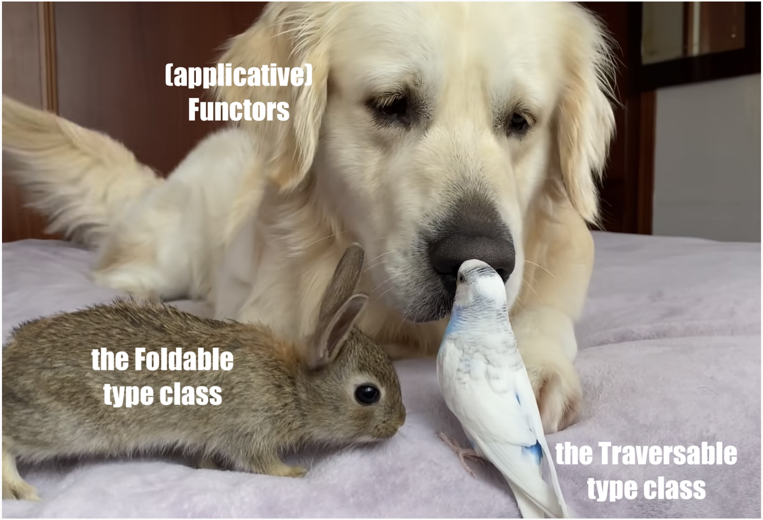
Functional programming, Seminar No. 5

Daniel Rogozin
Institute for Information Transmission Problems, RAS
Serokell OÜ
Higher School of Economics
The Faculty of Computer Science

Today

We will study



Foldable

Semigroup and Monoid: the definition

In this subsection, we study the generalisation of folds with the type class called `Foldable`. For that, we have a look at the classes called `Semigroup` and `Monoid`.

```
class Semigroup a where
  (<>) :: a -> a -> a

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

`Semigroup` and `Monoid`: the laws

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

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

- NOTE BENE: every `Semigroup`/`Monoid` instance should obey these laws.
- The compiler is not capable of proving such properties (note that tests $\neq$ proofs), so programmers have to ensure that required laws are valid for such instances themselves.
- The moral is that declaring a `Semigroup` instance where the operations happens to be non-associative is *mauvais goût*.

The `Monoid` instances

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

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

`(++)` is associative operation and `[]` is neutral. A semiformal proof:

```haskell
[] ++ (ys ++ zs) = ys ++ zs = ([] ++ ys) ++ zs

(x : xs) ++ (ys ++ zs) =
  x : (xs ++ (ys ++ zs)) = -- IH
  x : ((xs ++ ys) ++ zs) =
  (x : (xs ++ ys)) ++ zs =
  ((x : xs) ++ ys) ++ zs
```

Numbers and Booleans as monoids

```
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 because the same type cannot have two different instances.

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

Numbers and Booleans as monoids

```haskell
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
```

The similar for disjunction by putting `a <> b = a || b` and `mempty = False`.

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 to consider a broader class of foldable data structures

The `Foldable` type class

```haskell
class Foldable t where
  {-# MINIMAL foldMap | foldr #-}
  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
```

where

```haskell
  newtype Endo a = Endo { appEndo :: a -> a }
```

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

```haskell
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
```

Foldable instances. Other examples

```
instance Foldable NonEmpty
instance Foldable Set
instance Foldable (Map k)
instance Foldable (Array i)
instance Foldable Vector
```

Functor

Motivation

- Let us have a look at the following 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)
```

- These function are similar. Here we have an unary function that we carry through a element of a parametrised type.
- We generalise that with the type class `Functor`.

Functor: the definition and instances

```
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
  fmap _ [] = []
  fmap f (x : xs) = f x : map f xs
```

The full definition of `Functor`

```haskell
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
```

Another 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 `DeriveFunctor` extension

One may derive the `Functor` instance automatically for some data types.

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

import Data.Functor

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

The derived instance in this example is equivalent to our version above.

`Functor` instances for two-parametric data 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

Any `Functor` instance has to satisfy 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 really a functor by induction.

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

fmap (f . g) [] = []
fmap (f . g) (x : xs) =
  (f . g) x : fmap (f . g) xs =    -- IH
  (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 of 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
...
```

We cannot do that using only `fmap` for unary functions.

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 ($)

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

The Applicative class. A couple of examples

```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 (\f -> f y) <*> u -- interchange
```

The `Applicative` class. Another `Applicative` instance for lists

- The list data type might have an alternative `Applicative` instance
- Recall the function `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, as we've already said, we cannot have two instances for the same data type

The `Applicative` class. Another `Applicative` instance for lists

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

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` and preverse the applicative laws? If we define `pure` as below, that would break all axioms of an applicative functor.

```haskell
  pure x = ZipList [x]
```

The `Applicative` class. Another `Applicative` instance for lists

- Here is the proper instance:

```haskell
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
```

`Applicative` instance for tuples

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)
```

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

According to the documentation, `Traversable` describes "functors representing data structures that can be traversed from left to right".

```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`, `Foldable`, and `Traversable`

Summary

Today we

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

Next time, we will

- study monads!