Function Applicative (3A)

Young Won Lim
8/3/18

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

(->) r Applicative – pure

```
instance Applicative ((->) r) where
    pure x = (\_ -> x)
    f <*> g = \x -> f x (g x)
```

When we wrap a **value** into an applicative functor with **pure**,
the **result** it yields always has to be **applicative value**.
A minimal default context that still yields that **value** as a **result**.

pure takes a **value** and creates a **function**
that ignores its **parameter** and
always returns that taken **value**.

the **type** for **pure** for the **(->) r instance**,

```
pure :: a -> (r -> a)
```

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

(->) r Applicative – pure examples

```
> (pure 3) "blah"
  3
> pure 3 "blah"
  3
```

Because of **currying**, function application is **left**-**associative**,
the parentheses can be eliminated

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

(->) r Applicative – <*>

```
> :t (+) <$> (+3) <*> (*100)
(+) <$> (+3) <*> (*100) :: (Num a) => a -> a          -- a function

> (+) <$> (+3) <*> (*100) $ 5
508
```

Calling **<*>** with two **functions** (**applicative functors**) results in a **function** (an **applicative functor**)

(+) <$> (+3) <*> (*100) results in a **function** that uses **(+)** on the **results** of **(+3)** and **(*100)** and return that result.

the **5** first got applied to **(+3)** and **(*100)**, resulting in **8** and **500**. Then, (**+)** gets called with **8** and **500**, resulting in **508**.

using **functions (+3)**, **(*100)** as **applicative functors** in the **applicative style**

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

(->) r Applicative – <*>

functions as boxes that contain their eventual results
so doing **k <$> f <*> g** creates a **function**
that will call **k** with the eventual **results** from **f** and **g**.

(+) <$> Just 3 <*> Just 5, we're using + on **values**
that might or might not be there,
which also results in a **value**
that might or might not be there.

(+) <$> (+10) <*> (+5), we're using + on the **future return values**
of **(+10)** and **(+5)** and the **result** is also something
that will produce a **value** only when called with a parameter.

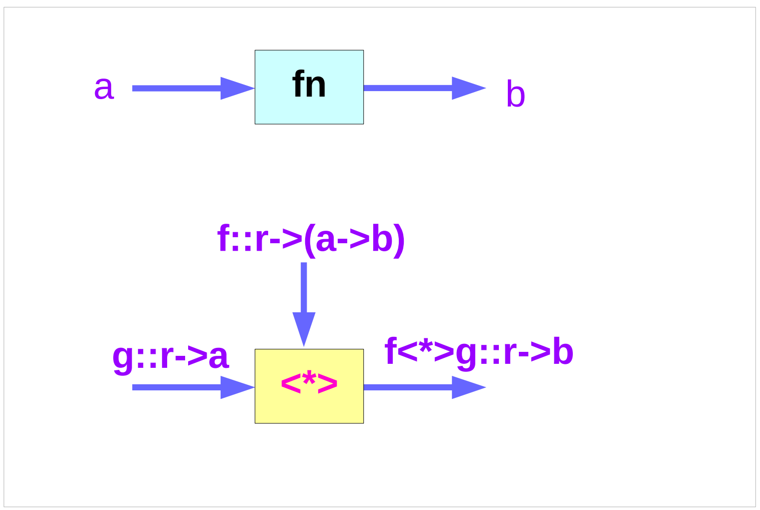
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

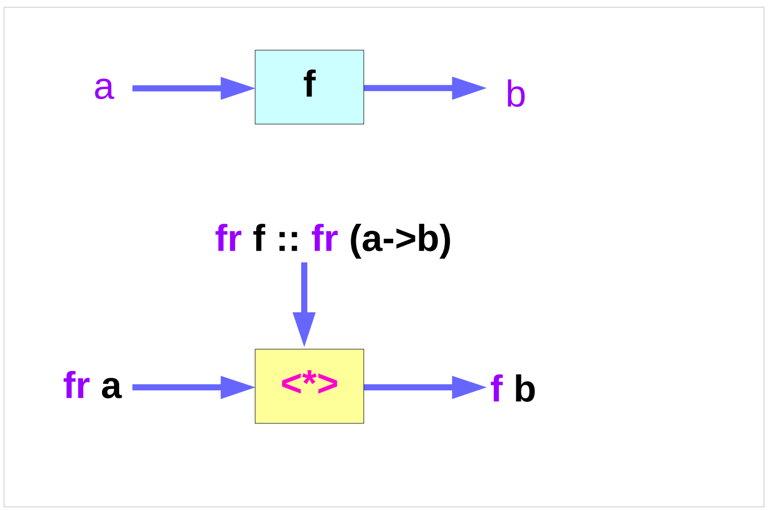
(->) r Applicative – pure

```
instance Applicative ((->) r) where
    pure x = (\_ -> x)
    f <*> g = \x -> f x (g x)
```

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

```
pure :: a -> (r -> a)
f <*> g :: r ->b
```





pure fn :: r->(a->b)

(->) r Applicative – <*>

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

(pure (+5)) <*> (*3) $ 4
(fmap (+ 5) (* 3)) 4
((+ 5) . (* 3)) 4          -- fmap = (.)
17
```

<*> essentially applies a **function** in the **functor** to a **value** in the **functor**.

Specializing to (->) r, this means it applies a function returned by a function from r to a value returned by a function from r. A function that returns a function is just a function of two arguments. So the real question is this: how would you apply a function of two arguments (r and a, returning b) to a value a returned by a function from r?

https://stackoverflow.com/questions/11810889/functions-as-applicative-functors-haskell-lyah

(->) r Applicative – <*>

must return a **value** of type **(->) r**
which means the **result** also has to be a **function** from **r**.

f <*> g = \x -> f x (g x)

Since we want to return a **function** taking a **value** of type **r**, **x :: r**.
The **function** we return has to have a type **r -> b**.

a **function f :: r -> a -> b**
Since **r** is going to be the **argument** of the result **function**,
a **function** from **a -> b**.
another **function g :: r -> a**.
So we can take our **value** of type **r** (the parameter **x**)
and use it to get a value of type **a**.

https://stackoverflow.com/questions/11810889/functions-as-applicative-functors-haskell-lyah

(->) r Applicative – <*>

we use the **parameter r** to first get a **value** of type **a**
by plugging it into **g :: r -> a**.
The **parameter** has type **r**, g has type **r -> a**, so we have an **a**.
plug both the **parameter r** and the new **value a** into **f :: r -> a -> b** .
Once we plug both an **r** and an **a** in, we have a **b**.
Since the **parameter** is in a lambda, the result has a type **r -> b**

https://stackoverflow.com/questions/11810889/functions-as-applicative-functors-haskell-lyah

(->) r Applicative – <*>

(+) <$> (+3) <*> (*100) $ 5
pure (+) <*> (+3) <*> (*100) $ 5

the **pure** before **(+)**, which has the effect of
boxing (+) as an **Applicative**.
to **unbox** it, you need to provide an additional **argument**,
which can be anything.
Applying **<*>** to **(+) <$> (+3)**, we get
\x -> (pure (+)) x ((+3) x)

Notice in **(pure (+)) x**, we are applying **x** to **pure** to unbox **(+)**.
\x -> (+) ((+3) x)

https://stackoverflow.com/questions/11810889/functions-as-applicative-functors-haskell-lyah

Adding **(*100)** to get **(+) <$> (+3) <*> (*100)**
and apply **<*>** again, we get
\x -> (+) ((+3) x) ((*100) x)

So in conclusion, the **x** after **f** is NOT the first argument
to our binary operator, it is used to UNBOX the operator inside pure.

https://stackoverflow.com/questions/11810889/functions-as-applicative-functors-haskell-lyah

(->) r Applicative – <*>

remeber that **pure (+5)** discards its first argument,
so it's **const (+5) 4 $ (4 * 3)** or **4 * 3 + 5**
which is consistent with **(+5) . (*3) $ 4**.

Additionally, **f <*> g = \x -> f (g x)**
is of type **(b -> c) -> (a -> b) -> (a -> c)**
which neither typechecks with **pure (+ 5) <*> (* 3) $ 4**
nor the class declaration of **Applicative**

https://stackoverflow.com/questions/11810889/functions-as-applicative-functors-haskell-lyah

References

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