Programare funcțională

Introducere în programarea funcțională folosind Haskell
C09

Ana Iova
Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Monoid

din nou foldr

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

Prelude> foldr (+) 0 [1,2,3]
6
Prelude> foldr (*) 1 [1,2,3]
6
Prelude> foldr (++) [] ["1","2","3"]
"123"
Prelude> foldr (||) False [True, False, True]
True
Prelude> foldr (&&) True [True, False, True]
False
```

Ce au in comun aceste operații?

Monoizi

$(M, \circ, e)$ este monoid dacă

- $\circ : M \times M \to M$ este asociativă
- $m \circ e = e \circ m = m$, oricare $m \in M$

Exemple de monoizi:

(Int, $+$,0), (Int, $*$, 1), (String, $++$, []),
({True,False}, &&, True), ({True,False}, ||, False)

Operația de monoid poate fi generalizată pe liste:

```
sum = foldr (+) 0
product = foldr (*) 1
concat = foldr (++) []
and = foldr (&&) True
or = foldr (||) False
```

Monoizi și semigrupuri

$(M, \circ, e)$ este monoid dacă

- $\circ : M \times M \to M$ este asociativă
- $m \circ e = e \circ m = m$, oricare $m \in M$

Un semigrup este un monoid fără element neutru.

$(M,\circ)$ este semigrup dacă

$\circ : M \times M \to M$ este asociativă.

Exemple

- Orice monoid este și semigrup
- Semigrupul numerelor naturale pozitive, cu adunarea $(\mathbb{N}^*, +)$
- Semigrupul numerelor intregi nenule, cu înmulțirea $(\mathbb{Z}^*, *)$
- Semigrupul listelor nevide, cu concatenarea

clasele Semigroup și Monoid

```haskell
class Semigroup a where
  (<>) :: a -> a -> a      -- operatia asociativa
infixr 6 <>

class Semigroup a => Monoid a where
    mempty  :: a              -- elementul neutru

    mconcat :: [a] -> a       -- generalizarea la liste
    mconcat = foldr (<>) mempty
```

Legi

- Asociativitate: x <> (y <> z) = (x <> y) <> z
- Identitate la dreapta: x <> mempty = x
- Identitate la stânga: mempty <> x = x
- Atenție! Acest lucru este responsabilitatea programatorului!

clasa Monoid

Instanta pentru liste

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

Prelude> mempty :: [a]
[]
Prelude> mconcat [[1,2,3],[4,5],[6]]
[1,2,3,4,5,6]
```

newtype

(Int, $+$,0), (Int, $*$, 1) sunt monoizi

({True,False}, &&, True), ({True,False}, ||, False) sunt monoizi

Cum definim instante diferite pentru acelasi tip?

- se creează o copie a tipului folosind **newtype**
- copia este definită ca instanță a tipului

```haskell
newtype Nat = MkNat Integer
```

- **newtype** se folosește cand un singur constructor este aplicat unui singur tip de date
- declarația cu **newtype** este mai eficientă decât cea cu **data**
- **type** redenumește tipul; **newtype** face o copie și permite redefinirea operațiilor

Bool ca monoid față de conjuncție

```haskell
newtype All = All { getAll :: Bool }
    deriving (Eq, Show)
instance Semigroup All where
    All x <> All y = All (x && y)
instance Monoid All where
    mempty = All True
```

Bool ca monoid față de disjuncție

```haskell
newtype Any = Any { getAny :: Bool }
    deriving (Eq, Show)
instance Semigroup Any where
    Any x <> Any y = Any (x || y)
instance Monoid Any where
    mempty = Any False
```

clasa Monoid

Num a ca monoid față de adunare

```
newtype Sum a = Sum { getSum :: a }
    deriving (Eq, Show)
instance Num a => Semigroup (Sum a) where
    Sum x <> Sum y = Sum (x + y)
instance Num a => Monoid (Sum a) where
    mempty = Sum 0
```

Num a ca monoid față de înmulțire

```
newtype Product a = Product { getProduct :: a }
    deriving (Eq, Show)
instance Num a => Semigroup (Product a) where
    Product x <> Product y = Product (x * y)
instance Num a => Monoid (Product a) where
    mempty = Product 1
```

Ord a ca semigrup față de operația de minim

```haskell
newtype Min a = Min { getMin :: a }
    deriving (Eq, Show)
instance Ord a => Semigroup (Min a) where
    Min x <> Min y = Min (min x y)
instance (Ord a, Bounded a) => Monoid (Min a) where
    mempty = Min maxBound
```

Ord a ca semigrup față de operația de maxim

```haskell
newtype Max a = Max { getMax :: a }
    deriving (Eq, Show)
instance Ord a => Semigroup (Max a) where
    Max x <> Max y = Max (max x y)
instance (Ord a, Bounded a) => Monoid (Max a) where
    mempty = Max minBound
```

```
Prelude> Sum 3
Sum {getSum = 3}
Prelude> Sum 3 <> Sum 4
Sum {getSum = 7}
Prelude> Product 3 <> Product 4
Product {getProduct = 12}
Prelude> mconcat [Any False,Any True,Any False]
Any {getAny = True}
Prelude> (getSum . mconcat) [Sum 3,Sum 4,Sum 5]
12
Prelude> getMax . mconcat . map Max $ [3,5,4]
5
```

Monoid Maybe

```
instance Semigroup a => Semigroup (Maybe a) where
    Nothing <> m        =  m
    m       <> Nothing  =  m
    Just m1 <> Just m2  =  Just (m1 <> m2)

instance Semigroup a => Monoid (Maybe a) where
    mempty =  Nothing
```

```
Prelude> Nothing <> (Just 3) :: Maybe Integer
<interactive>:35:1: error:

Prelude> Nothing <> (Just (Sum 3))
Just (Sum {getSum = 3})
```

Semigroup

Tipul listelor nevide

```
data NonEmpty a = a :| [a]      deriving (Eq, Ord)
instance Semigroup (NonEmpty a) where
    (a :| as) <> (b :| bs) = a :| (as ++ b : bs)
```

Concatenare pentru semigrupuri

```
sconcat :: Semigroup a => NonEmpty a -> a
sconcat (a :| as) = go a as
  where
    go a [] = a
    go a (b : bs) = a <> go b bs
```

```
Prelude>sconcat $ (Sum 1) :| [(Sum 2),(Sum 3)]
Sum {getSum = 6}
```

Foldable

din nou foldr

foldr pe liste

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f i [] = i
foldr f i (x:xs) = f x (foldr f i xs)
```

Problema: să generalizăm **foldr** la alte structuri recursive.

Exemplu: arbori binari

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

Cum definim **"foldr"** înlocuind listele cu date de tip **BinaryTree** ?

"foldr" folosind BinaryTree

```
data BinaryTree a =
        Leaf a
        | Node (BinaryTree a) (BinaryTree a)
        deriving Show

foldTree :: (a -> b -> b) -> b -> BinaryTree a -> b
foldTree f i (Leaf x) = f x i
foldTree f i (Node l r) = foldTree f (foldTree f i r) l


myTree = Node (Node (Leaf 1)(Leaf 2))(Node (Leaf 3)(
    Leaf 4))
Prelude> foldTree (+) 0 myTree
10
```

clasa Foldable

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

        fold = foldMap id
        ...
```

Observații:

- definiția minimală completă conține fie **foldMap**, fie **foldr**
- **foldMap** și **foldr** pot fi definite una prin cealaltă
- pentru a crea o instanță este suficient să definim una dintre **foldMap** și **foldr**, cealaltă va fi automat accesibilă

```
instance Foldable BinaryTree where
  foldr = foldTree

treeI = Node(Node(Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf
    4))
treeS = Node (Node(Leaf "1")(Leaf "2"))
             (Node (Leaf "3")(Leaf "4"))

Prelude> foldr (+) 0 treeI
10
Prelude> foldr (++) [] treeS
"1234"
```

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

        fold = foldMap id
        ...

instance Foldable BinaryTree where
   foldr = foldTree
```

Observație: în definiția clasei **Foldable**, variabila de tip t nu reprezintă un tip concret ([a], Sum a), ci un constructor de tip (BinaryTree)

Foldable cu foldr

Avem definite automat **foldMap** și alte funcții precum: **foldl, foldr’,foldr1,...**

```
Prelude> foldl (++) [] treeS
"1234"
Prelude> foldl (+) 0 treeI
10
Prelude> maximum treeI
4
Prelude> foldMap Sum treeI
Sum {getSum = 10}
Prelude> foldMap id treeS
"1234"
```

foldMap

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

newtype Sum a = Sum { getSum :: a }
                        deriving (Eq, Show)
instance Num a => Semigroup (Sum a) where
        Sum x <> Sum y = Sum (x + y)
instance Num a => Monoid (Sum a) where
        mempty = Sum 0

tree1 = Node(Node(Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf 4))

Prelude> foldMap Sum tree1    -- Sum :: a -> Sum a
Sum {getSum = 10}
```

sum cu foldMap

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

newtype Sum a = Sum { getSum :: a }
                        deriving (Eq, Show)
instance Num a => Semigroup (Sum a) where
        Sum x <> Sum y = Sum (x + y)
instance Num a => Monoid (Sum a) where
        mempty = Sum 0

sum as = getSum $ foldMap Sum as
-- sum = getSum . (foldMap Sum)

tree1 = Node(Node(Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf 4))
Prelude> foldMap Sum tree1    -- Sum :: a -> Sum a
Sum {getSum = 10}
Prelude> sum tree1
10
```

product cu foldMap

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

newtype Product a = Product { getProduct :: a }
        deriving (Eq, Show)
instance Num a => Semigroup (Product a) where
        Product x <> Product y = Product (x * y)
instance Num a => Monoid (Product a) where
        mempty = Product 1

product as = getProduct $ foldMap Product as
-- product = getProduct . (foldMap Product)

treeI = Node(Node(Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf 4))
Prelude> foldMap Product treeI
Product {getProduct = 24}
Prelude> product treeI
24
```

elem cu foldMap

```
foldMap :: Monoid m => (a -> m) -> t a -> m
newtype Any = Any { getAny :: Bool }
        deriving (Eq, Show)
instance Semigroup Any where
        Any x <> Any y = Any (x || y)
instance Monoid Any where
        mempty = Any False

any as = getAny $ foldMap Any as
-- any = getAny . (foldMap Any)
elem e = getAny . (foldMap (Any . (== e)))

tree1 = Node(Node(Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf 4))
Prelude> foldMap (Any . (== 1)) tree1
Any {getAny = True}
Prelude> elem 1  tree1
 True
```

Cum definim **foldMap** folosind **foldr**?

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

```haskell
foldMap f tr = foldr foo i tr      -- f :: a -> m
           where foo = ???         -- foo :: (a -> m -> m)
                 i = mempty
```

foldMap folosind foldr

Cum definim **foldMap** folosind **foldr**?

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

```haskell
foldMap f tr = foldr foo i tr      -- f :: a -> m
          where foo = ???         -- foo :: (a -> m -> m)
                i = mempty
```

```haskell
foo = \x acc -> f x <> acc
    = \x acc -> (<>) (f x) acc
    = \x -> (<>) $ f x
    = \x -> ((<>) . f)
    = (<>) . f
```

foldMap f = foldr ((<>) . f) mempty

Foldable cu foldMap

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

treeI = Node(Node(Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf
    4))
treeS = Node (Node(Leaf "1")(Leaf "2"))
             (Node (Leaf "3")(Leaf "4"))
```

Avem definite automat **foldr** și alte funcții precum: **foldl, foldr',foldr1,...**

```
Prelude> foldr (++) [] treeS
"1234"
Prelude> foldl (+) 0 treeI
10
```

foldr folosind foldMap - facultativ

Cum definim **foldr** folosind **foldMap**?

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

Idee

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

- pentru fiecare element de tip **a** din **t a** se crează o funcție de tip **(b->b)**

  obținem, de exemplu, o lista de funcții sau
  un arbore care are ca frunze funcții

- folosim faptul ca **(b->b)** este instanță a lui **Monoid** și aplicăm **foldMap**

Quiz time!

Seria 23: `https://questionpro.com/t/AT4qgZv37w`

Seria 24: `https://questionpro.com/t/AT4qgZv68m`

Seria 25: `https://questionpro.com/t/AT4qgZv37u`

Pe săptămâna viitoare!