# Functional programming
# Lecture 05 — Foldable and friends

Stéphane Vialette
stephane.vialette@univ-eiffel.fr

May 14, 2023

Laboratoire d'Informatique Gaspard-Monge, UMR CNRS 8049,
Université Gustave Eiffel

# Semigroup

Semigroup

Monoids

Foldable

# Semigroup

In mathematics, a Semigroup is a set together with an associative operator that combines two elements from the set.

# Semigroup

In mathematics, a Semigroup is a set together with an associative operator that combines two elements from the set.

## Commutative monoid

The set of positive integers $\mathbb{N} = \{1, 2, \dots\}$ is a semigroup under addition.

# Semigroup

In mathematics, a Semigroup is a set together with an associative operator that combines two elements from the set.

## Free semigroup

The set of all finite strings over some fixed alphabet $\Sigma$ forms a semigroup with string concatenation as the operation. The semigroup is called the free semigroup over $\Sigma$.

# Monoid

In Haskell, the notion of a semigroup is captured by the following built-in class declaration (in Prelude since GHC 8.4).

```
type Semigroup :: * -> Constraint
class Semigroup a where
  (<>) :: a -> a -> a
  GHC.Base.sconcat :: GHC.Base.NonEmpty a -> a
  GHC.Base.stimes :: Integral b => b -> a -> a
  {-# MINIMAL (<>) #-}
```

The binary operation <> must be associative :

```
(x <> y) <> z == x <> (y <> z)
```

# Methods

- `(<>) :: a -> a -> a`
  An associative binary operation.

- `sconcat :: GHC.Base.NonEmpty a -> a`
  Take a nonempty list of type `a` and apply the <> operation to all of them to get a single result.

- `stimes :: Integral b => b -> a -> a`
  Given a number `x` and a value of type `a`, combine `x` numbers of the value `a` by repeatedly applying <>.

# Prototypical example

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

```
λ> [1,2,3,4] <> [5,6,7,8]
[1,2,3,4,5,6,7,8]

λ> [1,2,3,4] ++ [5,6,7,8]
[1,2,3,4,5,6,7,8]

λ> Data.Semigroup.stimes 3 "a"
"aaa"

λ> Data.List.replicate 3 'a'
"aaa"
```

# Monoids

Semigroup

Monoids

Foldable

# Monoid

In mathematics, a monoid is a set together with an associative operator that combines two elements from the set, and an identity element for the operator.

# Monoid

In mathematics, a monoid is a set together with an associative operator that combines two elements from the set, and an identity element for the operator.

## Commutative monoid

The set of natural numbers $\mathbb{N} = \{0, 1, 2, \dots\}$ is a commutative monoid under addition (identity element 0) or multiplication (identity element 1).

# Monoid

In mathematics, a monoid is a set together with an associative operator that combines two elements from the set, and an identity element for the operator.

## Free monoid

The set of all finite strings over some fixed alphabet $\Sigma$ forms a monoid with string concatenation as the operation. The empty string serves as the identity element. This monoid is denoted $\Sigma^*$ and is called the free monoid over $\Sigma$.

# Monoid

In Haskell, the notion of a monoid is captured by the following built-in class declaration.

```
type Monoid :: * -> Constraint
class Semigroup a => Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mappend = (<>)
  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
  {-# MINIMAL mempty #-}
```

Identity laws:

```
x <> mempty = x
mempty <> x = x
```

# Trivial monoids

- Lists.
- Sum (`Sum`).
- Product (`Product`).
- Logical and (`And`).
- Logical or (`Any`).
- ...

# Prototypical example

The prototypical and perhaps most important example is lists, which form a monoid under concatenation.

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

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

# Prototypical example

The prototypical and perhaps most important example is lists, which form a monoid under concatenation.

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

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

```
λ> mempty :: [Int]
[]
λ> [1,2,3,4] `mappend` []
[1,2,3,4]
λ> [] `mappend` [1,2,3,4]
[1,2,3,4]
```

# Prototypical example

The prototypical and perhaps most important example is lists, which form a monoid under concatenation.

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

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

```
λ> [1,2,3,4] `mappend` [5,6,7,8]
[1,2,3,4,5,6,7,8]
λ> mconcat [[1,2],[3,4,5],[6,7,8,9]]
[1,2,3,4,5,6,7,8,9]
λ> mconcat [[1,2],[],[3,4,5],[],[],[6,7,8,9]]
[1,2,3,4,5,6,7,8,9]
```

# Product

Monoid under Addition.

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

# Product

Monoid under Addition.

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

instance Num a => Semigroup (Sum a) where
  x <> y =  Sum (getSum x + getSum y)
  -- (<>) = coerce ((+) :: a -> a -> a)

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

# Product

Monoid under Addition.

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

```
λ> Sum 1 <> Sum 2 <> Sum 3 <> Sum 4
Sum {getSum = 10}
λ> getSum $ Sum 1 <> Sum 2 <> Sum 3 <> Sum 4
10
λ> mconcat . map Sum $ [1,2,3,4]
Sum {getSum = 10}
λ> getSum . mconcat . map Sum $ [1,2,3,4]
10
```

# Product

Monoid under multiplication.

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

# Product

Monoid under multiplication.

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

instance Num a => Semigroup (Product a) where
  -- (<>) = coerce ((*) :: a -> a -> a)
  x <> y =  Product (getProduct x * getProduct y)
instance Monoid (Product a) where
  mempty = Product 1
```

# Product

Monoid under multiplication.

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

```
λ> Product 1 <> Product 2 <> Product 3 <> Product 4
Product {getProduct = 24}
λ> getProduct $ Product 1 <> Product 2 <> Product 3 <> Product 4
24
λ> mconcat . map Product $ [1,2,3,4]
Product {getProduct = 24}
λ> getProduct . mconcat . map Product $ [1,2,3,4]
24
```

# Disjunction

Boolean monoid under disjunction (||).

```
newtype Any = Any { getAny :: Bool }
```

# Disjunction

Boolean monoid under disjunction (||).

```haskell
newtype Any = Any { getAny :: Bool }

instance Semigroup Any where
  x <> y =  Any (getAny x || getAny y)
  -- (<>) = coerce (||)

instance Monoid (Any a) where
  mempty = Any False
```

# Disjunction

Boolean monoid under disjunction (||).

```
newtype Any = Any { getAny :: Bool }
```

```
λ>  Any True <> mempty <> Any False
Any {getAny = True}
λ>  getAny $ Any True <> mempty <> Any False
True
λ>  getAny (Any False <> mempty <> Any False)
False
λ>  getAny $ mconcat (map (\x -> Any (even x)) [2,4,6,7,8])
True
```

# Disjunction

Boolean monoid under conjunction (&&).

```
newtype All = All { getAll :: Bool }
```

# Disjunction

Boolean monoid under conjunction (&&).

```haskell
newtype All = All { getAll :: Bool }

instance Semigroup All where
  x <> y =  All (getAll x && getAll y)
  -- (<>) = coerce (&&)

instance Monoid (All a) where
  mempty = All True
```

# Disjunction

Boolean monoid under conjunction (&&).

```
newtype All = All { getAll :: Bool }
```

```
λ >  All True <> mempty <> All False
All {getAll = False}
λ >  getAll $ All True <> mempty <> All False
False
λ >  getAll $ All True <> mempty <> All True
True
λ >  getAll $ mconcat (map (\x -> All (even x)) [2,4,6,7,8])
False
```

# First

Maybe monoid returning the leftmost non-`Nothing` value.

```
newtype First a = First { getFirst :: Maybe a }
```

# First

Maybe monoid returning the leftmost non-`Nothing` value.

```haskell
newtype First a = First { getFirst :: Maybe a }

instance Semigroup (First a) where
  First Nothing <> b = b
  a             <> _ = a

instance Monoid (First a) where
  mempty = First Nothing
```

# First

Maybe monoid returning the leftmost non-`Nothing` value.

```
newtype First a = First { getFirst :: Maybe a }
```

```
λ> First (Just 'A') <> First Nothing <> First (Just 'B')
First {getFirst = Just 'A'}
λ> getFirst $ First (Just 'A') <> First Nothing <> First (Just 'B')
Just 'A'
λ> First Nothing <> First Nothing <> First Nothing
First {getFirst = Nothing}
λ> getFirst $ First Nothing <> First Nothing <> First Nothing
Nothing
```

# Last

Maybe monoid returning the rightmost non-`Nothing` value.

```
newtype Last a = Last { getLast :: Maybe a }
```

# Last

Maybe monoid returning the rightmost non-`Nothing` value.

```
newtype Last a = Last { getLast :: Maybe a }

instance Semigroup (Last a) where
        a <> Last Nothing = a
        _ <> b            = b

instance Monoid (First a) where
  mempty = Last Nothing
```

# Last

Maybe monoid returning the rightmost non-`Nothing` value.

```
newtype Last a = Last { getLast :: Maybe a }
```

```
λ> Last (Just 'A') <> Last Nothing <> Last (Just 'B')
Last {getLast = Just 'B'}
λ> getLast $ Last (Just 'A') <> Last Nothing <> Last (Just 'B')
Just 'B'
λ> Last Nothing <> Last Nothing <> Last Nothing
Last {getLast = Nothing}
λ> getLast $ Last Nothing <> Last Nothing <> Last Nothing
Nothing
```

# Foldable

Semigroup

Monoids

Foldable

# Motivating example

One of the primary applications of monoids in Haskell is to combine all the values in a data structure to give a single value

```
fold :: Monoid a => [a] -> a
fold []     = mempty
fold (x:xs) = x `mappend` fold xs
```

- `fold` provides a simple means of folding up a list using a monoid.
- `fold` bahaves in the same was as `mconcat` from the `Monoid` class (but is definced using explicit recursion rather than using `foldr`).

# Another otivating example

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

fold :: Monoid a => Tree a -> a
fold (Leaf x)     = x
fold (Node tl tr) = fold tl `mappend` fold tr
```

```
λ> fold (Leaf [1])
[1]

λ> fold (Node (Leaf [1]) (Leaf [2]))
[1,2]

λ> fold (Leaf (Just [1]))
Just [1]

λ> fold (Node (Leaf (Just [1])) (Leaf (Just [2])))
Just [1,2]
```

# Foldable

- The idea of folding up the values in data structure using a monoid isn't specific to types such as lists and binary trees, but can be abstracted to a range of parameterized types.
- In Haskell, this concept is captured by the class `Foldable` defined in `Data.Foldable`.

# Foldable

```
type Foldable :: (* -> *) -> Constraint
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
  foldl :: (b -> a -> b) -> b -> t a -> b
  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
  {-# MINIMAL foldMap | foldr #-}
```

# Turning lists to foldables

```haskell
instance Foldable [] where
  -- fold [] = mempty
  -- fold (x:xs) = x `mappend` fold xs

  foldMap _ []     = mempty
  foldMap f (x:xs) = f x `mappend` foldMap f xs

  foldr _ acc []     = acc
  foldr f acc (x:xs) = f x (foldr f acc xs)
```

# Turning lists to foldables

```
λ> fold [[1],[2],[3],[4]]
[1,2,3,4]
λ> foldMap (\x -> [x]) [1,2,3,4]
[1,2,3,4]
λ> foldMap (replicate 1) [1,2,3,4]
[1,2,3,4]
λ> foldMap (replicate 2) [1,2,3,4]
[1,1,2,2,3,3,4,4]
```

# Turning lists to foldables

```
λ> foldMap Sum [1,2,3,4]
Sum {getSum = 10}
λ> getSum $ foldMap Sum [1,2,3,4]
10
λ> foldMap Product [1,2,3,4]
Product {getProduct = 24}
λ> getProduct $ foldMap Product [1,2,3,4]
24
```

# Turning trees to foldables

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

instance Foldable Tree where
  -- foldMap :: Monoid a => Tree a -> a
  foldMap f (Leaf x)     = f x
  foldMap f (Node lt rt) = foldMap f lt `mappend`
                           foldMap f rt

  -- foldr :: (a -> b -> b) -> b -> Tree a -> b
  foldr f acc (Leaf x)     = f x acc
  foldr f acc (Node lt rt) = foldr f (foldr f acc rt) lt
```

# Turning trees to foldables

```
λ > t = Node (Node (Leaf 1) (Leaf 2)) (Node (Leaf 3) (Leaf 4))
λ > foldr (+) 0 t
10
λ > foldr (*) 1 t
24
λ > sum t
10
λ > product t
24
λ > foldr (:) [] t
[1,2,3,4]
```

# Turning trees to foldables

```
λ> t = Node (Node (Leaf 1) (Leaf 2)) (Node (Leaf 3) (Leaf 4))
λ> null t
False
λ> length t
4
λ> minimum t
1
λ> maximum t
4
λ> 3 `elem` t
True
λ> 5 `elem` t
False
```

# Generic functions

```
average :: [Int] -> Int
average xs = sum xs `div` length xs
```

`sum` and `length` are not specific to lists but can be used with any foldable type :

```
average :: Foldable t => t Int -> Int
average xs = sum xs `div` length xs
```

# Generic functions

```
average :: [Int] -> Int
average xs = sum xs `div` length xs
```

`sum` and `length` are not specific to lists but can be used with any foldable type :

```
average :: Foldable t => t Int -> Int
average xs = sum xs `div` length xs
```

As such, `average` can now be be applied to both lists and trees.

```
λ> average [1,2,3,4]
2

λ> average (Node (Node (Leaf 1) (Leaf 2)) (Node (Leaf 3) (Leaf 4)))
2
```

# Generic functions

In a similar way, `Data.Foldable` provides generic versions of a number of familiar functions that operate on lists of logical vlues :

```
and :: Foldable t => t Bool -> Bool
and = getAll . foldMap All

or :: Foldable t => t Bool -> Bool
or : getAny . foldMap Any

all :: Foldable t => (a -> Bool) -> t a -> Bool
all p = getAll . foldMap (All . p)

any :: Foldable t => (a -> Bool) -> t a -> Bool
any p = getAny . foldMap (Any . p)
```

# Generic functions

```
λ> and [True,False,True]
False
λ> and [True,True,True]
True
λ> and (Node (Node (Leaf True) (Leaf False)) (Leaf True))
False
λ> and (Node (Node (Leaf True) (Leaf True)) (Leaf True))
True
```

# Generic functions

```
λ> or [True,False,True]
True
λ> or [False,False,False]
False
λ> or (Node (Node (Leaf True) (Leaf False)) (Leaf True))
True
λ> or (Node (Node (Leaf False) (Leaf False)) (Leaf False))
False
```

# Generic functions

```
λ> all even [1,2,3,4]
False
λ> all (< 5) [1,2,3,4]
True
λ> all even (Node (Node (Leaf 1) (Leaf 2)) (Leaf 3))
False
λ> all (< 5) (Node (Node (Leaf 1) (Leaf 2)) (Leaf 3))
True
```

# Generic functions

```
λ > any even [1,2,3,4]
True
λ > any (> 5) [1,2,3,4]
False
λ > any even (Node (Node (Leaf 1) (Leaf 2)) (Leaf 3))
True
λ > any (> 5) (Node (Node (Leaf 1) (Leaf 2)) (Leaf 3))
False
```