An Introduction to Applicative Functors

Bocheng Zhou

What Is an Applicative Functor?

- An Applicative functor is a Monoid in the category of endofunctors, what's the problem?

- WAT?!

Functions in Haskell

- Functions in Haskell are first-order citizens
- Functions in Haskell are curried by default
  - f :: a -> b -> c is the curried form of g :: (a, b) -> c
  - f = curry g, g = uncurry f
- One type declaration, multiple interpretations
  - f :: a->b->c
  - f :: a->(b->c)
  - f :: (a->b)->c
  - Use parentheses when necessary:
    - >>= :: Monad m => m a -> (a -> m b) -> m b

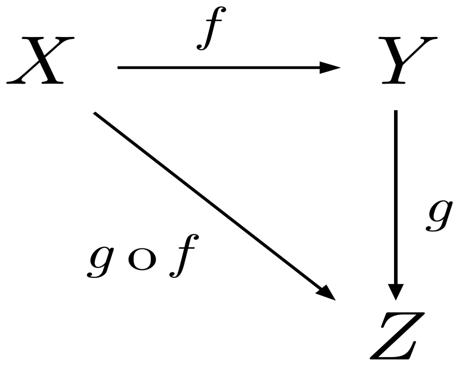
Functors

- A **functor** is a type of mapping between categories, which is applied in category theory.

- What the heck is category theory?

Category Theory 101

- A category is, in essence, a simple collection. It has three components:
    - A collection of **objects**
    - A collection of **morphisms**
    - A notion of **composition** of these morphisms

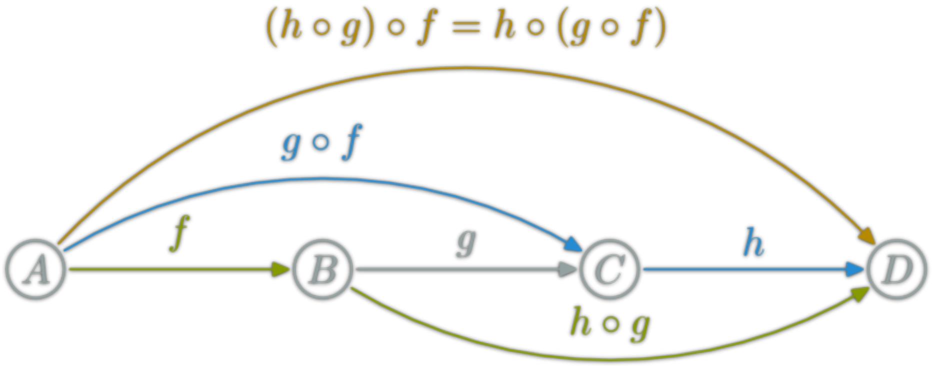


- Objects: X, Y, Z
- Morphisms: f :: X->Y, g :: Y->Z
- Composition: g . f :: X->Z

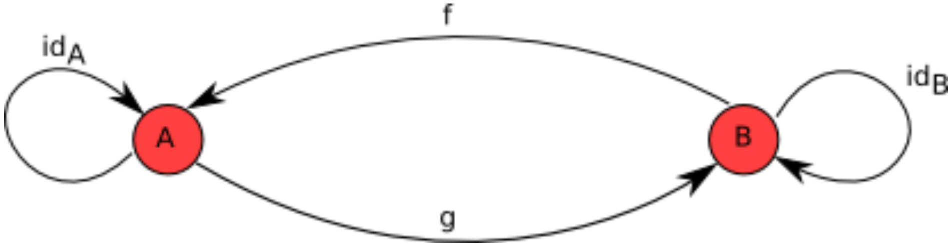
Category Theory 101

- Category laws:

$$f \circ (g \circ h) = (f \circ g) \circ h$$

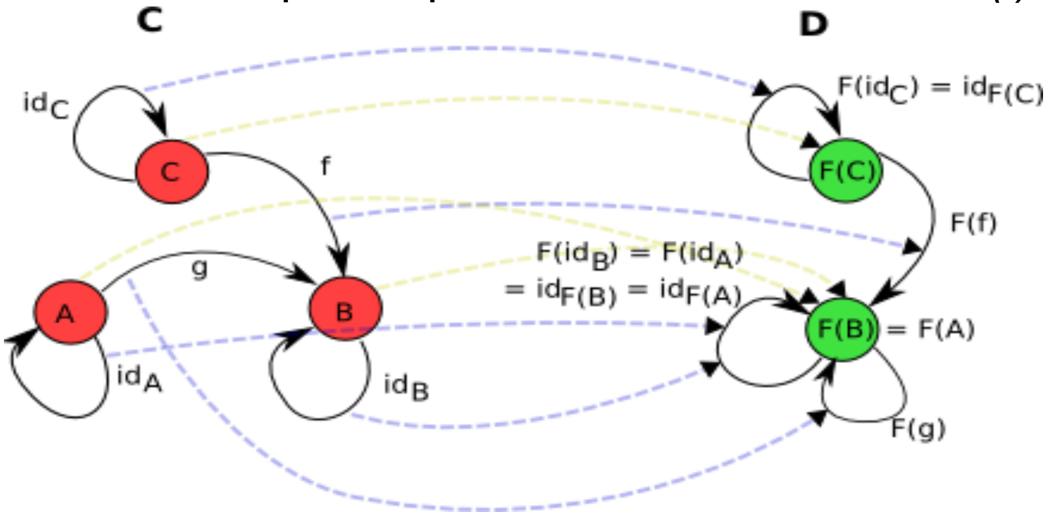


$$g \circ id_A = id_B \circ g = g$$



Functors Revisited

- Recall that a **functor** is a type of mapping between categories.
- Given categories ***C*** and ***D***, a functor ***F :: C -> D***
  - Maps any object *A* in ***C*** to *F(A)* in ***D***
  - Maps morphisms *f :: A -> B* in ***C*** to *F(f) :: F(A) -> F(B)* in ***D***



Functors in Haskell

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

- Recall that a functor maps morphisms *f :: A -> B* in ***C*** to *F(f) :: F(A) -> F(B)* in ***D***
- morphisms ~ functions
- ***C*** ~ category of primitive data types like Integer, Char, etc.
- ***D*** ~ category of “functorized types” like Maybe Integer, Maybe Chat, etc.
- fmap actually takes as parameter a function(g :: a -> b) , and returns a function(g’ :: f a -> f b)

Endofunctors

- A **functor** is a type of mapping between 2 categories.
- What if the 2 categories are the actually the same category? You got endofunctors
- Functors in Haskell are actually endofunctors

  We have a category **Hask**, which treats ALL Haskell types as objects and Haskell functions as morphisms and uses (.) for composition

Applicative Functors

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

  -- fmap
  <$>  ::  (a -> b) -> f a -> f b
```

Function-in-the-box

- Applicative functors are another mechanism for dealing with programming with effects(values wrapped in a context)
- Applicative functors are more powerful than functors because they are able to deal with functions in a context
- But how do functions get into a “box” in the first place?

Function-in-the-box

- How do functions get into a context?
  - Just use pure :: a -> f a
  - Use fmap:

```
fmap (+) [1]  or  (+) <$> [1]
>> [(+ 1)]

(+) <$> [1, 2] <*> [3, 4]
>> [4, 5, 5, 6]
```

A Use Case

```
data User = User { firstName :: Text,
                   LastName :: Text,
                   Email :: Text}
buildUser :: Profile -> Maybe User
```

```
buildUser p = User
  <$> lookup “first_name” p
  <*>  lookup “last_name” p
  <*>  lookup “email” p
```

```
buildUser p = do
  fn <- lookup “first_name” p
  ln <- lookup “last_name” p
  em <- lookup “email” p
  return $ User fn ln em
```

Why Applicatives?

Q: We already got this Monad dude, who is, like, super awesome. Why do we need to hire you for this task?

A: I’m flexible on salary, and I get shit done faster

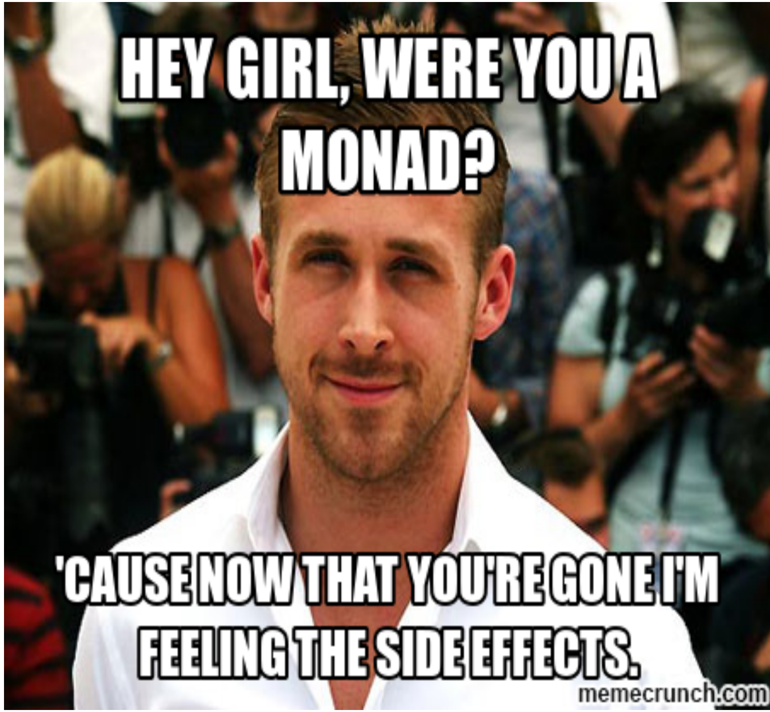
Q: Okay, what’s your name again?

A: Applicative Functor

Q: Geez, that’s a mouthful!

Applicatives vs. Monads

- Monads are about…
  - Effects
  - Composition
  - Sequence/Dependency
    - parsing context-sensitive grammar
    - branching on previous results
- Applicatives are about…
  - (less severe)Effects
  - Batching and aggregation
  - Concurrency/Independency
    - parsing context-free grammar
    - exploring all branches of computation



Disaster Averted (or Not)

- ```
  miffy :: Monad m => m Bool -> m a -> m a -> m a
  miffy mb mt me = do
      b <- mb
      if b then mt else me
  >> miffy (Just True) (Just “Yay!”) Nothing = Just “Yay!”
  ```

- ```
  iffy :: Applicative f => f Bool -> f a -> f a -> f a
  iffy fb ft fe = cond <$> fb <*> ft <*> fe   where
      cond b t e = if b then t else e
  >> iffy (Just True) (Just “Yay!”) Nothing = Nothing
  ```

Should It Always Fail Early?

- Monads have this inherent property that they can branch on the results of previous computations, which implies they always fail early(short-circuited)
- What if you want to design a signup page for your website?
- What if you actually don't really care whether the computation should fail early or not?

Weaker But Sometimes Better

- Applicatives are weaker than Monads, which also means they are more common than Monads
- Applicative code is usually cleaner and shorter than its monadic counterpart, and lends itself to optimization
  - Facebook’s Haxl provides a DSL that expose the monadic interfaces and converts them to applicatives when necessary
- Use the least powerful mechanism to get things done
- When there’s no dependency issues or branching, just use applicatives

Like Father, Like Son

- All monads are applicatives, but not all applicatives are monads
  - ZipList
- Applicative is actually a superclass of monad
- Fun fact: Actually applicatives were discovered **later** than monads
- Due to historical reasons, applicative is NOT a superclass of monad in Haskell yet (but it soon will be)

Applicative => Monad Proposal (AMP)

- Applicative becomes a superclass of Monad
- Why?
  - lack of unity means there is a lot of duplication of API:
    - liftA :: (Applicative f) => (a -> b) -> f a -> f b
    - liftM :: (Monad m) => (a -> b) -> m a -> m b
  - pure = return, <*> = ap
    - ap mf ma = do

```
f <- mf
a <- ma
return $ f a
```

  - Enforce the use of the least restrictive functions

So an Applicative Functor Is...

- A Monoid in the category of endofunctors. That’s it.
- Dammit! What the heck is a Monoid?
  - ```
    class Monoid m where
      mempty :: m
      mappend :: m -> m -> m
    ```
  - ```
    instance Monoid [a] where
      mempty = []
      la mappend lb = (++) <$>  la <*> lb
    ```

Resources

- http://learnyouahaskell.com/functors-applicative-functors-and-monoids
- Applicative programming with effects
- Applicative Functors: Hidden in plain view
- Haskell/Category Theory
- Introduction to functional programming
- Beginning Haskell: A Project-Based Approach
- Haskell Ryan Gosling