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`
- study such functor extensions as applicative functors and the `Travesable` class

Functor

Motivation

- Let us take a look at these functions

```haskell
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 with the type class `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 where
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just (f x)

instance Functor [] 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 example of a `Functor` instance

```haskell
import Data.Functor

data Tree a = Leaf a | Node (Tree a) a (Tree a)
  deriving Show

instance Functor Tree where
  fmap f (Leaf a) = Leaf (f a)
  fmap f (Node ls a rs) = Node (fmap f ls) (f a) (fmap f rs)

left = Node (Leaf 2) 3 (Leaf 5)
right = Node (Leaf 5) 7 (Leaf 11)
tree = Node left 13 right

treeWord = (\x -> show x ++ show x) <$> tree
voidTree = void tree
constTree = "Anna" <$ treeWord
```

The example of a `Functor` instance. The `DeriveFunctor`

One may derive the `Functor` automatically, if one has a permission to the structure of an observed data type.

```haskell
{-# LANGUAGE DeriveFunctor #-}

import Data.Functor

data Tree a = Leaf a | Node (Tree a) a (Tree a)
  deriving (Show, Functor)
```

One may ensure that the calls from the previous slide yield the same output:

```haskell
treeWord = (\x -> show x ++ show x) <$> tree
voidTree = void tree
constTree = "Anna" <$ treeWord
```

The `Functor` instances for two-parametric data types

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

```haskell
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:

```haskell
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)
```

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 where
  pure = Just
  Nothing <*> _ = Nothing
  _ <*> Nothing = Nothing
  Just f <*> Just x = Just (f x)

instance Applicative [] 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) -- composition

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 on a whiteboard

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:

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

The instance is implemented via `zipWith`:

```
instance Functor ZipList where
  fmap f = ZipList . getZipList . fmap f

instance Applicative ZipList 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?

The `Applicative` class. The list problem

Let us introduce the following data type:

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

The instance is implemented via `zipWith`:

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

Monoid

The `Monoid` definition

```
class Semigroup a where
  (<>) :: a -> a -> a
    -- a binary associative operation

class Semigroup a => Monoid a where
  mempty :: a
    -- a neutral element
  mappend :: a -> a -> a
  mappend = (<>)
  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

The operation in a semigroup should associative and `mempty` is a neutral element

```
a <> (b <> c) = (a <> b) <> c
a <> mempty = a = mempty <> a
```

The `Monoid` instances

```haskell
instance Semigroup [a] where
  (<>) = (++)

instance Monoid [a] where
  mempty = []
```

It is clear that `(++)` is an associative operation and `[]` is neutral.

Numbers and Booleans as monoids

```haskell
newtype Sum a = Sum { getSum :: a }
  deriving (Show, Eq, Ord)

instance Num a => Semigroup (Sum a) where
  Sum a <> Sum b = Sum (a + b)

instance Num a => Monoid (Sum a) where
  mempty = Sum 0
```

One has the `Monoid` instance for any numerical type with the product as a binary operation. For that one needs to introduce the following new type:

```haskell
newtype Sum a = Sum { getSum :: a }
  deriving (Show, Eq, Ord)
```

Numbers and Booleans as monoids

```
newtype All = All { getAll :: Bool }
  deriving (Show, Eq, Show)

instance Semigroup All where
  All a <> All b = All (a && b)

instance Monoid All where
  mempty = All True
```

One has the similar story with the Boolean disjunction.

Back to the `Applicative` class

The `Monoid` type class allows one to have the `Applicative` instance for tuples as follows:

```
instance Monoid a => Applicative ((,) a) where
  pure x = (mempty x, x)
  (a, f) <*> (b, x) = (a <> b, f x)
```

From the left, there is a counter or logging message which is accumulated by the monoidal operation. From the right, one has an application.

Foldable

Motivation

- Before we took a look at such fold functions as `foldr`

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

- One may generalise the idea of folding and consider a broader class of foldable data structures
- Before that, we introduce the type class called `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
```

Foldable instances.

```
instance Foldable [] where
    elem    = List.elem
    foldl   = List.foldl
    foldr   = List.foldr
    length  = List.length
    maximum = List.maximum
    product = List.product
```

Foldable instances. Other examples

```haskell
instance Foldable (Either a) where
    foldMap _ (Left _) = mempty
    foldMap f (Right y) = f y

    foldr _ z (Left _) = z
    foldr f z (Right y) = f y z

    length (Left _)  = 0
    length (Right _) = 1

    null             = isLeft

instance Foldable ((,) a) where
    foldMap f (_, y) = f y
    foldr f z (_, y) = f y z
```

Traversable

The motivating example

```haskell
dist :: Applicative f => [f a] -> f [a]
dist [] = pure []
dist (x : xs) = liftA2 (:) x (dist xs)
```

```
> dist (Just <$> [1,2,4])
Just [1,2,4]
> dist [Just 1, Nothing]
Nothing
> getZipList $ dist $ map ZipList [[1,2,3], [4,5,6], [7,8,9]]
[[1,4,7],[2,5,8],[3,6,9]]
```

The `Traversable` definition

The `Traversable` type class describe how exactly two computational contexts commute with each other:

```haskell
class (Functor t, Foldable t) => Traversable t where
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  traverse f = sequenceA . fmap f

  sequenceA :: Applicative f => t (f a) -> f (t a)
  sequenceA = traverse id
  {-# MINIMAL traverse | sequenceA #-}
```

The `Traversable` instances

```
instance Travesable Maybe where
  traverse _ Nothing = Nothing
  traverse f (Just x) = Just <$> f x

instace Traversable [] where
  traverse _ g = foldr consF (pure [])
    where
      consF x ys = liftA2 (:) (g x) ys
```

Summary

Summary

Today we

- introduced such type classes as `Functor`, `Applicative`, `Monoid`, `Traversable`, and `Traversable`

Summary

Today we

- introduced such type classes as `Functor`, `Applicative`, `Monoid`, `Traversable`, and `Traversable`

Next time, we

- study actions within a computational context and