Programare declarativă

Functori, Monoid, Foldable

Ioana Leuștean
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Cutii și computații

Tipuri parametrizate — „cutii”

Idee

O clasă largă de tipuri parametrizate pot fi gândite ca „cutii”, recipiente care pot conține elemente de tipul dat ca argument.

Exemple

- Clasa de tipuri opțiune asociază unui tip a, tipul **Maybe** a
  - cutii goale: **Nothing**
  - cutii care țin un element x de tip a: **Just** x
- Clasa de tipuri listă asociază unui tip a, tipul [a]
  - cutii care țin 0, 1, sau mai multe elemente de tip a: [1, 2, 3], [], [5]

Tipuri parametrizate — „cutii”

Idee

O clasă largă de tipuri parametrizate pot fi gândite ca „cutii”, recipiente care pot conține elemente de tipul dat ca argument.

Exemplu: tip de date pentru arbori binari

```haskell
data Arbore a = Nil
              | Nod a Arbore Arbore
```

- Un arbore este o „cutie” care poate ține 0, 1, sau mai multe elemente de tip a:
  Nod 3 Nil (Nod 4 (Nod 2 Nil Nil) Nil), Nil, Nod 3 Nil Nil

Generalizare: Tipuri parametrizate — „computații”

Idee

O clasă largă de tipuri parametrizate pot fi gândite ca „contexte computaționale”: computații care, atunci când se execută, pot produce rezultate de tipul dat ca argument.

Exemple

- **Maybe** a descrie rezultate de computații deterministe care pot eșua
  - computații care eșuează: **Nothing**
  - computații care produc un element de tipul dat: **Just** 4
- [**Int**] descrie liste de rezultate posibile ale unor computații nedeterministe
  - care pot produce oricare dintre rezultatele date: [1, 2, 3], [] , [5]

Tipuri parametrizate — „computații”

Idee

O clasă largă de tipuri parametrizate pot fi gândite ca „contexte computaționale”: computații care, atunci când se execută, pot produce rezultate de tipul dat ca argument.

Exemple

- **Either** e a descrie rezultate de tip a ale unor computații deterministe care pot eșua cu o eroare de tip *e*
  - **Right** 5 :: **Either** e **Int** reprezintă rezultatul unei computații reușite
  - **Left** "OOM":: **Either String** a reprezintă o excepție de tip **String**

Tipuri parametrizate — „computații”

Idee

O clasă largă de tipuri parametrizate pot fi gândite ca „contexte computaționale”: computații care, atunci când se execută, pot produce rezultate de tipul dat ca argument.

Exemplu: tipul funcțiilor de sursă dată

- t –> a descrie computații care atunci când primesc o intrare de tip t produc un rezultat de tip a
  - (++ "!") :: **String** –> **String** este o computație care dat fiind un șir, îi adaugă un semn de exclamare
  - **length** :: **String** –> **Int** este o computație care dat fiind un șir, îi prduce lungimea acestuia
  - **id** :: **String** –> **String** este o computație care produce șirul dat ca argument

Clase de tipuri pentru cutii și computații?

Întrebare

Care sunt trăsăturile comune ale acestor tipuri parametrizate care pot fi gândite intuitiv ca cutii care conțin elemente / computații care produc rezultate?

Problemă

Putem proiecta clase de tipuri care descriu funcționalități comune tuturor acestor tipuri?

Functori

Problemă

Formulare cu cutii

Dată fiind o funcție f :: a −> b și o cutie ca care conține elemente de tip a, vreau să să obțin o cutie cb care conține elemente de tip b obținute prin transformarea elementele din cutia ca folosind funcția f (și doar atât!)

Formulare cu computații

Dată fiind o funcție f :: a −> b și o computație ca care produce rezultate de tip a, vreau să să obțin o computație cb care produce rezultate de tip b obținute prin transformarea rezultatelor produse de computația ca folosind funcția f (și doar atât!)

Exemplu — liste

Dată fiind o funcție f :: a −> b și o listă *la* de elemente de tip a, vreau să să obțin o lista de elemente de tip b transformând fiecare element din *la* folosind funcția f (și doar atât!)

Clasa de tipuri Functor

Definiție

```
class Functor m where
  fmap :: (a -> b) -> m a -> m b
```

Dată fiind o funcție f :: a -> b și ca :: m a, fmap produce cb :: m b obținută prin transformarea rezultatelor produse de computația ca folosind funcția f (și doar atât!)

Instanță pentru liste

```
instance Functor [] where
  fmap = map
```

Clasa de tipuri Functor

Instanțe

```
class Functor f where
  fmap :: (a -> b) -> m a -> m b
```

Instanță pentru tipul optiune fmap :: (a –> b) –> **Maybe** a –> **Maybe** b

Instanță pentru tipul arbore fmap :: (a –> b) –> Arbore a –> Arbore b

Clasa de tipuri Functor

Instanțe

```
class Functor f where
  fmap :: (a -> b) -> m a -> m b
```

Instanță pentru tipul optiune fmap :: (a -> b) -> **Maybe** a -> **Maybe** b

```
instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

Instanță pentru tipul arbore fmap :: (a -> b) -> Arbore a -> Arbore b

```
instance Functor Arbore where
  fmap f Nil = Nil
  fmap f (Nod x l r) = Nod (f x) (fmap f l) (fmap f r)
```

Clasa de tipuri Functor

Instanțe

```
class Functor f where
  fmap :: (a -> b) -> m a -> m b
```

Instanță pentru tipul eroare fmap :: (a -> b) -> **Either** e a -> **Either** e b

Instanță pentru tipul funcție fmap :: (a -> b) -> (t -> a) -> (t -> b)

Clasa de tipuri Functor

Instanțe

```
class Functor f where
  fmap :: (a -> b) -> m a -> m b
```

Instanță pentru tipul eroare fmap :: (a –> b) –> **Either** e a –> **Either** e b

```
instance Functor (Either e) where
    fmap _ (Left x) = Left x
    fmap f (Right y) = Right (f y)
```

Instanță pentru tipul funcție fmap :: (a –> b) –> (t –> a) –> (t –> b)

```
instance Functor (->) a where
  fmap f g = f . g  -- sau, mai simplu, fmap = (.)
```

Exemple

```
Main> fmap (*2) [1..3]

Main> fmap (*2) (Just 200)

Main> fmap (*2) Nothing

Main> fmap (*2) (+100) 4

Main> fmap (*2) (Right 6)

Main> fmap (*2) (Left 1)
```

Exemple

```
Main> fmap (*2) [1..3]
[2,4,6]
Main> fmap (*2) (Just 200)
Just 400
Main> fmap (*2) Nothing
Nothing
Main> fmap (*2) (+100) 4
208
Main> fmap (*2) (Right 6)
Right 12
Main> fmap (*2) (Left 135)
Left 135
```

Proprietăți ale functorilor

- Argumentul m al lui **Functor** m definește o transformare de tipuri
  - m a este tipul a transformat prin functorul m
- fmap definește transformarea corespunzătoare a funcțiilor
  - fmap :: (a -> b) -> (m a -> m b)

Contractul lui fmap

- fmap f ca e obținută prin transformarea rezultatelor produse de computația ca folosind funcția f (și doar atât!)
- Abstractizat prin două legi:

  identitate fmap **id** == **id**

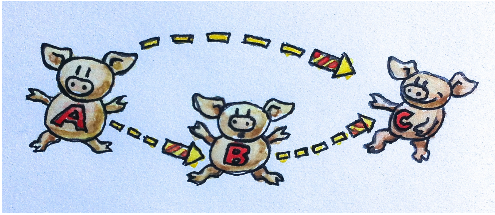
  compunere fmap (g . f) == fmap g . fmap f

Categorii și Functori

Categorii

O categorie $\mathbb{C}$ este dată de:

- O clasă $|\mathbb{C}|$ a obiectelor
- Pentru oricare două obiecte $A, B \in |\mathbb{C}|$,
  o mulțime $\mathbb{C}(A, B)$ a săgeților „de la A la B"
  $f \in \mathbb{C}(A, B)$ poate fi scris ca $f : A \to B$
- Pentru orice obiect A o săgeată $id_A : A \to A$ numită identitatea lui A
- Pentru orice obiecte A, B, C, o operație de compunere a săgeților
  $\circ : \mathbb{C}(B, C) \times \mathbb{C}(A, B) \to \mathbb{C}(A, C)$



Bartosz Milewski — Category: The Essence of Composition

- Compunerea este asociativă și are element neutru id

Exemplu: Categoria $\mathbb{S}\mathrm{et}$

- Obiecte: mulțimi
- Săgeți: funcții
- Identități: Funcțiile identitate
- Compunere: Compunerea funcțiilor

Exemplu: Categoria $\mathbb{H}$ask

- Obiectele: tipuri
- Săgețiile: funcții între tipuri

```
f :: A -> B
```

- Identități: funcția polimorfică **id**

```
Prelude> :t id
id :: a -> a
```

- Compunere: funcția polimorfică (.)

```
Prelude> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Subcategorii ale lui Hask date de tipuri parametrizate

- Obiecte: o clasă restânsă de tipuri din $|\mathbb{H}\mathrm{ask}|$
  - Exemplu: tipuri de forma [a]
- Săgeți: toate funcțiile din $\mathbb{H}$ask între tipurile obiecte
  - Exemple: **concat** :: [[a]] −> [a], **words** :: [**Char**] −> [**String**], **reverse** :: [a] −> [a]

Exemple

Liste	obiecte: tipuri de forma [a]
Optiuni	obiecte: tipuri de forma Maybe a
Arbori	obiecte: tipuri de forma Arbore a
Funcții de sursă t	obiecte: tipuri de forma t -> a

De ce categorii?

(Des)compunerea este esența programării

- Am de rezolvat problema P
- O descompun în subproblemele $P_1, \ldots P_n$
- Rezolv problemele $P_1, \ldots P_n$ cu programele $p_1, \ldots p_n$
  - Eventual aplicând recursiv procedura de față
- Compun rezolvările $p_1, \ldots p_n$ într-o rezolvare p pentru problema inițială

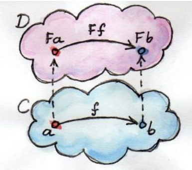
Categoriile rezolvă problema compunerii

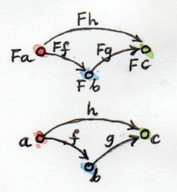
- Ne forțează să abstractizăm datele
- Se poate acționa asupra datelor doar prin săgeți (metode?)
- Forțează un stil de compunere independent de structura obiectelor

Functori

Date fiind două categorii $\mathbb{C}$ și $\mathbb{D}$, un functor $F : \mathbb{C} \to \mathbb{D}$ este dat de

- O funcție $F : |\mathbb{C}| \to |\mathbb{D}|$ de la obiectele lui $\mathbb{C}$ la cele ale lui $\mathbb{D}$
- Pentru orice $A, B \in |\mathbb{C}|$, o funcție $F : \mathbb{C}(A, B) \to \mathbb{D}(F(A), F(B))$
- Compatibilă cu identitățile și cu compunerea
    - $F(id_A) = id_{F(A)}$ pentru orice A
    - $F(g \circ f) = F(g) \circ F(f)$ pentru orice $f : A \to B, g : B \to C$, $h = g \circ f$





Bartosz Milewski — Functors

Functori în Haskell

În general un functor $F : \mathbb{C} \to \mathbb{D}$ este dat de

- O funcție $F : |\mathbb{C}| \to |\mathbb{D}|$ de la obiectele lui $\mathbb{C}$ la cele ale lui $\mathbb{D}$
- Pentru orice $A, B \in |\mathbb{C}|$, o funcție $F : \mathbb{C}(A, B) \to \mathbb{D}(F(A), F(B))$
- Compatibilă cu identitățile și cu compunerea
  - $F(id_A) = id_{F(A)}$ pentru orice A
  - $F(g \circ f) = F(g) \circ F(f)$ pentru orice $f : A \to B, g : B \to C$, $h = g \circ f$

În Haskell o instanță **Functor** m este dată de

- Un tip m a pentru orice tip a (deci m trebuie sa fie tip parametrizat)
- Pentru orice două tipuri a și b, o funcție

```
fmap :: (a -> b) -> (m a -> m b)
```

- Compatibilă cu identitățile și cu compunerea

```
fmap id == id
fmap (g . f) == fmap g . fmap f
```

pentru orice f :: a -> b și g :: b -> c

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 $m \in M$

Exemple de monoizi

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

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

Monoid

(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 monoid 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**

https://hackage.haskell.org/package/base/docs/Prelude.html#t:Semigroup

```
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**

Exemple

Listele ca instanța

```
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]
```

clasa **Monoid**

Exemple

Listele ca instanța

```
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]
```

Mai multe instanțe pentru același tip?

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

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

Problemă: 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?

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

```haskell
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**

All și Any

- **Bool** ca monoid față de conjuncție

```haskell
newtype All = All { getAll :: Bool }
    deriving (Eq, Read, 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, Read, Show)

instance Semigroup Any where
    Any x <> Any y = Any (x || y)
instance Monoid Any where
    mempty = Any False
```

clasa **Monoid**

Sum și Product

- **Num a** ca monoid față de adunare

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

clasa **Monoid**

Min și Max

- **Ord a** ca semigrup față de operația de minim

```
newtype Min a = Min { getMin :: a }
    deriving (Eq, Read, 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

```
newtype Max a = Max { getMax :: a }
    deriving (Eq, Read, 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
```

clasa **Monoid**

Exemple

```
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> Product 3 <> Product 4
Product {getProduct = 12}
Prelude Data.Monoid> mconcat [Any False,Any True,Any False]
Any {getAny = True}
Prelude Data.Monoid> (getSum . mconcat) [Sum 3,Sum 4,Sum 5]
12
Prelude Data.Monoid> getMax . mconcat . map Product $
    [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 Data.Monoid> Nothing <> (Just 3) :: Maybe Integer
<interactive>:35:1: error:

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

Funcții ca instanțe

(a -> a) ca instanța a clasei Monoid

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

instance Monoid Endo where
    mempty            = Endo id
    Endo g <> 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 where
    mempty            = Endo id
    Endo g <> 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 where
    mempty              = Endo id
    Endo g <> 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
```

Semigroup
NonEmpty

Tipul listelor nevide

```
data NonEmpty a = a :| [a]      deriving (Eq, Ord)

instance Semigroup (NonEmpty a) where
    (a :| as) <> (b :| bs) = a :| (as ++ b : bs)
```

Semigroup

NonEmpty

Tipul listelor nevide

```haskell
data NonEmpty a = a :| [a]      deriving (Eq, Ord)

instance Semigroup (NonEmpty a) where
    (a :| as) <> (b :| bs) = a :| (as ++ b : bs)
```

Concatenare pentru semigrupuri

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

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/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
*Main> maximum treeI
4
```

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> maximum treeI
4

*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 <> 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}
```

sum cu 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 <> Sum y = Sum (x + y)

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

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

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

product cu foldMap

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

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

*Main> foldMap Product treeI
Product {getProduct = 24}
*Main> product treeI
 24
```

elem cu foldMap

```
foldMap :: Monoid m => (a -> m) -> t a -> m
newtype Any = Any { getAny :: Bool }
    deriving (Eq, Read, 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))

*Main> foldMap (Any . (== 1)) tree1
Any {getAny = True}
*Main> elem 1 tree1
 True
```

prod cu 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 <> Sum y = Sum (x + y)


sum as = getsum $ foldMap Sum as


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


*Main> foldMap Sum treeI   -- Sum :: a -> Sum a
```

foldMap folosind **foldr**

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

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

```
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

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

```
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

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

```
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 ((<>) . 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
```

foldr folosind **foldMap**

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

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

```
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

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**

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 <> 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 <> 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
```