Класи типів Haskell: Semigroup Monoid (напівгрупи і моноїди)

Юрій Стативка

Лютий, 2023 р.

Зміст

Розділ 1

Класи Semigroup і Monoid

1.1 Алгебраїчні структури

1.1.1 Теоретичне підґрунтя

В математиці *алгебраїчна структура* – це непорожня множина із заданим на ній набором операцій та відношень, що задовільняють деякій системи аксіом. Ми розглянемо структури з однією бінарною асоціативною операцією, яку позначатимемо символом $\otimes$.

Напівгрупа[1] – це непорожня множина з визначеною на ній асоціативною бінарною операцією $(S, \otimes)$

$$\forall x, y, z \in S : \ x \otimes (y \otimes z) = (x \otimes y) \otimes z$$

Напівгрупи, в яких існує нейтральний елемент, виокремлюють як окрему структуру з власною назвою:

Моноїд[2] - це напівгрупа з нейтральним елементом $(M, \otimes, e)$

$$\forall x \in M : \ x \otimes e = e \otimes x = x$$

Якщо ж у напівгрупі з нейтральним елементом (у моноїді), крім того, для кожного елемента знайдеться обернений, тоді її називають групою:[3]

Група - це моноїд $(G, \otimes, e)$, в якому кожний елемент має обернений

$$\forall x \in G \quad \exists x^{-1} \in G : \ x \otimes x^{-1} = x^{-1} \otimes x = e$$

Далі будемо розглядати тільки напівгрупи та моноїди, оскільки у `Haskell` визначені відповідні класи типів.[4]

---

[1] Англійською – semigroup.

[2] Англійською – monoid.

[3] У 1916 р. Отто Шмідт, студент нині Київського національного університету імені Т. Г. Шевченка, захистив магістерську дисертацію на тему "Абстрактна теорія груп"!

[4] Клас `Semigroup` як суперклас класу `Monoid` з'явився тільки у версії 8.4, 2018 р.

1.1.2 Приклади

Наведені далі приклади ілюструють той факт, що багато добре відомих нам систем є напівгрупами і моноїдами. Множина чисел з бінарною операцією додавання + ($\otimes = +$) і нейтральним елементом 0 ($e = 0$):

```
Prelude> (2+3)+4
9

Prelude> 2+(3+4)
9

Prelude> 0+12
12

Prelude> 12+0
12
```

Множина чисел з бінарною операцією множення $*$ ($\otimes = *$) і нейтральним елементом 1 ($e = 1$):

```
Prelude> 2*(3*4)
24

Prelude> (2*3)*4
24

Prelude> 1*23
23

Prelude> 23*1
```

Множина списків з бінарною операцією конкатенації ++ ($\otimes = ++$) і нейтральним елементом [] ($e = []$):

```
Prelude> [1..2] ++ ([3..4] ++ [5..8])
[1,2,3,4,5,6,7,8]

Prelude> ([1..2] ++ [3..4]) ++ [5..8]
[1,2,3,4,5,6,7,8]

Prelude> [] ++ [1..4]
[1,2,3,4]

Prelude> [1..4] ++ []
[1,2,3,4]

Prelude> "asdf" ++ ("123" ++ "AS")
"asdf123AS"
```

```
Prelude> ("asdf" ++ "123") ++ "AS"
"asdf123AS"

Prelude> [] ++ "abc"
"abc"

Prelude> "abc" ++ []
"abc"
```

Множина логічних виразів з бінарною операцією кон'юнкції && ($\otimes = \&\&$) і нейтральним елементом $True$ ($e = True$):

```
Prelude> True && (1 /= 1  && False)
False

Prelude> (True && 1 /= 1) && False
False

Prelude> True && 1 /= 1
False

Prelude>  1 /= 1 && True
False
```

Множина логічних виразів з бінарною операцією диз'юнкції || ($\otimes = ||$) і нейтральним елементом $False$ ($e = False$):

```
Prelude> True || (1 /= 1 || False)
True

Prelude> (True || 1 /= 1) || False
True

Prelude> False || 1 /= 1
False

Prelude> 1 /= 1 || False
False
```

Інші приклади розглянемо далі.

1.2 Клас `Semigroup`

У модулі `Data.Semigroup` клас `Semigroup` визначається з декларуванням бінарного асоціативного оператора `<>` ($\otimes = $ `<>`) та сигнатури двох методів:

```
class Semigroup a where
   (<>) :: a -> a -> a
   sconcat :: NonEmpty a -> a
   stimes :: Integral b => b -> a -> a
```

Метод `sconcat` приймає як вхідний параметр непорожній список (тип `NonEmpty`) та редукує його, застосовуючи операцію `<>`.

Метод `stimes` приймає як вхідний параметр число `n :: Integral b` та `n-1` разів застосовує операцію `<>` до `n` значень, представлених другим параметром. Імпортуємо модуль.

```
Prelude> import Data.Semigroup
```

Перша спроба використання операції `<>` виявляється невдалою

```
Prelude Data.Semigroup> 2 <> 3

<interactive>:27:1: error:
    * Ambiguous type ...
```

Справді, звідки ж Haskell (та і будь хто) знає, йшла мова про напівгрупу над `Num a` з операцією `<> = +`, чи `<> = *` (в математичній нотації $\otimes = +$, $\otimes = *$), чи якоюсь іще іншою?

1.2.1 Класи-обгортки

З метою подолання цієї проблеми оголошені ізоморфні типи (типи-обгортки) за допомогою ключового слова `newtype`:

```
newtype Sum a = Sum { getSum :: a }

newtype Product a = Product { getProduct :: a }
```

та їх екземпляри:

```
instance Num a => Semigroup (Sum a) where
   Sum x <> Sum y = Sum (x + y)

instance Num a => Semigroup (Product a) where
   Product x <> Product y = Product (x * y)
```

Тепер використання `Sum a` чи `Product a` змістовно еквівалентно використанню `Num a` з бінароною опреацією `+` та `*` відповідно.

Справді

```
Prelude Data.Semigroup> Sum 2 <> Sum 3 <> Sum 4
Sum {getSum = 9}
```

Іменоване поле містить результат. Можемо легко дістатись результату, використовуючи дужки або оператор аплікації (застосування):

```
Prelude Data.Semigroup> getSum ( Sum 2 <> Sum 3 <> Sum 4)
9

Prelude Data.Semigroup> getSum $ Sum 2 <> Sum 3 <> Sum 4
9
```

Запитаємо типи значень:

```
Prelude Data.Semigroup> :t getSum $ Sum 2 <> Sum 3 <> Sum 4
getSum $ Sum 2 <> Sum 3 <> Sum 4 :: Num a => a

Prelude Data.Semigroup> getSum $ Sum 2.0 <> Sum 3 <> Sum 4
9.0

Prelude Data.Semigroup> :t getSum $ Sum 2.0 <> Sum 3 <> Sum 4
getSum $ Sum 2.0 <> Sum 3 <> Sum 4 :: Fractional a => a
```

Приклади роботи з напівгрупою над `Num a` з операцією множення:

```
Prelude Data.Semigroup> Product 3 <> Product 4
Product {getProduct = 12}

Prelude Data.Semigroup> getProduct $ Product 3 <> Product 4
12

Prelude Data.Semigroup> :t getProduct $ Product 3 <> Product 4
getProduct $ Product 3 <> Product 4 :: Num a => a

Prelude Data.Semigroup> getProduct $Product 3.0 <> Product 4
24.0

Prelude Data.Semigroup> :t getProduct $ Product 3.0 <> Product 4
getProduct $ Product 3.0 <> Product 4 :: Fractional a => a
```

Приклади роботи з напівгрупою над множиною списків з операцією конкатенації:

```
Prelude Data.Semigroup> "abc" <> "12" <>"DF"
"abc12DF"
```

Визначені й інші типи-обгортки, зокрема:

```
newtype All = All { getAll :: Bool }
newtype Any = Any { getAny :: Bool }
newtype Min a = Min { getMin :: a }
newtype Max a = Max { getMax :: a }
```

1.2.2 Методи класу `Semigroup`

Метод (<>)

Метод `(<>)`, представлений як бінарний інфіксний оператор, ми вже розглянули у попередньому підрозділі в застосуванні до ізоморфних типів (типів-обгорток).

Метод `sconcat`

Метод `sconcat :: NonEmpty a -> a` редукує непорожній список (застосовує операцію `<>` до елементів).

Для застосування методу `sconcat` необхідно імпортувати модуль `Data.List.NonEmpty` з конструктором непорожнього списку `:|`

```
Prelude Data.Semigroup> import Data.List.NonEmpty

Prelude Data.Semigroup Data.List.NonEmpty> 12 :| [10..15]
12 :| [10,11,12,13,14,15]

Prelude Data.Semigroup Data.List.NonEmpty> :t 12 :| [10..15]
12 :| [10..15] :: (Enum a, Num a) => NonEmpty a
```

Змінимо запрошення `ghci` на більш компактне та застосуємо до непорожнього списку типів-обгорток типу `Bool`:

```
Prelude Data.Semigroup Data.List.NonEmpty> :set prompt >

> sconcat (All (1/=1) :| [All True, All True])
All {getAll = False}

> sconcat (Any (1/=1) :| [Any True, Any True])
Any {getAny = True}
```

Метод `stimes`

Метод `stimes :: Integral b => b -> a -> a` застосовує бінарну операцію `<>` до кількох (перший параметр) екземплярів другого параметра.

```
>stimes 4 $ Sum 2
Sum {getSum = 8}

> Sum 2 <> Sum 2 <> Sum 2 <> Sum 2
Sum {getSum = 8}

> stimes 4 $ Product 2
Product {getProduct = 16}

> Product 2 <> Product 2 <> Product 2 <> Product 2
Product {getProduct = 16}
```

1.3 Клас Monoid

У модулі Data.Monoid клас Monoid визначається з декларуванням значення mempty – нейтрального елемента (e в математичній нотації), бінарного оператора mappend (оператор <> класу Semigroup чи $\otimes$ в математичній нотації) та методу mconcat:

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a

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

Математична нотація аксіом моноїда

$$\forall x, y, z \in M \ : \ x \otimes (y \otimes z) = (x \otimes y) \otimes z$$

$$\forall x \in M \ : \ x \otimes e = e \otimes x = x$$

представлена в декларації класу у формі

```
  mempty `mappend` x          = x
  x      `mappend` mempty     = x
  (x `mappend` y) `mappend` z = x `mappend` (y `mappend` z)
```

1.3.1 Методи класу Monoid

Метод mempty

Нуль-арний метод mempty – визначає нейтральний елемент моноїда, як то 0, 1, True і False для типів-обгорток Sum, Product, All та Any відповідно.

Метод mappend

Розробники обрали методу mappend не зовсім вдале ім'я для позначення бінарної моноїдальної операції, оскільки воно викликає асоціації, пов'язані зі списком. А його треба сприймати просто як ім'я бінарної операції. Зокрема, замість mappend можна використовувати інфіксний оператор <>, як і в класі Semigroup.[5]

```
Prelude> import Data.Monoid

Prelude Data.Monoid> Sum 2 `mappend` Sum 3
Sum {getSum = 5}

Prelude Data.Monoid> Sum 2 <> Sum 3
```

[5] У модулі Semigroup прямо вказано, що якщо напівгрупа є також і моноїдом, то (<>) = mappend.

```
Sum {getSum = 5}

Prelude Data.Monoid> Sum 2 <> Sum 3 <>  mempty
Sum {getSum = 5}

Prelude Data.Monoid> mempty <> Sum 2 <> Sum 3
Sum {getSum = 5}
```

Розглянуті раніше класи-обгортки, зокрема `Sum`, `Product`, `All`, `Any`, `Min`, `Max`, є екземплярами класу `Monoid` з визначеними відповідними нейтральними елементами.

Метод `mconcat`

Як видно з означення, метод

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

здійснює правоасоціативну згортку з моноїдною операцією `mappend`, приймаючи за стартове значення моноїдний нейтральний елемент `mempty` .

Розглянемо приклади, для цього створимо список моноїдних значень `All Bool` та `Any Bool` за допомогою функції вищого порядку `map` та відповідних конструкторів.

```
Prelude Data.Monoid> map Any [True, 1/=1, False]
[Any {getAny = True},Any {getAny = False},Any {getAny = False}]

Prelude Data.Monoid> :t map Any [True, 1/=1, False]
 map Any [True, 1/=1, False] :: [Any]
```

Такі списки можна згорнути, тобто отримати єдине моноїдне значення. Адже будь-яка згортка повертає єдине значення.

```
Prelude Data.Monoid> mconcat $ map All [True, 1/=1, False]
All {getAll = False}

Prelude Data.Monoid> :t mconcat $ map All [True, 1/=1, False]
mconcat $ map All [True, 1/=1, False] :: All

Prelude Data.Monoid> mconcat $ map Any [True, 1/=1, False]
Any {getAny = True}

Prelude Data.Monoid> :t mconcat $ map Any [True, 1/=1, False]
mconcat $ map Any [True, 1/=1, False] :: Any
```

Тип `Maybe a` оголошений екземпляром класів `Semigroup` та `Monoid`. Контекст декларацій становить обмеження – тип `a` має бути екземпляром відповідних класів:

```haskell
instance Semigroup a => Semigroup (Maybe a) where
  Nothing <> x       = x
  x       <> Nothing = x
  Just x  <> Just y  = Just (x <> y)
  stimes _ Nothing  = Nothing
  stimes n (Just x) = case compare n 0 of
    LT -> errorWithoutStackTrace "stimes: Maybe, negative multiplier"
    EQ -> Nothing
    GT -> Just (stimes n x)

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)
```

Ось кілька прикладів (запрошення змінено командою `:set prompt >` на >):

```haskell
>  Just "AB" <>  Just "CD"
Just "ABCD"

>  Nothing <> Just "AB"
Just "AB"

>  Just (Sum 2) <> Just (Sum 3)
Just (Sum {getSum = 5})

> Just (Sum 2) <> Nothing
Just (Sum {getSum = 2})

> Just (Product 4) <> Just (Product 5)
Just (Product {getProduct = 20})

> Just (Product 4) <> Nothing
Just (Product {getProduct = 4})

> First (Just 'D') <> First (Just 'B') <> First (Just '8')
First {getFirst = Just 'D'}

> First Nothing <> First (Just 'B') <> First (Just '8')
First {getFirst = Just 'B'}

> Last (Just 'D') <> Last (Just 'B') <> Last (Just '8')
Last {getLast = Just '8'}

> Last (Just 'D') <> Last (Just 'B') <> Last Nothing
Last {getLast = Just 'B'}
```

```
> map Last [Just 'D',Just 'B', Nothing]
[Last {getLast = Just 'D'},Last {getLast = Just 'B'},
Last {getLast = Nothing}]

> mconcat $ map Last [Just 'D',Just 'B', Nothing]
Last {getLast = Just 'B'}
```

Типи `First` та `Last` дозволяють визначити, відповідно, перше та останнє не-`Nothing`-значення.

1.4 Згортання структур

Практика програмування свідчить, що часто виникає необхідність виконати згортання (`folding`) не тільки списків, але й інших різноманітних структур даних. Вважається, що згортку можна визначити майже на будь-якій структурі. Тому в мову `Haskell` було введено клас `Foldable`.

1.4.1 Клас `Foldable`

У модулі `Data.Foldable` він визначається так:

```
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
   foldl :: (b -> a -> b) -> b -> t a -> b
   foldr1 :: (a -> a -> a) -> t a -> a
   foldl1 :: (a -> a -> a) -> t a -> a
   toList :: t a -> [a]
   null :: t a -> Bool
   length :: t a -> Int
   elem :: Eq a => a -> t a -> Bool
   maximum :: Ord a => t a -> a
   minimum :: Ord a => t a -> a
   sum :: Num a => t a -> a
   product :: Num a => t a -> a
```

Деякі з цих функцій (методів) нам добре відомі, проте застосовували ми їх тільки до списків, як от `length`, `sum`, `foldr`, `foldl` тощо. Проте, як бачимо, вони застосовні до будь-яких типів - екземплярів класу `Foldable`.

Розглянемо методи з моноїдним типом:

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

Обидва методи, очевидно, виконують згортку, повертаючи моноїдне значення.

Проте метод `fold :: Monoid m => t m -> m` згортає структуру даних (типу `t m`), кожен елементами якого має моноїдний тип `m`.

А метод `foldMap :: Monoid m => (a -> m) -> t a -> m` згортає струтуру (типу `t a`) з елементами немоноїдного типу `a`. Для поелементного відображення немоноїдних значень в моноїдні використовується перший параметр – функція з сигнатурою `(a -> m)`.

1.4.2 Приклад: згортання дерева

Як приклад `Foldable`-структури визначимо бінарне дерево типу:

```
data Tree a = Empty
             |Node a (Tree a) (Tree a)
                     deriving (Show)
```

Всі внутрішні вузли, з кореневим включно, містять значення типу `a`, термінальні – порожні (`Empty`).

У визначенні екземпляра класу `Foldable` реалізуємо метод `foldMap`

```
instance Foldable Tree where
   foldMap f Empty = mempty
   foldMap f (Node x l r) =
      foldMap f l  `mappend`
      foldMap f r  `mappend`
      f x
```

Для порожнього вузла метод повертає моноїдне значення `mempty`. Якщо ж у вузла зі значенням `x` є ліве `l` і праве `r` піддерева, то метод рекурсивно застосує метод до лівого піддерева `foldMap f l`, правого піддерева `foldMap f r`, застосує функцію відображення до значення у вузлі – `f x` і далі редукує все це за допомогою моноїдної операції `` `mappend` = (<>)``. Отже при згортанні реалізовано порядок обходу – LRN (ліве піддерево - праве піддерево - вузол).

Нехай у нас є дерево

```
   tree1 = Node 5
             (Node 3
                (Node 1 Empty Empty)
                (Node 2 Empty Empty)
              )
              (Node 8
                (Node 6 Empty Empty)
                (Node 7 Empty Empty)
              )
```

Графічно воно представлене на рис. 1.1

Для виконання прикладів треба імпортувати модулі

```
import Data.Foldable
import Data.Semigroup
```

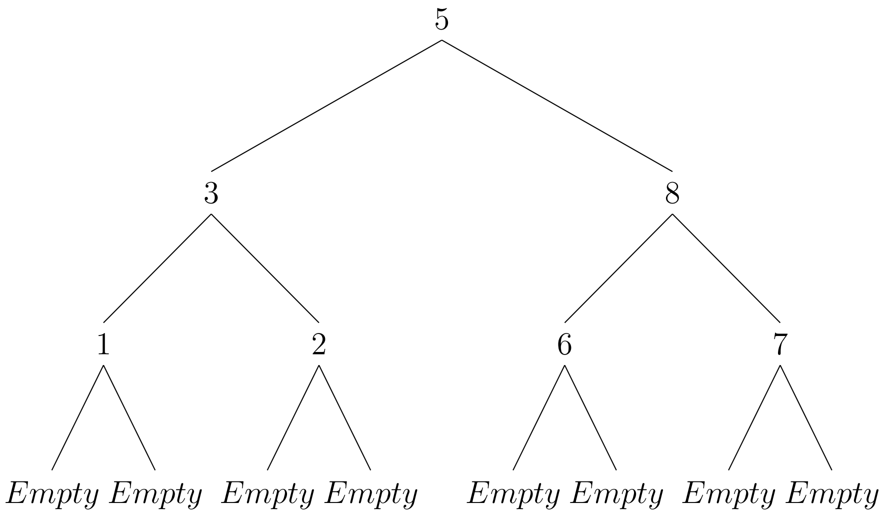

Рис. 1.1: Дерево tree1

Перший параметр метода `foldMap` - функцію `f::a -> m`, визначатимемо у формі λ-функції.

Обчислимо суму всіх елементів структури:

```
*Main> foldMap (\x-> Sum x) tree1
Sum {getSum = 32}
```

Визначимо кількість непорожніх (внутрішніх) вузлів:

```
*Main> foldMap (\x-> Sum 1) tree1
Sum {getSum = 7}
```

Якщо потрібно знати добуток всіх елементів:

```
*Main> foldMap (\x-> Product x) tree1
Product {getProduct = 10080}
```

Представлення у формі списку:

```
*Main> foldMap (\x->  [x]) tree1
[1,2,3,6,7,8,5]
```

Представлення у формі рядка:

```
*Main> foldMap (\x->  show x ++ " ") tree1
"1 2 3 6 7 8 5 "
```

Чи є елемент, для якого справедливе вказане твердження (дорівнює 4, більше 6):

```
*Main> foldMap (\x->  Any $ x == 4) tree1
Any {getAny = False}

*Main> foldMap (\x->  Any $ x > 6) tree1
Any {getAny = True}
```

Чи для всіх елементів справедливе вказане твердження (більше 6, більше 0):

```
*Main> foldMap (\x->  All $ x > 6) tree1
All {getAll = False}

*Main> foldMap (\x->  All $ x > 0) tree1
All {getAll = True}
```

Розглянемо приклад дерева, див. рис. 1.2, отриманого в результаті синтаксичного розбору арифметичного виразу $2 + 30 * 400$:

```
tree2 = Node "+"
          (Node "2" Empty Empty
          )
          (Node "*"
            (Node "30" Empty Empty)
            (Node "400" Empty Empty)
          )
```

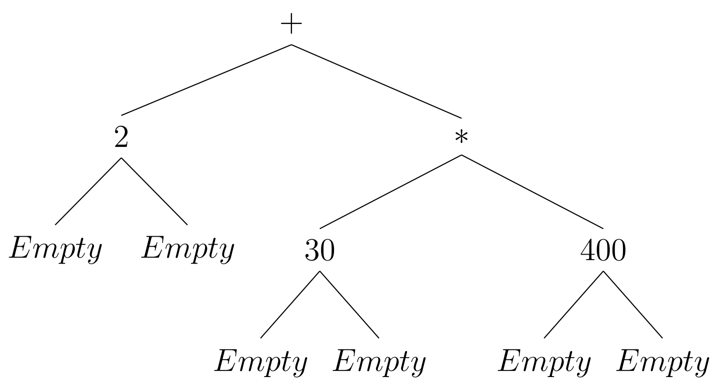

Рис. 1.2: Дерево tree2

Тепер можемо знайти постфіксне представлення у формі рядка чи списка

```
*Main> foldMap (\x-> x++" " ) tree2
"2 30 400 * + "

*Main> foldMap (\x-> [x]) tree2
["2","30","400","*","+"]
```

Чи визначити загальну довжину слів у непорожніх вузлах:

```
*Main> foldMap (\x-> Sum $ length x) tree2
Sum {getSum = 8}
```

Розглянемо дерево, вузли якого містять значення типу `Num b => ([Char], b)`, див. рис 1.3. Це можливо, оскільки змістовних обмежень щодо типів значень у вузлах дерева оголошення `data Tree a` немає.

```
tree3 = Node ("a",0)
          (Node ("a1",100) Empty Empty
          )
          (Node ("a2",200)
            (Node ("a21",215) Empty Empty)
            (Node ("a22",223) Empty Empty)
          )
```

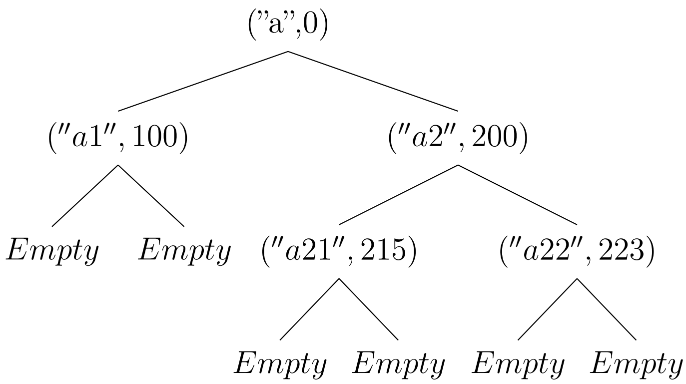

Рис. 1.3: Дерево tree3

Сигнатура метода `foldMap :: Monoid m => (a -> m) -> t a -> m` теж не містить обмежень щодо типу `a`, тому метод знову працює:

```
*Main> foldMap (\x-> Any $ fst x == "a21") tree3
Any {getAny = True}

*Main> foldMap (\x-> Any $ snd x == 100) tree3
Any {getAny = True}

*Main> foldMap (\x-> Any $ x == ("a22",223)) tree3
Any {getAny = True}

*Main> foldMap (\x-> Sum $ snd x) tree3
Sum {getSum = 738}
```

1.5 Узагальнений код

Як видно з наведених прикладів, код, написаний для екземплярів класу `Monoid`, може працювати з даними найрізноманітніших типів. Такий код називають узагальненим (**generic**).

Можна сказати, що під **generic** у `Haskell` розуміють форму абстракції, яка дозволяє визначати функції, здатні працювати з великим набором типів даних (великою кількістю типів - екземплярів певного класу).