PARADIGME DE PROGRAMARE

Curs 13

Limbajul Haskell – noțiuni avansate.

Limbajul Haskell – Cuprins

- Programare point-free
- Evaluare lenesă
- List comprehensions
- Comparație Racket-Haskell

Pointful versus point-free

Definirea funcțiilor

- Pointful: menționează explicit argumentele
- Point-free: nu menționează explicit argumentele

Exemplu

- Pointful

```
odds lst = filter odd lst
odds lst = (filter odd) lst
```

≡
(asociativitate la stânga)

- Point-free

```
odds = filter odd
```

(aplicație parțială)
(funcțiile sunt implicit curry)

Pointful versus point-free

Explicație (din punct de vedere al tipului expresiilor)

- Pointful
```
odds lst = filter odd lst
```
  - :t odds
```
odds :: Integral a => [a] -> [a]
```
- Point-free
```
odds = filter odd
```
  - :t filter
```
filter :: (a -> Bool) -> [a] -> [a]
```
  - :t odd
```
odd :: Integral a => a -> Bool
```
  - :t filter odd
```
filter odd :: Integral a => [a] -> [a]
```

- Primește o listă de întregi
- Întoarce o listă de întregi (întregii impari din input)

Mecanisme de sprijin

- Combinatori de bază
  - `.` `duplicateOdds = map (*2) . filter odd` (compunere)
  - `id` `onlyTrue = filter id` (id x = x)
  - `const` `resetList = map (const 0)` (const x y = x)

- Adaptarea funcțiilor
  - Secțiuni `isNegative = (< 0)` (operator binar ⟶ funcție unară)
  - `curry` `const' = curry fst` (curry f x y = f (x, y))
  - `uncurry` `sumPairs = map (uncurry (+))` (uncurry f (x, y) = f x y)
  - `flip` `rev = foldl (flip (:)) []` (flip f x y = f y x)

Implementări point-free – exerciții

- Implementări point-free – funcții pe liste

```
myLast =
myMin  =
myMax  =
mySum  =
myAnd  =
myAny  =
```

Implementări point-free – exerciții

- Implementări point-free – funcții pe liste

```
myLast = head . reverse
myMin  = head . sort
myMax  = myLast . sort
mySum  = foldl (+) 0
myAnd  = foldl (&&) True
myAny  = ((not . null) . ) . filter
```

Operatorul $ (aplicație de funcție)

Definiție: `f $ x = f x` (realizează aplicație de funcție)

- Prioritate minimă: evaluează ambele părți înainte de a efectua aplicația
- Asociativitate la dreapta: ~paranteză care se deschide aici și se închide la final (facilitează citirea codului de la dreapta la stânga)
  f (g (h … x)) devine f $ g $ h … x
- Rol:
  - Eliminarea unor paranteze (cele „de aici până la final”)
  - Utilizarea ca funcție: `map ($ 2) [(+1), (*2), (3^)]`

Exemplu

```
take 4 $ filter (odd . fst) $ zip [1..] [2..]
```

Operatorul $ (aplicație de funcție)

Definiție: `f $ x = f x` (realizează aplicație de funcție)

- Prioritate minimă: evaluează ambele părți înainte de a efectua aplicația
- Asociativitate la dreapta: ~paranteză care se deschide aici și se închide la final (facilitează citirea codului de la dreapta la stânga)
  f (g (h … x)) devine f $ g $ h … x
- Rol:
  - Eliminarea unor paranteze (cele „de aici până la final”)
  - Utilizarea ca funcție: `map ($ 2) [(+1), (*2), (3^)] --[3,4,9]`

Exemplu

```
take 4 $ filter (odd . fst) $ zip [1..] [2..] --[(1,2),(3,4),(5,6),(7,8)]
```

Limbajul Haskell – Cuprins

- Programare point-free
- Evaluare lenesă
- List comprehensions
- Comparație Racket-Haskell

Evaluare lenesă

- Funcții: toate funcțiile Haskell sunt nestricte
- Evaluare lenesă: **(sub)expresiile sunt evaluate doar când rezultatul lor este necesar, și sunt evaluate maximum o dată**
- Mecanism: o expresie neevaluată este o promisiune
  odată evaluată, expresia se înlocuiește cu valoarea ei (memoizare)

Exemplu

```
head [id 2, trace „8" 2*4]
```
nu evaluează al doilea element (→ nu afișează 8)

```
length [id 2, trace „8" 2*4]
```
nu evaluează niciun element (→ nu afișează 8)
(în calculul lungimii, valoarea elementelor nu e necesară)

Observație: În GHCi, :sprint x arată structura variabilei x (marcând părțile neevaluate cu _).

Evaluare lenesă

Exemplu care ilustrează memoizarea

```
1. f x = trace "x" 2*x
2. g x = f 2 + f 2
3. h x = x * x * x


g 5                 -- de câte ori se evaluează corpul funcției f?


h (f 2)             -- de câte ori se evaluează corpul funcției f?
```

Evaluare leneșă

Exemplu care ilustrează memoizarea

```
1. f x = trace "x" 2*x
2. g x = f 2 + f 2
3. h x = x * x * x


g 5                 -- 2 aplicări distincte ale lui f => se evaluează de 2 ori
                    -- f 2 + f 2  ->  4 + f 2  ->  4 + 4  ->  8

h (f 2)             -- argumentul se evaluează o dată și se folosește de 3 ori
                    -- x * x * x  -(eval x = f 2)->  4 * 4 * 4  ->  64
```

Consecință – toate listele sunt fluxuri!

Evaluare lenesă ⇒ toate listele se evaluează doar atât cât este necesar execuției

- Listă = Flux
- Listă infinită: se poate utiliza, cât timp programul consumă o porțiune limitată din listă

Exemple

```
naturals  =

ones      =

fibonacci =

evens     =
```

Consecință – toate listele sunt fluxuri!

Evaluare lenesă ⇒ toate listele se evaluează doar atât cât este necesar execuției

- Listă = Flux
- Listă infinită: se poate utiliza, cât timp programul consumă o porțiune limitată din listă

Exemple

```
naturals   = let loop n = n : loop (n+1) in loop 0

ones       = 1 : ones

fibonacci = 0 : 1 : zipWith (+) fibonacci (tail fibonacci)

evens      = filter even naturals
```

Generarea intervalelor

```
[start .. stop]          sau   [start ..]
[start, next .. stop] sau   [start, next ..] --next dă pasul
```

Exemple

```
[1 .. 5]
[1, 3 .. 10]
[10, 7 .. 0]
[20, 19.5 ..]
```

Generarea intervalelor

```
[start .. stop]          sau   [start ..]
[start, next .. stop] sau   [start, next ..] --next dă pasul
```

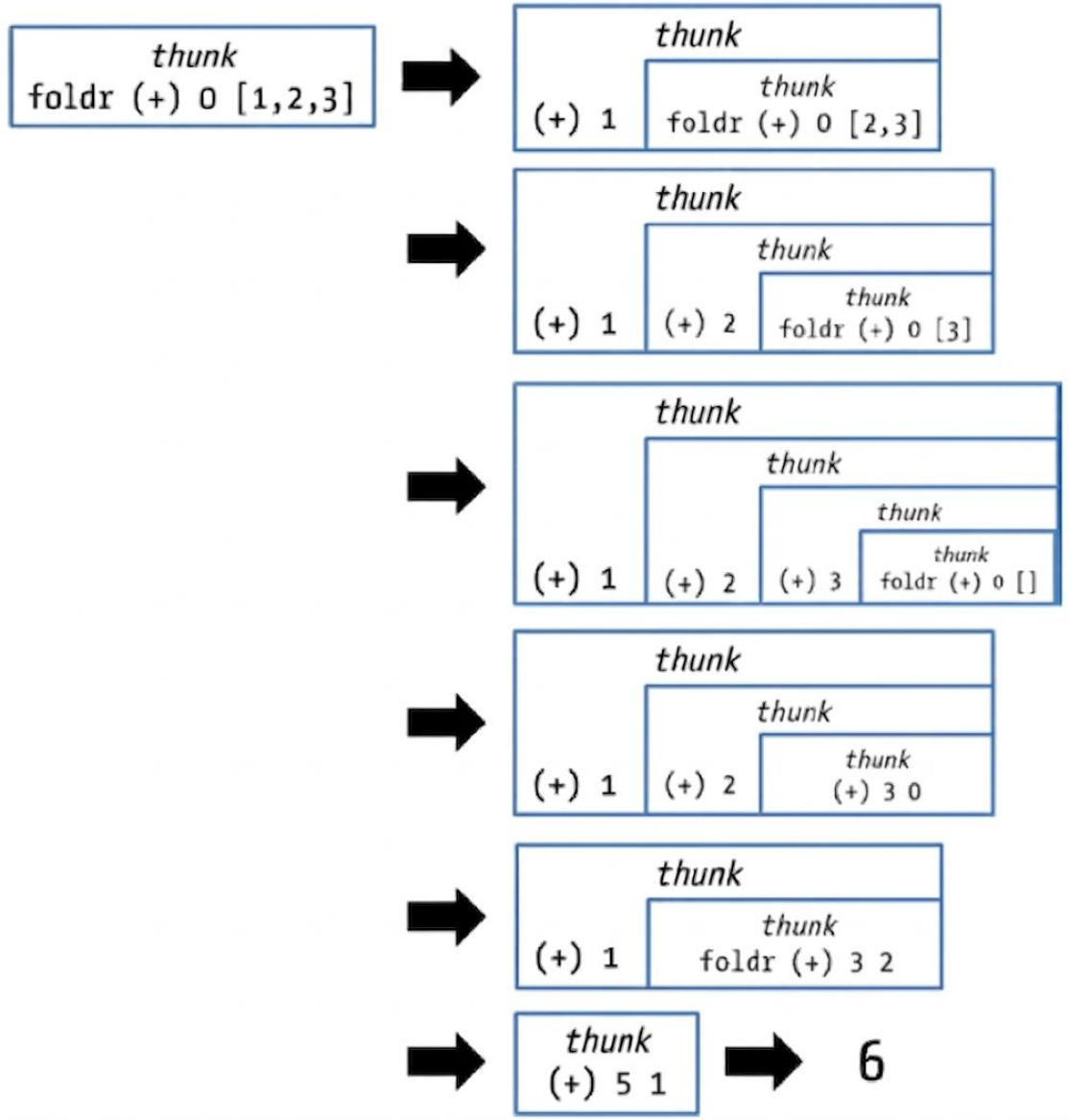
Exemple

```
[1 .. 5]          -- [1,2,3,4,5]
[1, 3 .. 10]      -- [1,3,5,7,9]
[10, 7 .. 0]      -- [10,7,4,1]
[20, 19.5 ..]     -- lista infinită [20,19.5,19,18.5..]
```

Limitări

Exemplu *(imagine: Beginning Haskell – Alejandro Serrano Mena)*

- Fiecare promisiune ocupă spațiu, până se ajunge la finalul listei
- Pe liste mari, poate genera stack overflow înainte de efectuarea primei adunări
- Observație:
  - `foldr` este recursivă pe stivă ⇒ nu se poate efectua nicio adunare înainte de a atinge finalul listei
  - contextul reținut pe stivă generează stack overflow și într-un limbaj cu evaluare strictă
  - `foldl` (pe coadă) rezolvă problema?



Limitări

- `foldl` (pe coadă) rezolvă problema?

- În Racket, da!

```
(foldl + 0 '(1 2 3)) →
(foldl + 1 '(2 3))   →
(foldl + 3 '(3))     →
(foldl + 6 '())      →
6
```

- În Haskell, nu!

```
foldl (+) 0                   [1, 2, 3] →
foldl (+) (0 + 1)             [2, 3]    →
foldl (+) ((0 + 1) + 2)       [3]       →
foldl (+) (((0 + 1) + 2) + 3) []        →
((0 + 1) + 2) + 3
```

Evaluarea leneșă consumă memorie

- Nu recursivitatea umple stiva
- Ci promisiunile de a aduna mai târziu, dacă este necesar
- O soluție: `foldl'` – forțează calculul acumulatorului la fiecare pas

Limbajul Haskell – Cuprins

- Programare point-free
- Evaluare lenesă
- List comprehensions
- Comparație Racket-Haskell

List comprehensions

List comprehension

- Metodă de a crea și procesa liste, inspirată din notația matematică pentru mulțimi
- Motto: începe cu rezultatul, apoi arată cum a fost generat

```
[ expr | generatori, predicate, legări locale ]
```

Exemple

```
lc1 = [ (x, y, z) | x <- [1..3], y <- [1..4], x < y, let z = x+y, odd z ]

fibo = 0 : 1 : [ x+y | (x, y) <- zip fibo (tail fibo) ]
```

Nu există list comprehension fără generator!

List comprehensions

List comprehension

- Metodă de a crea și procesa liste, inspirată din notația matematică pentru mulțimi
- Motto: începe cu rezultatul, apoi arată cum a fost generat

Explicație

```
lc1 = [ (x, y, z) | x <- [1..3], y <- [1..4], x < y, let z = x+y, odd z ]
```

• Expresia:	Ce să apară în noua listă?	Triplete (x, y, z)
• Generatorul:	Care este sursa datelor?	x <- [1..3], y <- [1..4]
• Predicatul:	Care din elementele generate interesează?	Doar variantele cu x < y, x + y impar
• Legarea locală:	Rol de lizibilitate și evitarea calculelor duplicate	

Generatoare multiple

- Ordinea contează
  - Generatorul din dreapta se schimbă cel mai repede (ca un `for` în interiorul altui `for`)

Explicație

```
lc1 = [ (x, y, z) | x <- [1..3], y <- [1..4], x < y, let z = x+y, odd z ]
```

- Pentru x = 1:
  - Pentru y = 1: x < y? (fail)
  - Pentru y = 2: x < y? z = 3 odd z? `[(1,2,3)`
  - Pentru y = 3: x < y? z = 4 odd z? (fail)
  - Pentru y = 4: x < y? z = 5 odd z? `[(1,2,3), (1,4,5)`
- Pentru x = 2:
  - Pentru y = 1: x < y? (fail)
  - Pentru y = 2: x < y? (fail)
  - Pentru y = 3: x < y? z = 5 odd z? `[(1,2,3), (1,4,5), (2,3,5)`

.....

Generatoare infinite

- Ordinea contează
  - Generatorul din dreapta se schimbă cel mai repede (ca un `for` în interiorul altui `for`)
  - Consecință: un generator infinit are sens doar dacă este primul!
    - Dacă generatorul din dreapta este infinit:
      - Bucla interioară nu se termină niciodată
      - Nu se trece niciodată la a doua valoare a generatorului din stânga

Exemple

```
increasingPairs  = [(x, sum - x) | sum <- [0 ..], x <- [0 .. div sum 2]]

increasingPairs' = [(x, sum - x) | x <- [0 ..], sum <- [2 * x ..]]
```

Generatoare infinite

- Ordinea contează
  - Generatorul din dreapta se schimbă cel mai repede (ca un `for` în interiorul altui `for`)
  - Consecință: un generator infinit are sens doar dacă este primul!
    - Dacă generatorul din dreapta este infinit:
      - Bucla interioară nu se termină niciodată
      - Nu se trece niciodată la a doua valoare a generatorului din stânga

Exemple

```
increasingPairs  = [(x, sum - x) | sum <- [0 ..], x <- [0 .. div sum 2]]
-- [(0,0),(0,1),(0,2),(1,1),(0,3),(1,2) ..

increasingPairs' = [(x, sum - x) | x <- [0 ..], sum <- [2 * x ..]]
-- [(0,0),(0,1),(0,2),(0,3),(0,4),(0,5) ..
```

Limbajul Haskell – Cuprins

- Programare point-free
- Evaluare lenesă
- List comprehensions
- Comparație Racket-Haskell

Racket versus Haskell

Criteriu	Racket	Haskell
Paradigmă	Multi-paradigmă	Pur funcțional
Sintaxă	Notație prefixată Paranteze pentru aplicația de funcție	Funcții – notație prefixată Operatori – notație infixată Paranteze pentru controlul priorității
Evaluare	Aplicativă	Lenșă
Legare	Statică Variante de legare dinamică top-level	Statică
Tipare	Dinamică – verificări la execuție	Statică – verificări la compilare
Recursivitate	TCO nativ	TCO leneș (optimizare afectată de reținerea promisiunilor în memorie)

Rezumat

Point-free

Combinatori

Mecanisme de adaptare a funcțiilor

Evaluare leneșă

Generarea intervalelor

List comprehensions

Rezumat

Point-free: definirea funcțiilor fără a menționa explicit argumentele

Combinatori

Mecanisme de adaptare a funcțiilor

Evaluare leneșă

Generarea intervalelor

List comprehensions

Rezumat

Point-free: definirea funcțiilor fără a menționa explicit argumentele

Combinatori: operatorul `.` (compunere), funcțiile `id` (identitate), `const` (ignoră al doilea argument)

Mecanisme de adaptare a funcțiilor

Evaluare leneșă

Generarea intervalelor

List comprehensions

Rezumat

Point-free: definirea funcțiilor fără a menționa explicit argumentele

Combinatori: operatorul `.` (compunere), funcțiile `id` (identitate), `const` (ignoră al doilea argument)

Mecanisme de adaptare a funcțiilor: secțiuni, `curry`, `uncurry`, `flip` (schimbă ordinea parametrilor)

Evaluare leneșă

Generarea intervalelor

List comprehensions

Rezumat

Point-free: definirea funcțiilor fără a menționa explicit argumentele

Combinatori: operatorul `.` (compunere), funcțiile `id` (identitate), `const` (ignoră al doilea argument)

Mecanisme de adaptare a funcțiilor: secțiuni, `curry`, `uncurry`, `flip` (schimbă ordinea parametrilor)

Evaluare leneșă: (sub)expresiile se evaluează doar când este necesar, și maxim o dată (memoizare)

Generarea intervalelor

List comprehensions

Rezumat

Point-free: definirea funcțiilor fără a menționa explicit argumentele

Combinatori: operatorul `.` (compunere), funcțiile `id` (identitate), `const` (ignoră al doilea argument)

Mecanisme de adaptare a funcțiilor: secțiuni, `curry`, `uncurry`, `flip` (schimbă ordinea parametrilor)

Evaluare leneșă: (sub)expresiile se evaluează doar când este necesar, și maxim o dată (memoizare)

Generarea intervalelor:

```
[start .. ],          [start .. stop],
[start, next .. ],    [start, next .. stop]
```

List comprehensions

Rezumat

Point-free: definirea funcțiilor fără a menționa explicit argumentele

Combinatori: operatorul `.` (compunere), funcțiile `id` (identitate), `const` (ignoră al doilea argument)

Mecanisme de adaptare a funcțiilor: secțiuni, `curry`, `uncurry`, `flip` (schimbă ordinea parametrilor)

Evaluare leneșă: (sub)expresiile se evaluează doar când este necesar, și maxim o dată (memoizare)

Generarea intervalelor:

```
[start .. ],          [start .. stop],
[start, next .. ],    [start, next .. stop]
```

List comprehensions: `[expr | generatori, predicate, legări locale]`

```
pattern <- listă      let pattern = expresie
```