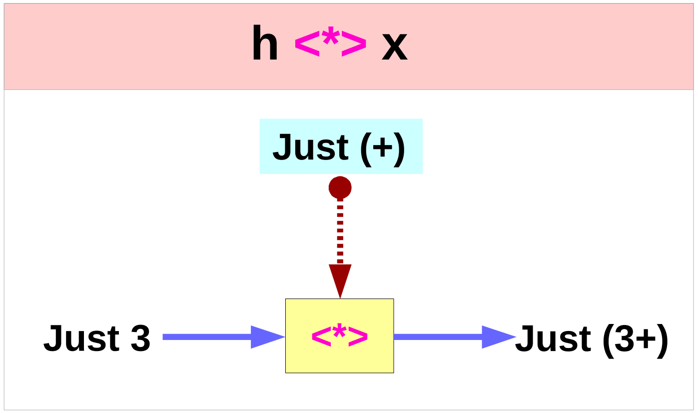


http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Infix Operators <*> vs <$> examples

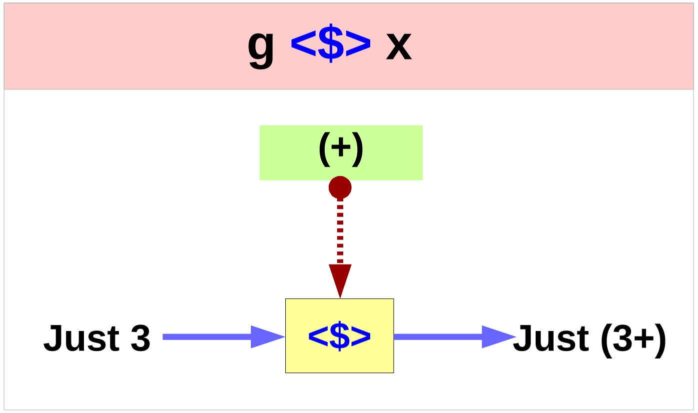
h <*> x <*> y

Just (+) <*> Just 3 <*> Just 2
Just (+3) <*> Just 2
Just 5



g <$> x <*> y

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



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

the minimal complete definition

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

the minimal complete definition

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

Not in the minimal complete definition

g :: a -> b,   x :: f a

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

requires **Functor** and **Applicative** instances.

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

fmap : defined in **Functors**

pure, (**<*>**) : defined in **Applicatives**

return, (**>>**) : defined in **Monads**

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

(<$>) vs ($)

(<$>) infix operator

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

g <$> x = **fmap g x**

The $ operator is for avoiding parentheses

```
putStrLn (show (1 + 1))
putStrLn $ show (1 + 1)
putStrLn $ show $ 1 + 1            – right associative
```

(**$**) calls the function which is its left-hand argument of **$**

on the value which is its right-hand argument of **$**

The Applicative Laws

The identity law:	pure id <*> v = v	id :: a -> a	v :: f a
Homomorphism:	pure g <*> pure x = pure (g x)	g :: a -> b	x :: a
Interchange:	u <*> pure y = pure ($ y) <*> u	u :: f (a -> b)	y :: a
Composition:	u <*> (v <*> w) = pure (.) <*> u <*> v <*> w	w :: f a	v :: f (a -> b) u :: f (b -> c)

Left associative u <*> v <*> w = (u <*> v) <*> w

u :: f (c -> b -> a)
v :: f c
u <*> v :: f (b -> a)
w :: f b
u <*> v <*> w = f a

The Identity Law

The identity law	pure id <*> v = v	id :: a -> a	v :: f a

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 g <*> pure x = pure (g x)**	**g :: a -> b**	**x :: a**

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

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

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

The Interchange Law

The interchange law **u <*> pure y = pure ($ y) <*> u** **u :: f (a -> b)** **y :: a**

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

Function $ Argument
$ y
(**y**) as a single argument

applying a morphism **u** to a "**pure**" value **pure y**
is the same as applying **pure** ($ **y**) to the morphism **u**

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

when evaluating the application of
an effectful function (**u)** to a pure argument (**pure y**),
the order doesn't matter – commutative.

u :: f (a -> b)	**y :: a**
u <*> pure y :: f b	**pure y :: f a**
pure ($ y) <*> u :: f b	**pure ($ y) :: f (a)**

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

The Composition Law

The composition law pure (.) <*> u <*> v <*> w = u <*> (v <*> w) w :: f a v :: f (a -> b) u :: f (b -> c)

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

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

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

w :: f a -- value
v :: f (a -> b) -- func1
u : f (b -> c) -- func2

v <*> w :: f b
u <*> (v <*> w) :: f c

pure (.) <*> u <*> v :: f (a -> c)
pure (.) <*> u <*> v <*> w :: f c

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

The Composition Law and Left Associativity

The composition law pure (.) <*> u <*> v <*> w = u <*> (v <*> w) w :: f a v :: f (a -> b) u :: f (b -> c)

pure (.) <*> pure g <*> pure h <*> pure x (g . h) x

((pure (.) <*> pure g) <*> pure h) <*> pure x

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

f (b -> c) f (a -> b)
pure g pure h
pure (.)
f (a -> c)

u = pure g :: f (b -> c) g :: (b -> c)
v = pure h :: f (a -> b) h :: (a -> b)
w = pure x :: f a x :: a

Left associative u <*> v <*> w = (u <*> v) <*> w u :: f (c -> b -> a) v :: f c w :: f b

u :: f (c -> b -> a)
v :: f c
u <*> v :: f (b -> a)
w :: f b
u <*> v <*> w = f a

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

liftA2

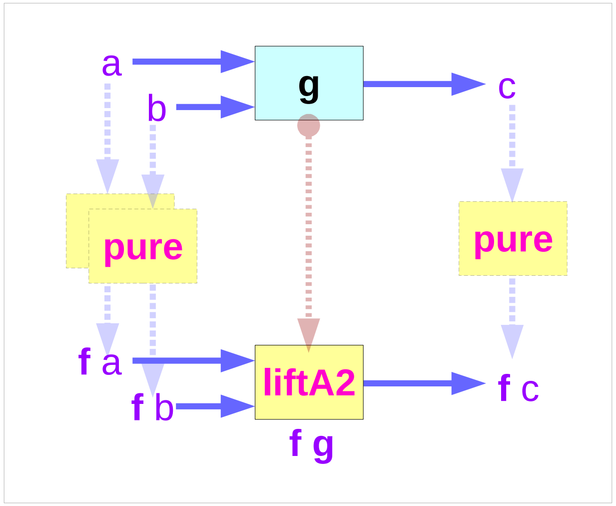
liftA2 :: (a -> b -> c) -> f a -> f b -> f c

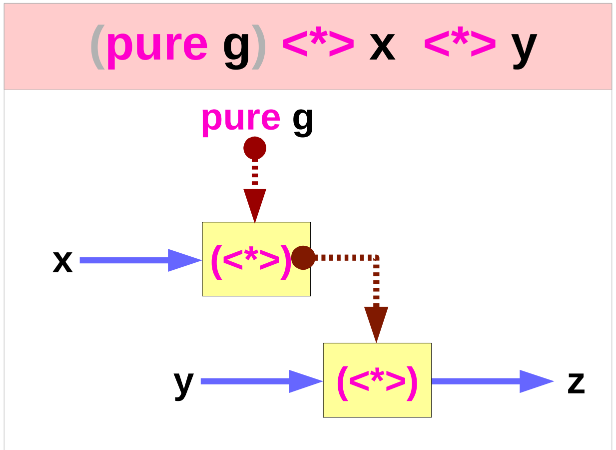
lift a binary function (**a->b->c**) to actions.

Some functors support an implementation of **liftA2** that is more efficient than the default one.

liftA2 may have an efficient implementation whereas **fmap** is an expensive operation,

sometimes better to use **liftA2** than to use **fmap** over the structure and then use **<*>**.

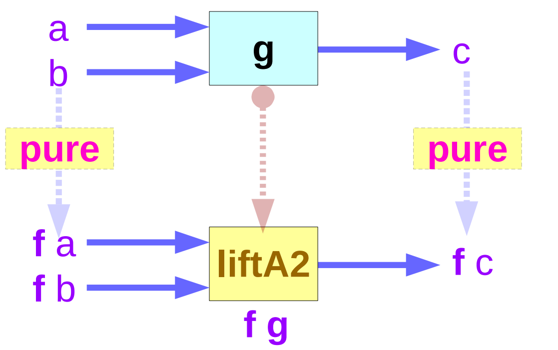




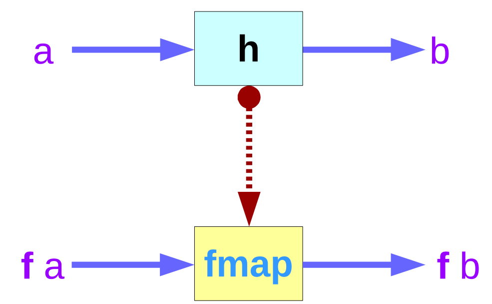
http://hackage.haskell.org/package/base-4.10.1.0/docs/Control-Applicative.html#v:liftA2

liftA2, <*>, fmap, <$>

liftA2 g x y



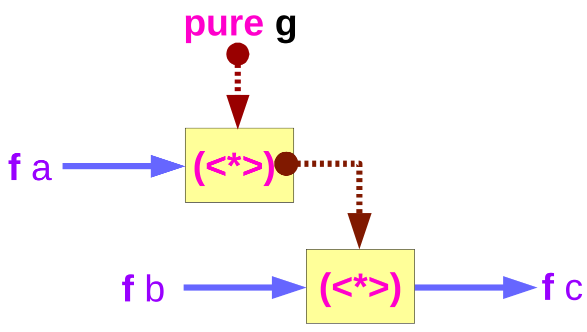
fmap h x



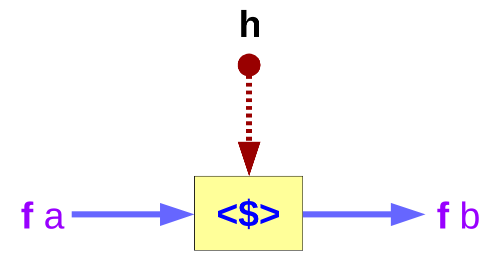
g :: a -> b -> c

h :: a -> b

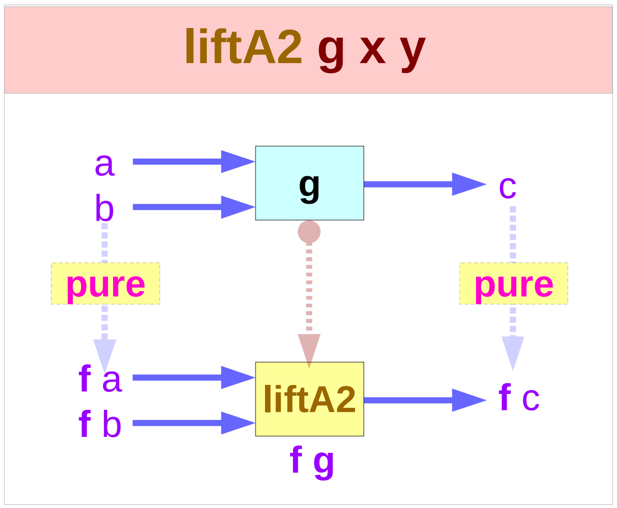
(pure g) <*> x <*> y

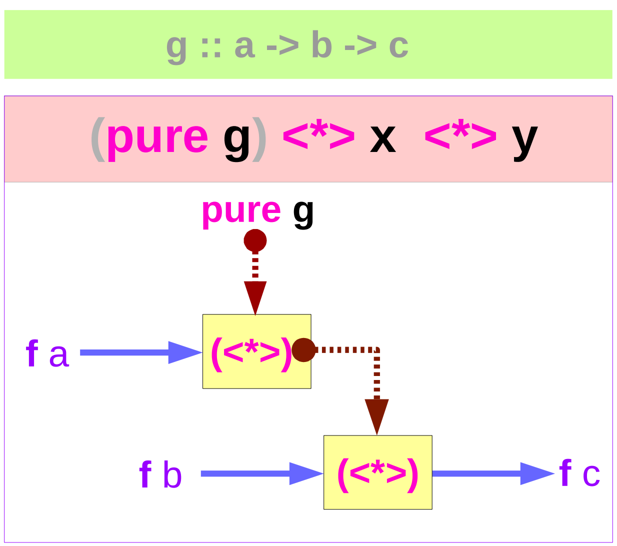


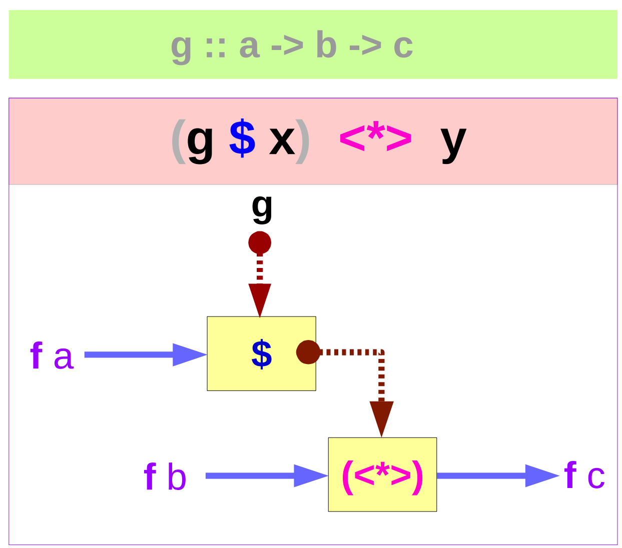
h <$> x

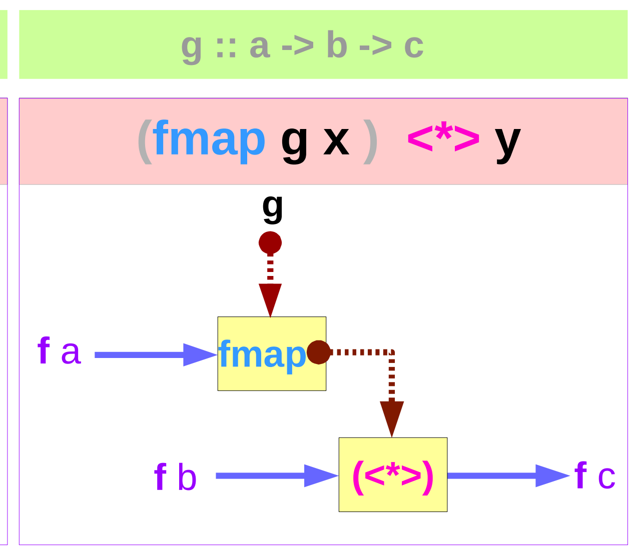


pure g <*> x <*> y equivalent









liftA2

liftA2 g x y

liftA2 :: (a -> b -> c) -> f a -> f b -> f c

g :: a -> b -> c

x :: f a

y :: f b

liftA2 g x y :: f c

pure g <*> x <*> y

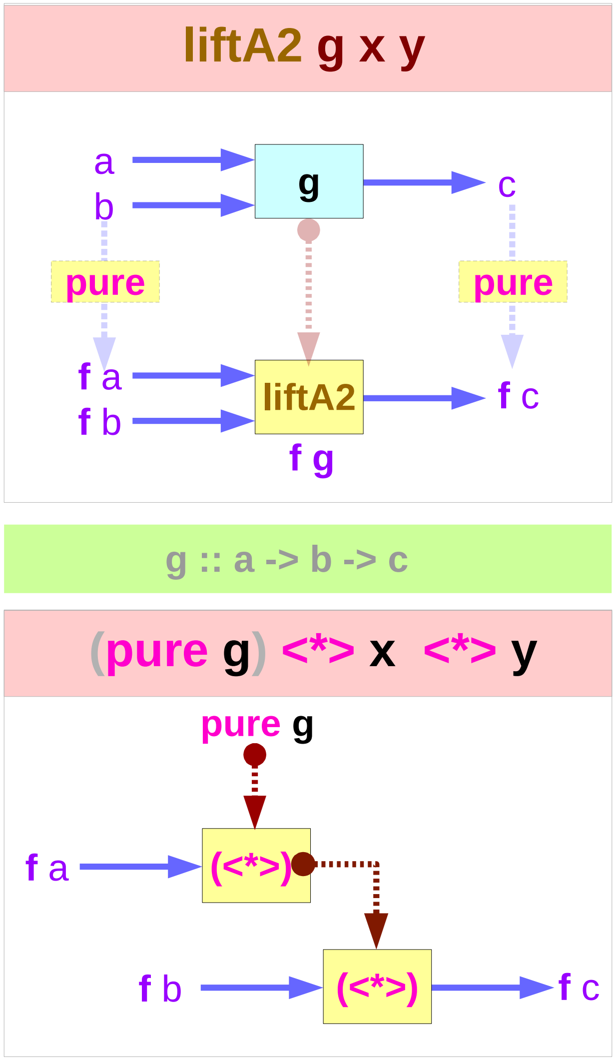
g :: a -> b -> c

x :: f a

y :: f b

z :: f c

pure g <*> x <*> y :: f c



https://wiki.haskell.org/Applicative_functor

Limitations of Functors

(a -> b -> c) -> (f a -> f b -> f c) – let fmap2 Functor as an extension of fmap

fmap :: (a -> b) -> (f a -> f b)
fmap2 :: Functor f => (a -> b -> c) -> (f a -> f b -> f c)

```
fmap2 h fa fb = undefined
h  :: a -> b -> c
fa :: f a
fb :: f b
```

```
h         :: a -> (b -> c)
fmap h    :: f a -> f (b -> c)
fmap h fa :: f (b -> c)
```
– now **f (b -> c)** must be applied to **f b**

fmap gives us a way to apply functions (**a -> b**) to values (**f a**) inside a Functor context,
but **fmap** cannot be used to apply a functions **f (b -> c)**
which are themselves in a Functor context to values **f b** in a Functor context.

http://www.openhaskell.com/lectures/applicative.html

pure, fmap, and liftA2

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

```
pure    :: a -> f a                                 – fmap0 → pure
fmap    :: (a -> b) -> f a -> f b                   – fmap1 → fmap
fmap2   :: (a -> b -> c) -> f a -> f b -> f c       – fmap2 → liftA2
```

```
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
liftA2 h fa fb = (h `fmap` fa) <*> fb
liftA2 h fa fb = h <$> fa <*> fb
```

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

```
liftA2 :: Applicative f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d
liftA3 h fa fb fc = ((h <$> fa) <*> fb) <*> fc
```

http://www.openhaskell.com/lectures/applicative.html

liftA2 examples

liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c

liftA2 (+) (Just 5) (Just 6) = Just 11

liftA2 h fa fb = (h `fmap` fa) <*> fb

liftA2 h fa fb = h <$> fa <*> fb

fmap (+) (Just 5) = Just (+5)

(+) <$> (Just 5) = Just (+5)

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

(Just (+5)) <*> (Just 6) = Just 11

let v1 = IO (Just (+5))

let v2 = IO (Just 6)

liftA2 (<*>) v1 v2 = IO (Just 11)

https://blog.ssanj.net/posts/2014-08-10-boosting-liftA2.html

<*> or liftA2 implementations

liftA2 :: (a -> b -> c) -> f a -> f b -> f c

A minimal complete definition :

either one of the two

1) pure and **<*>**
2) pure and **liftA2**

1) pure g <*> x <*> y
2) liftA2 g x y

If it defines both, then they must behave the same

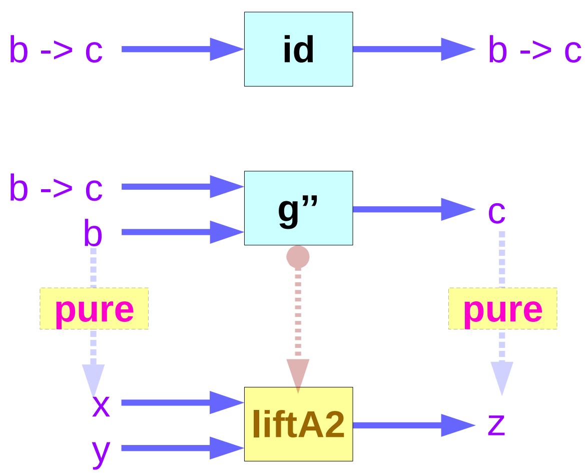
as their default definitions:

http://hackage.haskell.org/package/base-4.10.1.0/docs/Control-Applicative.html#v:liftA2

(<*>) = liftA2 id

liftA2 id x y = id <$> x <*> y = x <*> y

liftA2 id x y = x <*> y

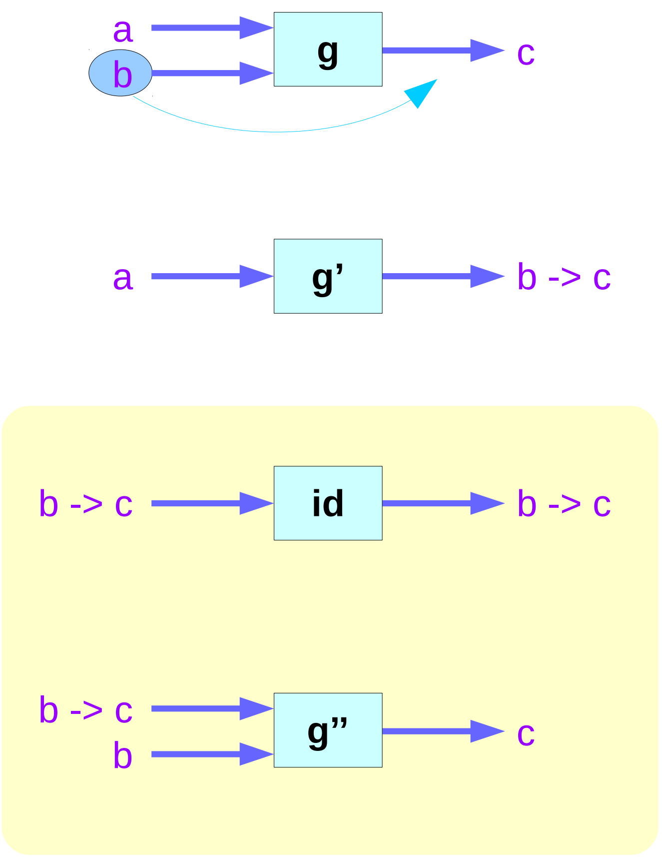


liftA2 g x y = g <$> x <*> y	g :: a -> b -> c	x :: f a	y :: f b
liftA2 g” x y = g” <$> x <*> y	g” :: (b -> c) -> b -> c	x :: f (b -> c)	y :: f b
liftA2 id x y = id <$> x <*> y = x <*> y	id :: (b -> c) -> (b -> c)	x :: f (b -> c)	y :: f b

(<*>) = liftA2 id

http://hackage.haskell.org/package/base-4.10.1.0/docs/Control-Applicative.html#v:liftA2

g” :: (b->c) -> b->c



g :: a -> b -> c

g’ :: a -> (b -> c)

id :: (b -> c) -> (b -> c)

g” :: (b -> c) -> (b -> c)

view the function

as having one input only

consider the case

when **a** is (**b -> c**)

Then **g”** is the same as **id**

Results and effects in a scope

Actually, using the **liftA** commands

we can pull results of applicative functors

into a scope where we can talk

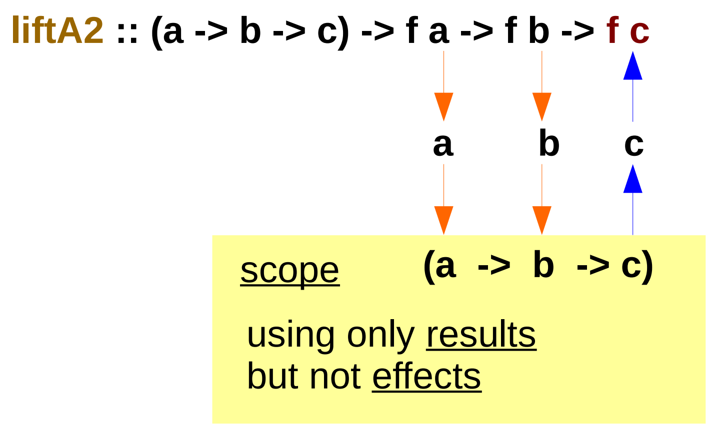
  exclusively about functor results **c**

  and not about effects. **f c**

Note that functor results can also be functions. **c**

This scope is simply a function,

which contains the code that we used in the non-functorial setting.

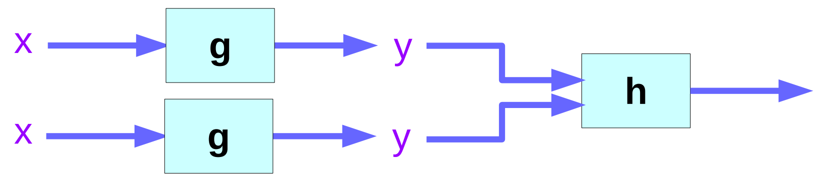


http://hackage.haskell.org/package/base-4.10.1.0/docs/Control-Applicative.html#v:liftA2

liftA3 – a non-functorial expression

Consider the non-functorial expression:

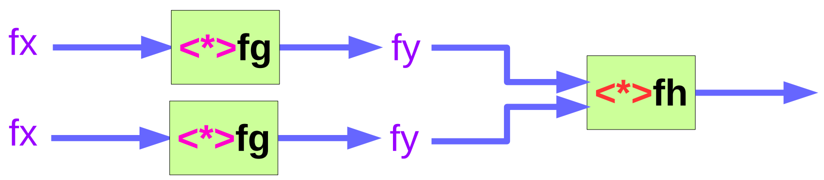
x :: x
g :: x -> y
h :: y -> y -> z



let y = g x
in h y y

generalization

fx :: f x
fg :: f (x -> y)
fh :: f (y -> y -> z)



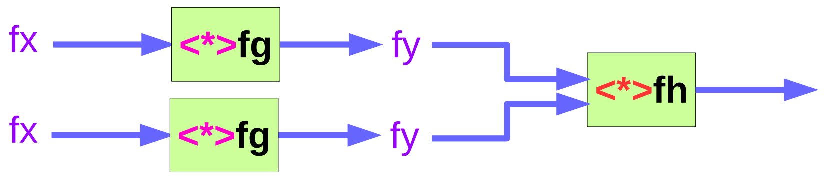
https://wiki.haskell.org/Applicative_functor

liftA3 – using <*> only

```
let fy = fg <*> fx
  in  fh <*> fy <*> fy
```

if **fy** writes something to the terminal

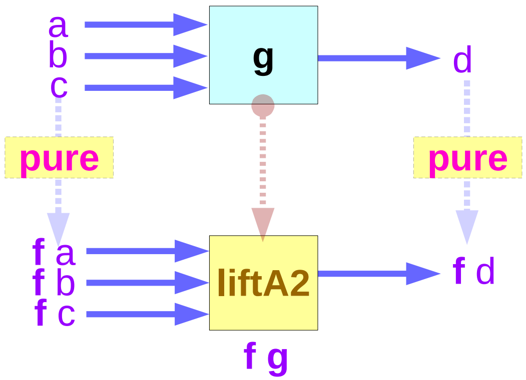
then **fh <*> fy <*> fy** writes twice.



this runs the effect of **fy** twice.

How the effect is run only once and the result is used twice?

→ utilize **liftA3**



https://wiki.haskell.org/Applicative_functor

liftA3 – using three input function

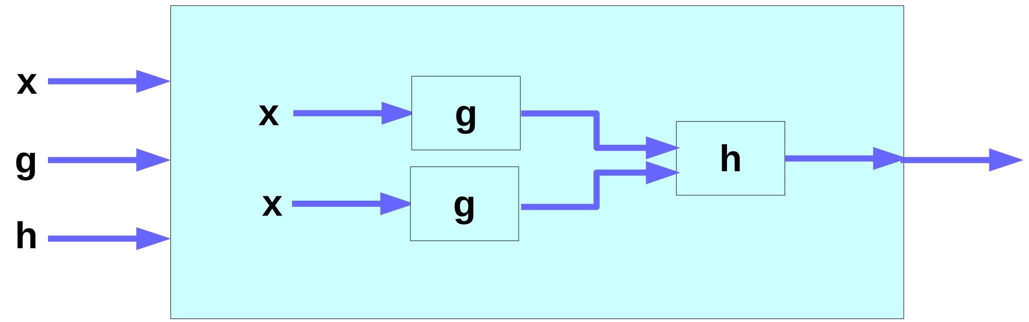
liftA3 :: Applicative f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d

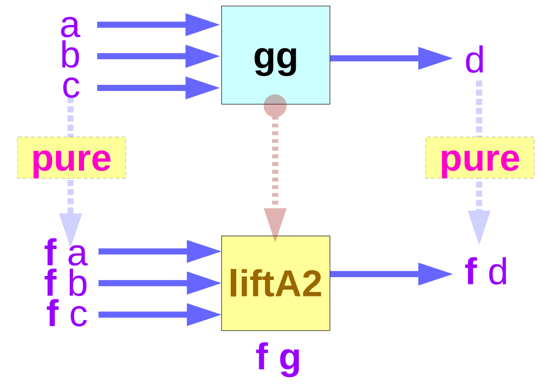
liftA3 h fa fb fc = ((h <$> fa) <*> fb) <*> fc

liftA3

(\x g h -> let y = g x in h y y)

fx fg fh





http://hackage.haskell.org/package/base-4.10.1.0/docs/Control-Applicative.html#v:liftA2

liftA3 – effects, results and scopes

Actually, using the **liftA** commands	
we can pull results of applicative functors	**y** from **fy**
into a scope where we can talk	**y -> y -> z**
exclusively about functor results	**y**
and not about effects.	**fy**
Note that functor results can also be functions.	**y**
This scope is simply a function,	**y -> y -> z**
which contains the code that we used in the non-functorial setting.	

```
liftA3
  (\x g h -> let y = g x in h y y)
  fx fg fh
```

The order of effects is entirely determined by the order of arguments to liftA3

https://wiki.haskell.org/Applicative_functor

liftA2 (<*>)

10
down vote
accepted

The wiki article says that **liftA2 (<*>)** can be used to compose applicative functors. It's easy to see how to use it from its type:

```
o :: (Applicative f, Applicative f1) =>
    f (f1 (a -> b)) -> f (f1 a) -> f (f1 b)
o = liftA2 (<*>)
```

https://stackoverflow.com/questions/12587195/examples-of-haskell-applicative-transformers

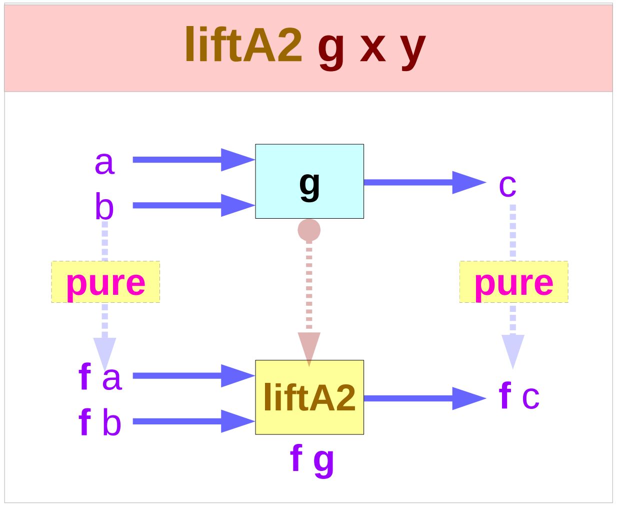
liftA2 (<*>) for composite applicative functors

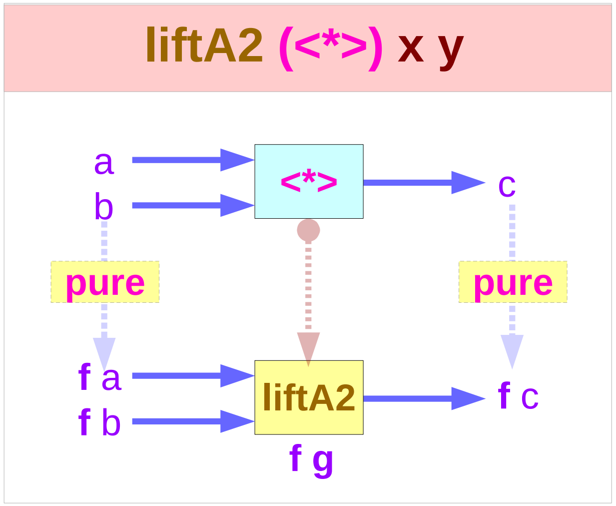
o :: (Applicative f, Applicative f1) =>
f (f1 (a -> b)) -> f (f1 a) -> f (f1 b)

o = liftA2 (<*>)

f1 (a -> b) <*> f1 a <*> f1 b

liftA2 (<*>) can be used to compose applicative functors.





https://stackoverflow.com/questions/12587195/examples-of-haskell-applicative-transformers

liftA2 (<*>) Examples (1)

if f is Maybe and f1 is [ ] we get:

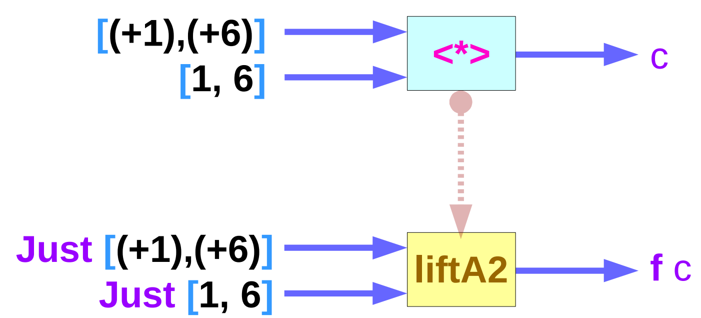
[(+1), (+6)]　　[1, 6]

(+1)　[1, 6]
(+6)　[1, 6]

Just [(+1),(+6)] \`o\` Just [1, 6]
Just [2,7,7,12]

liftA2 (<*>) Just [(+1),(+6)] Just [1, 6]
Just ([(+1),(+6)] <*> [1, 6])
Just   [2, 7, 7, 12])

liftA2 (<*>) Just [(+1),(+6)] Just [1, 6]



https://stackoverflow.com/questions/12587195/examples-of-haskell-applicative-transformers

liftA2 (<*>) Examples (2)

if f is Maybe and f1 is [ ] we get: [Just (+1),Just (+6)] [Just 1, Just 6]

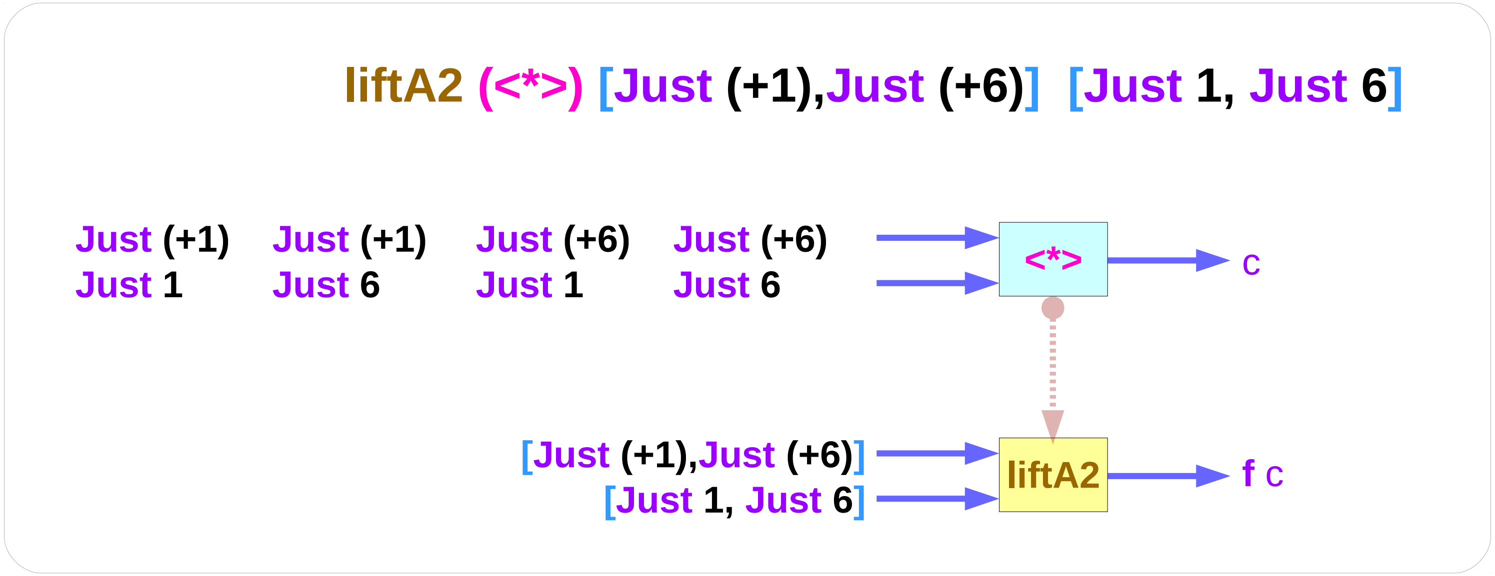
[Just (+1),Just (+6)] \`o\` [Just 1, Just 6] Just (+1) [Just 1, Just 6]

[Just 2, Just 7, Just 7, Just 12] Just (+6) [Just 1, Just 6]

liftA2 (<*>) [Just (+1),Just (+6)] [Just 1, Just 6]

[Just (+1) <*> Just 1, Just (+1) <*> Just 6, Just (+6) <*> Just 1, Just (+6) <*> Just 6]

[Just 2, Just 7, Just 7, Just 12]

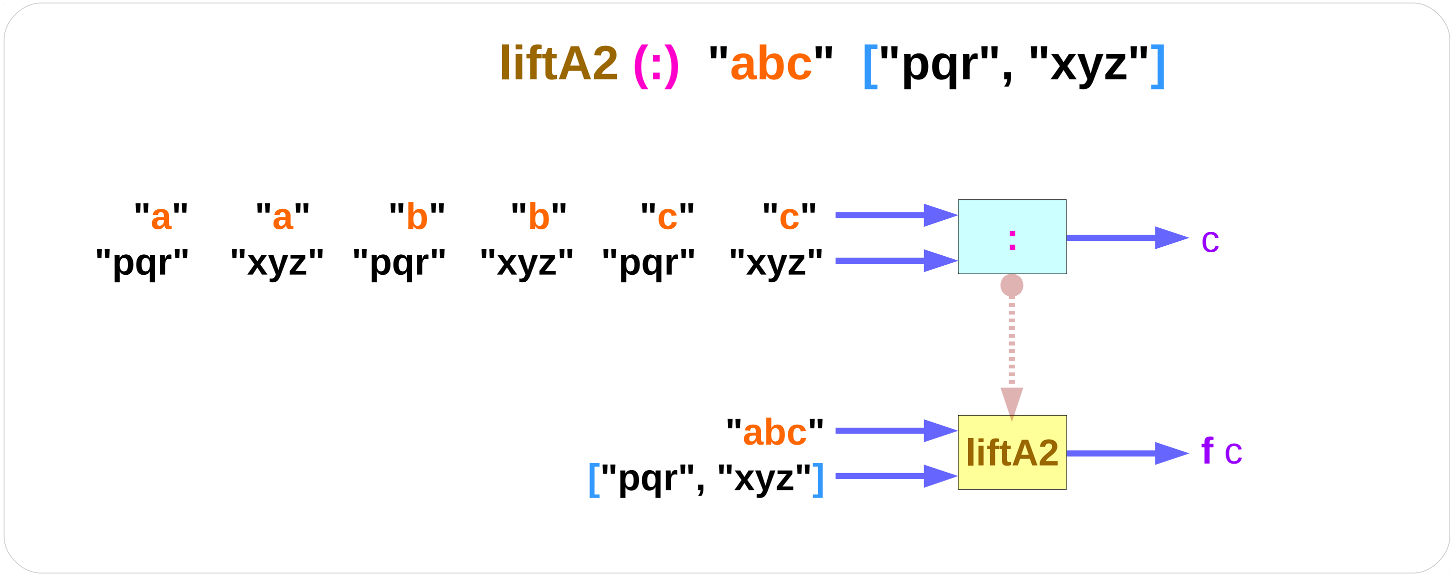


https://stackoverflow.com/questions/12587195/examples-of-haskell-applicative-transformers

liftA2 (:)

liftA2 (:) "abc" ["pqr", "xyz"]

["apqr","axyz","bpqr","bxyz","cpqr","cxyz"]

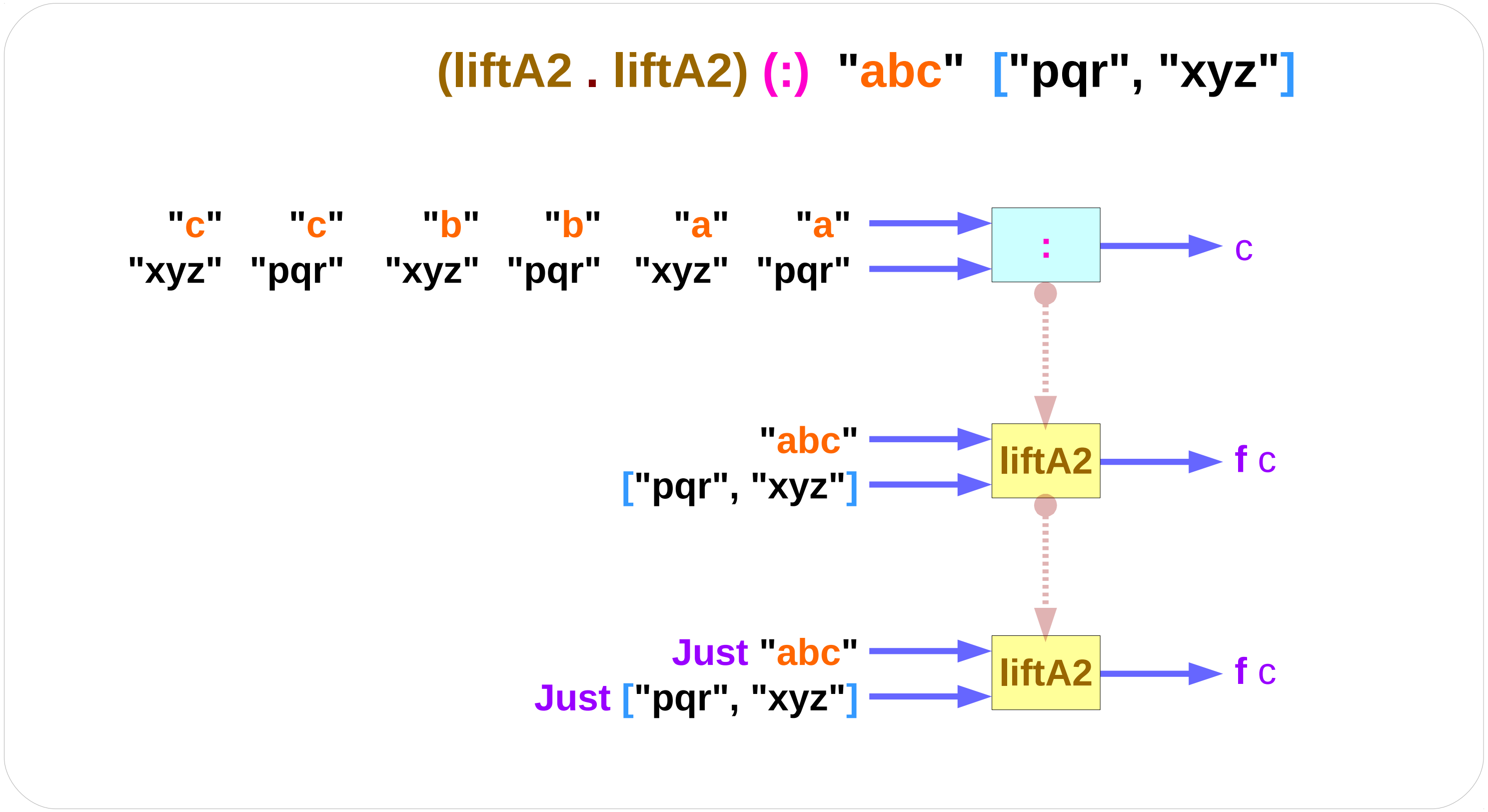


https://stackoverflow.com/questions/12587195/examples-of-haskell-applicative-transformers

liftA2 (:)

(liftA2 . liftA2) (:) (Just "abc") (Just ["pqr", "xyz"])

Just ["apqr","axyz","bpqr","bxyz","cpqr","cxyz"]



https://stackoverflow.com/questions/12587195/examples-of-haskell-applicative-transformers

<$> related operators

Functor map <$>

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

```
value <$ functor = const value <$> functor
functor $> value = const value <$> functor
```

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

<$, <$>, $> operators

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.

```
value <$ functor = const value <$> functor
functor $> value = const value <$> functor
```

```
x <$ y = y $> x        y :: functor
x $> y = y <$ x        x :: functor
```

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

<$, <$>, $> operators examples

```
import Data.Functor

Prelude> Just 1 $> 2
Just 2
Prelude> Just 2 $> 1
Just 1

Prelude>  1 <$ Just 3
Just 1
Prelude>  3 <$ Just 1
Just 3

Prelude>  1 <$ Just 3
Just 1
Prelude>  3 <$ Just 1
Just 3
```

```
import Data.Functor

Prelude> (+1) <$> Just 2
Just 3
Prelude> (+1) <$> Just 3
Just 4

Prelude> (+1) <$> Nothing
Nothing

Prelude> const 2 <$> Just 111
Just 2
```

https://www.schoolofhaskell.com/school/to-infinity-and-beyond/pick-of-the-week/Simple%20examples

<$> examples

```
#!/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

<*> related operators

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

*> operator

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

<* operator

<* 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

<*> examples

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

(*> 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

References

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