Applicative programming with effects

Luis Eduardo Bueso de Barrio
lbueso@acm.org

April 11, 2018

ACM

Contents

Why would you need Applicative functors?

TODO

Introduction

Introduction

This is the story of a pattern that popped up time and again in our daily work, programming in Haskell, until the temptation to abstract it became irresistible. – Simon Peyton Jones

Sequencing commands

Sequencing commands and collect the results is a very common pattern.

```
sequence :: (Monad m, Traversable t) => t (m a) -> m (t a)
sequence :: (Monad m) => [m a] -> m [a]
sequence :: [IO a] -> IO [a]
```

Let's take a look at some implementations for this pattern.

Sequencing commands

If we implement this sequence operator for the `IO` we will get something like this using do-notation:

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

Sequencing commands

We will rewrite it using the `ap` function from the `Monad` library.

```
ap :: Monad m => m (a -> b) -> m a -> m b
ap mf mx = do
  f <- mf
  x <- mx
  return (f x)
```

So we get something like this:

```
sequence :: [IO a] -> IO [a]
sequence []     = return []
sequence (c:cs) = return (:) `ap` c `ap` sequence cs
```

Transposing 'matrices'

If we model matrices as lists of lists, we can implement transposition as:

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

The `repeat` function is just:

```
repeat :: a -> [a]
repeat x = x : repeat x
```

Transposing 'matrices'

If we implement some zapp function like this:

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

We can rewrite transpose as:

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

Comparison between the functions

So let’s take a look at their types:

```
sequence :: [IO a] -> IO [a]

transpose :: [[a]] -> [[a]]
```

and their implementation:

```
sequence []     = return []
sequence (c:cs) = return (:) `ap` c `ap` sequence cs

transpose []       = repeat []
transpose (xs:xss) = repeat (:) `zapp` xs `zapp` transpose xss
```

Comparison between the functions

We can see a relation in the “application” function:

```
ap :: Monad m => m (a -> b) -> m a -> m b
zapp ::            [a -> b] -> [a] -> [b]
```

So we can create an abstraction of this operation called “apply” which will be the function:

```
(<*>) :: f (a -> b) -> f a -> f b
```

All this patterns will lead us to the creation of the `Applicative` type class.

The Applicative class

The Applicative class

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

We can check that applicative functors are functors by implementing `fmap`:

```
(<$>) :: Applicative f => (a -> b) -> f a -> f b
f <$> u = pure f <*> u
```

The Applicative class

Any expression can be transformed to “canonical form”

```
pure f <*> u_1 <*> ... <*> u_n
```

We can transform any monad into an applicative just with:

```
pure  = return
(<*>) = ap
```

The Applicative class

An Applicative instance for IO would be:

```
instance Applicative IO where
  pure  = return
  (<*>) = ap
```

So our sequence function becomes:

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

The Applicative class

The same way we have various instances of Monoid for a type, we have various of Applicative.

In our list example we need the Applicative ZipList instance:

```
instance Applicative [] where
  pure  = repeat
  (<*>) = zapp
```

So our transpose function becomes:

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

It must be notice that repeat are not the return and ap equivalents of any Monad.

Applicative laws

- Identity
  `pure id <*> u = u`
- Composition
  `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
- Homomorphism
  `pure f <*> pure x = pure (f x)`
- Interchange

QuickCheck

`QuickCheck` can help us writing correct `Applicative` functors using `Checkers`

In order to use it we must implement two instances:

```haskell
class Arbitrary a where
  arbitrary :: Gen a

class EqProp a where
  (=-=) :: a -> a -> Property
```

And to make the test we just need to:

```haskell
trigger = undefined :: Pair (String, Int, Bool)

main :: IO ()
main = do
  quickBatch $ applicative trigger
```

```
applicative:
  identity:     +++ OK, passed 500 tests.
  composition:  +++ OK, passed 500 tests.
  homomorphism: +++ OK, passed 500 tests.
  interchange:  +++ OK, passed 500 tests.
  functor:      +++ OK, passed 500 tests.
```

Traversing data structures

Traversing data structures

Let's take a look at `sequence` and `transpose` types again:

```haskell
sequence :: [IO a] -> IO [a]
```

```haskell
transpose :: [[a]] -> [[a]]
```

This common pattern is called *applicative distributor* for lists:

```haskell
dist :: Applicative => [f a] -> f [a]
dist []     = pure []
dist (v:vs) = pure (:) <*> v <*> (dist vs)
```

This is again in *applicative style*.

Traversing data structures

This is usually used with `map`, for example here we use it for "failure propagation":

```
flakyMap :: (a -> Maybe b) -> [a] -> Maybe [b]
flakyMap f ss = dist (fmap f ss)
```

We can implement a generalization of this:

```
traverse :: Applicative f => (a -> f b) -> [a] -> f [b]
traverse f []     = pure []
traverse f (x:xs) = pure (:) <*> (f x) <*> (traverse f xs)
```

Traversing data structures

This pattern is very useful so we can abstract in a type class:

```
class Traversable t where
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  dist     :: Applicative f => t (f a) -> f (t b)
  dist     = traverse id
```

If we implement an `Id` type like:

```
newtype Id a = An { an :: a }
```

We can implement `fmap` very easily:

```
fmap f = an . traverse (An . f)
```

Traversing data structures

Another interesting Traversable instance is:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)

instance Traversable Tree where
  traverse f Leaf         = pure Leaf
  traverse f (Node l x r) = pure Node <*> (traverse f l) <*> (f
```

In the latest GHC we need Functor, Applicative and also Foldable in order to have a Traversable instance.

Monoids are phantom Applicative functors

Monoids are phantom Applicative functors

As we know the `Monoid` is just:

```
class Monoid o where
  mempty :: o
  (<>)   :: o -> o -> o -- also called "mappend"
```

Monoids are very useful in functional programming, we have them in numeric types, lists, booleans... And every monoid also induces an applicative functor!

Monoids are phantom Applicative functors

If we define:

```
newtype Accy o a = Acc { acc :: o }
```

We can implement an applicative functor that accumulates computations:

```
instance Monoid o => Applicative (Accy o) where
  pure _              = Acc mempty
  (Acc x) <*> (Acc y) = Acc (x <> y)
```

Monoids are phantom Applicative functors

This accumulation can be seen as a special kind of traversal:

```
accumulate :: (Traversable t, Monoid o) => (a -> o) -> t a -> o
accumulate f = acc . traverse (Acc . f)

reduce :: (Traversable t, Monoid o) => t o -> o
reduce = accumulate id
```

With this monoid instance we get operations like `flatten` and `concat` nearly for free!

```
flatten :: Tree a -> [a]     -- our Tree data type will need a
flatten = accumulate (:[])   -- Monoid and Traversable instance

concat :: [[a]] -> [a]       -- the same for our list
concat = reduce
```

Applicative versus Monad

Applicative versus Monad

TODO

Composing applicative functors

TODO

Accumulating exceptions

If we define a data type to model exceptions:

```
data Except err a = OK a | Failed err
```

If we use a `Monad` instance, it will abort once the computation fails, however we can define the `Applicative` instance as:

```
instance Monoid err => Applicative (Except err) where
  pure                              = OK
  (OK f) <*> (OK x)                 = OK (f x)
  (OK f) <*> (Failed err)           = Failed err
  (Failed err) <*> (OK x)           = Failed err
  (Failed err) <*> (Failed err')    = Failed (err <> err')
```

With this instance we can collect errors using the list `Monoid`