Applicatives

Sanchayan Maity

Agenda

- Recap of Functors
- Applicative

Functor[1,2]

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

Functors Laws

- Must preserve identity

```haskell
fmap id = id
```

- Must preserve composition of morphism

```haskell
fmap (f . g)  ==  fmap f . fmap g
```

[1] Category Design Pattern
[2] Functor Design Pattern

Higher order kinds[3]

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

[3] Haskell's Kind System

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 <*> 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 cases[4]

```
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)
```

[4] FP Complete - Crash course to Applicative syntax

Use cases[5]

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

[5] FP Complete - Crash course to Applicative syntax

Use cases[6]

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

parsePerson :: Parser Person
parsePerson = helper
  <$> string "Name: "
  <*> takeWhile (/= 'n')
  <*> endOfLine
  <*> string "Age: "
  <*> decimal
  <*> endOfLine
```

[6] FP Complete - Crash course to Applicative syntax

Lifting

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

```
liftA0 :: Applicative f => (a)                     -> (f a)
liftA1 :: Functor     f => (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 d)
liftA4 :: 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 ::     f          f      f
($) ::       (a -> b) ->   a ->   b
<*> ::     f (a -> b) -> f a -> f b
```

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

Function apply

- Applying a function to an `effectful` argument

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

Contrasts with monad

- 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`

Applicative laws

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

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

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

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

Operators[7]

- `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
- `*>`, `<*`

[7] FP Complete - Crash course to Applicative syntax

Applicative vs monads

- Applicative
  - 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 but better

- 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

Resources

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

Questions

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