Programare declarativă[1]

Introducere în programarea funcțională folosind Haskell

Traian Florin Șerbănuță

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

12 octombrie 2016

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

De ce programare funcțională? De ce Haskell?

- E bine să știm cât mai multe limbaje de programare
- Programarea funcțională e din ce în ce mai importantă în industrie
  - Haskell e folosit în proiecte de Facebook, Google, Microsoft, ...
  - mai multe la `https://wiki.haskell.org/Haskell_in_industry`
- Programare funcțională în limbajul vostru preferat de programare:
  - Java 8, C++x11, C#, Python, PHP, JavaScript
  - Funcții anonime (λ-abstracții)
  - Funcții de procesare a fluxurilor de date: filter, map, reduce

De ce Haskell? (din cartea Real World Haskell)

[It] is a deep language and [...] learning it is a hugely rewarding experience.

Nou	Radical diferit de limbajele cu care suntem obișnuiți
Puternic	Cod concis, rapid și sigur
Plăcut	Tehnici elegante pentru rezolvarea de probleme concrete

Nou

Programarea funcțională

O cale profund diferită de a concepe ideea de software

- În loc să modificăm datele existente, calculăm valori noi din valorile existente, folosind funcții
- Funcțiile sunt pure: aceleași rezultate pentru aceleași intrări
- Distincție clară între părțile pure și cele care comunică cu mediul extern
- 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..]

Puternic

- Puritatea asigură consistență
  - O bucată de cod nu poate corupe datele altei bucăți de cod.
  - Mai ușor de testat decât codul care interacționează cu mediul
- Evaluarea lenesă poate fi exploatată pentru a reduce timpul de calcul fără a denatura codul

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

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

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`

Plan (orientativ) cursuri

1. Introducere
2. Funcții, liste, recursie
3. Operatori, funcții, recursie
4. Funcții de ordin înalt; Map, Filter, Fold
5. Tipuri de date algebrice
6. Tipuri de date algebrice - evaluarea expresiilor
7. Module și clase de tipuri
8. Intrare/Ieșire
9. Functori, Functori Applicativi, Monoizi
10. Monade I
11. Monade II
12. Zippers

Resurse

- Pagina Moodle a cursului:
  http://moodle.fmi.unibuc.ro/course/view.php?id=449
  - Prezentările cursurilor, forumuri, resurse electronice
  - Știri legate de curs vor fi postate pe Moodle
  - Notele la teste vor fi postate tot pe Moodle
  - Parola pentru accesarea paginii Moodle: progdecl
- http://bit.do/progdecl
  - Cele mai noi variante ale cursurilor si laboratoarelor.
- 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

Evaluare

Notare

- 3 teste (test1, test2, test3) și activitate laborator (lab)
- Nota finală: 40% test1 + 40% test2 + 20% test3 + 10% lab
- Notele vor fi postate (doar) pe pagina Moodle a cursului.

Promovabilitate

- Nota finală cel puțin 5
- Prezența la laborator e obligatorie
  - minim 50% din nr. total de prezențe

Test 1

- Pondere în calcularea notei finale: 40%
- La laborator în a 6-a săptămână
- Durată: 1 oră
- Acoperă materia din cursurile 1–4
- Cu acces la materiale descărcate pe calculator
- Fără acces la rețea/internet

Test 2

- Pondere în acordarea notei finale: 40%
- La curs, înainte de vacanță (16 decembrie)
- Durată: 1 oră
- Materia din cursurile 5–7
- Cu acces la materiale tipărite
- Fără acces la rețea/internet

Test 3

- Pondere în acordarea notei finale: 20%
- În sesiune
- Materia din cursurile 8–12
- Durată: 2 ore
- Cu acces la materiale tipărite
- Fără acces la rețea/internet

Activitate laborator

- Se va puncta activitatea în plus față de cerințele obșnuite
  - Probleme suplimentare
  - La alegerea profesorului din tematica laboratorului respectiv
  - Rezolvate în timpul laboratorului
- Maxim un punct pe laborator, la latitudinea profesorului
- Maxim 10 puncte în total
- Se adaugă la nota finală cu o pondere de 10%

Observații despre teste și notare

- Nu este necesar să dați toate testele pentru promovare
  - $test1 = 7,50 \wedge test2 = 5 \implies final = 5$
  - Dar este necesară nota agregată cel puțin 5
- Pentru a lua nota finală 10 sunt necesare toate cele 3 teste
  - $test1 = 10 \wedge test2 = 10 \wedge lab = 10 \implies final = 9$
  - $test1 = 10 \wedge test2 = 10 \wedge lab = 10 \wedge test3 = 2,50$
    $\implies final = 9,50 \approx 10$
  - $test1 = 9 \wedge test2 = 9 \wedge lab = 5 \wedge test3 = 9$
    $\implies final = 9,50 \approx 10$
- Cursurile se bazează unele pe altele
  - $\forall j < i$. Testul i presupune construcții și concepte acoperite de Testul j

Programare declarativă vs. imperativă

Ce vs. cum

Programare imperativă (Cum)

Explic mașinii, pas cu pas, algoritmic, cum să facă ceva și ca rezultat, se întâmplă ce voiam să se întâmple.

Programare declarativă vs. imperativă

Ce vs. cum

Programare imperativă (Cum)

Explic mașinii, pas cu pas, algoritmic, cum să facă ceva și ca rezultat, se întâmplă ce voiam să se întâmple.

Programare declarativă (Ce)

Îi spun mașinii ce vreau să se întâmple și o las pe ea să se prindă cum să realizeze acest lucru. :-)

Operații iterative pe colecții

http://latentflip.com/imperative-vs-declarative

Imperativ (JS)

```
var numbers = [1,2,3,4,5]
var doubled = []
for(var i = 0; i < numbers.length; i++) {
  doubled.push(numbers[i] * 2)
}
```

Operații iterative pe colecții

http://latentflip.com/imperative-vs-declarative

Imperativ (JS)

```
var numbers = [1,2,3,4,5]
var doubled = []
for(var i = 0; i < numbers.length; i++) {
  doubled.push(numbers[i] * 2)
}
```

Declarativ (Haskell)

```
numbers = [1,2,3,4,5]
doubled = map (\n -> n * 2) numbers
```

Agregarea datelor dintr-o colecție (JS)

http://latentflip.com/imperative-vs-declarative

Imperativ (JS)

```
var numbers = [1,2,3,4,5]
var total = 0

for(var i = 0; i < numbers.length; i++) {
  total += numbers[i]
}
```

Agregarea datelor dintr-o colecție (JS)

http://latentflip.com/imperative-vs-declarative

Imperativ (JS)

```
var numbers = [1,2,3,4,5]
var total = 0

for(var i = 0; i < numbers.length; i++) {
  total += numbers[i]
}
```

Declarativ (Haskell)

```
numbers = [1,2,3,4,5]
total = foldl (+) 0 numbers
```

Extragerea informației din tabele asociate

http://latentflip.com/imperative-vs-declarative

Imperativ (JS)

```
var dogsWithOwners = []
for(var di=0; di < dogs.length; di++) {
  dog = dogs[di]
  for(var oi=0; oi < owners.length; oi++) {
    owner = owners[oi]
    if (owner && dog.owner_id == owner.id) {
      dogsWithOwners.push({  dog: dog,   owner: owner })
    }
  }}
}
```

Extragerea informației din tabele asociate

http://latentflip.com/imperative-vs-declarative

Imperativ (JS)

```
var dogsWithOwners = []
for(var di=0; di < dogs.length; di++) {
  dog = dogs[di]
  for(var oi=0; oi < owners.length; oi++) {
    owner = owners[oi]
    if (owner && dog.owner_id == owner.id) {
      dogsWithOwners.push({  dog: dog,   owner: owner })
    }
  }}
}
```

Declarativ (SQL)

```
SELECT * FROM dogs INNER JOIN owners
WHERE dogs.owner_id = owners.id
```

Programare imperativă vs. declarativă

Diferențe

- Modelul de computație: algoritm vs. relație
- Ce exprimă un program: cum vs. ce
- Variabile/parametrii: atribuire distructivă vs. non-distructivă
- Structuri de date: alterabile vs. explicite
- Ordinea de execuție: efecte laterale vs. neimportantă
- Expresii ca valori: nu vs. da
- Controlul execuției: responsabilitatea programatorului vs a mașinii

Expresii și funcții

Signatura unei funcții

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

Definiții folosind if

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

Definiții folosind ecuații

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

Definiții folosind cazuri

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

Definiții de liste

- 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]
progresieInfinita = [3,7..]    -- [3,7,11,15,19,..]
```

- Definiții prin selecție

```
pare :: [Integer] -> [Integer]
pare xs = [x | x<-xs, even x]

pozitiiPare :: [Integer] -> [Integer]
pozitiiPare xs = [i | (i,x) <- [1..] `zip` xs, even x]
```

Tipuri. Clase de tipuri. Variabile de tip

```
Prelude> :t 'a'
'a' :: Char
Prelude> :t "ana"
"ana" :: [Char]
Prelude> :t 1
1 :: Num a => a
Prelude> :t [1,2,3]
[1,2,3] :: Num t => [t]
Prelude> :t 3.5
3.5 :: Fractional a => a
Prelude> :t (+)
(+) :: Num a => a -> a -> a
Prelude> :t (+3)
(+3) :: Num a => a -> a
Prelude> :t (3+)
(3+) :: Num a => a -> a
```

Expresii ca valori

Funcțiile — „cetățeni de rangul I”

- Funcțiile sunt valori care pot fi luate ca argument sau întoarse ca rezultat

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f = \ x y -> f y x
-- sau alternativ folosind matching
flip f x y = f y x
-- sau flip ca valoare de tip functie
flip = \ f x y -> f y x
-- Currying
flip = \f -> \x -> \y -> f y x
```

- Aplicare parțială a funcțiilor

```
injumatateste :: Integral a => a -> a
injumatateste = (`div` 2)
```

Funcții de ordin înalt

map, filter, foldl, foldr

```
Prelude> map (*3) [1,3,4]
[3,9,12]
Prelude> filter (>=2) [1,3,4]
[3,4]
Prelude> foldr (*) 1  [1,3,4]
12
Prelude> foldl (flip (:)) [] [1,3,4]
[4,3,1]
```

Compunere si aplicare

```
Prelude> map (*3) ( filter (<=3) [1,3,4])
[3,9]
Prelude> map (*3) . filter (<=3) $ [1,3,4]
[3,9]
```

Lenevire

- Argumentele sunt evaluate doar cand e necesar si doar cat e necesar

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

- Liste infinite (fluxuri de date)

```
ones = [1,1..]
zeros = [0,0..]
both = zip ones zeros
short = take 5 both   -- [(1,0),(1,0),(1,0),(1,0),(1,0)]
```

Intteractiune cu mediul extern

- Monade
- Acțiuni
- Secvențiere