Programare declarativă

Monoid

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

```
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

```
Prelude Data.Monoid> Nothing `mappend` (Just 3)
<interactive>:35:1: error:
```

```
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 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 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 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
```