Programare declarativă[1]

Module si Clase de Tipuri

Ioana Leuștean
Traian Florin Șerbănuță

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

17 noiembrie 2017

[1] bazat pe cursul Informatics 1: Functional Programming de la University of Edinburgh

Module (simple)

Module

Fișierul Geometry.hs

```haskell
module Geometry
( sphereVolume, sphereArea
, cuboidVolume, cuboidArea
) where

sphereVolume :: Float -> Float
sphereVolume radius = (4.0 / 3.0) * pi * (radius ^ 3)
sphereArea :: Float -> Float
sphereArea radius = 4 * pi * (radius ^ 2)

cuboidVolume :: Float -> Float -> Float -> Float
cuboidVolume a b c = rectangleArea a b * c
cuboidArea :: Float -> Float -> Float -> Float
cuboidArea a b c = rectangleArea a b * 2 +
    rectangleArea a c * 2 + rectangleArea c b * 2
rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```

Module Structurate

Fișierul Geometry/Sphere.hs

```haskell
module Geometry.Sphere ( volume, area ) where
volume :: Float -> Float
volume radius = (4.0 / 3.0) * pi * (radius ^ 3)
area :: Float -> Float
area radius = 4 * pi * (radius ^ 2)
```

Fișierul Geometry/Cuboid.hs

```haskell
module Geometry.Cuboid ( volume, area ) where
volume :: Float -> Float -> Float -> Float
volume a b c = rectangleArea a b * c
area :: Float -> Float -> Float -> Float
area a b c = rectangleArea a b * 2 +
    rectangleArea a c * 2 + rectangleArea c b * 2
rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```

Importare: redenumire și ascundere

Fișierul Geometry/Cube.hs

```
module Geometry.Cube ( volume, area ) where

import qualified Geometry.Cuboid as Cuboid

volume :: Float -> Float
volume side = Cuboid.volume side side side

area :: Float -> Float
area side = Cuboid.area side side side
```

Fișierul Main.hs

```
import qualified Geometry.Sphere as Sphere (volume)
import Geometry.Cuboid (volume)
import Geometry.Cube hiding (volume)
```

Abstractizare

Mulțimi implementate folosind liste

Fișierul SetListImpl.hs (1)

```
module SetListImpl
(Set, empty, insert, set, element, equal, check) where
import Test.QuickCheck

type Set a = [a]

empty :: Set a
empty = []

insert :: a -> Set a -> Set a
insert x xs = x:xs

set :: [a] -> Set a
set xs = xs
```

Mulțimi implementate folosind liste

Fișierul SetListImpl.hs (2)

```
element :: Eq a => a -> Set a -> Bool
x `element` xs = x `elem` xs

equal :: Eq a => Set a -> Set a -> Bool
xs `equal` ys = xs `subset` ys && ys `subset` xs
  where
    xs `subset` ys = and [ x `elem` ys | x <- xs ]
```

Mulțimi implementate folosind liste

Proprietăți și teste

Fișierul SetListImpl.hs (3)

```
prop_element :: [Int] -> Bool
prop_element ys =
    and [ x `element` s == odd x | x <- ys ]
  where
    s = set [ x | x <- ys, odd x ]
check =
    quickCheck prop_element
```

GHCI

```
SetListImpl> check
+++ OK, passed 100 tests.
```

Probleme de abstractizare

Fișierul SetListImplTest.hs

```
module SetListImplTest where
import SetListImpl
test :: Int -> Bool
test n =
   s `equal` t
  where
    s = set [1,2..n]
    t = set [n,n-1..1]

breakAbstraction :: Set a -> a
```

Probleme de abstractizare

Fișierul SetListImplTest.hs

```
module SetListImplTest where
import SetListImpl
test :: Int -> Bool
test n =
    s `equal` t
  where
    s = set [1,2..n]
    t = set [n,n-1..1]

breakAbstraction :: Set a -> a
breakAbstraction = head
```

Probleme de abstractizare

Fișierul SetListImplTest.hs

```
module SetListImplTest where
import SetListImpl
test :: Int -> Bool
test n =
   s `equal` t
  where
    s = set [1,2..n]
    t = set [n,n-1..1]

breakAbstraction :: Set a -> a
breakAbstraction = head
```

- Nu e funcție

Probleme de abstractizare

Fișierul SetListImplTest.hs

```
module SetListImplTest where
import SetListImpl
test :: Int -> Bool
test n =
   s `equal` t
  where
   s = set [1,2..n]
   t = set [n,n-1..1]

breakAbstraction :: Set a -> a
breakAbstraction = head
```

- Nu e funcție

```
head (set [1,2,3]) == 1 /= 3 == head (set [3,2,1])
```

Încapsulare

Încapsulare

Fișierul SetListAbs.hs (1)

```
module SetListAbs
(Set, empty, insert, set, element, equal, check) where
import Test.QuickCheck

data Set a    =   MkSet [a]

empty :: Set a
empty = MkSet []

insert :: a -> Set a -> Set a
insert x (MkSet xs) = MkSet (x:xs)

set :: [a] -> Set a
set xs = MkSet xs
```

Încapsulare

Fișierul SetListAbs.hs (2)

```
element :: Eq a => a -> Set a -> Bool
x `element` (MkSet xs) = x `elem` xs

equal :: Eq a => Set a -> Set a -> Bool
MkSet xs `equal` MkSet ys =
    xs `subset` ys && ys `subset` xs
  where
    xs `subset` ys = and [ x `elem` ys | x <- xs ]
```

Încapsulare

Proprietăți și teste

Fișierul SetListAbs.hs (3)

```
prop_element :: [Int] -> Bool
prop_element ys =
    and [ x `element` s == odd x | x <- ys ]
  where
    s = set [ x | x <- ys, odd x ]

check =
    quickCheck prop_element
```

GHCI

```
SetListAbs> check
+++ OK, passed 100 tests.
```

Probleme de abstractizare?

Fișierul SetListAbsTest.hs

```
module SetListAbsTest where
import SetListAbs
test :: Int -> Bool
test n =
   s `equal` t
  where
    s = set [1,2..n]
    t = set [n,n-1..1]

breakAbstraction :: Set a -> a
```

Probleme de abstractizare?

Fișierul SetListAbsTest.hs

```
module SetListAbsTest where
import SetListAbs
test :: Int -> Bool
test n =
   s `equal` t
  where
    s = set [1,2..n]
    t = set [n,n-1..1]

breakAbstraction :: Set a -> a
breakAbstraction = head   -- eroare de tipuri
```

Încapsulare = Ascundere de informație

module ListAbs(Set,empty,**insert**,set,element,equal)

```
> ghci SetListAbs.hs
Ok, modules loaded: SetListAbs
*SetListAbs> let s0 = set [2,7,1,8,2,8]
*SetListAbs> let MkSet xs = s0 in xs
Not in scope: data constructor ‘MkSet‘
```

module SetListAbs(Set(MkSet),empty,**insert**,set,element,equal)

```
> ghci SetListAbs.hs
*SetListAbs> let s0 = set [2,7,1,8,2,8]
*SetListAbs> let MkSet xs = s0 in xs
[2,7,1,8,2,8]
*SetListAbs> head xs
2
```

Clase de tipuri

Test de apartenență

```
elem :: Eq a => a -> [a] -> Bool
```

Folosind descrieri de liste

Folosind recursivitate

Folosind funcții de nivel înalt

Test de apartenență

```
elem :: Eq a => a -> [a] -> Bool
```

Folosind descrieri de liste

```
elem x ys      = or [ x == y | y <- ys ]
```

Folosind recursivitate

```
elem x []      = False
elem x (y:ys)  = x == y || elem x ys
```

Folosind funcții de nivel înalt

```
elem x ys      = foldr (||) False (map (x ==) ys)
```

Funcția elem este polimorfică

Dar nu pentru orice tip

```
*Main> elem 1 [2,3,4]
False

*Main> elem 'o' "word"
True

*Main> elem (1,'o') [(0,'w'),(1,'o'),(2,'r'),(3,'d')]
True

*Main> elem "word" ["list","of","word"]
True

*Main> elem (\x -> x) [(\x -> -x), (\x -> -(-x))]
No instance for (Eq (a -> a)) arising from a use of `elem'
Possible fix: add an instance declaration for (Eq (a -> a))
```

Clasa de tipuri pentru egalitate

```
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Int         where
  (==) = eqInt

instance    Eq Char      where
  x == y                 = ord x == ord y

instance (Eq a, Eq b) => Eq (a,b) where
  (u,v) == (x,y)         = (u == x) && (v == y)

instance Eq a => Eq [a] where
  [] == []               = True
  [] == y:ys             = False
  x:xs == []             = False
  x:xs == y:ys           = (x == y) && (xs == ys)
```

Clasă de tipuri = dicționar de funcții

```
type EqDict a =  EqD { eq :: a -> a -> Bool }
elem :: EqDict a -> a -> [a] -> Bool
```

Folosind descrieri de liste

Folosind recursivitate

Folosind funcții de nivel înalt

Clasă de tipuri = dicționar de funcții

```haskell
type EqDict a =  EqD { eq :: a -> a -> Bool }
elem :: EqDict a -> a -> [a] -> Bool
```

Folosind descrieri de liste

```haskell
elem (EqD eq) x ys      = or [ x `eq` y | y <- ys ]
```

Folosind recursivitate

```haskell
elem _  x []           = False
elem (EqD eq) x (y:ys) = x `eq` y || elem x ys
```

Folosind funcții de nivel înalt

```haskell
elem d x ys      = foldr (||) False (map (eq d x) ys)
```

Clasă de tipuri = dicționar de funcții   Instanțieri

```
dInt :: EqDict Int
dInt = EqD eqInt

dChar ::  EqDict Char
dChar = EqD (\ x y -> ord x == ord y)

dPair :: (EqDict a, EqDict b) -> EqDict (a,b)
dPair (EqD eqa, EqD eqb) = EqD eq
  where (u,v) `eq` (x,y) = (u `eqa` x) && (v `eqb` y)

dList :: EqDict a -> EqDict [a]
dList (EqD eqa) = EqD eq
  where
    [] `eq` []            = True
    [] `eq` (y:ys)        = False
    (x:xs) `eq []         = False
    (x:xs) `eq` (y:ys) = (x `eqa` y) && (xs `eq` ys)
```

Polimorfism cu dicționare de funcții

```
*Main> elem dInt 1 [2,3,4]
False

*Main> elem dChar 'o' "word"
True

*Main> elem (dPair (dInt,dChar))
            [(1,'o') [(0,'w'),(1,'o'),(2,'r'),(3,'d')]
True

*Main> elem (dList dChar) "word" ["list","of","word"]
True
```

Eq, Ord, Show

Eq, Ord, Show

```haskell
class  Eq a  where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  -- minimum definition: (==)
  x /= y = not (x == y)

class  (Eq a) => Ord a  where
  (<)  ::   a -> a ->   Bool
  (<=) ::   a -> a ->   Bool
  (>)  ::   a -> a ->   Bool
  (>=) ::   a -> a ->   Bool
  -- minimum    definition: (<=)
  x < y   =    x <= y && x /= y
  x > y   =    y < x
  x >= y  =     y <= x

class  Show a  where
  show :: a -> String
```

Bool

```
instance  Eq Bool  where
  False == False = True
  False == True  = False
  True  == False = False
  True  == True  = True

instance  Ord Bool  where
  False <= False = True
  False <= True  = True
  True  <= False = False
  True  <= True  = True

instance  Show Bool  where
  show False     = "False"
  show True      = "True"
```

Perechi

```
instance (Eq a, Eq b) => Eq (a,b) where
  (x,y) == (x',y') = x == x' && y == y'

instance (Ord a, Ord b) => Ord (a,b) where
  (x,y) <= (x',y') = x < x' || (x == x' && y <= y')

instance (Show a, Show b) => Show (a,b) where
  show (x,y) = "(" ++ show x ++ "," ++ show y ++ ")"
```

Liste

```
instance  Eq a => Eq [a]  where
  []     == []     = True
  []     == (y:ys) = False
  (x:xs) == []     = False
  (x:xs) == (y:ys) = x == y && xs == ys

instance  Ord a => Ord [a]  where
  []     <= ys     = True
  (x:xs) <= []     = False
  (x:xs) <= (y:ys) = x < y || (x == y && xs <= ys)

instance Show a => Show [a] where
  show []     = "[]"
  show (x:xs) = "[" ++ showSep x xs ++ "]"
    where
      showSep x []     = show x
      showSep x (y:ys) = show x ++ "," ++ showSep y ys
```

Derivare automata pentru tipuri algebrice

```haskell
data Bool = False | True
     deriving (Eq, Ord, Show)

data Pair a b = MkPair a b
     deriving (Eq, Ord, Show)

data List a = Nil | Cons a (List a)
     deriving (Eq, Ord, Show)
```

Mulțimi ca instanță a lui Eq

```
instance Eq (Set a) where
  s == t = s `equal` t
```

Observație

- Diferit față de implementarea implicită dată de **deriving**
- Deoarece egalitatea de mulțimi e mai mult decât egalitatea sintactică

Numere

Clase de tipuri pentru numere

```
class (Eq a, Show a) => Num a where
  (+),(-),(*)     :: a -> a -> a
  negate          :: a -> a
  fromInteger     :: Integer -> a
  -- minimum definition: (+),(-),(*),fromInteger
  negate x       =   fromInteger 0 - x

class (Num a) => Fractional a where
  (/)             :: a -> a -> a
  recip           :: a -> a
  fromRational   :: Rational -> a
  -- minimum definition: (/), fromRational
  recip x        =   1/x

class (Num a, Ord a) => Real a where
  toRational      :: a -> Rational

class (Real a, Enum a) => Integral a where
  div, mod        :: a -> a -> a
  toInteger       :: a -> Integer
```

Instanțe pentru tipul predefinit Float

```
instance Num Float    where
   (+)             =      builtInAddFloat
   (-)             =      builtInSubtractFloat
   (*)             =      builtInMultiplyFloat
   negate          =      builtInNegateFloat
   fromInteger     =      builtInFromIntegerFloat

instance Fractional Float where
   (/)             = builtInDivideFloat
   fromRational = builtInFromRationalFloat
```

Să definim numerele naturale

Fișierul Natural.hs (1)

```
module Natural(Nat) where
import Test.QuickCheck

data Nat = MkNat Integer

invariant :: Nat -> Bool
invariant (MkNat x) = x >= 0

instance Eq Nat where
  MkNat x == MkNat y = x == y

instance Ord Nat where
  MkNat x <= MkNat y = x <= y

instance Show Nat where
  show (MkNat x) = show x
```

Naturale ca instanță a lui Num

Fișierul Natural.hs (2)

```
instance Num Nat where
  MkNat x + MkNat y  =  MkNat (x + y)
  MkNat x - MkNat y
    | x >= y     =  MkNat (x - y)
    | otherwise =  error (show (x-y) ++ " is negative")
  MkNat x * MkNat y  =  MkNat (x * y)
  fromInteger x
    | x >= 0     =  MkNat x
    | otherwise =  error (show x ++ " is negative")
  negate  =   undefined
```

Teste de consistență

Fișierul Natural.hs (3)

```
prop_plus :: Integer -> Integer -> Property
prop_plus m n =
  (m >= 0) && (n >= 0) ==> (m+n >= 0)

prop_times :: Integer -> Integer -> Property
prop_times m n =
  (m >= 0) && (n >= 0) ==> (m*n >= 0)

prop_minus :: Integer -> Integer -> Property
prop_minus m n =
  (m >= 0) && (n >= 0) && (m >= n) ==> (m-n >= 0)
```

Fișierul NaturalTest.hs

```
module NaturalTest where
import Natural

m, n :: Nat
m = fromInteger 2
n = fromInteger 3
```

Testare

```
> ghci NaturalTest
Ok, modules loaded: NaturalTest, Natural.
*NaturalTest> m
2
*NaturalTest> n
3
*NaturalTest> m+n
5
*NaturalTest> n-m
1
*NaturalTest> m-n
*** Exception: -1 is negative
*NaturalTest> m*n
6
*NaturalTest> fromInteger (-5) :: Nat
*** Exception: -5 is negative
*NaturalTest> MkNat (-5)
Not in scope: data constructor `MkNat`
```