Paradigme de Programare

Conf. dr. ing. Andrei Olaru

andrei.olaru@cs.pub.ro | cs@andreiolaru.ro
Departamentul de Calculatoare

2020

Cursul 6

Programare funcțională în Haskell

Cursul 6: Programare funcțională în Haskell

1. Introducere
2. Sintaxă
3. Evaluare

Introducere

Haskell

[https://en.wikipedia.org/wiki/Haskell_(programming_language)]

- din 1990;
- GHC – Glasgow Haskell Compiler (The Glorious Glasgow Haskell Compilation System)
  - dialect Haskell standard *de facto*;
  - compilează în/folosind C;
- Haskell Stack
- nume dat după logicianul Haskell Curry;
- aplicații: Pugs, Darcs, Linspire, Xmonad, Cryptol, seL4, Pandoc, web frameworks.

Paralelă între limbaje

Criteriu	Racket	Haskell
Funcții	*Curry* sau *uncurry*	*Curry*
Tipare	Dinamică, tare (-liste)	Statică, tare
Legarea variabilelor	Statică	Statică
Evaluare	Aplicativă	Normală (Lenesă)
Transferul parametrilor	*Call by sharing*	*Call by need*
Efecte laterale	`set!`*	Interzise

Sintaxă

Funcții

- toate funcțiile sunt *Curry*;
- aplicabile asupra oricâtor parametri la un moment dat.

Ex | Exemplu : Definiții echivalente ale funcției `add`:

Funcții

- toate funcțiile sunt *Curry*;
- aplicabile asupra oricâtor parametri la un moment dat.

Ex | Exemplu : Definiții echivalente ale funcției `add`:

```
1  add1          =    \x y -> x + y
2  add2          =    \x -> \y -> x + y
3  add3 x y      =    x + y
4
5  result        =    add1 1 2      -- echivalent, ((add1 1) 2)
6  result2       =    add3 1 2      -- echivalent, ((add3 1) 2)
7  inc           =    add1 1
```

Funcții vs operatori

- Aplicabilitatea parțială a operatorilor infixați
- Transformări operator $\rightarrow$ funcție și funcție $\rightarrow$ operator

Funcții vs operatori

- Aplicabilitatea parțială a operatorilor infixați
- Transformări operator → funcție și funcție → operator

Ex | Exemplu Definiții echivalente ale funcțiilor `add` și `inc`:

```
1  add4          =    (+)
2  result1       =    (+) 1 2
3  result2       =    1 `add4` 2
4
5  inc1          =    (1 +)
6  inc2          =    (+ 1)
7  inc3          =    (1 `add4`)
8  inc4          =    (`add4` 1)
```

Pattern matching

- Definirea comportamentului funcțiilor pornind de la structura parametrilor → traducerea axiomelor TDA.

Ex Exemplu

```
1  add5 0 y          =   y                    -- add5 1 2
2  add5 (x + 1) y    =   1 + add5 x y
3
4  sumList []        =   0                    -- sumList [1,2,3]
5  sumList (hd:tl)   =   hd + sumList tl
6
7  sumPair (x, y)    =   x + y                -- sumPair (1,2)
8
9  sumTriplet (x, y, z@(hd:_)) =              -- sumTriplet
10     x + y + hd + sumList z                 --  (1,2,[3,4,5])
```

List comprehensions

- Definirea listelor prin proprietățile elementelor, ca într-o specificare matematică

Ex Exemplu

```
1  squares lst       =    [x * x | x <- lst]
2
3  quickSort []      =    []
4  quickSort (h:t)   =    quickSort [x | x <- t, x <= h]
5                    ++   [h]
6                    ++   quickSort [x | x <- t, x > h]
7
8  interval          =    [0 .. 10]
9  evenInterval      =    [0, 2 .. 10]
10 naturals          =    [0 ..]
```

Evaluare

Evaluare

- Evaluare lenesă: parametri evaluați la cerere, cel mult o dată, eventual parțial, în cazul obiectelor structurate
- Transferul parametrilor: *call by need*
- Funcții nestricte!

Ex | Exemplu

```
1  f (x, y) z = x + x
```

Evaluare:

```
1  f (2 + 3, 3 + 5) (5 + 8)
2
3
4
```

Evaluare

- Evaluare lenesă: parametri evaluați la cerere, cel mult o dată, eventual parțial, în cazul obiectelor structurate
- Transferul parametrilor: *call by need*
- Funcții nestricte!

Ex Exemplu

```
1  f (x, y) z = x + x
```

Evaluare:

```
1  f (2 + 3, 3 + 5) (5 + 8)
2  → (2 + 3) + (2 + 3)
3
4
```

Evaluare

- Evaluare lenesă: parametri evaluați la cerere, cel mult o dată, eventual parțial, în cazul obiectelor structurate
- Transferul parametrilor: *call by need*
- Funcții nestricte!

Ex | Exemplu

```
1  f (x, y) z = x + x
```

Evaluare:

```
1  f (2 + 3, 3 + 5) (5 + 8)
2  → (2 + 3) + (2 + 3)
3  → 5 + 5          reutilizăm rezultatul primei evaluări!
4  → 10               ceilalți parametri nu sunt evaluați
```

Pași în aplicarea funcțiilor
Exemplu

Exemplu

```
1  frontSum (x:y:zs)    =    x + y
2  frontSum [x]         =    x
3
4  notNil []            =    False
5  notNil (_:_)         =    True
6
7  frontInterval m n
8      | notNil xs =    frontSum xs
9      | otherwise =    n
10     where
11         xs       =    [m .. n]
```

Pași în aplicarea funcțiilor
Ordine

1. Pattern matching: evaluarea parametrilor suficient cât să se constate (ne-)potrivirea cu *pattern*-ul;
2. Evaluarea gărzilor (`|`);
3. Evaluarea variabilelor locale, la cerere (`where`, `let`).

Pași în aplicarea funcțiilor

Exemplu – revisited

Ex | execuția exemplului anterior

```
1  frontInterval 3 5                          evaluare pattern
2  ??  notNil xs                              evaluare prima gardă
3  ??      where                              necesar xs → evaluare where
4  ??          xs = [3 .. 5]
5  ??          → 3:[4 .. 5]
6  ??  → notNil (3:[4 .. 5])
7  ??  → True
8  → frontSum xs                              evaluare valoare gardă
9      where
10         xs = 3:[4 .. 5]                    xs deja calculat
11         → 3:4:[5]
12 → frontSum (3:4:[5])
13 → 3 + 4 → 7
```

Consecințe

- Evaluarea parțială a structurilor – liste, tupluri etc.
- Listele sunt, implicit, văzute ca fluxuri!

Ex Exemplu

```
1   ones                =    1 : ones
2
3   naturalsFrom n      =    n : (naturalsFrom (n + 1))
4   naturals1           =    naturalsFrom 0
5   naturals2           =    0 : (zipWith (+) ones naturals2)
6
7   evenNaturals1       =    filter even naturals1
8   evenNaturals2       =    zipWith (+) naturals1 naturals2
9
10  fibo                =    0 : 1 : (zipWith (+) fibo (tail fibo
        ))
```

Sfârșitul cursului 6
Elemente esențiale

- Haskell, diferențe față de Racket
- pattern matching și list comprehensions
- evaluare în Haskell