Functional programming, Seminar No. 5

Danya Rogozin
Lomonosov Moscow State University,
Serokell OÜ
Higher School of Economics
The Department of Computer Science

Intro

On the previous seminar, we

- introduced data types, new types, records, and type synonyms
- told about right and left folds
- discussed lazy evaluation enforcing

Intro

On the previous seminar, we

- introduced data types, new types, records, and type synonyms
- told about right and left folds
- discussed lazy evaluation enforcing

Today we

- motivate and introduce functors
- generalise left and right folds with the type class `Foldable`

Functor

Motivation

- Let us take a look at these functions

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

mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe _ Nothing = Nothing
mapMaybe f (Just x) = Just (f x)
```

- One has the same pattern, a unary function carries through a computational context (lists and optional values)
- This idea has a generalisation introducing functor, a Haskell counterpart of categorical functor

Here comes the Functor

Instances of the type class `Functor` are type constructors that has kind `* -> *`:

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

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

instance Functor [a] where
  map _ [] = []
  map f (x : xs) = f x : map f xs
```

The full definition of a functor

```
class Functor (f :: * -> *) where
  fmap        :: (a -> b) -> f a -> f b
  (<$)        :: a -> f b -> f a
  (<$)        =  fmap . const

infixl 4 <$>, <$

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

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

The `Functor` instances for two-parametric types

Let us take a look at the `Functor` for type constructors that have kind `* -> * -> *`

```
instance Functor ((,) a) where
  fmap f (x,y) = (x, f y)

instance Functor ((->) r) where
  fmap = (.)

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

The `Functor` laws

Functor has the following axioms:

```
fmap id fx = fx
```

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

The `Functor` laws. Example

Let us check that the list data type is a Functor. In other words, let us check that the `Functor` instance satisfies required conditions.

```
fmap id [] = map id [] = []
fmap id (x : xs) =
  id x : fmap id xs =
  x : fmap id xs =        -- Induction hypothesis
  x : xs

fmap (f . g) [] = []
fmap (f . g) (x : xs) =
  (f . g) x : fmap (f . g) xs =    -- Induction hypothesis
  (f . g) x : (fmap f . fmap g) xs =
  f (g x) : fmap f (fmap g xs)
```

Monoid

Foldable

The `Foldable` type class

```haskell
class Foldable t where
  {-# MINIMAL foldMap | foldr #-}
  fold :: Monoid m => t m -> m
  fold = foldMap id

  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldMap f = foldr (mappend . f) mempty

  foldr :: (a -> b -> b) -> b -> t a -> b
  foldr f z t = appEndo (foldMap (Endo . f) t) z

  foldl :: (b -> a -> b) -> b -> t a -> b
    foldl f z t =
      appEndo (getDual (foldMap (Dual . Endo . flip f) t)) z
```

Useful functions for foldable data types

Here we provide type signatures only:

```
toList :: Foldable t => t a -> [a]

null :: Foldable => t a -> Bool

length :: Foldable t => t a -> Int

elem :: (Eq a, Foldable t) => a -> t a -> Bool

maximum :: (Ord a, Foldable t) => t a -> a

sum, product :: (Num a, Foldable t) => t a -> a
```

Applicative functors

Motivation

- It is clear that we would like to have something like `fmap` for functions that have an arbitrary arity:

```
fmap2
  :: (a -> b -> c)
  -> f a -> f b -> f c
fmap3
  :: (a -> b -> c -> d)
  -> f a -> f b -> f c -> f d
fmap4
  :: (a -> b -> c -> d -> e)
  -> f a -> f b -> f c -> f d -> e
...
```

- That is, one needs to generalise a functor
- The solution is the `Applicative` type class that extends `Functor`

The `Applicative` class

```haskell
class Functor f => Applicative f where
    {-# MINIMAL pure, ((<*>) | liftA2) #-}
    pure :: a -> f a

    (<*>) :: f (a -> b) -> f a -> f b
    (<*>) = liftA2 id  -- the same as liftA2 ($)

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

The `Applicative` class. Example

```haskell
instance Applicative (Maybe a) where
  pure = Just
  Nothing <*> _ = Nothing
  _ <*> Nothing = Nothing
  Just f <*> Just x = Just (f x)

instance Applicative [a] where
  pure x = [x]
  fs <*> fx = [ f x | f <- fs, x <- xs]
```

The `Applicative` laws

```haskell
fmap f x = pure f <*> x

pure id <*> v = v   -- identity

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

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

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

Let us check some of these laws for the `Maybe` data type

The `Applicative` laws

```
Nothing <*> _ = Nothing
_ <*> Nothing = Nothing
Just f <*> Just x = Just (f x)

fmap f Nothing = Nothing = Just f <*> Nothing
fmap f (Just x) = Just (f x) = Just f <*> Just x

pure id <*> v = fmap id v = v

Nothing <*> Just y = Nothing = Just ($ y) <*> Nothing
Just f <*> Just y = Just (f y) = Just ($ y) <*> Just f
```

The `Applicative` class. The list problem

- The list data type might have an alternative `Applicative` instance
- One has the function called `zipWith`:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

- The signature of `zipWith` corresponds to the signature of `liftA2`
- On the other hand, we cannot have two instances for the same data type

The `Applicative` class. The list problem

Let us introduce the following data type:

```haskell
newtype ZipList a = ZipList { getZipList :: [a]}
```

The instance is implemented via `zipWith`:

```haskell
instance Applicative (ZipList a) where
  liftA2 f (ZipList xs) (ZipList ys) =
    ZipList (zipWith f xs ys)
  zipF <*> zipX = liftA2 ($)
  pure = ???
```

How to implement `pure` to preverse the identity law?

Traversable

The `Traversable` definition

```
class (Functor t, Foldable t) => Traversable t where
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
  {-# MINIMAL traverse | sequenceA #-}
```

Summary