

# Applicatives Methods (3B)

---

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](mailto: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
```

$f (a \rightarrow b) ::$  a **function** wrapped in f

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

The class has 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](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](https://en.wikibooks.org/wiki/Haskell/Applicative_functors)

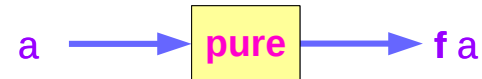
# An Instance of the Applicative Typeclass

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

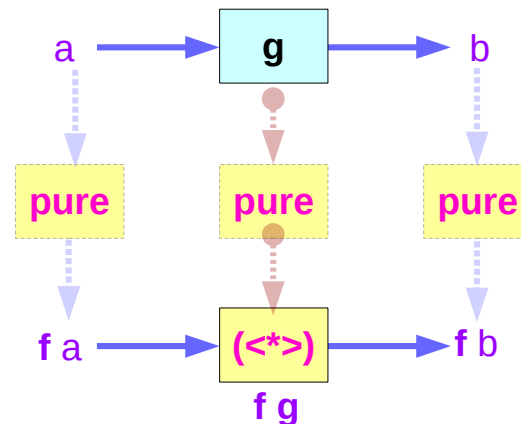
**f** : Functor, Applicative

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

**f** : function in a context



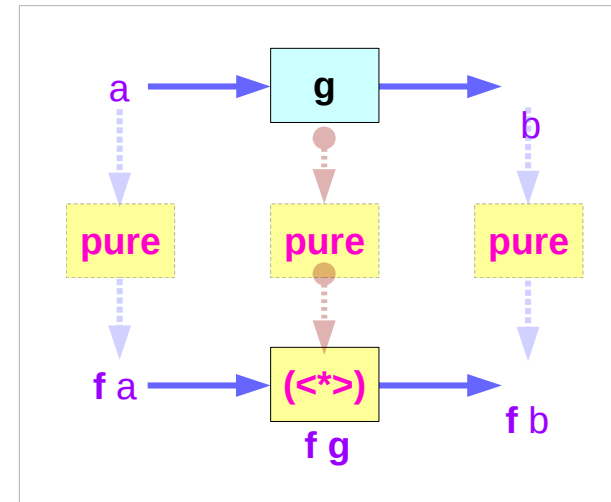
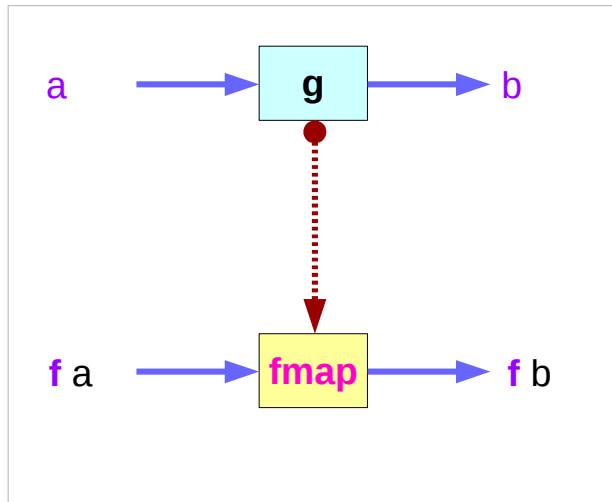
(Functor f) => Applicative f



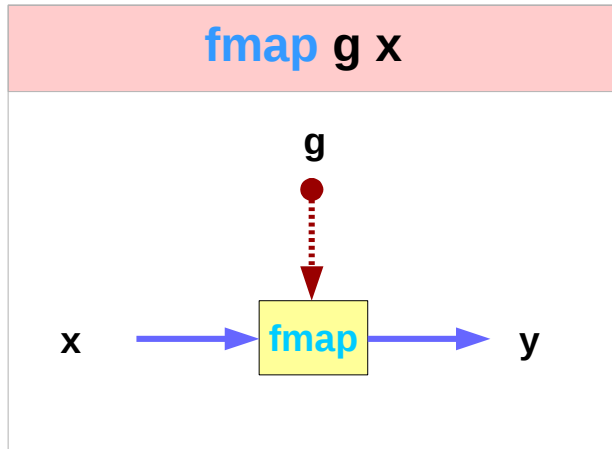
(Functor f) => Applicative f

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

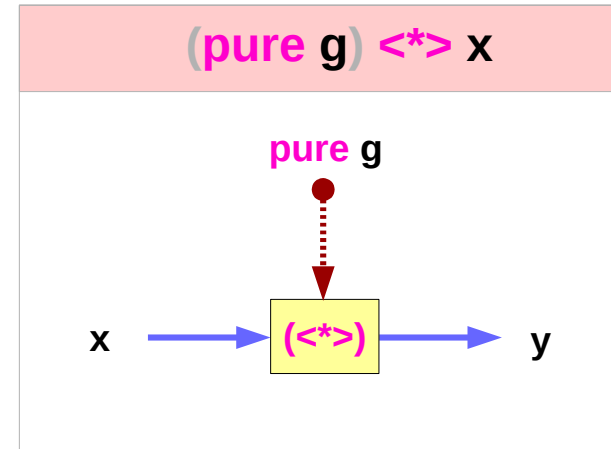
$$\text{fmap } g \ x = (\text{pure } g) \langle * \rangle x$$



pure = f



$x :: f \ a$   
 $y :: f \ b$



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

# Left Associative <\*>

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

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

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

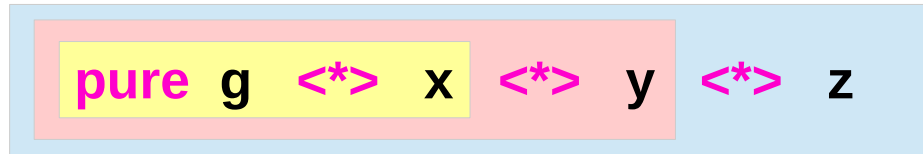
```
Just 8
```

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

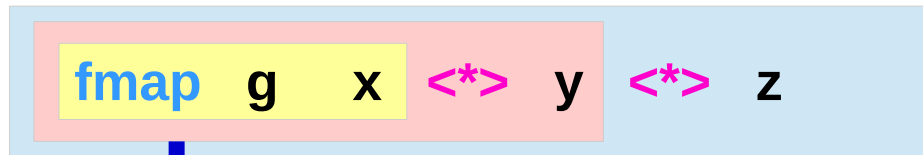
```
ghci> pure (+) <*> Nothing <*> Just 5  
Nothing
```

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

# Infix Operator $\langle \$ \rangle$

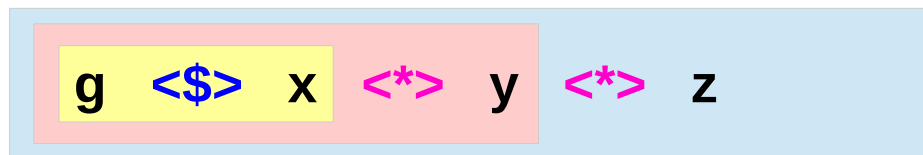


$g :: a \rightarrow b$

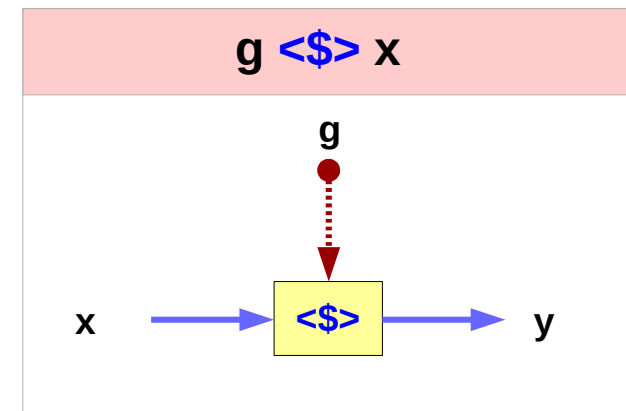
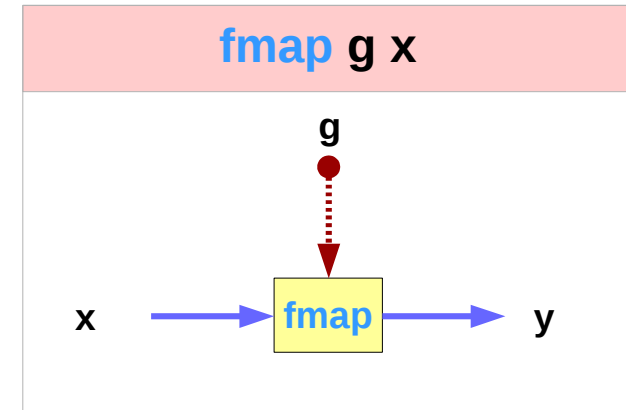


$x :: f\ a$   
 $y :: f\ b$   
 $z :: f\ c$

infix operator



$g :: a \rightarrow b$   
 $x :: f\ a$



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



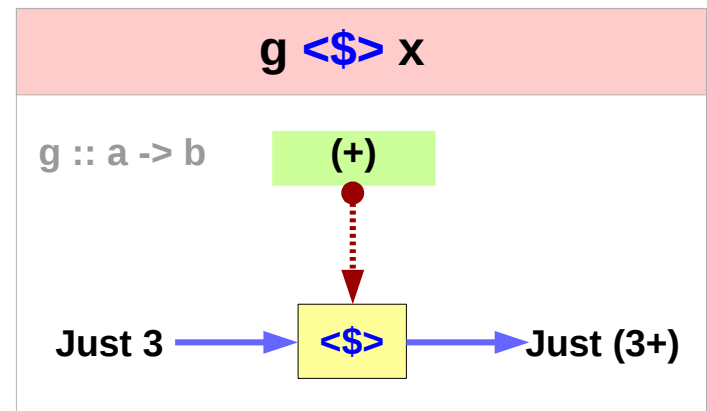
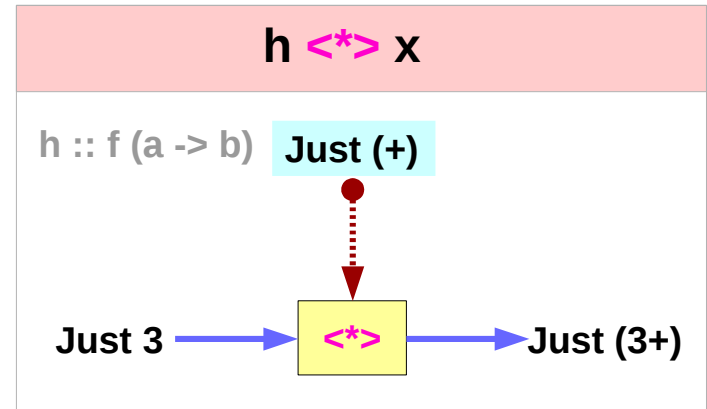
# Infix Operators $\langle * \rangle$ vs $\langle \$ \rangle$

$h \langle * \rangle x \langle * \rangle y$

Just (+)  $\langle * \rangle$  Just 3  $\langle * \rangle$  Just 2  
Just (+3)  $\langle * \rangle$  Just 2  
Just 5

(+)  $\langle \$ \rangle$  Just 3  $\langle * \rangle$  Just 2  
Just (+3)  $\langle * \rangle$  Just 2  
Just 5

$g \langle \$ \rangle x \langle * \rangle y$



<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*

$g :: a \rightarrow b, \quad x :: f a$

```
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 **monads**.

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](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
```

(\$) 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:  $\text{pure id} \langle * \rangle v = v$

$\text{id} :: a \rightarrow a$

$v :: f\ a$

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

$g :: a \rightarrow b$

$x :: a$

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

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

$y :: a$

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**  $\langle * \rangle$   $v = v$

**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](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](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  $u$

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

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

$(\$ y)$  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](https://en.wikibooks.org/wiki/Haskell/Applicative_functors)



# The Composition Law

The composition law

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

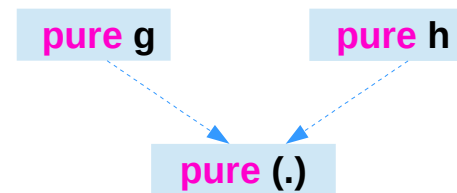
`pure (.)` composes morphisms similarly to how `(.)` composes functions:

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

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

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

$$(g . h) x = g (h x)$$



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

[https://en.wikibooks.org/wiki/Haskell/Applicative\\_functors](https://en.wikibooks.org/wiki/Haskell/Applicative_functors)

# The Composition Law

The composition law

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

$w :: f a$

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

$u : f (b \rightarrow c)$

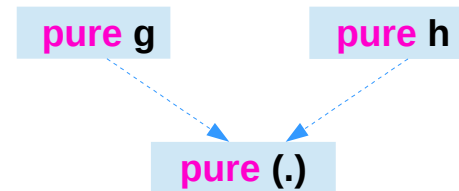
$v \langle * \rangle w :: f b$

$u \langle * \rangle (v \langle * \rangle w) :: f c$

$\text{pure } (.) \langle * \rangle u \langle * \rangle v :: f (a \rightarrow c)$

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

$(g . h) x = g (h x)$



[https://en.wikibooks.org/wiki/Haskell/Applicative\\_functors](https://en.wikibooks.org/wiki/Haskell/Applicative_functors)

# liftA2

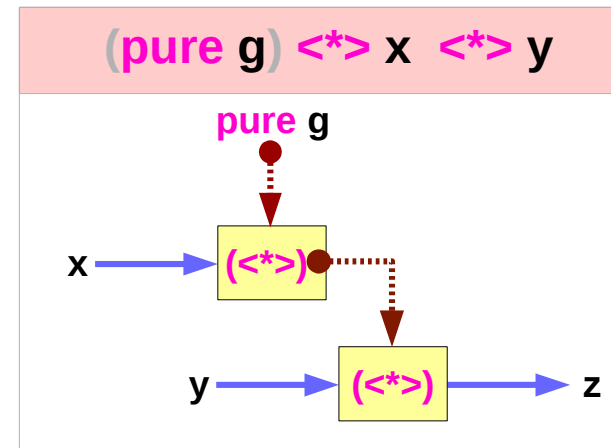
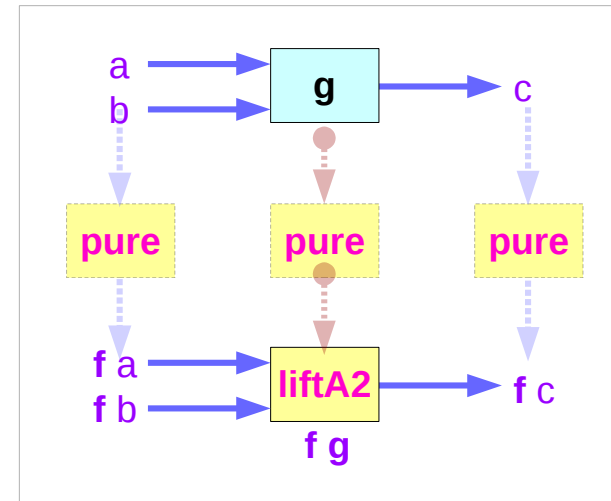
`liftA2 :: (a -> b -> c) -> f a -> f b -> f c`

Lift a binary function 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, then 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

variables

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

```
x :: f a
```

```
y :: f b
```

```
z :: f c
```

```
pure g <*> x <*> y
```

```
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
```

[https://wiki.haskell.org/Applicative\\_functor](https://wiki.haskell.org/Applicative_functor)

# liftA2

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.

## 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

.

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

# liftA2

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

A minimal complete definition must include implementations of **pure** and of either **<\*>** or **liftA2**.

**pure** and **<\*>**

**pure** and **liftA2**

If it defines both, then they must behave the same as their default definitions:

**<\*> = liftA2 id liftA2**

**f x y = f <\$> x <\*> y**

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

# 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 (<*>)
```

So to if `f` is `Maybe` and `f1` is `[]` we get:

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

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

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

# liftA2

The other way around is:

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

your ex function is equivalent to liftA2 (:):

```
test1 = liftA2 (:) "abc" ["pqr", "xyz"]
```

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



# liftA2

To use (:) with deeper applicative stack  
you need multiple applications of liftA2:

```
*Main> (liftA2 . liftA2) (:) (Just "abc") (Just ["pqr", "xyz"])  
Just ["apqr", "axyz", "bpqr", "bxyz", "cpqr", "cxyz"]
```

However it only works when both operands are equally deep.  
So besides double liftA2 you should use pure to fix the level:

```
*Main> (liftA2 . liftA2) (:) (pure "abc") (Just ["pqr", "xyz"])  
Just ["apqr", "axyz", "bpqr", "bxyz", "cpqr", "cxyz"]
```

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

# liftA2

Consider the non-functorial expression:

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

```
let y = g x  
in h y y
```

Very simple. Now we like to generalize this to

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

[https://wiki.haskell.org/Applicative\\_functor](https://wiki.haskell.org/Applicative_functor)

# liftA2

However, we note that

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

runs the effect of fy  
twice. E.g. if fy

writes something to the terminal then fh <\*> fy <\*> fy

writes twice. This could be intended, but how can we achieve, that the effect is run only once and the result is used twice? Actually, using the liftA

commands we can pull results of applicative functors into a scope where we can talk exclusively about functor results and not about effects. Note that functor results can also be functions. This scope is simply a function, 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](https://wiki.haskell.org/Applicative_functor)

# <\$> 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

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>

# <\$> 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>

# <\$, \$> operators

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.

**value <\$ functor = const value <\$> functor**

**functor \$> value = const value <\$> functor**

**x <\$ y = y \$> x**

**x \$> y = y <\$ x**

<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>

# <\*> 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>



# \*> 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>

# (\*> 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](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>