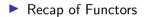
Applicatives

Sanchayan Maity

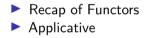






Agenda





Functor¹²



class Functor f where fmap :: (a -> b) -> f a -> f b (<\$) :: a -> f b -> f a

Functors Laws

Must preserve identity

fmap id = id

fmap (f . g) == fmap f . fmap g

Functor¹²



class Functor f where fmap :: (a -> b) -> f a -> f b (<\$) :: a -> f b -> f a

Functors Laws

Must preserve identity

fmap id = id

Must preserve composition of morphism

fmap (f . g) == fmap f . fmap g

¹Category Design Pattern ²Functor Design Pattern





For something to be a functor, it has to be a first order kind.

Applicative

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

fmap f x = pure f $\langle * \rangle$ x

Examples

```
pure (+1) <*> [1..3]
[2, 3, 4]
```

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

```
("Woo", (+1)) <*> (" Hoo!", 0)
("Woo Hoo!", 1)
```

```
(Sum 2, (+1)) <*> (Sum 0, 0)
(Sum {getSum = 2}, 1)
```

```
(Product 3, (+9)) <*> (Product 2, 8)
(Product {getProduct = 6}, 17)
```

```
(,) <$> [1, 2] <*> [3, 4]
[(1,3),(1,4),(2,3),(2,4)]
```

Use cases⁴

Person

- <\$> parseString "name" o
- <*> parseInt "age" o
- <*> parseTelephone "telephone" o

Can also be written as

```
liftA3 Person
  (parseString "name" o)
  (parseInt "age" o)
  (parseTelephone "telephone" o)
```

⁴FP Complete - Crash course to Applicative syntax

Use cases⁵

```
parsePerson :: Parser Person
parsePerson = do
  string "Name: "
  name <- takeWhile (/= 'n')
  endOfLine
  string "Age: "
  age <- decimal
  endOfLine
  pure $ Person name age</pre>
```

⁵FP Complete - Crash course to Applicative syntax

Use cases⁶

```
helper :: () -> Text -> () -> () -> Int -> () -> Person
helper () name () () age () = Person name age
```

```
parsePerson :: Parser Person
parsePerson = helper
```

- <\$> string "Name: "
- <*> takeWhile (/= 'n')
- <*> endOfLine
- <*> string "Age: "
- <*> decimal
- <*> endOfLine

⁶FP Complete - Crash course to Applicative syntax



Seeing Functor as unary lifting and Applicative as n-ary lifting

liftA0 :: Applicative f => (a)-> (f a)liftA1 :: Functorf => (a -> b)-> (f a -> f b)liftA2 :: Applicative f => (a -> b -> c)-> (f a -> f b -> f c)liftA3 :: Applicative f => (a -> b -> c -> d)-> (f a -> f b -> f c -> f dliftA4 :: Applicative f => ..

Where liftA0 = pure and liftA1 = fmap.

Monoidal functors

Remember	Monoid?
----------	---------

class Monoid m where mempty :: m mappend :: m -> m -> m (\$) :: (a -> b) -> a -> b (<\$>) :: (a -> b) -> f a -> f b (<*>) :: f (a -> b) -> f a -> f b

mappend ::fff(\$) :: $(a \rightarrow b) \rightarrow a \rightarrow b$ <*> ::f $(a \rightarrow b) \rightarrow f a \rightarrow f b$

instance Monoid a => Applicative ((,) a) where
pure x = (mempty, x)
 (u, f) <*> (v, x) = (u `mappend` v, f x)



Where are monoids



```
fmap (+1) ("blah", 0)
("blah",1)
```

```
("Woo", (+1)) <*> (" Hoo!", 0)
("Woo Hoo!", 1)
```

```
(,) <$> [1, 2] <*> [3, 4]
[(1,3),(1,4),(2,3),(2,4)]
```

```
liftA2 (,) [1, 2] [3, 4]
[(1,3),(1,4),(2,3),(2,4)]
```

Applying a function to an effectful argument

(<\$>)	::	Functor m	=>		(a	->	b)	->	m	а	->	m	b
(<*>)	::	Applicative m	=>	m	(a	->	b)	->	m	а	->	m	b
(=<<)	::	Monad m	=>		(a	->	m b)	->	m	а	->	m	b



No data dependency between f a and f b



No data dependency between f a and f b

Result of f a can't possibly influence the behaviour of f b

- No data dependency between f a and f b
- Result of f a can't possibly influence the behaviour of f b
- ► That needs something like a -> f b

-- Interchange u <*> pure y = pure (\$ y) <*> u

-- Homomorphism pure f <*> pure x = pure (f x)

```
-- Composition
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
```

```
-- Identity
pure id <*> v = v
```



Applicative laws







pure wraps up a pure value into some kind of Applicative



- pure wraps up a pure value into some kind of Applicative
- liftA2 applies a pure function to the values inside two Applicative wrapped values



- pure wraps up a pure value into some kind of Applicative
- liftA2 applies a pure function to the values inside two Applicative wrapped values
- <\$> operator version of fmap



- pure wraps up a pure value into some kind of Applicative
- liftA2 applies a pure function to the values inside two Applicative wrapped values
- <\$> operator version of fmap
- <*> apply a wrapped function to a wrapped value



- pure wraps up a pure value into some kind of Applicative
- liftA2 applies a pure function to the values inside two Applicative wrapped values
- <\$> operator version of fmap
- <*> apply a wrapped function to a wrapped value
- ▶ *>, <*

⁷FP Complete - Crash course to Applicative syntax











- Effects
- Batching and aggregation



- Effects
- Batching and aggregation
- Concurrency/Independent



- Effects
- Batching and aggregation
- Concurrency/Independent
 - Parsing context free grammar



- Effects
- Batching and aggregation
- Concurrency/Independent
 - Parsing context free grammar
 - Exploring all branches of computation (see Alternative)



- Effects
- Batching and aggregation
- Concurrency/Independent
 - Parsing context free grammar
 - Exploring all branches of computation (see Alternative)
- Monads



- Effects
- Batching and aggregation
- Concurrency/Independent
 - Parsing context free grammar
 - Exploring all branches of computation (see Alternative)
- Monads
 - Effects



- Effects
- Batching and aggregation
- Concurrency/Independent
 - Parsing context free grammar
 - Exploring all branches of computation (see Alternative)
- Monads
 - Effects
 - Composition



- Effects
- Batching and aggregation
- Concurrency/Independent
 - Parsing context free grammar
 - Exploring all branches of computation (see Alternative)
- Monads
 - Effects
 - Composition
 - Sequence/Dependent



- Effects
- Batching and aggregation
- Concurrency/Independent
 - Parsing context free grammar
 - Exploring all branches of computation (see Alternative)
- Monads
 - Effects
 - Composition
 - Sequence/Dependent
 - Parsing context sensitive grammar



- Effects
- Batching and aggregation
- Concurrency/Independent
 - Parsing context free grammar
 - Exploring all branches of computation (see Alternative)
- Monads
 - Effects
 - Composition
 - Sequence/Dependent
 - Parsing context sensitive grammar
 - Branching on previous results



Weaker than monads but thus also more common

- Weaker than monads but thus also more common
- Lends itself to optimisation (See Facebook's Haxl project)

- Weaker than monads but thus also more common
- Lends itself to optimisation (See Facebook's Haxl project)
- Always opt for the least powerful mechanism to get things done

- Weaker than monads but thus also more common
- Lends itself to optimisation (See Facebook's Haxl project)
- Always opt for the least powerful mechanism to get things done
- ▶ No dependency issues or branching? just use applicative



Applicative Programming with Effects



- Applicative Programming with Effects
- optparse-applicative

- Applicative Programming with Effects
- optparse-applicative
- Control Applicative

- Applicative Programming with Effects
- optparse-applicative
- Control Applicative
- Applicative functors for fun & parsing









Email: sanchayan@sanchayanmaity.net



- Email: sanchayan@sanchayanmaity.net
- Mastodon: sanchayanmaity.com

Reach out on

- Email: sanchayan@sanchayanmaity.net
- Mastodon: sanchayanmaity.com
- Telegram: t.me/SanchayanMaity

Reach out on

- Email: sanchayan@sanchayanmaity.net
- Mastodon: sanchayanmaity.com
- Telegram: t.me/SanchayanMaity
- Blog: sanchayanmaity.net