Programare declarativă

Introducere în programarea funcțională folosind Haskell

Ana Cristina Țurlea

ana.turlea@fmi.unibuc.ro

Organizare

Plan curs

Programare funcțională (folosind Haskell)

- Funcții, recursie, funcții de ordin înalt, tipuri
- Operații pe liste: filtrare, transformare, agregare
- Polimorfism, clase de tipuri, modularizare
- Tipuri de date algebrice - evaluarea expresiilor
- Operațiuni Intrare/Ieșire
- Agregare pe tipuri algebrice
- Functori, monade

Resurse

- Pagina cursului:
  - https://moodle.unibuc.ro/course/view.php?id=3041 (ambele serii?)
  - Materiale: https://drive.google.com/drive/folders/18UMgYsM5VYaMG55Vl6g6IuW8lA5VcJRJ
- Cartea online „Learn You a Haskell for Great Good” http://learnyouahaskell.com/
- Pagina Haskell http://haskell.org
- Hoogle https://www.haskell.org/hoogle
- Haskell Wiki http://wiki.haskell.org

Legătură foarte utilă!

https://wiki.haskell.org/H-99:_Ninety-Nine_Haskell_Problems

Evaluare

Notare

- Testare (t), examen (e)
- Nota finală: 1 (oficiu) + t + e

Condiție de promovabilitate

- Nota finală cel puțin 5
  - $5 > 4.99$

Activitate laborator

- Se poate nota activitatea in cadrul laboratoarelor.
- Maxim 1 punct (bonus la nota finală)

Test laborator

- Valorează 5 puncte din nota finală
- După săptămâna a 7-a (va fi anuntat ulterior)
- Pe moodle
- Materiale ajutătoare: cursul

Examen final

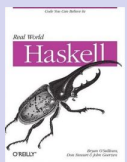
- Valorează 4 puncte din nota finală
- În sesiune
- Fizic sau online, în funcție de cum se vor desfășura examenele. Va fi anunțat atunci.
- Acoperă toată materia
- Materiale ajutătoare: cursul

Programare funcțională

Programare funcțională în Haskell

- Haskell e folosit în proiecte de Facebook, Google, Microsoft, . . .
  - Programarea funcțională e din ce în ce mai importantă în industrie
  - mai multe la `https://wiki.haskell.org/Haskell_in_industry`
- Oferă suport pentru paralelism și concurență.

De ce Haskell? (din cartea Real World Haskell)



The illustration on our cover is of a Hercules beetle. These beetles are among the largest in the world. They are also, in proportion to their size, the strongest animals on Earth, able to lift up to 850 times their own weight. Needless to say, we like the association with a creature that has such a high power-to-weight ratio.

Haskell este un limbaj funcțional pur

- Funcțiile sunt valori.
- În loc să modificăm datele existente, calculăm valori noi din valorile existente, folosind funcții
- Funcțiile sunt pure: aceleași rezultate pentru aceleași intrări.
- O bucată de cod nu poate corupe datele altei bucăți de cod.
- Distincție clară între părțile pure și cele care comunică cu mediul extern.

Haskell este un limbaj elegant

- Idei abstracte din matematică devin instrumente puternice practice
  - recursivitate, compunerea de funcții, functori, monade
  - folosirea lor permite scrierea de cod compact și modular
- Rigurozitate: ne forțează să gândim mai mult înainte, dar ne ajută să scriem cod mai corect și mai curat
- Curbă de învățare în trepte
  - Putem scrie programe mici destul de repede
  - Expertiza în Haskell necesită multă gândire și practică
  - Descoperirea unei lumi noi poate fi un drum distractiv și provocator
    `http://wiki.haskell.org/Humor`

Haskell

- Haskell e leneș: orice calcul e amânat cât de mult posibil
  - Schimbă modul de concepere al programelor
  - Permite lucrul cu colecții potențial infinite de date precum [1..]
  - Evaluarea lenească poate fi exploatată pentru a reduce timpul de calcul fără a denatura codul

```
firstK k xs = take k (sort xs)
```

- Haskell e minimalist: mai puțin cod, în mai puțin timp, și cu mai puține defecte
  - ...rezolvând totuși problema :-)

```
numbers = [1,2,3,4,5]
total = foldl (*) 1 numbers
doubled = map (* 2) numbers
```

Exemplu

```
qsort :: Ord a => [a] -> [a]

qsort []     = []

qsort (p:xs) = (qsort lesser) ++ [p] ++ (qsort greater)
    where
        lesser  = filter (< p) xs
        greater = filter (>= p) xs
```

Exemplu

```
import Control.Monad (foldM)
import Data.List ((\\))

queens :: Int -> [[Int]]
queens n = foldM f [] [1..n]
  where
    f qs _ = [q:qs | q <- [1..n] \\ qs, q `notDiag` qs]
    q `notDiag` qs = and [abs (q - qi) /= i |
                          (qi,i) <- qs `zip` [1..]]
```

https://rosettacode.org/wiki/N-queens_problem

Instalare

Instalare

Descărcare și instalare

- https://www.haskell.org/downloads/
- https://docs.haskellstack.org/en/stable/install_and_upgrade/#installupgrade

IDE

- **Atom** https://atom.io/ -> Haskell package `ide-haskell`, `ide-haskell-repl`, `language-haskell`
- Altele: https://wiki.haskell.org/IDEs

Elemente de sintaxă

Sintaxă

- Comentarii

```
-- comentariu pe o linie
{- comentariu pe
   mai multe
   linii -}
```

- Identificatori
  - șiruri formate din litere, cifre, caracterele _ și ’ (apostrof)
  - identificatorii pentru variabile încep cu literă mică sau _
  - identificatorii pentru tipuri și constructori încep cu literă mare
  - Haskell este sensibil la majuscule (case sensitive)

```
double x = 2 * x
data Point a = Pt a a
```

Sintaxă

Blocuri și indentare

Blocurile sunt delimitate prin indentare.

```
fact n =  if n == 0
          then 1
          else  n * fact (n-1)

trei =  let
            a = 1
            b = 2
        in a + b
```

- echivalent, putem scrie

```
trei  =  let {a = 1; b = 2} in a + b
trei  =  let a = 1; b = 2 in a + b
```

Legarea variabilelor

Variabile

Presupunem că fisierul `test.hs` conține

x=1
x=2

- Ce valoare are x?

```
Prelude> :l test.hs

test.hs:2:1: error:
    Multiple declarations of ‘x’
    Declared at: test.hs:1:1
                 test.hs:2:1
  |
2 | x=2
  | ^
```

Variabile

În Haskell, variabilele sunt imuabile, adică:

- = **nu** este operator de atribuire
- x = 1 reprezintă o *legatură* (binding)
- din momentul în care o variabilă este legată la o valoare, acea valoare nu mai poate fi schimbată

Legarea variabilelor

let .. in ...

este o *expresie* care crează scop local

Presupunem că fișierul `testlet.hs` conține

```
x=1
z= let x=3 in x
```

```
Prelude> :l testlet.hs
[1 of 1] Compiling Main
Ok, 1 module loaded.
*Main> z
3
*Main> x
1
```

Legarea variabilelor

- **let .. in ...** crează scop local

```
x = let
        z = 5
        g u = z + u
    in  let
            z = 7
        in g 0 + z                    -- x=12
```

```
x = let z = 5; g u = z + u in let z = 7 in g 0    -- x=5
```

- clauza **... where ...** creaza scop local

```
f x = g x + g x + z
  where
    g x = 2*x
    z =  x-1
```

Legarea variabilelor

- **let .. in ...** este o expresie

```
x = [let y =8 in y, 9] -- x=[8,9]
```

- **where** este o clauză, disponibilă doar la nivel de definiție

```
x = [y where y =8, 9] – error: parse error ...
```

- Variabile pot fi legate și prin "pattern matching" la definirea unei funcții sau expresii **case**.

```
h x | x == 0    = 0
    | x == 1    = y + 1
    | x == 2    = y * y
    | otherwise = y
  where y = x*x
```

```
f x = case x of
        0 -> 0
        1 -> y + 1
        2 -> y * y
        _ -> y
  where y = x*x
```

Tipuri de date

Sistemul tipurilor

"There are three interesting aspects to types in Haskell: they are strong, they are static, and they can be automatically inferred."
http://book.realworldhaskell.org/read/types-and-functions.html

tare garanteaza absenta anumitor erori

static tipul fiecari valori este calculat la compilare

dedus automat compilatorul deduce automat tipul fiecarei expresii

```
Prelude> :t [('a',1,"abc")]
[('a',1,"abc")] :: Num b => [(Char, b, [Char])]
```

Sistemul tipurilor

Tipurile de baza

Int, Integer, Float, Double, Bool, Char, String

- tipuri compuse: tupluri si liste

```
Prelude> :t ('a', True)
('a', True) :: (Char, Bool)

Prelude> :t ["ana", "ion"]
["ana", "ion"] :: [[Char]]
```

- tipuri noi definite de utilizator

```
data RGB = Rosu | Verde | Albastru
data Point a = Pt a a     -- tip parametrizat
                          -- a este variabila de tip
```

Tipuri de date

- Integer: 4, 0, -5

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

```
Prelude> mod 4 3
Prelude> 4 `mod` 3
```

- Float: 3.14

```
Prelude> truncate 3.14
Prelude> sqrt 4
```

```
Prelude> let x = 4 :: Int
Prelude> sqrt (fromIntegral x)
```

- Char: 'a','A', '\n'

```
Prelude> import Data.Char
Prelude Data.Char> chr 65
Prelude Data.Char> ord 'A'
Prelude Data.Char> toUpper 'a'
Prelude Data.Char> digitToInt '4'
```

Tipuri de date

- Bool: True, False

```
data Bool = True | False
```

```
Prelude> True && False || True
Prelude> not True
```

```
Prelude> 1 /= 2
Prelude> 1 == 2
```

- String: "prog\ndec"

```
type String = [Char] -- sinonim pentru tip
```

```
Prelude> "aa"++"bb"
"aabb"
Prelude> "aabb" !! 2
'b'
```

```
Prelude> lines "prog\ndec"
["prog","dec"]
Prelude> words "pr og\nde cl"
["pr","og","de","cl"]
```

Tipuri de date compuse

- Tipul tuplu - secvențe de de tipuri deja existente

```
Prelude> :t (1 :: Int, 'a', "ab")
(1 :: Int, 'a', "ab") :: (Int, Char, [Char])

Prelude> fst (1,'a') -- numai pentru perechi
Prelude> snd (1,'a')
```

- Tipul unit

```
Prelude> :t ()
() :: ()
```

- Tipul listă

```
Prelude>:t [True, False, True]
[True, False, True] :: [Bool]
```

Tipuri. Clase de tipuri. Variabile de tip

Ce răspuns primim in GHCi dacă introducem comanda

```
Prelude> :t 1
```

Răspunsul primit este:

```
1 :: Num a => a
```

Semnificația este următoarea:

- a este un *parametru de tip*
- Num este o clasă de tipuri
- 1 este o valoare de tipul a din clasa Num

```
Prelude> :t 1
1 :: Num a => a

Prelude> :t [1,2,3]
[1,2,3] :: Num t => [t]
```

Funcții

Funcții în Haskell. Terminologie

Prototipul funcției

double :: Integer -> Integer

- **numele funcției**
- signatura funcției

Definiția funcției

double elem = elem + elem

- **numele funcției**
- parametrul formal
- corpul funcției

Aplicarea funcției

double 5

- **numele funcției**
- parametrul actual (argumentul)

Exemplu: funcție cu două argumente

Prototipul funcției

add :: Integer -> Integer -> Integer

- **numele funcției**
- signatura funcției

Definiția funcției

add elem1 elem2 = elem1 + elem2

- **numele funcției**
- parametrii formali
- corpul funcției

Aplicarea funcției

add 3 7

- **numele funcției**
- argumentele

Exemplu: funcție cu **un** argument de tip tuplu

Prototipul funcției

dist :: (Integer, Integer) -> Integer

- **numele funcției**
- signatura funcției

Definiția funcției

dist (elem1, elem2) = abs (elem1 - elem2)

- **numele funcției**
- parametrul formal
- corpul funcției

Aplicarea funcției

dist (elem1, elem2)

- **numele funcției**
- argumentul

Tipuri de funcții

```
Prelude> :t abs
abs :: Num a => a -> a
Prelude> :t div
div :: Integral a => a -> a -> a

Prelude> :t (:)
(:) :: a -> [a] -> [a]
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]


Prelude> :t zip
zip :: [a] -> [b] -> [(a, b)]
```

Definirea funcțiilor

```
fact :: Integer -> Integer
```

- Definiție folosind **if**

```
fact n = if n == 0 then 1
         else n * fact(n-1)
```

- Definiție folosind ecuații

```
fact 0 = 1
fact n = n * fact(n-1)
```

- Definiție folosind cazuri

```
fact n
  | n == 0     = 1
  | otherwise = n * fact(n-1)
```

Șabloane (patterns)

- `x:y = [1,2,3]` *-- x=1 si y =[2,3]*

  Observați că : este constructorul pentru liste.

- `(u,v)=('a',[(1,'a'),(2,'b')])` *-- u='a',*
  -- v=[(1,'a'),(2,'b')]

  Observați că („) este constructorul pentru tupluri.

- Definitii folosind șabloane

```
selectie :: Integer -> String -> String
```

```
-- case...of
selectie x s =
    case (x,s) of
        (0,_) -> s
        (1, z:zs) -> zs
        (1, []) -> []
        _ -> (s ++ s)
```

```
-- stil ecuational
selectie 0 s = s
selectie 1 (_:s) = s
selectie 1 "" = ""
selectie _ s = s + s
```

Tipuri de funcții

Fie foo o funcție cu următorul tip

```
foo :: a -> b -> [a] -> [b]
```

- are trei argumente, de tipuri a, b și [a]
- întoarce un rezultat de tip [b]

Schimbăm signatura funcției astfel:

```
ffoo :: (a -> b) -> [a] -> [b]
```

- are două argumente, de tipuri (a -> b) și [a],
  adică o funcție de la a la b și o listă de elemente de tip a
- întoarce un rezultat de tip [b]

```
Prelude> :t map
map :: (a -> b) -> [a] -> [b]
```

Liste

Liste

Definiție

Observație

Orice listă poate fi scrisă folosind doar constructorul (:) și lista vidă []

- [1,2,3] == 1 : (2 : (3 : [])) == 1 : 2 : 3 : []
- "abcd" == ['a','b','c','d'] == 'a' : ('b' : ('c' : ('d' : []))) == 'a' : 'b' : 'c' : 'd' : []

Definiție recursivă

O listă este

- vidă, notată []; sau
- compusă, notată x:xs, dintr-un un element x numit capul listei (head) și o listă xs numită coada listei (tail).

Definirea listelor. Operații

Intervale și progresii

```
interval = ['c'..'e']          -- ['c','d','e']
progresie = [20,17..1]         -- [20,17,14,11,8,5,2]
progresie' = [2.0,2.5..4.0] -- [2.0,2.5,3.0,3.5,4.0]
```

Operații

```
Prelude> [1,2,3] !! 2
3
Prelude> "abcd" !! 0
'a'
Prelude> [1,2] ++ [3]
[1,2,3]
Prelude> import Data.List
```

Definiția prin selecție $\{x \mid P(x)\}$

[E(x)| x <- [x1,...,xn], P(x)]

```
Prelude> let xs = [0..10]
Prelude> [x | x <- xs, even x]
[0,2,4,6,8,10]

Prelude> let xs = [0..6]
Prelude> [(x,y) | x <- xs, y <- xs,  x + y == 10]
[(4,6),(5,5),(6,4)]
```

Folosirea lui `let` pentru declarații locale:

```
Prelude> [(i,j) | i <- [1..2], let k = 2 * i, j <- [1..k]]
[(1,1),(1,2),(2,1),(2,2),(2,3),(2,4)]

Prelude> let xs = ['A'..'Z']
Prelude> [x | (i,x) <- [1..] `zip` xs, even i]
"BDFHJLNPRTVXZ"
```

zip xs ys

```
Prelude> let xs = ['A'..'Z']
Prelude> [x | (i,x) <- [1..] `zip` xs, even i]
"BDFHJLNPRTVXZ"

Prelude> :t zip
zip :: [a] -> [b] -> [(a, b)]

Prelude> let ys = ['A'..'E']
Prelude> zip [1..]  ys
[(1,'A'),(2,'B'),(3,'C'),(4,'D'),(5,'E')]
```

Observați diferența!

```
Prelude> zip [1..3] ['A'..'D']
[(1,'A'),(2,'B'),(3,'C')]

Prelude> [(x,y) | x <- [1..3], y <- ['A'..'D']]
[(1,'A'),(1,'B'),(1,'C'),(1,'D'),(2,'A'),(2,'B'),(2,'C')
    ,(2,'D'),(3,'A'),(3,'B'),(3,'C'),(3,'D')]
```

Lenevire (Lazyness)

Argumentele sunt evaluate doar când e necesar și doar cât e necesar

```
Prelude> head[]
*** Exception: Prelude.head: empty list
Prelude> let x = head []
Prelude> let f a = 5
Prelude> f x
5
Prelude> [1,head [],3] !! 0
1
Prelude> [head [],3] !! 1
*** Exception: Prelude.head: empty list
```

Liste infinite

Drept consecință a evaluării lenește, se pot defini liste infinite (fluxuri de date)

```
Prelude> let natural = [0,..]
Prelude> take 5 natural
[0,1,2,3,4]

Prelude> let evenNat = [0,2..] -- progresie infinita
Prelude> take 7 evenNat
[0,2,4,6,8,10,12]

Prelude> let ones = [1,1..]
Prelude> let zeros = [0,0..]
Prelude> let both = zip ones zeros
Prelude> take 5 both
[(1,0),(1,0),(1,0),(1,0),(1,0)]
```

Recursivitate

Recursivitate

Programarea declarativă VS Programarea imperativă

- modalitatea de abordare a problemei iterării;
- **Imperativă:** bucle (`while`, `for`...);
- **Declarativă:** recursie.

Un avantaj al recursiei față de bucle este acela că ușurează sarcina de scriere și verificare a corectitudinii programelor prin raționamente de tip inductiv: construiește rezultatul pe baza rezultatelor unor subprobleme mai simple (aceeași problemă, dar pe o dimensiune mai mică a datelor).

Recursivitate

Fibonacci

Un foarte simplu exemplu de recursie este acela al calculării unui element de index dat din secvența numerelor Fibonacci, definită recursiv de:

$$F_n = \begin{cases} n & \text{dacă } n \in \{0, 1\} \\ F_{n-1} + F_{n-2} & \text{dacă } n > 1 \end{cases}$$

Putem transcrie această definiție direct in Haskell:

```
fibonacciCazuri :: Integer -> Integer
fibonacciCazuri n
    | n < 2      = n
    | otherwise = fibonacciCazuri (n - 1) + fibonacciCazuri
      (n - 2)
```

Recursivitate

Fibonacci

Alternativ, putem folosi o definiție în stil ecuațional (cu șabloane):

```
fibonacciEcuational :: Integer -> Integer
fibonacciEcuational 0 = 0
fibonacciEcuational 1 = 1
fibonacciEcuational n =
    fibonacciEcuational (n - 1) + fibonacciEcuational (n -
        2)
```

Recursivitate

Fibonacci liniar

- definiția de mai sus: timp de execuție exponențial;
- rezultatul este compus din rezultatele a 2 subprobleme de mărime aproximativ egală cu cea inițială.

Recursia depinde doar de precedentele 2 valori
→ funcție care calculează recursiv perechea (F_{n-1}, F_n).

Recursivitate

Completați definiția funcției `fibonacciPereche`

- **Observație 1. inducție:** `fibonacciPereche (n-1)` va calcula perechea (F_{n-2}, F_{n-1}) și o folosim pe aceasta pentru a calcula perechea (F_{n-1}, F_n).
- **Observație 2.** Recursia este liniară doar dacă expresia care reprezintă apelul recursiv apare o singură dată. Folosiți `let`, `case`, sau `where` pentru a vă asigura de acest lucru.

Recursivitate

Fibonacci Pereche

```
fibonacciLiniar :: Integer -> Integer
fibonacciLiniar 0 = 0
fibonacciLiniar n = snd (fibonacciPereche n)
  where
    fibonacciPereche :: Integer -> (Integer, Integer)
    fibonacciPereche 1 = (0, 1)
    fibonacciPereche n = let (a,b)=fibonacciPereche (n-1) in
        (b,a+b)
```

Pe săptămâna viitoare!