Programare declarativă

Foldable

Traian Florin Șerbănuță
Ioana Leuștean

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

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)

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

*Main> foldMap Sum treel    -- 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


treel = 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 treel
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


treel = 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 treel
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 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 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
```

Exemplu

```
data Tree a = Null
            |TNode { value :: a , children :: [Tree a] }
                  deriving Show

instance Foldable Tree where
      foldMap f Null = mempty
      foldMap f (TNode val xs) = foldr mappend (f val)
                                       [foldMap f x | x <- xs]
```

Exemplu

```
data Tree a = Null
            |TNode { value :: a , children :: [Tree a] }
                  deriving Show

instance Foldable Tree where
      foldMap f Null = mempty
      foldMap f (TNode val xs) = foldr mappend (f val)
                                 [foldMap f x | x <- xs]
```

```
*Main> let f k s = concat (replicate k s)
*Main> foldr f "a" [1,2,3]
"aaaaaa"

*Main> arb = TNode 1 [TNode 2 [Null] , TNode 3 [Null]]
*Main> foldr f "a" arb
"aaaaaa"
```