Applicatives Methods (3B)

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

This document was produced by using LibreOffice.

Young Won Lim
3/7/18

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

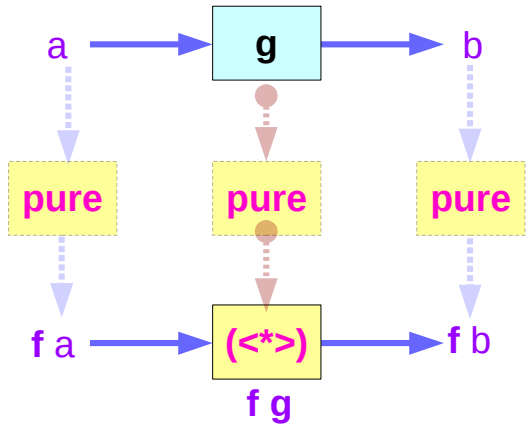
```
class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

f : Functor, Applicative

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

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

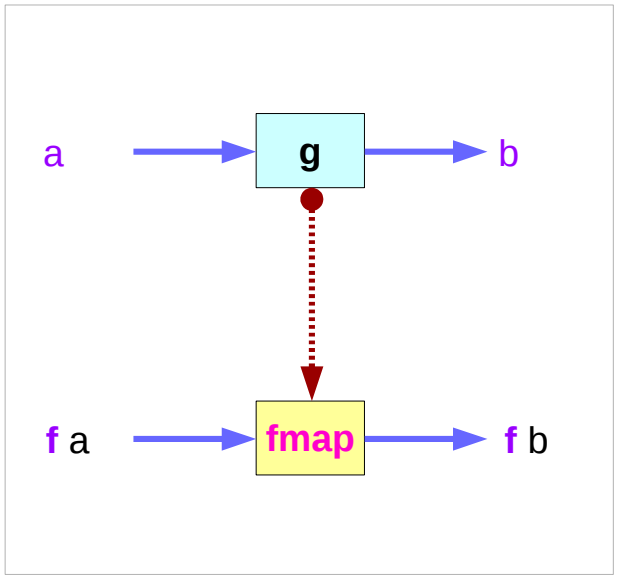
f : function in a context

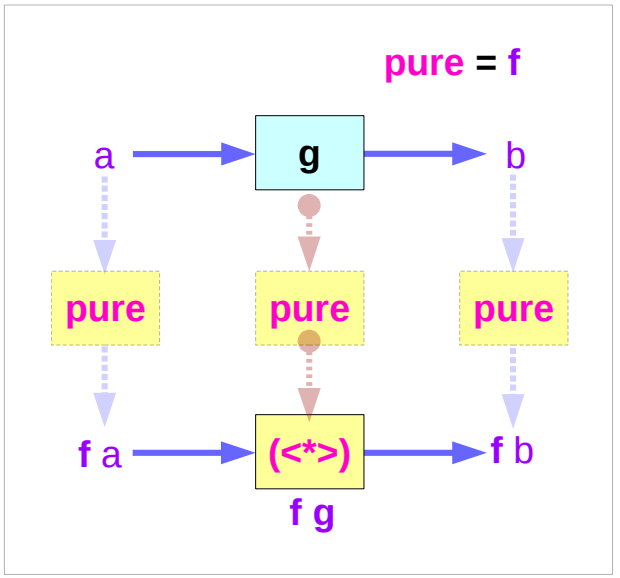


(**Functor f**) => **Applicative f**

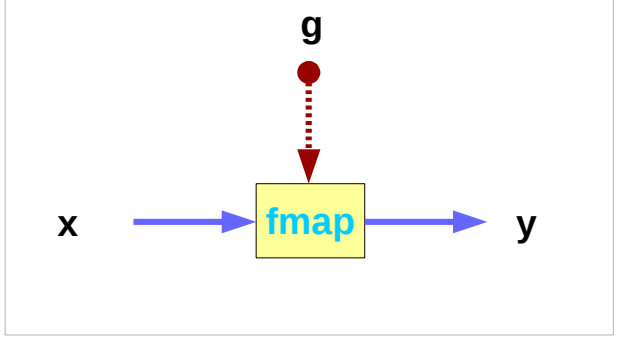
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

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

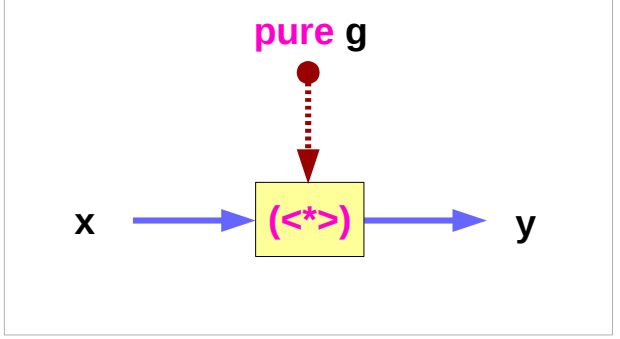




fmap g x



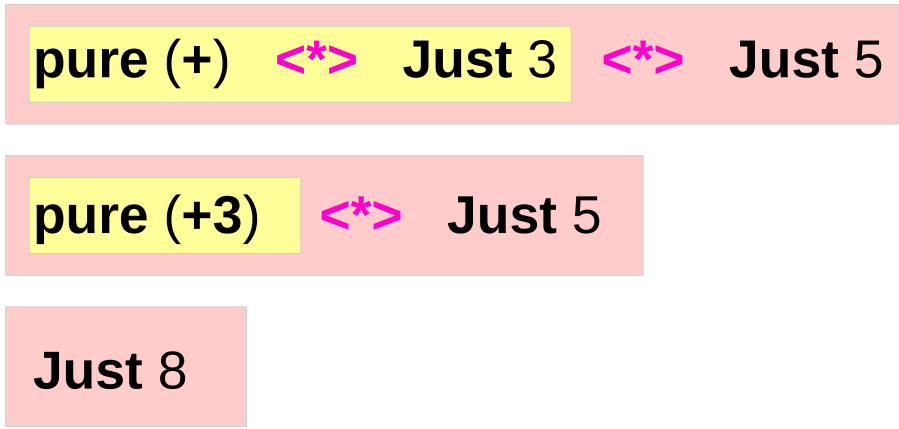
(pure g) <*> x



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

Left Associative <*>

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




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

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

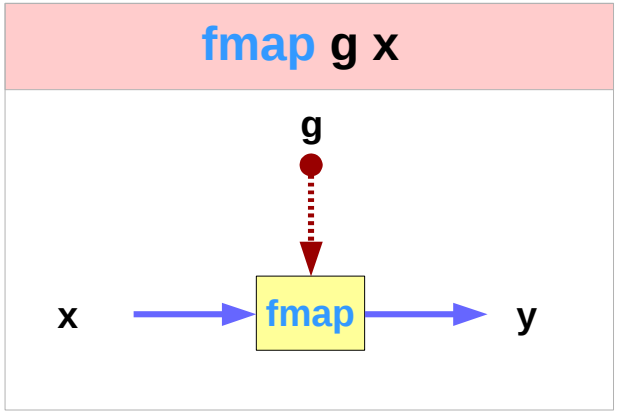
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

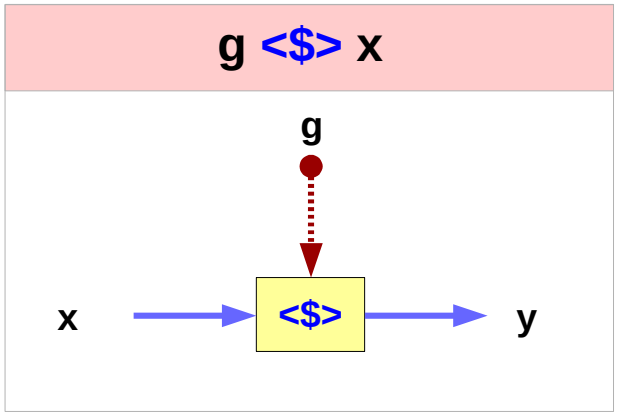
Infix Operator <$>

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

```
fmap f x <*> y <*> z
```

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





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

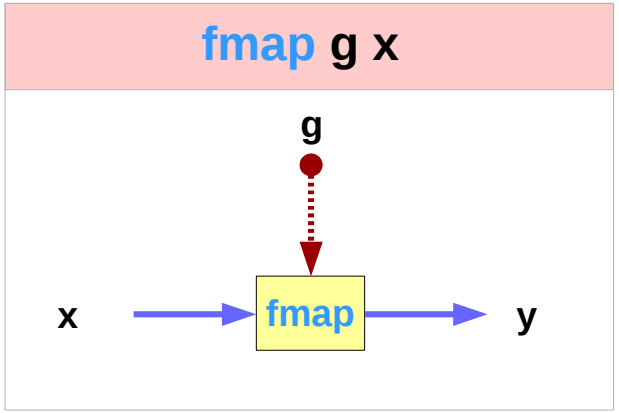
Infix Operator <$> : not a class method

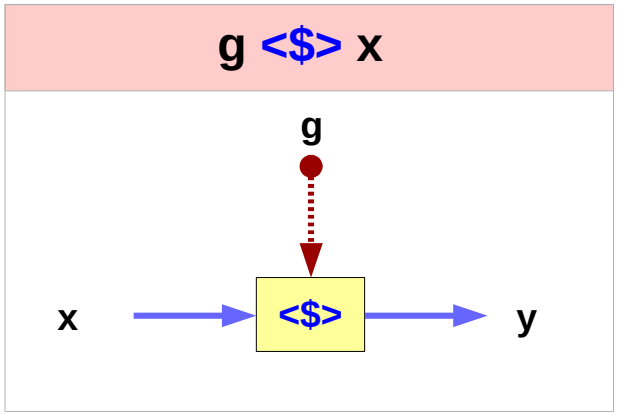
```
class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

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

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