

Лениво оценяване и програмиране от по-висок ред

Трифон Трифонов

Функционално програмиране, 2018/19 г.

5 декември 2018 г.

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 != NULL ? 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)) → (f (car '(3)) (cadr '(3)))
           → (f 3 (cadr '(3))) → Грешка!
```

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

```
g [3] → f (head [3]) (head (tail [3]))
      → if head [3] < 5 then head [3] else head (tail [3])
      → if 3 < 5 then head [3] else head (tail [3])
      → if True then head [3] else head (tail [3])
      → head [3] → 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, Church-Rosser)

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

Следствие

*Ако с нормално оценяване програмата даде грешка или не завърши, то няма да получим резултат с **никоя 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 израза:


```
\x_1... x_n -> if g_1 then t_1 else ... if g_k then t_k
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 $(\backslash p \rightarrow t) e$?

- 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
 - ако се установи, че тя е от вида (e_1, e_2, \dots, e_n) , преминава се към оценката на израза $(\backslash p_1 p_2 \dots p_n \rightarrow t) e_1 e_2 \dots e_n$

Кога се оценяват изразите при използване на образци?

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

- 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 $(\setminus p_h p_t \rightarrow t) e_h e_t$
- 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 $(\setminus p_1 p_2 \dots p_n \rightarrow t) e_1 e_2 \dots e_n$
 - 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

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

```
  sumFirst [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
```