

Лениво оценяване и програмиране от по-висок ред

Трифон Трифонов

Функционално програмиране, 2025/26 г.

11–18 декември 2025 г.

Тази презентация е достъпна под лиценза Creative Commons Признание-Некомерсиално-Споделяне на споделеното 4.0 Международен 

Щипка λ -смятане

- λ -изрази: $E ::= x \mid E_1(E_2) \mid \lambda x E$
- Изчислително правило: $(\lambda x E_1)(E_2) \mapsto E_1[x := E_2]$
- В какъв ред прилагаме изчислителното правило?
- Нека $f := \lambda x x!$, $g := \lambda z z^2 + z$
- $g(f(4)) \longrightarrow ?$
- $g(\underline{f(4)}) \longrightarrow g(\underline{4!}) \longrightarrow \underline{g(24)} \longrightarrow 24^2 + 24 \longrightarrow 600$
 - оценява се **отвътре навън**
 - **стриктно** (апликативно, лакомо) оценяване
- $\underline{g(f(4))} \longrightarrow (\underline{f(4)})^2 + \underline{f(4)} \longrightarrow (\underline{4!})^2 + \underline{4!} \longrightarrow 24^2 + 24 \longrightarrow 600$
 - оценява се **отвън навътре**
 - **нестриктно** (нормално, лениво) оценяване

Стриктно и нестриктно оценяване

Стриктното оценяване

- се използва в повечето езици за програмиране
- се нарича още „call-by-value“ (извикване по стойност)
- позволява лесно да се контролира редът на изпълнение
- пестеливо откъм памет, понеже „пази чисто“

Нестриктното оценяване

- е по-рядко използвано
- въпреки това се среща в някаква форма в повечето езици!
 - `x = p != nullptr ? p->data : 0;`
 - `found = i < n && a[i] == x`
- нарича се още „call-by-name“ (извикване по име)
- може да спести сметки, понеже „изхвърля боклуците“

Кога мързелът помага

```
(define (f x y) (if (< x 5) x y))
(define (g l) (f (car l) (cadr l)))
```

(g '(3)) \rightarrow (f (car '(3)) (cadr '(3)))
 \rightarrow (f 3 (cadr '(3))) \rightarrow **Грешка!**

```
f x y = if x < 5 then x else y
g l    = f (head l) (head (tail l))
```

g [3] \rightarrow f (head [3]) (head (tail [3]))
 \rightarrow if head [3] < 5 then head [3] else head (tail [3])
 \rightarrow if 3 < 5 then head [3] else head (tail [3])
 \rightarrow if True then head [3] else head (tail [3])
 \rightarrow head [3] \rightarrow 3

Теорема за нормализация

- всеки път когато апликативното оценяване дава резултат и нормалното оценяване дава резултат
- има случаи, когато нормалното оценяване дава резултат, но апликативното не!
- нещо повече:

Теорема (за нормализация, Curry)

Ако има някакъв ред на оценяване на програмата, който достига до резултат, то и с нормална стратегия на оценяване ще достигнем до същия резултат.

Следствие

*Ако с нормално оценяване програмата даде грешка или не завърши, то няма да получим резултат с **никая друга стратегия на оценяване**.*

Извикване при нужда („call-by-need“)

Ако $g(z) = z^2 + z$, $g(g(g(2))) = ?$

$$\begin{aligned} g(g(g(2))) &\mapsto g(g(2))^2 + g(g(2)) \mapsto (g(2)^2 + g(2))^2 + g(2)^2 + g(2) \mapsto \\ &\mapsto ((2^2 + 2)^2 + 2^2 + 2) + (2^2 + 2)^2 + 2^2 + 2 \mapsto \dots \end{aligned}$$

Времето и паметта нарастват експоненциално!

Идея: $(\lambda x E_1)(E_2) \mapsto \text{let } x = E_2 \text{ in } E_1$

$$\begin{aligned} g(g(g(2))) &\mapsto \text{let } x = g(g(2)) \text{ in } x^2 + x \mapsto \\ &\mapsto \text{let } y = g(2) \text{ in let } x = y^2 + y \text{ in } x^2 + x \mapsto \\ &\mapsto \text{let } z = 2 \text{ in let } y = z^2 + z \text{ in let } x = y^2 + y \text{ in } x^2 + x \mapsto \\ &\mapsto \text{let } y = 6 \text{ in let } x = y^2 + y \text{ in } x^2 + x \mapsto \\ &\mapsto \text{let } x = 42 \text{ in } x^2 + x \mapsto 1806 \end{aligned}$$

- Избягва се повторението чрез споделяне на общи подизрази
- Заместването се извършва чак когато е **абсолютно наложително**

Кога се налага оценяване на израз?

Във всеки даден момент Haskell оценява някой израз s .

- ако $s \equiv \text{if } e \text{ then } e_1 \text{ else } e_2$
 - първо се оценява e
 - ако оценката е **True**, се преминава към оценката на e_1
 - ако оценката е **False**, се преминава към оценката на e_2
- ако $s \equiv f\ e_1\ e_2\ \dots\ e_n$, за f — n -местна примитивна функция:
 - оценяват се последователно e_1, \dots, e_n
 - прилага се примитивната операция над оценките им
- нека сега да допуснем, че $s \equiv f\ e$
- първо се оценява f , за да разберем как да продължим
- ако $f\ x_1\ \dots\ x_n \mid g_1 = t_1\ \dots \mid g_k = t_k$ е дефинирана чрез пазачи:
 - тогава f се замества с изказа:

$$\backslash x_1\ \dots\ x_n \rightarrow \text{if } g_1 \text{ then } t_1 \text{ else } \dots \text{ if } g_k \text{ then } t_k \text{ else error } "..."$$
- ако f е конструктор (константа), **оценката остава $f\ e$**
- ако $f = \backslash p \rightarrow t$, където p е образец, редът на оценяване зависи от образца!

Кога се оценяват изразите при използване на образци?

Как се оценява $(\lambda p \rightarrow t)$ е?

- ако $p \equiv c$ е константа
 - преминава се към оценката на аргумента e
 - ако се установи че оценката тя съвпада с константата c , преминава се към оценката на тялото t
- ако $p \equiv _$ е анонимният образец
 - преминава се директно към оценката на t **без да се оценява e**
- ако $p \equiv x$ е променлива
 - преминава се към оценка на израза t **като се въвежда локалната дефиниция $x = e$**
- ако $p \equiv (p_1, p_2, \dots, p_n)$
 - преминава се към оценката на e
 - като се установи, че тя е от вида (e_1, e_2, \dots, e_n) , преминава се към оценката на израза $(\lambda p_1 p_2 \dots p_n \rightarrow t) e_1 e_2 \dots e_n$

Кога се оценяват изразите при използване на образци?

Как се оценява $(\lambda p \rightarrow t) e$?

- ако $p \equiv (p_h : p_t)$
 - преминава се към оценката на e
 - ако се установи, че тя е от вида $(e_h : e_t)$, преминава се към оценката на израза $(\lambda p_h p_t \rightarrow t) e_h e_t$
- ако $p \equiv [p_1, p_2, \dots, p_n]$
 - преминава се към оценката на e
 - ако се установи, че тя е от вида $[e_1, e_2, \dots, e_n]$, преминава се към оценката на израза $(\lambda p_1 p_2 \dots p_n \rightarrow t) e_1 e_2 \dots e_n$
 - всъщност е еквивалентно да разгледаме p като $p_1 : p_2 : \dots : p_n : []$
- ако има няколко равенства за f с използване на различни образци, се търси кой образец пасва отгоре надолу

Оценяване в Haskell: пример 1

`sumHeads (x:xs) (y:ys) = x + y`

`sumHeads [1..10] [5..50]`

→ `(\ (x:xs) -> \ (y:ys) -> x + y) [1..10] [5..50]`

→ `(\ (x:xs) -> \ (y:ys) -> x + y) (1:[2..10]) [5..50]`

→ `let x=1; xs=[2..10] in (\ (y:ys) -> x + y) [5..50]`

→ `let x=1; xs=[2..10] in (\ (y:ys) -> x + y) (5:[6..50])`

→ `let x=1; xs=[2..10]; y=5; ys=[6..50] in x + y`

→ `1 + 5` → 6

Оценяване в Haskell: пример 2

```

    (filter isPrime [4..1000]) !! 1
→ (\(_:xs) n -> xs !! (n-1)) (filter isPrime [4..1000]) 1
→ (\(_:xs) n -> xs !! (n-1)) (filter isPrime [4..1000]) 1
→ ... (\p (z:zs) ->      if p z then z:filter p zs
                        else filter p zs) isPrime [4..1000]...
→ ...let p=isPrime in (\(z:zs) -> if p z then z:filter p zs
                                else filter p zs) [4..1000]...
→ ...let p=isPrime in (\(z:zs) -> if p z then z:filter p zs
                                else filter p zs) (4:[5..1000]))...
→ ...let p=isPrime; z=4; zs=[5..1000] in
    if p z then z:filter p zs else filter p zs...
→ ...let p=isPrime; z=4; zs=[5..1000] in
    if False then z:filter p zs else filter p zs...

```

Оценяване в Haskell: пример 2

```

→ ... (\p (z:zs) -> if p z then z:filter p zs
      else filter p zs) isPrime [5..1000]...
→ ...let p=isPrime in (\ (z:zs) -> if p z then z:filter p zs
      else filter p zs) (5:[6..1000]))...
→ ...let p=isPrime; z=5; zs=[6..1000] in
    if p z then z:filter p zs else filter p zs...
→ ...let p=isPrime; z=5; zs=[6..1000] in
    if True then z:filter p zs else filter p zs...
→ (\(_:xs) n -> xs !! (n-1)) (5:filter isPrime [6..1000]) 1
→ let xs=filter isPrime [6..1000] in (\n -> xs !! (n-1)) 1
→ let xs=filter isPrime [6..1000]; n=1 in xs !! (n-1)
→ (\ (y:_) 0 -> y) (filter isPrime [6..1000]) 0

```

Оценяване в Haskell: пример 2

```

→ ...(\p (z:zs) ->    if p z then z:filter p zs
                      else filter p zs) isPrime [6..1000]...
→ ...let p=isPrime in (\(z:zs) -> if p z then z:filter p zs
                      else filter p zs) (6:[7..1000]))...
→ ...let p=isPrime; z=6; zs=[7..1000] in
    if p z then z:filter p zs else filter p zs...
→ ...let p=isPrime; z=6; zs=[7..1000] in
    if False then z:filter p zs else filter p zs...
→ ...(\p (z:zs) ->    if p z then z:filter p zs
                      else filter p zs) isPrime [7..1000]...
→ ...let p=isPrime in (\(z:zs) -> if p z then z:filter p zs
                      else filter p zs) (7:[8..1000]))...
→ ...let p=isPrime; z=7; zs=[8..1000] in
    if p z then z:filter p zs else filter p zs...

```

Оценяване в Haskell: пример 2

```
→ ...let p=isPrime; z=7; zs=[8..1000] in  
    if True then z:filter p zs else filter p zs ...  
→ (\ (y:_) 0 -> y) (7:filter isPrime [8..1000]) 0  
→ let y=7 in y  
→ 7
```

Потоци в Haskell

- Можем да си мислим, че аргументите в Haskell са **обещания**, които се изпълняват при нужда
- В частност, $x:xs = (:) \ x \ xs$, където
 - x е обещание за глава
 - xs е обещание за опашка
- **списъците в Haskell всъщност са потоци!**
- можем да работим с безкрайни списъци
 - `ones = 1 : ones`
 - `length ones` \longrightarrow ...
 - `take 5 ones` \longrightarrow `[1,1,1,1,1]`

Генериране на безкрайни списъци

- $[a..] \rightarrow [a, a+1, a+2, \dots]$
- Примери:
 - `nats = [0..]`
 - `take 5 [0..] → [0,1,2,3,4]`
 - `take 26 ['a'..] → "abcdefghijklmnopqrstuvwxyz"`
- Синтактична захар за `enumFrom from`
- $[a, a + \Delta x ..] \rightarrow [a, a + \Delta x, a + 2\Delta x, \dots]$
- Примери:
 - `evens = [0,2..]`
 - `take 5 evens → [0,2,4,6,8]`
 - `take 7 ['a','e'..] → "aeimquy"`
- Синтактична захар за `enumFromThen from then`

Генериране на безкрайни списъци

- `repeat :: a -> [a]`
 - създава безкрайния списък `[x,x,...]`
 - `repeat x = [x,x..]`
 - `repeat x = x : repeat x`
 - `replicate n x = take n (repeat x)`
- `cycle :: [a] -> [a]`
 - `cycle [1,2,3] → [1,2,3,1,2,3,...]`
 - `cycle 1 = 1 ++ cycle 1`
 - създава безкраен списък повтаряйки подадения (краен) списък
- `iterate :: (a -> a) -> a -> [a]`
 - `iterate f z` създава безкрайния списък `[z,f(z),f(f(z)),...]`
 - `iterate f z = z : iterate f (f z)`

Отделяне на безкрайни списъци

Отделянето на списъци работи и за безкрайни списъци.

- `oddSquares = [x^2 | x <- [1,3..]]`
- `twins = [(x,x+2) | x <- [3,5..], isPrime x, isPrime (x+2)]`
- `pairs = [(x,y) | x <- [0..], y <- [0..x - 1]]`
- `pythagoreanTriples = [(a,b,c) | c <- [1..],
b <- [1..c-1],
a <- [1..b-1],
a^2 + b^2 == c^2,
gcd a b == 1]`

Функции от по-висок ред над безкрайни списъци

Повечето функции от по-висок ред работят и над безкрайни списъци!

- `powers2 = 1 : map (*2) powers2`
- `notdiv k = filter (\x -> x `mod` k > 0) [1..]`
- `fibs = 0:1:zipWith (+) fibs (tail fibs)`
- `foldr (+) 0 [1..] → ...`
 - **Внимание:** `foldr` не работи над безкрайни списъци с операции, които изискват оценка на десния си аргумент!
 - `triplets = iterate (map (+3)) [3,2,1]`
 - `take 3 triplets → [[3,2,1],[6,5,4],[9,8,7]]`
 - `take 5 (foldr (++) [] triplets) → [3,2,1,6,5]`
 - `take 5 (foldl (++) [] triplets) → ...`
 - `foldl` не може да работи с безкрайни списъци!

Ленив вход и изход

- Ленивото оценяване в Haskell ни позволява да работим с входно/изходни потоци
- `getContents :: IO String` — връща списък от **ВСИЧКИ** символи на стандартния вход
- списъкът се оценява лениво, т.е. прочита се при нужда

- **Пример:**

```
noSpaces = do text <- getContents
             putStr (filter (/=' ') text)
```

- `interact :: (String -> String) -> IO ()` — лениво прилага функция над низове над стандартния вход и извежда резултата на стандартния изход

- **Пример:**

```
noSpaces = interact (filter (/=' '))
```

Апликация

- Операцията „апликация“ се дефинира с $f \$ x = f\ x$
- За какво може да бъде полезна?
- Операцията $\$$ е с най-нисък приоритет и е дясноасоциативна
 - за разлика от прилагането на функции, което е с най-висок приоритет и лявоасоциативно
- Може да бъде използвана за спестяване на скоби вложени надясно
- $(\dots ((f\ x_1)\ x_2)\ \dots\ x_n) = f\ x_1\ x_2\ \dots\ x_n$
- $f_1\ (f_2\ \dots\ (f_n\ x)\ \dots) = f_1\ \$\ f_2\ \$\ \dots\ \$\ f_n\ \$\ x$
- Примери:
 - `head $ tail $ take 5 $ drop 7 $ l`
 - `sum $ map (^2) $ filter odd $ [1..10]`
 - `map ($2) [(+2), (3^), (*5)] → [4,9,10]`

Композиция

- $(f \ . \ g) \ x = f \ (g \ x)$ — операция „композиция“
- с най-висок приоритет, дясноасоциативна
- Може да бъде използвана за спестяване на скоби вложени надясно
- $f_1 \ (f_2 \ \dots \ (f_n \ x) \ \dots) = f_1 \ . \ f_2 \ . \ \dots \ . \ f_n \ \$ \ x$
- **Примери:**
 - `sublist n m = take m . drop n`
 - `sumOddSquares = sum . map (^2) . filter odd`
 - `repeated n f x = foldr ($) x (replicate n f)`
 - `repeated n f = foldr (.) id (replicate n f)`
 - `repeated n f = foldr (.) id ((replicate n) f)`
 - `repeated n = foldr (.) id . replicate n`
 - `repeated n = (foldr (.) id .) (replicate n)`
 - `repeated = (foldr (.) id .) . replicate`

Разходване на памет при лениво оценяване

Ленивото оценяване може да доведе до голям разход на памет.

В Scheme:

- `(define (f x) (f (- 1 x)))`
- `(f 0)` \longrightarrow забива, но не изразходва памет
- `f` е **опашково-рекурсивна** и се реализира чрез итерация
- `(f 0)` \longrightarrow `(f 1)` \longrightarrow `(f 0)` \longrightarrow `(f 1)` \longrightarrow ...

В Haskell:

- `f x = f (1-x)`
- `f 0` \longrightarrow **забива с изтичане на памет!**
- `f 0` \longrightarrow `f (1-0)` \longrightarrow `f (1-(1-0))` \longrightarrow `f (1-(1-(1-0)))` ... \longrightarrow

Стриктно оценяване в Haskell

- в Haskell може да изискаме даден израз да се оцени веднага
- еквивалентно на форсиране на обещание
- `seq :: a -> b -> b`
- оценява първия си аргумент и връща втория като резултат
- **Примери:**
 - `second _ y = y`
 - `second (101010) 2 \longrightarrow 2`
 - `seq (101010) 2 \longrightarrow 2`
 - `f x = seq x (f (1-x))`
 - `f 0 \longrightarrow забива, но не изразходва памет!`
- `f $! x = seq x $ f x`
 - първо оценява `x` и след това прилага `f` над оценката на `x`
 - прилага `f` над `x` със стриктно оценяване
 - `f x = f $! (1-x)`

Изразходване на памет при foldl

```
foldl (+) 0 [1..4]
→ foldl (+) (0 + 1) [2..4]
→ foldl (+) ((0 + 1) + 2) [3..4]
→ foldl (+) (((0 + 1) + 2) + 3) [4..4]
→ foldl (+) ((((0 + 1) + 2) + 3) + 4) []
→ (((((0 + 1) + 2) + 3) + 4)
→ (((1 + 2) + 3) + 4)
→ ((3 + 3) + 4)
→ (6 + 4)
→ 10
```

Проблем: Изразходва памет при оценяване, понеже отлага изчисления!

Стриктен вариант на foldl

```
foldl' _ nv [] = nv
```

```
foldl' op nv (x:xs) = (foldl' op $! op nv x) xs
```

```
    foldl' (+) 0 [1..4]
```

```
→ foldl' (+) 1 [2..4]
```

```
→ foldl' (+) 3 [3..4]
```

```
→ foldl' (+) 6 [4..4]
```

```
→ foldl' (+) 10 []
```

```
→ 10
```