# Function Applicative (3A)

Young Won Lim
8/4/18

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.

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.

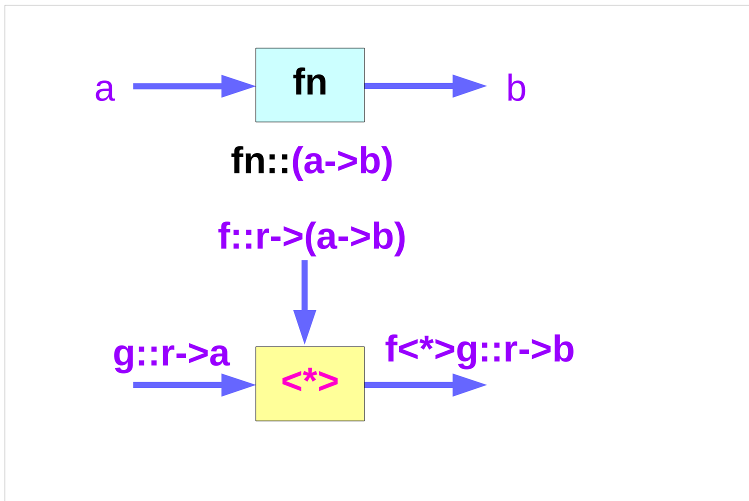
http://learnyouahaskell.com/functors-applicative-functors-and-monoids

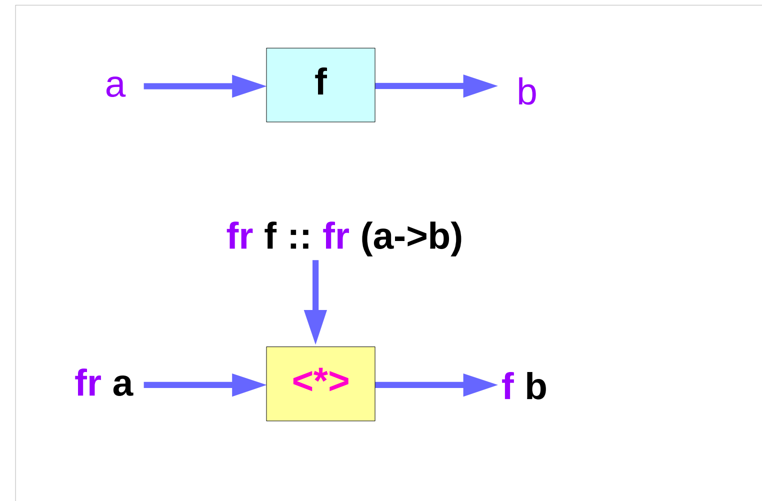
# (->) r Applicative – pure

```
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)

```
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 left **functor** to a **value** in the right **functor**.

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

# (->) r Applicative – <*>

The **function functor** specializes to **(->) r**

**<*>** applies
a **function (a->b)** returned by a **function from r** **r->(a->b)**
to a **value a** returned by a **function from r** **r->a**

the **result** of **<*>** must
be a **function from r** **r->b**
return a **value** of type **(->) r** an **function applicative value**

A **function** that returns a **function** is
just a **function** of two **arguments**. **r->a->b**

how to suply two **arguments** (**r** and **a**, returning **b**) (**r->a**)**->b**

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

# (->) r Applicative – <*>

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

Since a **function** taking a **value** of type **r** must be returned, **x :: r**.

The result **function** of **<*>** must have a type **r -> b**.

a **function f :: r -> a -> b**
**r** is the **argument** type of **f**
a **function a -> b** is returned by the **function f**

another **function g :: r -> a**
take the **value** of type **r** (the parameter **x**)
**g** is used to get a **value** of type **a**.

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

# (->) r Applicative – <*>

use the **parameter r**
to 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** .

plug both an **r** and an **a** in **f :: r -> a -> b**, 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
```

**pure (+) : boxing (+)** as an **Applicative**.

to **unbox pure (+)**, provide an additional **argument**,
with a type of **r** whose **value** which can be anything **_->x**

Applying **<*>** to **(+) <$> (+3)**,
we get **\x -> (pure (+)) x ((+3) x)**

```
pure (+) <*> (+3)
f <*> g = \x -> f x (g x)
```

**(pure (+)) x**, we are applying **x** to **pure** to unbox **(+)**.

```
\x -> (pure (+)) x ((+3) x)
\x -> (+) ((+3) x)
```

```
pure :: a -> fr a
pure :: a -> (r -> a)
pure x = (\_ -> x)

pure :: (a->a->a) -> (r -> (a->a->a))
pure (+) = (\_ -> (+))
```

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

# (->) r Applicative – <*>

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

appending **(*100)** to get **(+) <$> (+3) <*> (*100)**
and apply **<*>** again, we get

**\x -> (pure (+)) x ((+3) x) ((*100) x)**
**\x -> (+) ((+3) x) ((*100) x)**

the **x** after **f** is NOT the first **argument** **(pure (+)) x**
to our binary operator **(+)**,
**x** is used to UNBOX the operator **(+)** inside **pure**.

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

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