Applicative Sequencing (3C)

Young Won Lim

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

<$> 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
```

replace b in f b with a … f a

replace a in f a with b … f b

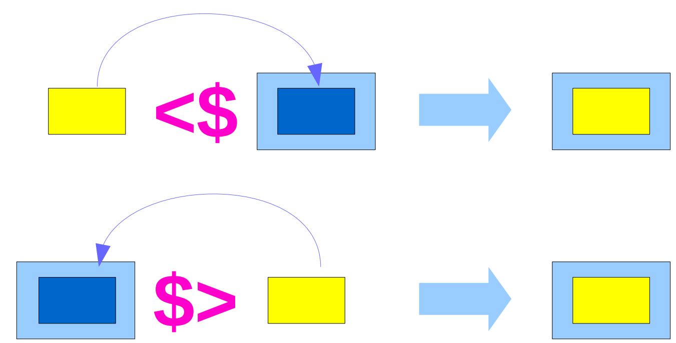
The <$> operator is just a synonym

for the fmap function in the Functor typeclass.

fmap generalizes map for **lists**

to other data types : **Maybe**, **IO**, **Map**.

Replacing the core



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

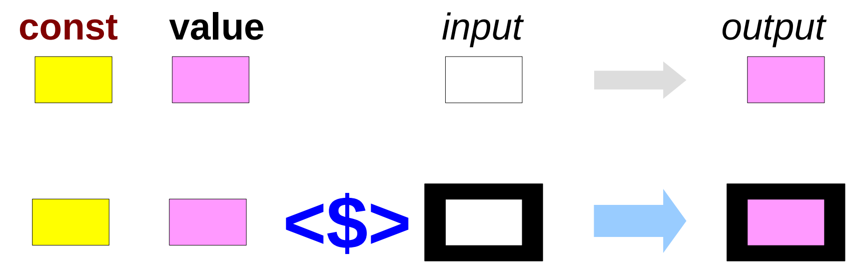
<$ / <$> / $> operators

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
```

```
getLine :: IO String
```

```
Input: read "12"::Double
Output: 12.0
```

```
-- this infix synonym for mappend is found in Data.Monoid
x <> y = mappend x y
infixr 6 <>
```

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

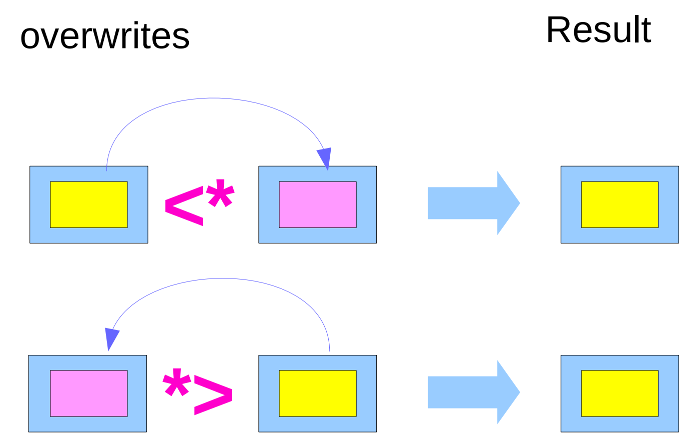
<*> related operators

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



<*> is an operator that applies

a wrapped function to a wrapped value.

<*> is a part of the Applicative typeclass,

<*>is very often used as follows

foo <$> bar <*> baz

faa <*> bar <*> baz

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

*> operator

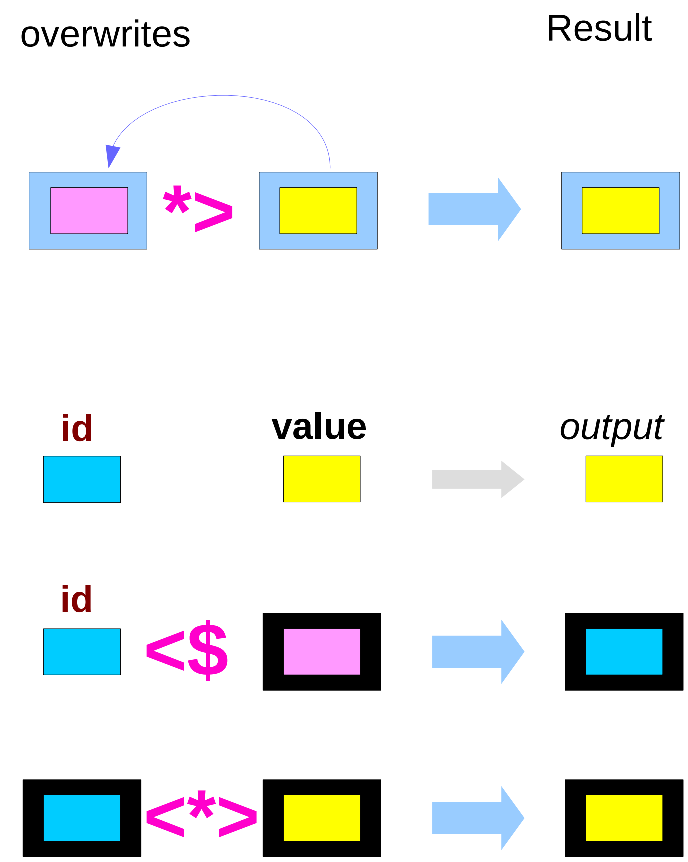
two helper operators

*> ignores the value from the first argument.

*> is completely equivalent to >> in Monad

a1 *> a2 = (id <$ a1) <*> a2

```
a1 *> a2 = do
    _ <- a1
    a2
```



(id <$ a1)

(id <$ a1) <*> a2

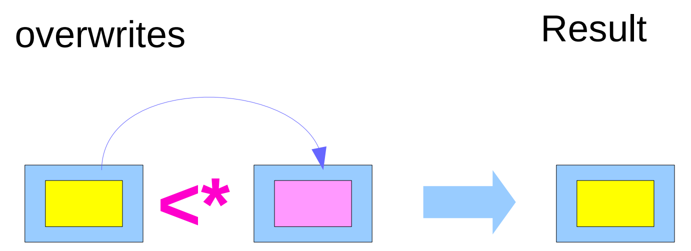
https://haskell-lang.org/tutorial/operators

<* operator

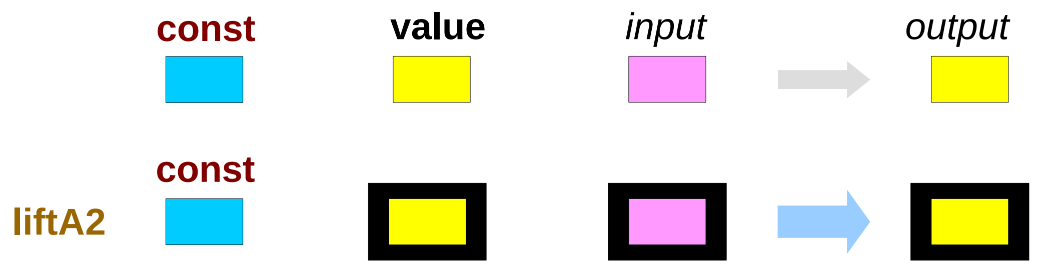
<* is the same thing in reverse: perform the first action then the second,

but only take the value from the first action.

(<*) = liftA2 const



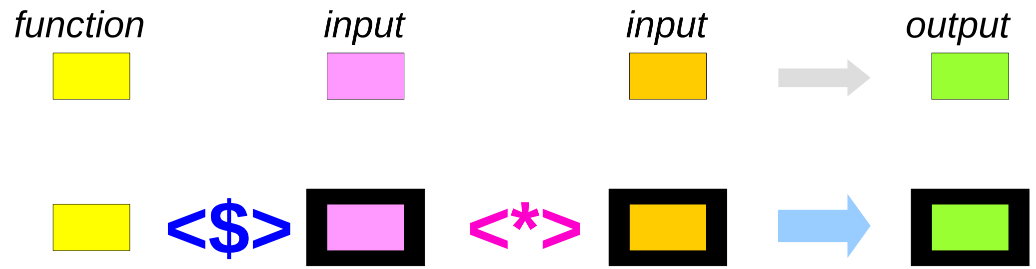

```
a1 <* a2 = do
  res <- a1
    _ <- a2
  return res
```



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

<*> examples

foo <$> bar <*> baz

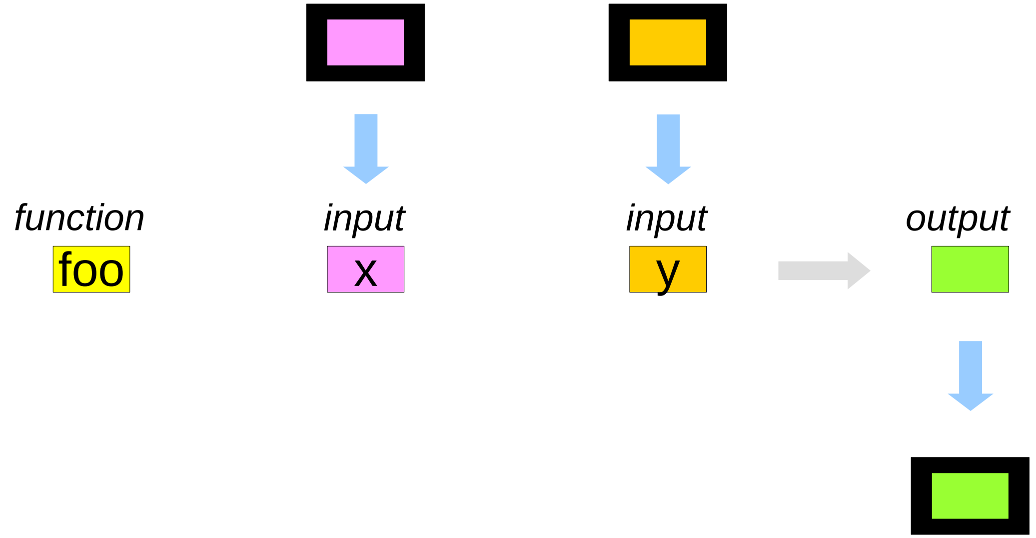


With a Monad, this is equivalent to:

do x <- bar

y <- baz

return (foo x y)



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

<*> examples

examples including parsers and serialization libraries.

using the **aeson** package: (handling **JSON** data)

```
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
```

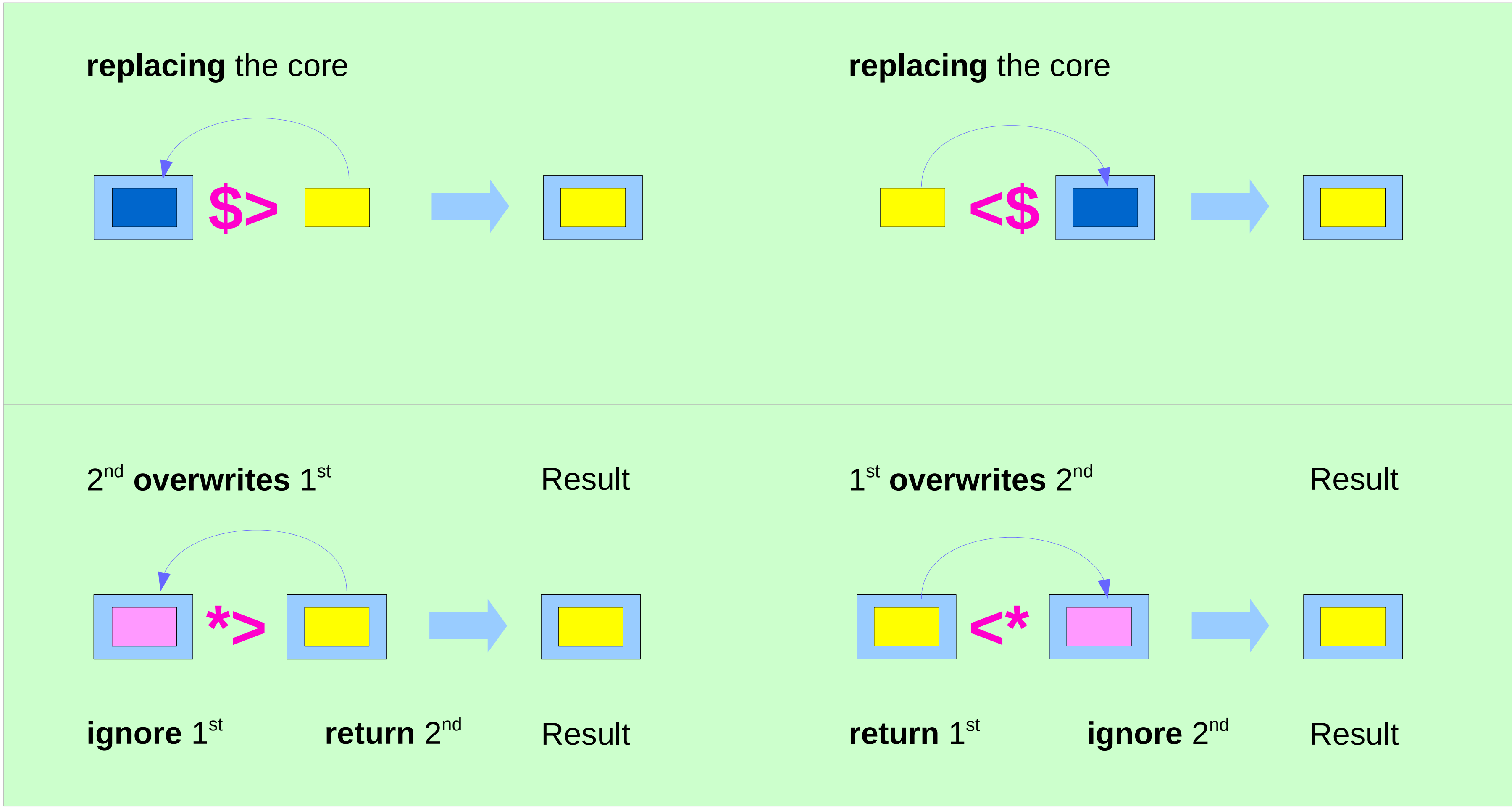
: append-head operator (cons)

. function composition operators

. name qualifier

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

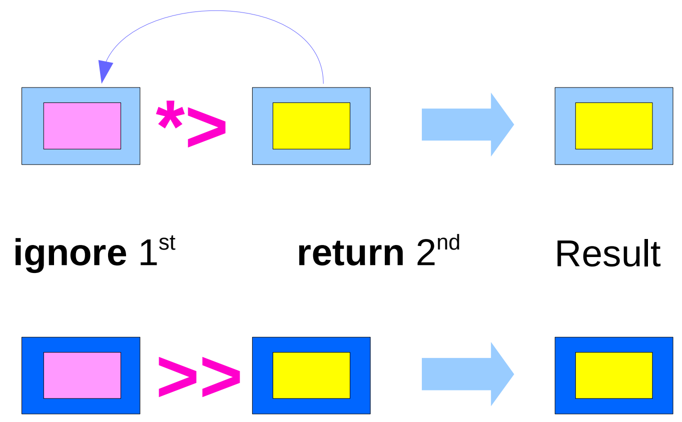
($> v.s. <$) and (*> v.s. <*)



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

Sequencing of Effects

commutative monads in Haskell,

the concept involved is the same, only specialised to Monad.

Commutativity (or the lack thereof) affects

other functions which are derived from **(<*>)** as well.

(*>) is a clear example:

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

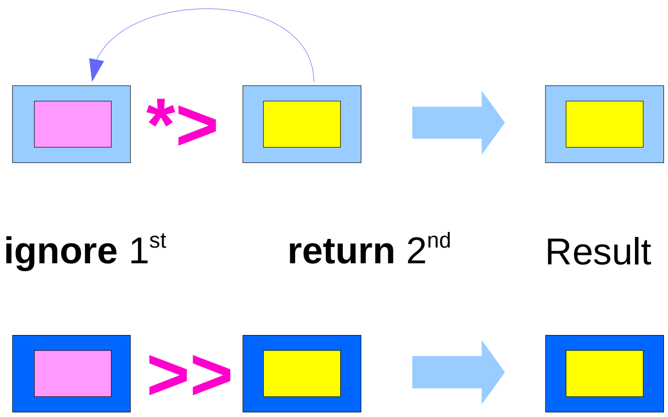
(*>) combines **effects** while preserving

only the **values** of its second argument.

For monads, it is equivalent to **(>>)**.

Here is a demonstration of it using Maybe,

which is commutative:



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

Left-to-right sequencing

The convention in Haskell is to always implement (<*>)

and other applicative operators using **left-to-right sequencing**.

Even though this convention helps reducing confusion,

it also means appearances sometimes are misleading.

For instance, the **(<*)** function is not flip **(*>)**,

as it sequences effects from left to right just like **(*>)**:

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

<**> operators

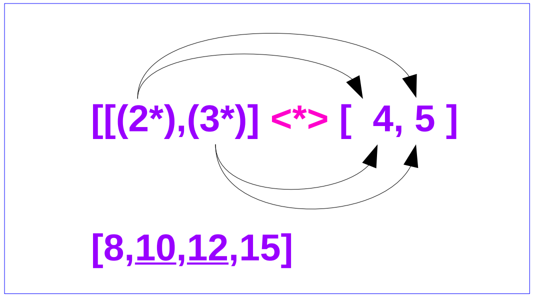
(<>) :: Applicative f => f a -> f (a -> b) -> f b**

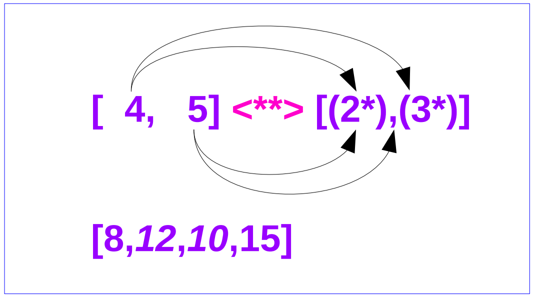
(<*>) :: Applicative f => f (a -> b) -> f a -> f b

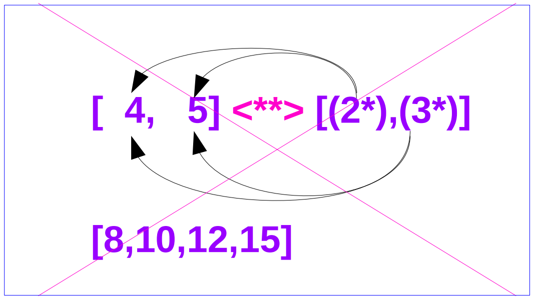
from **Control.Applicative**

not flip **(<*>)**

a way of inverting the sequencing







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

Sequencing examples (1)

```
Prelude> [(2*),(3*)] <*> [4,5]
[8,10,12,15]
Prelude> [4,5] <**> [(2*),(3*)]
[8,12,10,15]
```

```
[(2*)] <*> [4,5],   [(3*)] <*> [4,5]

[4] <**> [(2*),(3*)],     [5] <**> [(2*),(3*)]
```

```
Prelude> Just 2 *> Just 3
Just 3
Prelude> Just 3 *> Just 2
Just 2
Prelude> Just 2 *> Nothing
Nothing
Prelude> Nothing *> Just 2
Nothing
```

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

Sequencing examples (2)

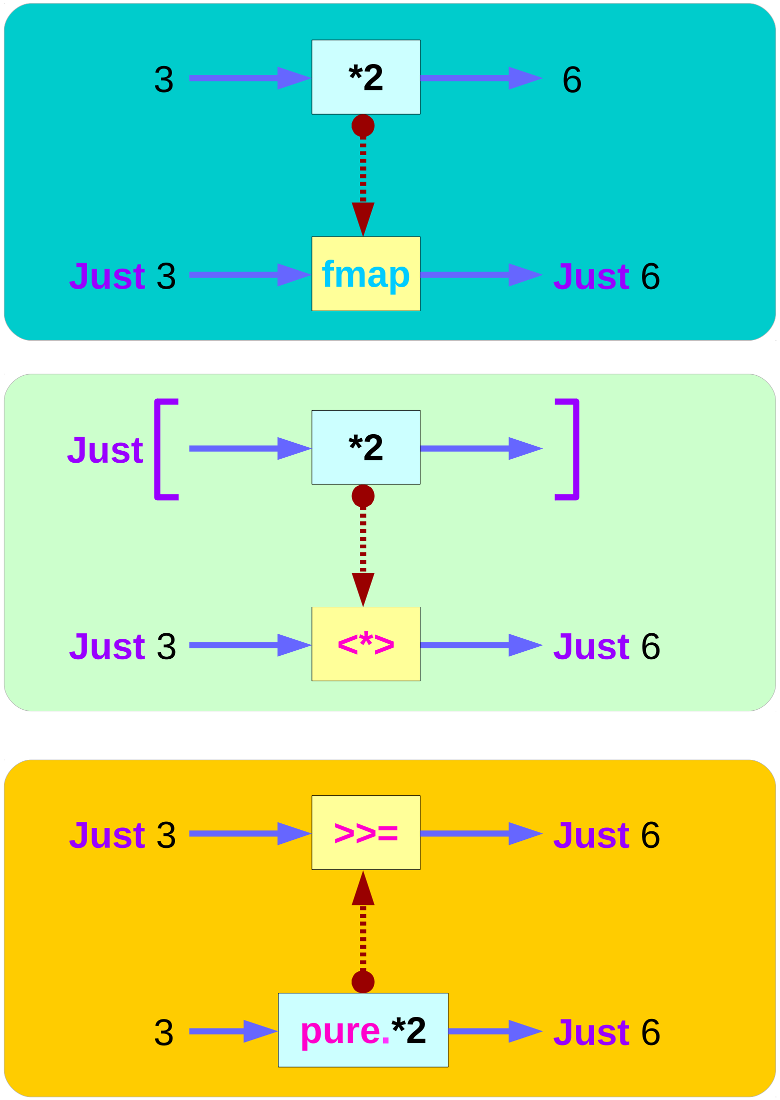
```
Prelude> (print "foo" *> pure 2) *> (print "bar" *> pure 3)
"foo"
"bar"
3
Prelude> (print "bar" *> pure 3) *> (print "foo" *> pure 2)
"bar"
"foo"
2
```

```
Prelude> (print "foo" *> pure 2) <* (print "bar" *> pure 3)
"foo"
"bar"
2
```

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

Functors, Applicative, and Monad

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

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

(>>=) :: Monad       m => m a        -> (a -> m b)   -> m b
```



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

Functors, Applicative, and Monad Examples

```
fmap  :: Functor     f  =>  (a -> b)    -> f a            -> f b
(<*>) :: Applicative f  =>  f (a -> b)  -> f a            -> f b
(>>=) :: Monad       m  =>  m a         -> (a -> m b)     -> m b
```

```
Prelude> fmap (*2) (Just 3)
Just 6
```

```
Prelude> (Just (*2)) <*>  (Just 3)
Just 6
```

```
Prelude> (Just 3) >>= (pure . (*2))
Just 6
Prelude> (Just 3) >>= (return . (*2))
Just 6
```

Comparing the three characteristic methods

replace **fmap** by its infix synonym, **(<$>)**

replace **(>>=)** by its flipped version, **(=<<)**

```
fmap  :: Functor       f  =>   (a -> b)  -> f a        -> f b
(<*>) :: Applicative   f  => f (a -> b)  -> f a        -> f b
(>>=) :: Monad         m  => m a         -> (a -> m b) -> m b
```

```
(<$>) :: Functor t      =>   (a ->   b)    -> (t a -> t b)
(<*>) :: Applicative t  => t (a ->   b)    -> (t a -> t b)
(=<<) :: Monad t        =>   (a -> t b)    -> (t a -> t b)
```

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

All mapping functions over Functors

fmap, **(<*>)** and **(=<<)** are all mapping functions over **Functors**.

The differences between them are in what is being mapped over in each case:

```
(<$>) :: Functor t      =>   (a ->   b)     -> (t a -> t b)
(<*>) :: Applicative t  => t (a ->   b)     -> (t a -> t b)
(=<<) :: Monad t        =>   (a -> t b)     -> (t a -> t b)
```

fmap maps **(a -> b)** arbitrary functions over functors.

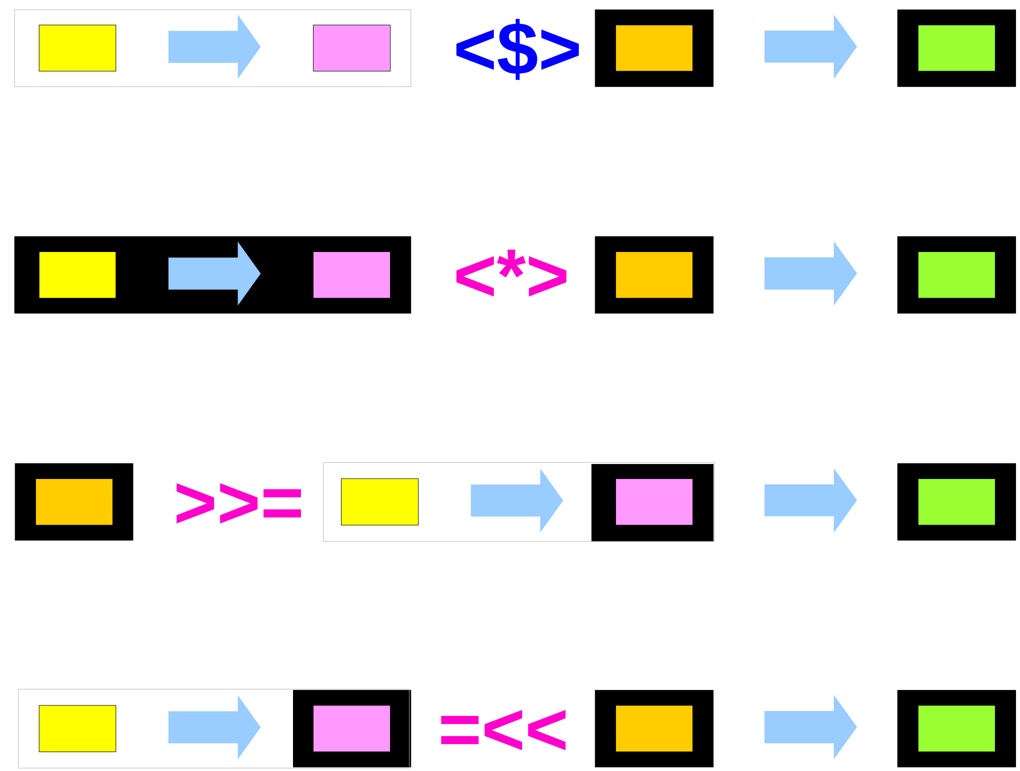
(<*>) maps **t (a -> b)** morphisms over (applicative) functors.

(=<<) maps **a -> t b** functions over (monadic) functors.

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

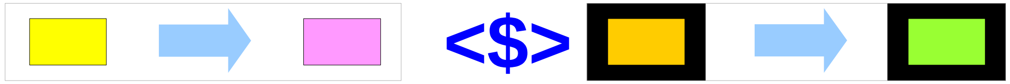
<$>, <*>, >>=, and =<< examples

```
Prelude> (*2) <$> (Just 3)
Just 6

Prelude> (Just (*2)) <*>  (Just 3)
Just 6

Prelude> (Just 3) >>= (pure . (*2))
Just 6

Prelude> (pure . (*2)) =<< (Just 3)
Just 6
```

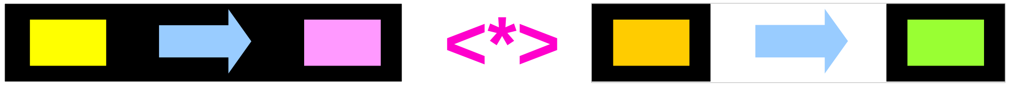


<$>, <*>, >>=, and =<< examples

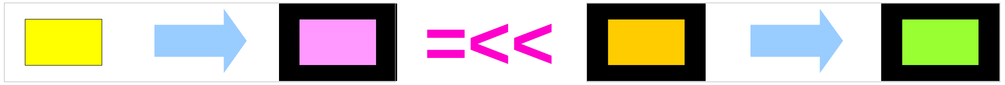
```
Prelude> (*2) <$> (Just 3)
Just 6
```




```
Prelude> (Just (*2)) <*>  (Just 3)
Just 6
```




```
Prelude> (pure . (*2)) =<< (Just 3)
Just 6
```



Sliding scale of power

The differences of **Functor**, **Applicative** and **Monad** follow from what the types of those three mapping functions allow you to do.

As you move from **fmap** to **(<*>)** and then to **(>>=)**, you gain in power, versatility and control, at the cost of guarantees about the results.

We will now slide along this scale.
While doing so, we will use the contrasting terms **values** and **context** to refer to plain values within a functor and to whatever surrounds them, respectively.

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

Changing the context

Prelude> **fmap (2*) [2,5,6]**

[4,10,12]

That can be taken as a safety guarantee

or as an unfortunate restriction,

depending on what you intend.

In any case, **(<*>)** is clearly able to change the context:

Prelude> **[(2*),(3*)] <*> [2,5,6]**

[4,10,12,6,15,18]

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

Carrying a context

The **t (a -> b)** morphism carries a context of its own,

which is combined with that of the **t** a functorial value.

(<*>), however, is subject to a more subtle restriction.

While **t (a -> b)** morphisms carry context,

within them there are plain **(a -> b)**,

which are still unable to modify the context.

That means the changes to the context **(<*>**) performs

are fully determined by the context of its arguments,

and the values have no influence over the resulting context.

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

Carrying a context

```
Prelude> (print "foo" *> pure (2*)) <*> (print "bar" *> pure 3)
"foo"
"bar"
6
Prelude> (print "foo" *> pure 2) *> (print "bar" *> pure 3)
"foo"
"bar"
3
Prelude> (print "foo" *> pure undefined) *> (print "bar" *> pure 3)
"foo"
"bar"
3
```

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

Creating a context

Thus with list **(<*>)** you know that the length of the resulting list
will be the product of the lengths of the original lists,
with IO **(<*>)** you know that all real world effect
will happen as long as the evaluation terminates, and so forth.

With Monad, however, we are in a very different game.
(>>=) takes a (**a -> t b)** function, and so it is able
to create context from values. That means a lot of flexibility:

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

Creating a context

```
Prelude> [1,2,5] >>= \x -> replicate x x
[1,2,2,5,5,5,5,5]

Prelude> [0,0,0] >>= \x -> replicate x x
[]

Prelude> return 3 >>= \x -> print $ if x < 10 then "Too small" else "OK"
"Too small"

Prelude> return 42 >>= \x -> print $ if x < 10 then "Too small" else "OK"
"OK"
```

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

Flexibility

Taking advantage of the extra flexibility, however,
might mean having less guarantees about,
for instance, whether your functions are able
to unexpectedly erase parts of a data structure for pathological inputs,
or whether the control flow in your application remains intelligible.

In some situations there might be performance implications as well,
as the complex data dependencies monadic code makes possible
might prevent useful refactorings and optimisations.

All in all, it is a good idea to only use as much power as needed for the task at hand.

If you do need the extra capabilities of Monad, go right ahead;
however, it is often worth it to check whether Applicative or Functor are sufficient.

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

Changing the context

The type of **fmap** ensures that it is impossible to use it to **change the context**,

no matter which function it is given.

In **(a -> b) -> t a -> t b**, the **(a -> b)** function

has nothing to do with the **t** context of the **t** a functorial value,

and so applying it cannot affect the context.

For that reason, if you do **fmap** **f xs** on some list **xs**

the number of elements of the list will never change.

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

Monadic binding / composition operators

```
(>>=)  :: Monad m => m a          -> (a -> m b)  -> m b
(=<<)  :: Monad m => (a -> m b) -> m a          -> m b
(>>)   :: Monad m => m a          -> m b          -> m b
(>=>)  :: Monad m => (a -> m b) -> (b -> m c)  -> (a -> m c)
(<=<)  :: Monad m => (b -> m c) -> (a -> m b)  -> (a -> m c)
```

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

Monadic binding operators (1)

```
(>>=)  :: Monad m => m a        -> (a -> m b)  -> m b
(=<<)  :: Monad m => (a -> m b) -> m a         -> m b
(>>)   :: Monad m => m a        -> m b         -> m b
(>=>)  :: Monad m => (a -> m b) -> (b -> m c)  -> (a -> m c)
(<=<)  :: Monad m => (b -> m c) -> (a -> m b)  -> (a -> m c)
```

There are a few different monadic binding operators.

The two most basic are >>= and >>,

as they can be trivially expressed in do-notation.

And as previously mentioned, >> is just a synonym for *> from the Applicative class,

so it's even easier. =<< is just >>= with the arguments reversed.

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

Monadic binding operators (2)

```
(>>=)  :: Monad m => m a          -> (a -> m b)  -> m b
(>>)   :: Monad m => m a          -> m b         -> m b
(<=<)  :: Monad m => (b -> m c) -> (a -> m b)  -> (a -> m c)

m1 >>= f = do
    x <- m1
    f x

m1 >> m2 = do
    _ <- m1
    m2

f =<< m1 = do
    x <- m1
    f x
```

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

Monadic composition operators (1)

```
(>=>)  :: Monad m => (a -> m b)  -> (b -> m c)  -> (a -> m c)
(<=<)  :: Monad m => (b -> m c)  -> (a -> m b)  -> (a -> m c)
```

In addition to these two operators,

there are also composition operators for when you have two monadic functions.

>=> pipes the result from the left side to the right side,

while <=< pipes the result the other way. In other words:

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

Monadic composition operators (2)

```
(>=>)  :: Monad m => (a -> m b) -> (b -> m c)  -> (a -> m c)
(<=<)  :: Monad m => (b -> m c) -> (a -> m b)  -> (a -> m c)

f >=> g = \x -> do
    y <- f x
    g y

g <=< f = \x -> do
    y <- f x
    g y

f >=> g = g <=< f
g >=> f = f <=< g
```

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