Functional Programming
Lecture 13

Rostislav Horčík
Niklas Heim

Czech Technical University in Prague
Faculty of Electrical Engineering
xhorcik@fel.cvut.cz
heimnikl@fel.cvut.cz

Folding

Functions `foldl` and `foldr` allow traversing a list and aggregating its elements by a given function.

```
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

```
> foldl max 0 [3,4,5,2,0]
5

> foldl (+) 0 [3,4,5,2,0]
14
```

`Foldable` splits the traversing part from the aggregating part.

Monoids

Abstract aggregating

Semigroup $\langle S, * \rangle$ is a set S endowed with a binary operation $*$ satisfying

$$(x * y) * z = x * (y * z)$$

Monoid $\langle M, *, u \rangle$ is a semigroup $\langle M, * \rangle$ with a constant $u \in M$ satisfying

$$u * x = x = x * u$$

Examples

- $\langle \mathbb{N}, +, 0 \rangle$
- $\langle \mathbb{R}, \cdot, 1 \rangle$
- $\langle$`[a],++,[]`$\rangle$ lists over a type `a` (free monoid)
- $\langle A^A, \circ, id \rangle$ selfmaps $f : A \to A$ form a monoid under composition $\circ$
- $\langle [a, b], \min, b \rangle$ and $\langle [a, b], \max, a \rangle$

Type classes Semigroup and Monoid

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

class Semigroup a => Monoid a where
    mempty :: a
    mappend :: a -> a -> a
    mconcat :: [a] -> a
```

```haskell
mappend = (<>)
```

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

As a type can have only a single instance of `Monoid`, we use type wrappers to define various monoidal instances over a type.

```
newtype Sum a = Sum {getSum :: a}

instance Num a => Semigroup (Sum a)
    (Sum x) <> (Sum y) = Sum (x+y)

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

> (Sum 7) <> (Sum 4)
Sum {getSum = 11}

newtype Product a = Product {getProduct :: a}
```

Further instances

`Any` (resp. `All`) is the disjunctive (resp. conjunctive) monoid on `Bool`.

```
> (Any False) <> (Any True) <> (Any False)
Any {getAny = True}
```

For a monoid `m` its dual monoid is `Dual m`

```
> (Dual "a") <> (Dual "b") <> (Dual "c")
Dual {getDual = "cba"}
```

Product of monoids

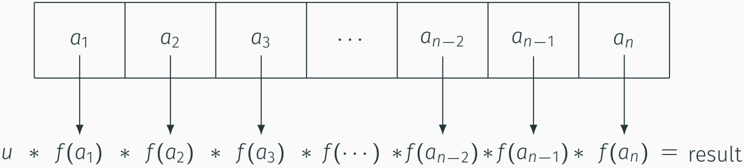
```
> (Sum 2,Product 3) <> (Sum 5,Product 7)
(Sum {getSum = 7},Product {getProduct = 21})
```

Foldables

Folding lists

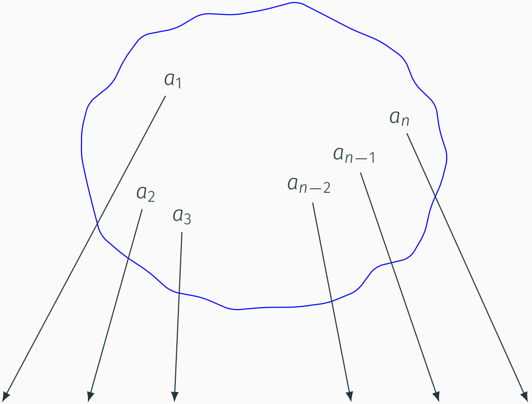
Let $M = \langle M, *, u \rangle$ be a monoid, $f : A \to M$ and $lst = [a_1, \dots, a_n]$ a list of elements from A.

foldMap of *lst* w.r.t. M and f is the composition of *map f* followed by the aggregation.



FoldMap

Once we are able to traverse a data structure and collect some elements, we can do foldMap.



$$u * f(a_1) * f(a_2) * f(a_3) * f(\cdots) * f(a_{n-2}) * f(a_{n-1}) * f(a_n) = \text{result}$$

Foldable

In the library `Data.Foldable`

```
class Foldable t where
  fold :: Monoid m => t m -> m
  foldMap :: Monoid m => (a -> m) -> t a -> m
  foldr :: (a -> b -> b) -> b -> t a -> b
  ...
  {-# MINIMAL foldMap | foldr #-}
```

Foldable functions

If we define `foldMap` or `foldr`, the following functions are defined automatically:

```
toList :: t a -> [a]
null :: t a -> Bool
length :: t a -> Int
elem :: Eq a => a -> t a -> Bool
maximum :: Ord a => t a -> a
minimum :: Ord a => t a -> a
sum :: Num a => t a -> a
product :: Num a => t a -> a
```

Foldable instances

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

instance Foldable [] where
  foldMap f = mconcat . map f

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

foldMap :: Monoid m => (a -> m) -> Tree a -> m

instance Foldable Tree where
    foldMap f (Leaf x) = f x
    foldMap f (Node l r) =
      foldMap f l <> foldMap f r
```

Further instances: `Set`, `Map` (foldMap traverses through values)

Summary

- `Semigroup` and `Monoid` are type classes abstracting value aggregation.
- `Foldable` is a type class generalizing `foldr` and `foldl`.

Exams