Haskell: Legarea variabilelor. Structuri de date infinite. Funcționale

- Responsabil: Tiberiu Lepadatu [mailto:tiberiulepadatu14@gmail.com]

- Data publicării: 01.04.2019
- Data ultimei modificări: 01.04.2019

Obiective

Scopul acestui laborator îl reprezintă acomodarea cu noțiuni avansate de programare în Haskell, având în vedere scrierea de cod **clar** și **concis**.

Aspectele urmărite sunt:

- domenii de vizibilitate ale definițiilor: top-level și local
- definirea de structuri de date infinite
- funcționalele ca șabloane de proiectare
- programare „point-free“

Domenii de vizibilitate

Spre deosebire de Racket, unde legarea variabilelor la nivelul cel mai de sus (top-level) este dinamică, Haskell leagă definițiile **static**, acestea fiind vizibile implicit la nivel **global**. De exemplu, o definiție de forma:

```
theAnswer = 42
```

va putea fi utilizată implicit în toate fișierele încărcate de către compilator/interpretor la un moment dat. Domeniul de vizibilitate al definițiilor top-level poate fi însă redus cu ajutorul definirii de module: consultați capitolul 11 din A Gentle Introduction to Haskell [http://www.haskell.org/tutorial/modules.html] pentru mai multe detalii.

La fel ca Racket, Haskell permite definirea în cadrul domeniilor de vizibilitate locală (mai exact în cadrul funcțiilor), cu ajutorul clauzelor `let` și `where`.

let

Forma generală a clauzei `let` este următoarea:

```
let id1 = val1
    id2 = val2
    ...
    idn = valn
  in expr
```

unde `expr` este o expresie Haskell care poate depinde de `id1, id2, …, idn`. De asemenea, domeniul de vizibilitate ale definițiilor locale este întreaga clauză `let` (similar cu `letrec` în Racket). Astfel, definiția următoare:

```
p = let x = y + 1
        y = 2
        b n = if n == 0 then [] else n : b (n - 1)
  in (x + y, b 2)
```

este corectă. `x` poate să depindă de `y` datorită **evaluării leneșe**: în fapt `x` va fi evaluat în corpul clauzei, în cadrul expresiei `(x + y, b 2)`, unde `y` e deja definit.

where

Clauza `where` este similară cu `let`, diferența principală constând în folosirea acesteia **după** corpul funcției. Forma generală a acesteia este:

```
def = expr
  where
  id1 = val1
  id2 = val2
  ...
  idn = valn
```

cu aceleași observații ca în cazul `let`.

Un exemplu de folosire este implementarea metodei de sortare QuickSort:

```
qsort [] = []
qsort (p : xs) = qsort left ++ [p] ++ qsort right
  where
  left = filter (< p) xs
  right = filter (>= p) xs
```

Clauzele de tip `let` și `where` facilitează **reutilizarea** codului. De exemplu, funcția:

```
inRange :: Double -> Double -> String
inRange x max
  | f < low               = "Too low!"
  | f >= low && f <= high = "In range"
  | otherwise             = "Too high!"
  where
  f = x / max
  (low, high) = (0.5, 1.0)
```

verifică dacă o valoare normată se află într-un interval fixat. Expresia dată de `f` este folosită de mai multe ori în corpul funcției, motiv pentru care este urmărită încapsularea ei într-o definiție. De asemenea, se observă că definițiile locale, ca și cele top-level, permit pattern matching-ul pe constructorii de tip, în cazul acesta constructorul tipului pereche.

Observăm că `where` și `let` sunt de asemenea utile pentru definirea de funcții auxiliare:

```
naturals = iter 0
  where iter x = x : iter (x + 1)
```

`iter` având în exemplul de mai sus rolul de generator auxiliar al listei numerelor naturale.

Structuri de date infinite, funcționale

După cum am observat în cadrul laboratorului introductiv, Haskell facilitează definirea structurilor de date infinite, dată fiind natura **leneșă** a evaluării expresiilor. De exemplu mulțimea numerelor naturale poate fi definită după cum urmează:

```
naturals = [0..]
```

Definiția de mai sus este însă **zahăr sintactic** pentru următoarea expresie (exemplificată anterior):

```
naturals = iter 0
  where iter x = x : iter (x + 1)
```

Putem de asemenea să generăm liste infinite folosind funcționala iterate [http://hackage.haskell.org/package/base-4.6.0.1/docs/Prelude.html#v:iterate], având prototipul:

```
> :t iterate
iterate :: (a -> a) -> a -> [a]
```

`iterate` primește o funcție `f` și o valoare inițială `x` și generează o listă infinită din aplicarea repetată a lui `f`. Implementarea listei numerelor naturale va arăta deci astfel:

```
naturals = iterate (\ x -> x + 1) 0
```

Observăm că `iterate` este nu numai o funcțională, ci și un **șablon de proiectare** (design pattern). Alte funcționale cu care putem genera liste infinite sunt:

- `repeat`: repetă o valoare la infinit
- `intersperse`: introduce o valoare între elementele unei liste
- `zipWith`: generalizare a lui `zip` care, aplică o funcție binară pe elementele a două liste; e completată de `zipWith3` (pentru funcții ternare), `zipWith4` etc.
- `foldl`, `foldr`, `map`, `filter`

Exemple de utilizare:

```
ones = repeat 1 -- [1, 1, 1, ..]
onesTwos = intersperse 2 ones -- [1, 2, 1, 2, ..]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs) -- sirul lui Fibonacci
powsOfTwo = iterate (* 2) 1 -- puterile lui 2
palindromes = filter isPalindrome [0..] -- palindroame
  where
  -- truc: reprezint numarul ca String
  isPalindrome x = show x == reverse (show x)
```

Point-free programming

"Point-free style" [https://wiki.haskell.org/Pointfree] reprezintă o paradigmă de programare în care evităm menționarea explicită a parametrilor unei funcții în definiția acesteia. Cu alte cuvinte, se referă la scrierea unei funcții ca o succesiune de compuneri de funcții. Această abordare ne ajută, atunci când scriem sau citim cod, să ne concentrăm asupra obiectivului urmărit de algoritm deoarece expunem mai transparent ordinea în care sunt efectuate operațiile. În multe situații, codul realizat astfel este mai compact și mai ușor de urmărit.

De exemplu, putem să definim operația de însumare a elementelor unei liste in felul următor:

```
> let sum xs = foldl (+) 0 xs
```

Alternativ, în stilul „point-free“, vom evita descrierea explicită a argumentului funcției:

```
> let sum = foldl (+) 0
```

Practic, ne-am folosit de faptul că în Haskell toate funcțiile sunt în formă curry [https://wiki.haskell.org/Currying], aplicând parțial funcția `foldl` pe primele doua argumente, pentru a obține o expresie care așteaptă o listă și produce rezultatul dorit.

Pentru a compune două funcții, vom folosi operatorul `.`:

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

Dacă analizăm tipul operatorului, observăm că acesta primește ca argumente o funcție care acceptă o intrare de tipul `b` și întoarce o valoare de tipul `c`, respectiv încă o funcție care primește o intrare de tipul `a` și produce o valoare de tipul `b`, compatibilă cu intrarea așteptată de prima funcție. Rezultatul întors este o funcție care acceptă valori de tipul `a` și produce rezultate de tipul `c`.

Cu alte cuvinte, expresia `(f . g)(x)` este echivalentă cu `f(g(x))`.

Spre exemplu, pentru a calcula expresia `2*x+1` pentru orice element dintr-o listă, fără a folosi o funcție lambda, putem scrie direct:

```
> let sm = map ((+ 1) . (* 2))
> :t sm
sm :: [Integer] -> [Integer]
```

De observat că au fost necesare paranteze pentru ca întreaga expresie `(+ 1) . (* 2)` să reprezinte un singur argument pentru funcția `map`. Am putea obține un cod mai concis folosind operatorul `$`:

```
> let sm = map $ (+ 1) . (* 2)
> :t ($)
($) :: (a -> b) -> a -> b
```

`$` este un operator care aplică o funcție unară pe parametrul său. Semantica expresiei `f $ x` este aceeași cu cea a `f x`, diferența fiind pur sintactică: precedența `$` impune ordinea de evaluare astfel încât expresia din stânga `$` este aplicată pe cea din dreapta. Avantajul practic este că putem sa omitem parantezele în anumite situații, în general atunci când ar trebui să plasăm o paranteză de la punctul unde se află `$` până la sfârșitul expresiei curente.

De exemplu, expresiile următoare sunt echivalente:

```
> length (tail (zip [1,2,3,4] ("abc" ++ "d")))
> length $ tail $ zip [1,2,3,4] $ "abc" ++ "d"
```

Alternativ, folosind operatorul de compunere `.`:

```
> (length . tail . zip [1,2,3,4]) ("abc" ++ "d")
> length . tail . zip [1,2,3,4] $ "abc" ++ "d"
```

Atenție! Cei doi operatori, `.` și `$` nu sunt, în general, interschimbabili:

```
> let f = (+ 1)
> let g = (* 2)
> :t f . g
f . g :: Integer -> Integer
> :t f $ g
-- eroare, f așteaptă un Integer, g este o funcție
> f . g $ 2
5
> f . g 2
-- eroare, echivalent cu f . (g 2),  f . g (2)
> f $ g $ 2
5
> f $ g 2
5
```

Uneori, ne dorim să schimbăm ordinea de aplicare a argumentelor pentru o funcție. Un exemplu de funcție care ne-ar putea ajuta în acest sens este funcția `flip`, care interschimbă parametrii unei funcții binare:

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

De exemplu, dacă vrem să construim o funcție point-free care aplică o funcție primită ca argument pe fluxul numerelor naturale, am putea scrie:

```
> :t map
map :: (a -> b) -> [a] -> [b]
> :t flip map
flip map :: [a] -> (a -> b) -> [b]
> let f = flip map $ [0..]
> take 10 $ f (*2)
[0,2,4,6,8,10,12,14,16,18]
```

Un șablon de proiectare des folosit este `concatMap`, sau map/reduce [https://en.wikipedia.org/wiki/MapReduce], care implică compunerea funcționalelor `map` și `fold`. De exemplu, funcția `intersperse`, prezentată anterior, poate fi definită sub forma:

```
myIntersperse :: a -> [a] -> [a]
myIntersperse y = foldr (++) [] . map (: [y])
```

Observăm că al doilea argument al funcției `myIntersperse` nu este menționat explicit.

Atenție! Folosirea stilului „point-free“ poate să scadă în anumite cazuri lizibilitatea codului! Ideal este să folosiți construcții „point-free“ doar atunci când acestea fac programul **mai ușor** de înțeles. Proiectele Haskell de dimensiuni mari încurajează stilul „point-free“ **doar** împreună cu clauze `let` și `where`, și uneori cu funcții anonime.

Un exemplu de aplicație uzuală de „point-free style programming“, cu care probabil sunteți deja familiari, este folosirea operatorului pipe („|“) în unix shell scripting [http://www.vex.net/~trebla/weblog/pointfree.html].

Resurse

- Exerciții

- Solutii
- Cheatsheet

Referințe

- *Let vs Where [https://wiki.haskell.org/Let_vs._Where]*
- *Haskell: function composition (.) vs function application ($) (SO) [http://stackoverflow.com/questions/3030675/haskell-function-composition-and-function-application-idioms-correct-us]*
- *Pointfree (Haskell Wiki) [http://www.haskell.org/haskellwiki/Pointfree]*
- *Higher order function (Haskell Wiki) [http://www.haskell.org/haskellwiki/Higher_order_function]*
- *Learn You a Haskell [http://learnyouahaskell.com/chapters]*
- *Real World Haskell [http://book.realworldhaskell.org/read/]*
- *Hoogle [http://www.haskell.org/hoogle/]*
- *Hayoo! [http://holumbus.fh-wedel.de/hayoo/hayoo.html]*
- *Hackage [http://hackage.haskell.org/packages/hackage.html]*
- *Local definitions [http://en.wikibooks.org/wiki/Haskell/Variables_and_functions#Local_definitions]*
- *Comparație [http://www.lambdadays.org/static/upload/media/14254629275479beameraghfuneval2.pdf] între diferiți algoritmi de sortare implementați in limbaje funcționale*
