Functional Programming

Lecture 14: Foldable and Traversable

Twan van Laarhoven

20 December 2021

Outline

- Foldable containers
- Traversable containers
- Ad hoc traversals
- Lenses

Foldable

Containers - again

What is a (finite) container?

- A container holds some number of 'values'
- It supports fmap (Functor)

Containers - again

What is a (finite) container?

- A container holds some number of ‘values’
- It supports `fmap` (Functor)
- We can `fold` over the values in the container

What is a fold?

Left and right folds

```
foldr :: (a → b → b) → b → [a] → b
foldl :: (b → a → b) → b → [a] → b
```

For lists: `foldr` matches structure.

```
foldr (⊗) z (1 : (2 : (3 : []))) = (1 ⊗ (2 ⊗ (3 ⊗ z)))
```

What is a fold?

```
class Monoid m where
  mempty :: m
  (<>)   :: m → m → m
```

Left and right folds

$$\mathsf{foldr} :: (a \to b \to b) \to b \to [a] \to b$$
$$\mathsf{foldl} :: (b \to a \to b) \to b \to [a] \to b$$

For lists: foldr matches structure.

$$\mathsf{foldr}\ (\otimes)\ z\ (1 : (2 : (3 : []))) = (1 \otimes (2 \otimes (3 \otimes z)))$$

Tree structured folds

$$\mathsf{fold} :: (\mathsf{Monoid}\ m) \Rightarrow [m] \to m$$

Often combined with a map

$$\mathsf{foldMap} :: (\mathsf{Monoid}\ m) \Rightarrow (a \to m) \to [a] \to m$$

```
class Monoid m where
  mempty :: m
  (<>)   :: m -> m -> m
```

fold/foldMap for lists

Collapsing a list of values into a single value:

```
fold :: Monoid m => [m] -> m
fold [ ]      = mempty
fold (x : xs) = x <> fold xs
```

Is the same as

```
fold = foldr (<>) mempty
```

Combining fold with map

```
foldMap :: Monoid m => (a -> m) -> [m] -> m
foldMap f [ ]      = mempty
foldMap f (x : xs) = f x <> foldMap f xs
```

foldMap examples

With the Sum monoid

```haskell
newtype Sum a = Sum { getSum :: a }
instance Num a => Monoid (Sum a) where
  mempty = Sum 0
  Sum x <> Sum y = Sum (x + y)
```

We can implement

```haskell
sum :: Num a => [a] -> a
sum = getSum . foldMap Sum
```

and

```haskell
length :: [a] -> Int
length = getSum . foldMap (\_ -> Sum 1)
```

Foldable containers

Generalize fold to any container type

```
class Foldable t where
  foldr   :: (a → b → b) → b → t a → b
  foldl   :: (b → a → b) → b → t a → b
  fold    :: Monoid m ⇒ t m → m
  foldMap :: Monoid m ⇒ (a → m) → t a → m
```

Foldable includes default implementations, defining either foldr or foldMap is sufficient.

Foldable containers

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
instance Foldable Tree where
  foldMap :: Monoid m => (a -> m) -> Tree a -> m
  foldMap f (Leaf x)   = f x
  foldMap f (Node l r) = foldMap f l <> foldMap f r
```

foldMap follows tree structure:

```
foldMap f (Node (Node (Leaf 1) (Leaf 2)) (Node (Leaf 3) (Leaf 4)))
        = (f 1 <> f 2) <> (f 3 <> f 4)
```

And is much easier to implement than foldr

Other primitives

What primitives can we define for Foldable containers?

```
toList  :: (Foldable t) ⇒ t a → [a]
null    :: (Foldable t) ⇒ t a → Bool
length  :: (Foldable t) ⇒ t a → Int
elem    :: (Foldable t, Eq a) ⇒ a → t a → Bool
maximum :: (Foldable t, Ord a) ⇒ t a → a
```

Exercise: define these functions in terms of `foldr` or `foldMap`.

Defining foldMap using foldr and vice-versa

foldMap from foldr

```
foldMap :: Monoid m => (a -> m) -> t a -> m
foldMap f = foldr (\a m -> f a <> m) mempty
```

Defining foldMap using foldr and vice-versa

foldMap from foldr

```
foldMap :: Monoid m => (a -> m) -> t a -> m
foldMap f = foldr (\a m -> f a <> m) mempty
```

foldr from foldMap

```
foldr :: (a -> b -> b) -> b -> t a -> b
foldr f e x = appEndo (foldMap (Endo . f) x) e
```

Endo explained

Functions in the same type (a → a) are a monoid (endomorphisms)

```haskell
newtype Endo a = Endo { appEndo :: a -> a }
instance Monoid (Endo a) where
  mempty = Endo id
  Endo f <> Endo g = Endo (f . g)
```

For example

```haskell
   foldr f e [1,2,3]
 = (appEndo (foldMap (Endo . f) [1,2,3])) e
 = (appEndo (Endo (f 1) <> Endo (f 2) <> Endo (f 3) <> Endo id)) e
 = (f 1 . f 2 . f 3 . id) e
 = f 1 (f 2 (f 3 e))
```

Traversable containers

Traversables

Mapping a function that may fail over a list

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

Generalize to

- any applicative f instead of Maybe (seen before as mapM).
- any container t instead of list

Gives

```
class (Functor t, Foldable t) ⇒ Traversable t where
  traverse :: Applicative f ⇒ (a → f b) → t a → f (t b)
```

Traversable: examples

Instance for lists:

```
instance Traversable [] where
  traverse :: Applicative f ⇒ (a → f b) → [a] → f [b]
  traverse g [ ]      = pure []
  traverse g (x : xs) = pure (:) <*> g x <*> traverse g xs
```

Instance for trees:

```
instance Traversable Tree where
  traverse :: Applicative f ⇒ (a → f b) → Tree a → f (Tree b)
  traverse g (Leaf x)   = pure Leaf <*> g x
  traverse g (Node l r) = pure Node <*> traverse g l <*> traverse g r
```

Traverse generalizes mapM

Apply an action to every element of a list

```
askUser :: String → IO Bool

twoQuestions :: IO [Bool]
twoQuestions = mapM askUser
                 ["Do you like cake?",
                  "Are you still here?"]
```

Traverse generalizes mapM

Apply an action to every element of a container

```
askUser :: String → IO Bool

twoQuestions :: IO [Bool]
twoQuestions = traverse askUser
                 ["Do you like cake?",
                  "Are you still here?"]

optQuestion :: Maybe String → IO (Maybe Bool)
optQuestion = traverse askUser
```

Traverse generalizes map

Recall: identity functor

```haskell
newtype Id a = Id { getId :: a }
instance Functor Id where
  fmap f (Id x) = Id (f x)
instance Applicative Id where
  pure = Id
  Id f <*> Id x = Id (f x)
```

Then

```haskell
amap :: Traversable f => (a -> b) -> f a -> f b
amap f = getId . traverse (Id . f)
```

So Traversable containers are Functors

Traverse generalizes fold

The constant functor

```
newtype Const c a = Const { getConst :: c }
instance Functor (Const c) where
  fmap _ (Const x) = Const x
instance Monoid c => Applicative (Const c) where
  pure _ = Const mempty
  Const x <*> Const y = Const (x <> y)
```

Gives

```
afoldMap :: (Traversable f, Monoid m) => (a -> m) -> f a -> m
afoldMap f = getConst . traverse (Const . f)
```

So Traversable containers are Foldable

Ad hoc traversals

Lenses

Ad hoc traversals

Traversable is about visiting all elements in a container

```
traverse :: (Traversable t, Applicative f) ⇒ (a → f a) → t a → f (t a)
```

We can also traverse all literals in an expression

```
lits :: Applicative f ⇒ (Integer → f Integer) → Expr → f Expr
lits f (Lit x)   = Lit <$> f x
lits f (Add x y) = Add <$> lits f x <*> lits f y
lits f (Mul x y) = Mul <$> lits f x <*> lits f y
```

Or the first element of a pair

```
fst' :: Applicative f ⇒ (a → f a) → (a,b) → f (a,b)
fst' f (x,y) = (\x' → (x',y)) <$> f x
```

Using ad hoc traversals

Sum of all literals in an expression

```
sumLits :: Expr → Integer
sumLits = getSum . getConst . lits (Const . Sum)
```

Count number of literals

```
countLits :: Expr → Int
countLits = getSum . getConst . lits (\_ → Const (Sum 1))
```

```
>>> sumLits (Add (Lit 1) (Mul (Lit 3) (Lit 3)))
7
>>> countLits (Add (Lit 1) (Mul (Lit 3) (Lit 3)))
3
```

Look familiar? Compare fold from traverse; sum and length from fold.

Using ad hoc traversals as folds

Let's give this a name

```
type Traversal s a = forall f. Applicative f ⇒ (a → f a) → (s → f s)
lits :: Traversal Expr Integer
lits = -- as before

foldMapOf :: Monoid m ⇒ Traversal s a → (a → m) → (s → m)
foldMapOf trav f = getConst . trav (Const . f)

lengthOf :: Traversal s a → s → Int
lengthOf trav = getSum . foldMapOf trav (\_ → Sum 1)
```

Now

```
length = lengthOf traverse
countLits = lengthOf lits
```


Note: 'forall' requires language extension

Using ad hoc traversals as maps

Also

```
type Traversal s a = forall f. Applicative f ⇒ (a → f a) → (s → f s)
mapOf :: Traversal s a → (a → a) → (s → s)
mapOf trav f = getId . trav (Id . f)    -- sometimes called over
```

Now

```
fmap = mapOf traverse
```

Example

```
>>> mapOf lits (*2) (Add (Lit 1) (Mul (Lit 3) (Lit 3)))
(Add (Lit 2) (Mul (Lit 6) (Lit 6)))
```

More traversals

Traversing all items in a container

```
each :: Traversable t ⇒ Traversal (t a) a
each = traverse
```

Traversing a single item in a list

```
at :: Applicative f ⇒ Int → (a → f a) → [a] → f [a]
at 0 f (x:xs) = (:xs) <$> f x
at n f (x:xs) = (x:) <$> at (n−1) xs
at _ f _ = error "index out of range"
```

More traversals – examples

```
>>> foldMapOf (at 2) return "hello"
"l"
>>> mapOf each toUpper "hello"
"HELLO"
>>> mapOf (at 2) toUpper "hello"
"heLlo"
```

Odd and even

```
>>> mapOf evens toUpper "hello world"
"HeLlO WoRlD"
```

Where

```
-- Traverse the even positions
evens :: Applicative f => (a -> f a) -> [a] -> f [a]   -- Traversal [a] a
evens _ []     = pure []
evens f (x:xs) = (:) <$> f x <*> odds f xs
-- Traverse the odd positions
odds :: Applicative f => (a -> f a) -> [a] -> f [a]
odds _ []     = pure []
odds f (x:xs) = (x:) <$> evens f x -- don't apply f to x
```

Lenses

If the function is used exactly once, then Functor is enough

```
fst' :: Functor f ⇒ (a₁ → f a₁) → (a₁,b) → f (a₂,b)
fst' f (x₁,y) = (\x₂ → (x₂,y)) <$> f x₁
```

Such a traversal is called a *lens*.

```
type Lens s a = forall f. Functor f ⇒ (a → f a) → (s → f s)
```

Compare

```
type Traversal s a = forall f. Applicative f ⇒ (a → f a) → (s → f s)
```

Viewing through a lens

Const c is only an Applicative functor if c is a Monoid.

```
foldOf :: Monoid m => Traversal s m -> s -> m
foldOf = foldMapOf id
```

But Const c is always a *functor* even if c is not a Monoid, so

```
view :: Lens s a -> s -> a
view lens = getConst . lens Const
```

Also

```
modify :: Lens s a -> (a -> a) -> s -> s
modify = mapOf

set :: Lens s a -> a -> s -> s
set lens x = modify (const x)
```

Lens examples

```
>>> view (at 2) [1,2,3,4]
3
>>> modify (at 2) negate [1,2,3,4]
[1,2,-3,4]
>>> view fst' (123,"hello")
123
>>> set fst' 456 (123,"hello")
(456,"hello")
```

Composing traversals

Visit the even elements, and for each of those visit the third element

```
evens . at 2 :: Traversal [[a]] a
```

So

```
>>> foldMapOf (evens . at 2) return ["hello","world","where","I","live"]
"lev"
```

```
>>> mapOf (each . each) (+1) [Just 2, Nothing, Just 3]
[Just 3, Nothing, Just 4]
```

Composing two lenses gives a lens

```
>>> view (fst' . snd') (('a',3),"thing")
3
```

Conditional traversal

Visit a value if a condition holds

```haskell
when :: (a → Bool) → Traversal a a
when pred f x
  | pred x      = f x
  | otherwise = pure x

>>> mapOf (each . when (> 0)) succ [1,3,−4,2,0]
[2,4,−4,3,0]
```

Traversing children

Traverse direct children of an expression

```
children :: Traversal Expr Expr
children f (Lit x)   = pure (Lit x)
children f (Add x y) = Add <$> f x <*> f y
children f (Mul x y) = Mul <$> f x <*> f y
```

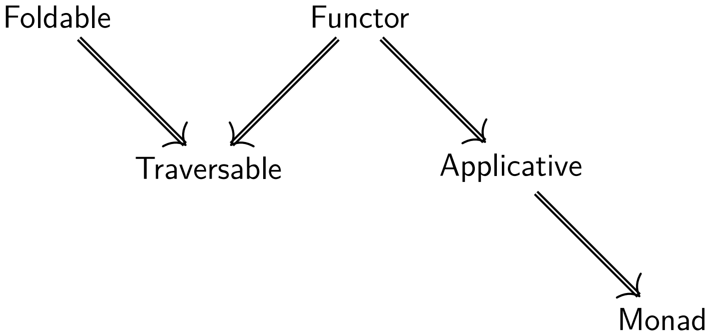
Use as a default

```
flipMul :: Expr → Expr
flipMul (Mul x y) = Mul (flipMul y) (flipMul x)
flipMul expr      = mapOf children flipMul expr
```

Flips arguments to all Mul constructors

Take away

The container-like class hierarchy



Summary

- Foldable generalizes folds to other containers
- Traversable generalizes `mapM`.
- Ad-hoc traversals can view and modify parts of a data structure