Lecture 14. Foldables and traversables

Functional Programming 2019/20

Matthijs Vákár

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Goals

- How do we calculate summaries of data structures other than lists?
  - Learn about Monoid and Foldable type classes
- How do we `map` impure functions over data structures?
  - Learn about Applicative and Traversable type classes
- See some examples of monadic and applicative code in action.

Chapter 14 from Hutton's book

Our three example data types for today

Binary trees:

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

Rose trees:

```
data Rose a = RLeaf | RNode a [Rose a]
```

ASTs:

```
data Expr x = Var x
            | Val Int
            | Add (Expr x) (Expr x)
```

Linear summaries: Monoids and Foldables

Summaries to calculate

- Sums, products of entries
- And/or of entries
- Used variables
- Composition of all functions in data structure
- Parity of Booleans in data structure

Monoids and folds abstract the idea of combining elements in a well-behaved way!

Monoids

Some types have an intrinsic notion of *combination*

- We already hinted at it when describing folds
- **Monoids** provide an *associative* binary operation with an *identity* element

```
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
  mconcat :: [m] -> m
  mconcat = foldr mappend mempty
```

Monoids: example

Lists `[T]` are monoids *regardless* of their contained type `T`

```
instance Monoid [t] where
  mempty  = []      -- empty list
  mappend = (++)    -- concatenation
  mconcat = concat
```

The simplest monoids in a sense (jargon: *free* monoids)

Monoid laws

Monoids capture *well-behaved* notion of combination, respecting these laws:

```
mempty <> y      = y                 -- left identity
x      <> mempty = x                 -- right identity
(x <> y) <> z    = x <> (y <> z)     -- associativity
```

We write `mappend` infix as `<>`.

Do these remind of you anything?

Some examples of monoids

Can you come up with some examples of monoids?

Folds and Monoids

Recall, folding on lists:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldl :: (b -> a -> b) -> b -> [a] -> b
```

We have seen, because of associativity and identity laws:

```
foldr mappend mempty = foldl mappend mempty
```

for any monoid!

Folds and Monoids

Recall, folding on lists:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldl :: (b -> a -> b) -> b -> [a] -> b
```

We have seen, because of associativity and identity laws:

```
foldr mappend mempty = foldl mappend mempty
```

for any monoid!

Note that monoids may be non-commmutative, so that

```
foldr mappend mempty /= foldr (flip mappend) mempty
```

Generalizing foldr?

```
foldr :: (a -> b -> b) -> b -> [a] -> b
         (a -> b -> b) -> [a] -> b -> b
         (a -> End b ) -> [a] -> End b

foldMap :: Monoid m => (a -> m) -> [a] -> m
```

Does this buy us anything?

Foldables

Want:

```
foldr :: (a -> b -> b) -> b -> t a -> b
```

or

```
foldMap :: Monoid m => (a -> m) -> t a -> m
```

for some other container type `t`.

`t` had better be a functor…

Foldables

Data structure we can fold over, like lists:

```
class Functor t => Foldable t where
    foldr :: (a -> b -> b) -> b -> t a -> b
    foldr op i = ???

    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldMap f = ???

    toList :: t a -> [a]
    toList = ???

    fold :: Monoid m => t m -> m
    fold = ???
```

Foldables

Data structure we can fold over, like lists:

```
class Functor t => Foldable t where
    foldr :: (a -> b -> b) -> b -> t a -> b
    foldr op i = foldr op i . toList

    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldMap f = mconcat . map f . toList

    toList :: t a -> [a]
    toList = foldr (:) []

    fold :: Monoid m => t m -> m
    fold = foldMap id
```

In essence, a data type that we can linearize to a list...

Some examples

Let's implement `Foldable` for `Tree`, `Rose` and `Expr`!

See how they solve our initial problem!

Mapping Impure Functions: Applicatives and Traversables

Mapping impure functions

- A stateful walk of a tree
- A walking a rose tree while performing IO
- Trying to evaluate an expression while accumulating errors
- Idea: keep shape of data structure; replace entries using impure function; accumulate side effects on the outside

Applicatives and traversals abstract the idea of mapping impure functions in a well-behaved way!

The functor - applicative - monad hierarchy

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

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

class Applicative f => Monad f where
  return :: a -> f a -- equals Applicative's pure
  (>>=) :: f a -> (a -> f b) -> f b
```

Recall: Applicatives from Monads

Every monad induces an applicative

```
pure = return
af <*> ax = do f <- af
               x <- ax
               return (f x)
```

But not every applicative arises that way!

Example: Error Accumulation

Example of Applicative that does not come from Monad

```
data Error m a = Error m | OK a
```

How is `Error m` a functor? An applicative?

Example: Error Accumulation

Example of Applicative that does not come from Monad

```
data Error m a = Error m | OK a
```

How is `Error m` a functor? An applicative?

```
instance Monoid m => Applicative (Error m) where
    pure                         = OK
    (Error m1) <*> (Error m2) = Error (m1 <> m2)
    (Error m1) <*> (OK a)     = Error m1
    (OK f)     <*> (Error m2) = Error m2
    (OK f)     <*> (OK a)     = OK (f a)
```

Why not from a monad?

Traversables

Data structure we can traverse/walk:

```
class Foldable t => Traversable t where
  traverse :: Applicative f =>
                (a -> f b) -> t a -> f (t b)
  traverse g ta = ???

  sequenceA :: Applicative f => t (f a) -> f (t a)
  sequenceA = ???
```

Think of `traverse` **as a map over** `t` **using an impure function** `:: a -> f b`

Traversables

Data structure we can traverse/walk:

```
class Foldable t => Traversable t where
  traverse :: Applicative f =>
                (a -> f b) -> t a -> f (t b)
  traverse g ta = sequenceA (fmap g ta)

  sequenceA :: Applicative f => t (f a) -> f (t a)
  sequenceA = traverse id
```

Think of `traverse` **as a map over** `t` **using an impure function** `:: a -> f b`

Some examples

Let's implement `Foldable` for `Tree`, `Rose` and `Expr`!

Traversing with the Identity Applicative

We have the identity monad

```
newtype Identity' a = Identity' {runIdentity' :: a}

instance Monad Identity' where
    return x = Identity' x
    x >>= f  = f (runIdentity' x)
```

Traversing with the Identity Applicative

We have the identity monad

```
newtype Identity' a = Identity' {runIdentity' :: a}

instance Monad Identity' where
    return x = Identity' x
    x >>= f  = f (runIdentity' x)
```

Traversals are impure maps:

```
fmap :: Traversable t => (a -> b) -> t a -> t b
fmap == runIdentity . traverse (Identity . f)
```

Relating Monoids and Applicatives, Folds and Traversals

Phantom Types: All Monoids Are Applicatives

Introduce fake type dependency:

```haskell
newtype Const a b = Const { getConst :: a }

instance Monoid m => Applicative (Const m) where
  pure _ = Const mempty
  (<*>) (Const f) (Const b) = Const (f <> b)
```

Phantom Types: All Monoids Are Applicatives

Introduce fake type dependency:

```
newtype Const a b = Const { getConst :: a }

instance Monoid m => Applicative (Const m) where
  pure _ = Const mempty
  (<*>) (Const f) (Const b) = Const (f <> b)
```

Claim: traversing with `Const` is the same as folding:

```
foldMap :: (Traversable t, Monoid m) =>
             (a -> m) -> t a -> m
foldMap f = getConst . sequenceA . fmap (Const . f)
```

Foldables and Traversables in practice

- We can derive Foldable and Traversable instances (using a compiler extension)!
- The built-in instances for tuples can be *very* confusing
- A little game

```
Prelude> minimum(1,100)

Prelude> let splat = splitAt 5 [0..10]
Prelude> splat
([0,1,2,3,4],[5,6,7,8,9,10])
Prelude> concat splat
```

Foldables and Traversables in practice

- We can derive Foldable and Traversable instances (using a compiler extension)!
- The built-in instances for tuples can be *very* confusing
- A little game

```
Prelude> minimum(1,100)
100
Prelude> let splat = splitAt 5 [0..10]
Prelude> splat
([0,1,2,3,4],[5,6,7,8,9,10])
Prelude> concat splat
[5,6,7,8,9,10]
```

```
Prelude> fmap (+1) [1,2]
[2,3]
Prelude> fmap (+1) (1,2)

Prelude> let xs = [(1,"hello"),(2,"world")]
Prelude> length "world"
5
Prelude> length (lookup 2 xs)

Prelude> let y = lookup 100 xs
Prelude> null y
True
Prelude> length y
```

```
Prelude> fmap (+1) [1,2]
[2,3]
Prelude> fmap (+1) (1,2)
(1,3)
Prelude> let xs = [(1,"hello"),(2,"world")]
Prelude> length "world"
5
Prelude> length (lookup 2 xs)
1
Prelude> let y = lookup 100 xs
Prelude> null y
True
Prelude> length y
0
```

Summary

- Monoids capture a notion of summary/combination of values
- Foldables are data types that can be cast to a list
- They let us calculate summaries of the values in a data type

Summary

- Monoids capture a notion of summary/combination of values
- Foldables are data types that can be cast to a list
- They let us calculate summaries of the values in a data type

- Applicatives capture a notion of side effect
- Traversables are data types that we can map effectful functions over
- Monoids/foldables are a special case of applicatives/traversables (by using Phantom types)

Where to from here?

Talen en compilers!

- Efficient parsing using applicatives
- Lots of traversals
- Recursion schemes: much more interesting generalization of folds to other data types
- Plenty of other cool FP tricks