Programare declarativă

Monoid, Foldable

Traian Florin Șerbănuță
Ioana Leuștean

Departamentul de Informatică, FMI, UB
traian.serbanuta@fmi.unibuc.ro
ioana@fmi.unibuc.ro

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 $m \in M$

Monoizi

$(M, \circ, e)$ este monoid dacă
$\circ : M \times M \to M$ este asociativă
$m \circ e = e \circ m = m$ oricare $m \in M$

Observații:

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

Monoizi

$(M, \circ, e)$ este monoid dacă
$\circ : M \times M \to M$ este asociativă
$m \circ e = e \circ m = m$ oricare $m \in M$

Observații:

- $(Int, +, 0)$, $(Int, *, 1)$, $(String, ++, [])$, $(\{True, False\}, \&\&, True)$ sunt monoizi
- Operația de monoid poate fi generalizată pe liste:

```
sum = foldr (+) 0
product = foldr (*) 1
concat = foldr (++) []
all = foldr (&&) True
```

clasa **Monoid**

https://en.wikibooks.org/wiki/Haskell/Monoids//https://hackage.haskell.org/package/base-4.10.0.0/docs/Data-Monoid.html

Data.Monoid

```
class Monoid a where
    mempty  :: a              -- elementul neutru
    mappend :: a -> a -> a    -- operatia de monoid

    mconcat :: [a] -> a       -- generalizarea la liste
    mconcat = foldr mappend mempty
```

Observație: În loc de *mappend* se poate folosi (<>)

```
infixr 6 <>
(<>) :: Monoid m => m -> m -> m
(<>) = mappend        -- notatie infixa
```

clasa **Monoid**

Legile monoizilor

Instanțele clasei **Monoid** trebuie să satisfacă următoarele ecuații:

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

Atenție! Acest lucru este responsabilitatea programatorului!

clasa **Monoid**

Legile monoizilor

Instanțele clasei **Monoid** trebuie să satisfacă următoarele ecuații:

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

Atenție! Acest lucru este responsabilitatea programatorului!

Listele ca instanța

```
instance Monoid [a] where
     mempty  = []
     mappend = (++)

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

clasa **Monoid**

(Int, $+$,0), (Int, $*$, 1) sunt monoizi
({True,False}, &&, True), ({True,False}, ||, False) sunt monoizi

Cum definim instante diferite pentru acelasi tip?

clasa **Monoid**

(Int, $+$,0), (Int, $*$, 1) sunt monoizi
({True,False}, &&, True), ({True,False}, ||, False) sunt monoizi

Cum definim instante diferite pentru acelasi tip?

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

newtype

```
newtype Nat = MkNat Integer
```

- **newtype** se folosește cînd 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

clasa **Monoid**

- **Num a** ca monoid față de adunare

```
newtype Sum a = Sum { getSum :: a }
                    deriving (Eq, Read, Show)

instance Num a => Monoid (Sum a) where
    mempty = Sum 0
    Sum x `mappend` Sum y = Sum (x + y)
```

- **Num a** ca monoid față de înmulțire

```
newtype Product a = Product { getProduct :: a }
                         deriving (Eq, Read, Show)

instance Num a => Monoid (Product a) where
    mempty = Product 1
    Product x `mappend` Product y = Product (x * y)
```

clasa **Monoid**

```
Prelude> Sum 3
<interactive>:15:1: error:

Prelude> :m + Data.Monoid
Prelude Data.Monoid> Sum 3
Sum {getSum = 3}
Prelude Data.Monoid> Sum 3 <> Sum 4
Sum {getSum = 7}
Prelude Data.Monoid> Sum 3 + Sum 4
Sum {getSum = 7}
Prelude Data.Monoid> mconcat [Sum 3,Sum 4,Sum 5]
Sum {getSum = 12}
Prelude Data.Monoid> (getSum . mconcat) [Sum 3,Sum 4,Sum 5]
12
Prelude Data.Monoid> (getSum . mconcat) $ map Sum [3,4,5]
12
Prelude Data.Monoid> getSum . mconcat . (map Sum) $ [3,4,5]
12
```

Monoid Maybe

```haskell
instance Monoid a => Monoid (Maybe a) where
    mempty                          =  Nothing
    Nothing `mappend` m             =  m
    m       `mappend` Nothing       =  m
    Just m1 `mappend` Just m2       =  Just (m1 `mappend` m2)
```

Atentie! Monoid a => este o constrangere de tip

```haskell
Prelude Data.Monoid> Nothing `mappend` (Just 3)
```

<interactive>:35:1: error:

```haskell
Prelude Data.Monoid> Nothing `mappend` (Just (Sum 3))
Just (Sum {getSum = 3})
```

Funcții ca instanțe

(a -> a) ca instanța a clasei Monoid

```
newtype Endo a = Endo { appEndo :: a -> a }

instance Monoid (Endo a) where
    mempty                   = Endo id
    Endo g `mappend` Endo f = Endo (g . f)
```

Funcții ca instanțe

(a -> a) ca instanța a clasei **Monoid**

```
newtype Endo a = Endo { appEndo :: a -> a }

instance Monoid (Endo a) where
    mempty                  = Endo id
    Endo g `mappend` Endo f = Endo (g . f)

Prelude> :m + Data.Monoid
>let f = mconcat [Endo (+1), Endo (+2), Endo (+3)]
>:t f
f :: Num a => Endo a
```

Funcții ca instanțe

(a -> a) ca instanța a clasei Monoid

```
newtype Endo a = Endo { appEndo :: a -> a }

instance Monoid (Endo a) where
    mempty                    = Endo id
    Endo g `mappend` Endo f   = Endo (g . f)

Prelude> :m + Data.Monoid
>let f = mconcat [Endo (+1), Endo (+2), Endo (+3)]
>:t f
f :: Num a => Endo a

> (appEndo f) 0
6
> (appEndo . mconcat) [Endo (+1), Endo (+2), Endo (+3)] $ 0
6
```

Foldable

din nou **foldr**

foldr pe liste

```
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

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

din nou **foldr**

foldr pe liste

```
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

```
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

```
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
```

foldTree

```
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))

*Main> foldTree (+) 0 myTree
10
```

clasa **Foldable**

https://en.wikibooks.org/wiki/Haskell/Foldable
https://hackage.haskell.org/package/base-4.10.0.0/docs/Data-Foldable.html

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

Foldable cu foldr

```
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"))

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

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
        ...
```

```
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

```
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"))
```

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

```
*Main> foldl (++) [] treeS
"1234"
*Main> foldl (+) 0 treeI
10
```

Foldable cu foldr

```
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"))
```

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

```
*Main> foldl (++) [] treeS
"1234"
*Main> foldl (+) 0 treeI
10

*Main Data.Monoid> foldMap Sum treeI
Sum {getSum = 10}
*Main Data.Monoid> foldMap id treeS
"1234"
```

foldMap

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

```
newtype Sum a = Sum { getSum :: a }
                deriving (Eq, Read, Show)

instance Num a => Monoid (Sum a) where
    mempty = Sum 0
    Sum x `mappend` Sum y = Sum (x + y)

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

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

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

foldMap folosind foldr

http://cmsc-16100.cs.uchicago.edu/2016/Lectures/13-monoid-foldable.php

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

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

foldMap folosind foldr

http://cmsc-16100.cs.uchicago.edu/2016/Lectures/13-monoid-foldable.php

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

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

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

foldMap folosind foldr

http://cmsc-16100.cs.uchicago.edu/2016/Lectures/13-monoid-foldable.php

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

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

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

foldMap f = foldr (mappend . 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,...**

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

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,...**

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

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

foldr folosind **foldMap**

https://en.wikibooks.org/wiki/Haskell/Foldable

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

foldr folosind **foldMap**

https://en.wikibooks.org/wiki/Haskell/Foldable

```
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**

foldr folosind **foldMap**

https://en.wikibooks.org/wiki/Haskell/Foldable

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

(b->b) instanță a lui **Monoid**

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

foldr folosind **foldMap**

https://en.wikibooks.org/wiki/Haskell/Foldable

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

(b->b) instanță a lui **Monoid**

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

Definim funcția ajutătoare

```
foldComposing :: (a -> (b -> b)) -> t a -> Endo b
```

astfel încât

```
foldr f i tr = appEndo (foldComposing f tr) $ i
```

foldr folosind **foldMap**

https://en.wikibooks.org/wiki/Haskell/Foldable

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

foldr folosind foldMap

https://en.wikibooks.org/wiki/Haskell/Foldable

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

foldComposing f = foldMap (Endo . f)
```

foldr folosind foldMap

https://en.wikibooks.org/wiki/Haskell/Foldable

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

foldComposing f = foldMap (Endo . f)
```

Exemplu:

```
foldComposing (+) [1, 2, 3]
foldMap (Endo . (+)) [1, 2, 3]
(Endo . (+)) 1 <> (Endo . (+)) 2 <> (Endo . (+)) 3
Endo (+1) <> Endo (+2) <> Endo (+3)
Endo ((+1) . (+2) . (+3))
Endo (+6)
```

foldr folosind **foldMap**

https://en.wikibooks.org/wiki/Haskell/Foldable

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

foldComposing f = foldMap (Endo . f)
```

Exemplu:

```
foldComposing (+) [1, 2, 3]
foldMap (Endo . (+)) [1, 2, 3]
(Endo . (+)) 1 <> (Endo . (+)) 2 <> (Endo . (+)) 3
Endo (+1) <> Endo (+2) <> Endo (+3)
Endo ((+1) . (+2) . (+3))
Endo (+6)
```

```
foldr f i tr = appEndo (foldComposing f tr) $ i
```