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:

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

Where `f` represents a type of kind `* -> *`
A proper instance of functor should define `fmap` in such a way that it satisfies the following laws:

\[
\begin{align*}
\text{fmap id} & \quad = \quad \text{id} \\
\text{fmap (f . g)} & \quad = \quad \text{fmap f . fmap g}
\end{align*}
\]

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.
Lists are naturally functors:

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

Making `Maybe` a functor is quite obvious too:

```haskell
instance Functor Maybe where
  fmap f (Just x) = Just $ f x
  fmap _ Nothing = Nothing
```
For the `Either` datatype, which 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 analogous to `Nothing` in `Maybe`.
I/O actions can be mapped as well:

```haskell
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:

```haskell
main = do
    line <- fmap ("You said" ++) getLine
    print line
```
Functions are functors too. Consider the type \( r \to a \). It can be written as: \((\to)\ r\ a\). Then \((\to)\) is simply a type constructor of kind \( * \to * \to * \)!
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, \texttt{fmap} is function composition in this context.
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
Applicative functors are beefed up functors. Consider the code:

```haskell
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.
The **Applicative** class is defined as follows:

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

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

```haskell
pure f <*> u1 <*> u2 <*> ... <*> uk
```
We require the following laws for applicative functors:

\[
\begin{align*}
\text{pure id } \langle \star > u &= u \quad \text{-- identity} \\
\text{-- composition} \\
\text{pure (.) } \langle \star > u \langle \star > v \langle \star > w &= u \langle \star > (v \langle \star > w) \\
\text{pure f } \langle \star > \text{ pure x} &= \text{ pure } (f \ x) \quad \text{-- homomorphism} \\
u \langle \star > \text{ pure y} &= \text{ pure } (\$ y) \langle \star > u \quad \text{-- interchange}
\end{align*}
\]
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
Control.Applicative provides `fmap` operator:

\[
(\langle\rangle) : (\text{Functor } f) \Rightarrow (a \to b) \to f a \to f b
\]

\[
f \langle\rangle x = \text{fmap } f x
\]

So these two lines of code are equivalent:

```haskell
pure f <*> x <*> y <*> ...
f <$> x <*> y <*> ...
```
Another useful function is \texttt{liftA2}, which applies a function between two applicatives, hiding the applicative style:

\[
\text{liftA2} :: (a \rightarrow b \rightarrow c) \rightarrow f\ a \rightarrow f\ b \rightarrow f\ c \\
\text{liftA2}\ f\ a\ b = f\ \langle\$\rangle\ a\ \langle\ast\rangle\ b
\]

For example:

\[
\text{ghci}\> \text{liftA2}\ (\:)(\text{Just\ 2})(\text{Just\ [1]}) \\
\text{Just\ [2,1]}
\]
Making an applicative instance for IO is easy:

```haskell
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

```haskell
main = do
  print "Type Your name:"
nname <- ((++)((++)"
  putStrLn name
  putStrLn "Your name is " ++ name
```
Consider the following example of sequencing IO operations:

\[
\text{sequence} :: [\text{IO } a] \rightarrow \text{IO } [a]
\]

\[
\text{sequence } [] = \text{return } []
\]

\[
\text{sequence } (c : cs) = \text{do}
\]

\[
x <- c
\]

\[
xs <- \text{sequence } cs
\]

\[
\text{return } (x : xs)
\]

We collect values of effectful computations, but we don’t use them until (:) is applied.
With applicative style we avoid the need for names of the intermediate values:

\[
\text{sequence} :: [\text{IO } a] \rightarrow \text{IO } [a]
\]

\[
\text{sequence } [] = \text{pure } []
\]

\[
\text{sequence } (c : cs) = \text{pure } (:) \triangleright\triangleright c \triangleright\triangleright (\text{sequence } cs)
\]

E.g. reading multiple values:

```haskell
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 $(-\rightarrow)\ r$, we define **Applicative** instance as follows:

```haskell
instance Applicative ((-\rightarrow)\ r) where
    pure x = (__ \rightarrow x)
    f <*> g = \x \rightarrow 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(ly)** here is more tricky, but an example makes it clearer:

```haskell
ghci> let f = (+) <$> (+5) <*> (*100)
ghci> :t f
f :: Integer \rightarrow Integer
ghci> f 5
510
```
Let’s take a look at a simple evaluator:

```haskell
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 explicitly makes this code a bit messy, so we can define special functions to avoid it.
Applicative functors - evaluator (applicative-like)

eval :: \text{Exp v} \rightarrow \text{Env v} \rightarrow \text{Int}
eval (\text{Var} \; x) = \text{fetch} \; x
\text{eval (Val} \; i) = \text{K} \; i
\text{eval (Add} \; p \; q) = \text{K} \; (\text{K}) \; 'S' \; (\text{eval} \; p) \; 'S' \; (\text{eval} \; q)

\text{K} :: a \rightarrow \text{env} \rightarrow a
\text{K} \; x \; e = x

\text{S} :: (\text{env} \rightarrow a \rightarrow b) \rightarrow (\text{env} \rightarrow a) \rightarrow (\text{env} \rightarrow b)
\text{S} \; \text{ef} \; \text{es} \; e = (\text{ef} \; e) \; (\text{es} \; e)

Fairly applicative style, but...
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

$$S :: (( -> ) \text{env} \ (a -> b)) \rightarrow (( -> ) \text{env} \ a) \rightarrow (( -> ) \text{env} \ b)$$

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

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

\[
\text{transpose} :: [[a]] \rightarrow [[a]]
\]
\[
\text{transpose} [] = \text{repeat} []
\]
\[
\text{transpose} (xs : xss) = \text{zipWith} (:) xs (\text{transpose} xss)
\]

We simply zip all the rows and obtain columns (which correspond to rows in the transposed matrix)
The binary `zipWith` function can be generalized like this:

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

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

With `zapp`, we can do transposition as follows:

```haskell
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!
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 were 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.
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?