APPLICATIVE FUNCTORS

Sebastian Cielemęcki

June 13, 2015

FUNCTORS

FUNCTOR CLASS

Functor is a class for types which can be mapped over. It lets us generalize the well-known **map** function.

The class is defined in **Prelude** as follows:

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

Where **f** represents a type of kind *** -> ***

THE FUNCTOR LAWS

A proper instance of functor shoud define **fmap** in such a way that it satisfies the following laws:

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

The laws are quite intuitive. Mapping the identity function over a functor should not change it in any way, and it should not be relevant whether we map two functions composed or one after the other.

We say that **fmap**, when applied to a function working on given types, lifts it so that it operates on functors over the same types.

FUNCTOR EXAMPLES: LIST AND MAYBE

Lists are naturally functors:

```
instance Functor [] where
  fmap = map
```

Making **Maybe** a functor is quite obvious too:

```
instance Functor Maybe where
  fmap f (Just x) = Just $ f x
  fmap _ Nothing = Nothing
```

FUNCTOR EXAMPLES: EITHER A

For the **Either** datatype, wich takes two type parameters, the functor instance is defined as below:

```haskell
data Either a b = Left a | Right b

instance Functor (Either a) where
  fmap f (Right x) = Right $ f x
  fmap f (Left x) = Left x
```

We map over the second parameter. **Left** is analoguous to **Nothing** in **Maybe**.

FUNCTOR EXAMPLES: IO

I/O actions can be mapped as well:

```
instance Functor IO where
  fmap f action = do
      result <- action
      return $ f result
```

For example, one might want to change input value directly before binding:

```
main = do
  line <- fmap ("You said" ++) getLine
  print line
```

FUNCTOR EXAMPLES: (->) R

Functions are functors too. Consider the type **r -> a**. It can be written as: **(->) r a**. Then **(->)** is simply a type constructor of kind *** -> * -> ***! Thus we define:

```haskell
instance Functor ((->) r) where
  fmap f g = (\x -> f (g x))
  -- fmap = (.)
```

The mapping function changes a value of type **a** - which is the result of the mapped function. In other words, **fmap** is function compisition in this context.

FUNCTORS - WHAT THEY LACK

So far, we can map functions over functors to get modified ones but what if we want to use functions like `(+)` to combine two functors?
We need additional tools!

APPLICATIVE FUNCTORS

Functors and multiparameter functions

Applicative functors are beefed up functors. Consider the code:

```
plusTwo :: Maybe (Double -> Double)
plusTwo = fmap (+) (Just 2.5) -- Just ((+) 2.5)
```

The **(+)** operator is partially applied to content of the **Just** value. If we want to 'apply' it to another **Just** value, it is clear we need a function of a type:

```
Maybe (Double -> Double) -> Maybe Double -> Maybe Double
```

Applicative introduces special **ap(ply)** operator for this purpose.

Applicative class

The **Applicative** class is defined as follows:

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

The **Applicative** allowes us to embed pure computations into pure fragments of an effectful world in convenient style, like

```
pure f <*> u1 <*> u2 <*> ... <*> uk
```

Applicative functors - laws

We require the following laws for applicative functors:

```
pure id <*> u  ==  u -- identity
-- composition
pure (.) <*> u <*> v <*> w  ==  u <*> (v <*> w)
pure f <*> pure x = pure (f x) -- homomorphism
u <*> pure y = pure ($ y) <*> u -- interchange
```

Applicative functors - some advantages

Applicative functors have more structure than functors, but less than monads.

- We often don't need to use monadic style. Code using only the Applicative interface is more general than code using the Monad interface.
- With Applicative programming has a more applicative/functional feel, whereas monadic style encourages more sequential and imperative style

Applicative functors - fmap operator

Control.Applicative provides **fmap operator**:

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

So these two lines of code are equivalent:

```
pure f <*> x <*> y <*> ...
f <$> x <*> y <*> ...
```

Another useful function is liftA2, which applies a function between two applicatives, hiding the applicative style:

```
liftA2 :: (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b
```

For example:

```
ghci> liftA2 (:) (Just 2) (Just [1])
Just [2,1]
```

Applicative functors - IO

Making an applicative instance for **IO** is easy:

```
instance Applicative IO where
  pure = return
  a <*> b = do
      f <- a
      x <- b
      return (f x)
```

E.g. one can perform a sequence of IO operations and process them with a function

```
main = do
  print ”Type Your name:”
  name <- ((++).(++” ”)) <$> getLine <*>  getLine
  print $ ”Your name is ” ++ name
```

Applicative functors - sequencing (monadic)

Consider the following example of sequencing IO operations:

```
sequence :: [IO a ] -> IO [a]
sequence [] = return []
sequence (c : cs) = do
x <- c
xs <- sequence cs
return (x : xs)
```

We collect values of effectful computations, but we don't use them until (:) is applied.

Applicative functors - sequencing (applicative)

With applicative style we avoid the need for names of the intermediate values:

```
sequence :: [IO a] -> IO [a]
sequence [] = pure []
sequence (c : cs) = pure (:) <*> c <*> (sequence cs)
```

E.g. reading multiple values:

```
main = do
  print "How many values do You want to input?"
  k <- getLine
  print $ "Type " ++ k ++ " values"
  seqs <- sequence (replicate (read k) getLine)
  print "The values are:"
  print seqs
```

For **(->) r)**, we define **Applicative** instance as follows:

```
instance Applicative ((->) r) where
  pure x = (\_ -> x)
  f <*> g = \x -> f x (g x)
```

pure should take a pure value and put it in minimum context that still yields that value - in this case it is a function, which always returns the same value **x**. The **ap(ply)** here is more tricky, but an example makes it clearer:

```
ghci> let f = (+) <$> (+5) <*> (*100)
ghci> :t f
f :: Integer -> Integer
ghci> f 5
510
```

Applicative functors - evaluator

Let's take a look at a simple evaluator:

```
data Exp v = Var v
           | Val Int
           | Add (Exp v) (Exp v)
eval :: Exp v -> Env v -> Int
eval (Var x ) e = fetch x e
eval (Val i) e = i
eval (Add p q) e = eval p e + eval q e
```

Threading environment explicitliy makes this code a bit messy, so we can define special functions to avoid it

Applicative functors - evaluator (applicative-like)

```haskell
eval :: Exp v -> Env v -> Int
eval (Var x) = fetch x
eval (Val i) = K i
eval (Add p q) = K (+) 'S' (eval p) 'S' (eval q)

K :: a -> env -> a
K x e = x

S :: (env -> a -> b) -> (env -> a) -> (env -> b)
S ef es e = (ef e) (es e)
```

Fairly applicative style, but...

Applicative functors - evaluator - applicative style

S and K are actually the same as monadic **return** and **ap**, which correspond to **pure** and **<*>** in **Applicative**.

Note that the type of S can be written as

```
S :: ((->) env (a -> b)) -> ((->) env a) -> ((->) env b)
```

Which is the exact type of **<*>**. We can rewrite **eval** using real **Applicative** syntax:

```
eval :: Exp v -> Env v -> Int
eval (Var x) = fetch x
eval (Val i) = pure i
eval (Add p q) = pure (+) <*> (eval p) <*> (eval q)
```

When we now apply evaluation to environment, all sub-evaluations will be feeded with it and the results will eventually be combined.

Applicative functors - transpose - with zipWith

If we represent matrices by list of lists, we can define transposition as follows:

```
transpose :: [[a]] -> [[a]]
transpose [] = repeat []
transpose (xs : xss) = zipWith (:) xs (transpose xss)
```

We simply zip all the rows and obtain columns (which correspond to rows in the transposed matrix)

Applicative functors - transpose (applicative-like)

The binary **zipWith** function can be generalized like this:

```
zipN :: (a1 ->...-> an -> b) -> [a1] ->...-> [an] -> [b]
zipN f xs1...xsn = repeat f 'zapp' xs1 'zapp'...'zapp' xsn

zapp :: [a -> b] -> [a] -> [b]
zapp (f : fs) (x : xs) = f x : zapp fs xs
zapp _ _ = []
```

With **zapp**, we can do tansposition as follows:

```
transp :: [[a]] -> [[a]]
transp [] = repeat []
transp (xs:xss) = repeat (:) 'zapp' xs 'zapp' transp xss
```

With infinite list of conses and recursion we can "zapp" arbitrary number of lists of different length!

Applicative functors - transpose - applicative style

Finally, the definition of **transp** in applicative style:

```
transp :: [[a]] -> [[a]]
transp [ ] = pure []
transp (xs:xss) = pure (:) <*> xs <*> (transp xss)
```

Here, **pure** == **repeat** and **<*>** == **zapp**.

In reality, **Applicative []** is implemented in a different way (we associate lists with nondeterministic computations). Applicatives like **ZipList** would be more suitable in this case.

Final remarks

- Monads are more powerful than functors. The bind operator gives a possibility to choose next computation depending on the value returned from the previous one
- Using functions from Control.Monad, we can define pure and <*> as return and ap. Thus we can make a monad an instance of Applicative
- It was a historical accident that Applicative is not a superclass of Monad in Haskell. This is simply because Monads where discovered and popularized earlier. This will change in GHC 7.10 under Applicative-Monad Proposal - any instance of Monad will also have to be an instance of Applicative

Summary

A few useful links:

staff.city.ac.uk/~ross/papers/Applicative.pdf

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

en.wikibooks.org/wiki/Haskell/Applicative_Functors

wiki.haskell.org/Applicative_functor

Questions?