Лениво оценяване и програмиране от по-висок ред

Трифон Трифонов

Функционално програмиране, 2018/19 г.

5 декември 2018 г.

Щипка λ-смятане

- λ-изрази: $E ::= x \mid E_1(E_2) \mid \lambda x\, E$

Щипка λ-смятане

- λ-изрази: $E ::= x \mid E_1(E_2) \mid \lambda x\, E$
- Изчислително правило: $(\lambda x\, E_1)(E_2) \mapsto E_1[x := E_2]$

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?

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 $f := \lambda x\, x!$, $g := \lambda z\, z^2 + z$

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 $f := \lambda x\; x!$, $g := \lambda z\; z^2 + z$
- $g(f(4))$

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 $f := \lambda x\; x!$, $g := \lambda z\; z^2 + z$
- $g(f(4)) \longrightarrow \quad ?$

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 $f := \lambda x\; x!$, $g := \lambda z\; z^2 + z$
- $g(f(4)) \longrightarrow \quad ?$
- $g(\underline{f(4)})$

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 $f := \lambda x\; x!$, $g := \lambda z\; z^2 + z$
- $g(f(4)) \longrightarrow \quad ?$
- $g(\underline{f(4)}) \longrightarrow g(\underline{4!})$

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 $f := \lambda x\; x!$, $g := \lambda z\; z^2 + z$
- $g(f(4)) \longrightarrow \quad ?$
- $g(\underline{f(4)}) \longrightarrow g(\underline{4!}) \longrightarrow \underline{g(24)}$

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 $f := \lambda x\; x!$, $g := \lambda z\; z^2 + z$
- $g(f(4)) \longrightarrow \quad ?$
- $g(\underline{f(4)}) \longrightarrow g(\underline{4!}) \longrightarrow \underline{g(24)} \longrightarrow 24^2 + 24$

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 $f := \lambda x\; x!$, $g := \lambda z\; z^2 + z$
- $g(f(4)) \longrightarrow \quad ?$
- $g(\underline{f(4)}) \longrightarrow g(\underline{4!}) \longrightarrow \underline{g(24)} \longrightarrow 24^2 + 24 \longrightarrow 600$

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 $f := \lambda x\; x!$, $g := \lambda z\; z^2 + z$
- $g(f(4)) \longrightarrow \quad ?$
- $g(\underline{f(4)}) \longrightarrow g(\underline{4!}) \longrightarrow \underline{g(24)} \longrightarrow 24^2 + 24 \longrightarrow 600$

- $\underline{g(f(4))}$

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 $f := \lambda x\; x!$, $g := \lambda z\; z^2 + z$
- $g(f(4)) \longrightarrow \quad ?$
- $g(\underline{f(4)}) \longrightarrow g(\underline{4!}) \longrightarrow \underline{g(24)} \longrightarrow 24^2 + 24 \longrightarrow 600$

- $\underline{g(f(4))} \longrightarrow (\underline{f(4)})^2 + \underline{f(4)}$

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 $f := \lambda x\; x!$, $g := \lambda z\; z^2 + z$
- $g(f(4)) \longrightarrow \quad ?$
- $g(\underline{f(4)}) \longrightarrow g(\underline{4!}) \longrightarrow \underline{g(24)} \longrightarrow 24^2 + 24 \longrightarrow 600$

- $\underline{g(f(4))} \longrightarrow (\underline{f(4)})^2 + \underline{f(4)} \longrightarrow (\underline{4!})^2 + \underline{4!}$

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 $f := \lambda x\; x!$, $g := \lambda z\; z^2 + z$
- $g(f(4)) \longrightarrow \quad ?$
- $g(\underline{f(4)}) \longrightarrow g(\underline{4!}) \longrightarrow \underline{g(24)} \longrightarrow 24^2 + 24 \longrightarrow 600$

- $\underline{g(f(4))} \longrightarrow (\underline{f(4)})^2 + \underline{f(4)} \longrightarrow (\underline{4!})^2 + \underline{4!} \longrightarrow 24^2 + 24$

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 $f := \lambda x\, x!$, $g := \lambda z\, z^2 + z$
- $g(f(4)) \longrightarrow \quad ?$
- $g(\underline{f(4)}) \longrightarrow g(\underline{4!}) \longrightarrow \underline{g(24)} \longrightarrow 24^2 + 24 \longrightarrow 600$
  - оценява се отвътре навън

- $\underline{g(f(4))} \longrightarrow (\underline{f(4)})^2 + \underline{f(4)} \longrightarrow (\underline{4!})^2 + \underline{4!} \longrightarrow 24^2 + 24 \longrightarrow 600$
  - оценява се отвън навътре

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 $f := \lambda x\, x!$, $g := \lambda z\, z^2 + z$
- $g(f(4)) \longrightarrow \quad ?$
- $g(\underline{f(4)}) \longrightarrow g(\underline{4!}) \longrightarrow \underline{g(24)} \longrightarrow 24^2 + 24 \longrightarrow 600$
  - 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

Стриктно и нестриктно оценяване

Стриктното оценяване

- се използва в повечето езици за програмиране
- се нарича още “call-by-value” (извикване по стойност)

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

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”

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

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

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!

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;`

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`

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)

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)))
```

Кога мързелът помага

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

(g '(3)) $\longrightarrow$ (f (car '(3)) (cadr '(3)))

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)))
```

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!
```

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))
```

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]
```

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]))

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])
```

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])

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])
```

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]
```

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

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!

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.

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

Извикване при нужда (“call-by-need”)

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

$g(g(g(2)))$

Извикване при нужда (“call-by-need”)

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

$$g(g(g(2))) \quad \mapsto g(g(2))^2 + g(g(2))$$

Извикване при нужда (“call-by-need”)

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

$$g(g(g(2))) \mapsto g(g(2))^2 + g(g(2)) \mapsto (g(2)^2 + g(2))^2 + g(2)^2 + g(2)$$

Извикване при нужда (“call-by-need”)

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

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

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!

Извикване при нужда (“call-by-need”)

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

$$g(g(g(2))) \mapsto g(g(2))^2 + g(g(2)) \mapsto (g(2)^2 + g(2))^2 + g(2)^2 + g(2) \mapsto$$
$$\mapsto ((2^2 + 2)^2 + 2^2 + 2) + (2^2 + 2)^2 + 2^2 + 2 \mapsto \dots$$

Времето и паметта нарастват експоненциално!

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

Извикване при нужда (“call-by-need”)

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

$$g(g(g(2))) \mapsto g(g(2))^2 + g(g(2)) \mapsto (g(2)^2 + g(2))^2 + g(2)^2 + g(2) \mapsto$$
$$\mapsto ((2^2 + 2)^2 + 2^2 + 2) + (2^2 + 2)^2 + 2^2 + 2 \mapsto \ldots$$

Времето и паметта нарастват експоненциално!

Идея: $(\lambda x\, E_1)(E_2) \mapsto$ **let** $x = E_2$ **in** E_1

$g(g(g(2)))$

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: $(\lambda x\, E_1)(E_2) \mapsto \textbf{let } x = E_2 \textbf{ in } E_1$

$$g(g(g(2))) \mapsto \textbf{let } x = g(g(2)) \textbf{ in } x^2 + x$$

Извикване при нужда (“call-by-need”)

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

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

Времето и паметта нарастват експоненциално!

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

$$\begin{aligned} g(g(g(2))) &\mapsto \textbf{let } x = g(g(2)) \textbf{ in } x^2 + x \mapsto \\ &\mapsto \textbf{let } y = g(2) \textbf{ in let } x = y^2 + y \textbf{ in } x^2 + x \end{aligned}$$

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: $(\lambda x\, E_1)(E_2) \mapsto \textbf{let } x = E_2 \textbf{ in } E_1$

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

Извикване при нужда (“call-by-need”)

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

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

Времето и паметта нарастват експоненциално!

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

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

Извикване при нужда (“call-by-need”)

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

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

Времето и паметта нарастват експоненциално!

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

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

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: $(\lambda x\, E_1)(E_2) \mapsto \textbf{let } x = E_2 \textbf{ in } E_1$

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

- Избягва се повторението чрез споделяне на общи подизрази

Извикване при нужда (“call-by-need”)

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

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

Времето и паметта нарастват експоненциално!

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

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

- 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.

Кога се налага оценяване на израз?

Във всеки даден момент Haskell оценява някой израз s.

- ако $s \equiv$ `if` e `then` e_1 `else` e_2

Кога се налага оценяване на израз?

Във всеки даден момент Haskell оценява някой израз s.

- ако $s \equiv$ `if` e `then` e_1 `else` e_2
  - първо се оценява e

Кога се налага оценяване на израз?

Във всеки даден момент Haskell оценява някой израз s.

- ако $s \equiv$ `if` e `then` e_1 `else` e_2
  - първо се оценява e
  - ако оценката е `True`, се преминава към оценката на e_1

Кога се налага оценяване на израз?

Във всеки даден момент Haskell оценява някой израз s.

- ако $s \equiv$ `if` e `then` e_1 `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

Кога се налага оценяване на израз?

Във всеки даден момент Haskell оценява някой израз s.

- ако $s \equiv$ `if` e `then` e_1 `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` $e_1\ e_2\ \ldots\ e_n$, за `f` — n-местна примитивна функция:

Кога се налага оценяване на израз?

Във всеки даден момент Haskell оценява някой израз s.

- ако $s \equiv$ `if` e `then` e_1 `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` $e_1\ e_2\ \ldots\ e_n$, за `f` — n-местна примитивна функция:
  - оценяват се последователно $e_1, \ldots, e_n$

Кога се налага оценяване на израз?

Във всеки даден момент Haskell оценява някой израз s.

- ако $s \equiv$ `if` e `then` e_1 `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` $e_1\ e_2\ \ldots\ e_n$, за `f` — n-местна примитивна функция:
  - оценяват се последователно $e_1, \ldots, e_n$
  - прилага се примитивната операция над оценките им

Кога се налага оценяване на израз?

Във всеки даден момент Haskell оценява някой израз s.

- ако $s \equiv$ `if` e `then` e_1 `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` $e_1\ e_2\ \ldots\ e_n$, за `f` — n-местна примитивна функция:
  - оценяват се последователно $e_1, \ldots, e_n$
  - прилага се примитивната операция над оценките им
- нека сега да допуснем, че $s \equiv$ `f` e

Кога се налага оценяване на израз?

Във всеки даден момент Haskell оценява някой израз s.

- ако $s \equiv$ `if` e `then` e_1 `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` $e_1\ e_2\ \ldots\ e_n$, за `f` — n-местна примитивна функция:
  - оценяват се последователно $e_1, \ldots, e_n$
  - 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

Кога се налага оценяване на израз?

Във всеки даден момент Haskell оценява някой израз s.

- ако $s \equiv$ `if` e `then` e_1 `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 $s \equiv \texttt{f}\ e_1\ e_2\ \ldots\ e_n$, за `f` — n-местна примитивна функция:
  - оценяват се последователно $e_1, \ldots, e_n$
  - прилага се примитивната операция над оценките им
- нека сега да допуснем, че $s \equiv \texttt{f}\ e$
- първо се оценява `f`, за да разберем как да продължим
- ако $\texttt{f}\ x_1\ \ldots\ x_n \mid g_1 = t_1\ \ldots\ \mid g_k = t_k$ е дефинирана чрез пазачи:

Кога се налага оценяване на израз?

Във всеки даден момент Haskell оценява някой израз s.

- ако $s \equiv$ `if` e `then` e_1 `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` $e_1\ e_2\ \ldots\ e_n$, за `f` — n-местна примитивна функция:
  - оценяват се последователно $e_1, \ldots, e_n$
  - 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` $x_1\ \ldots\ x_n$ `|` g_1 `=` $t_1\ \ldots$ `|` g_k `=` t_k е дефинирана чрез пазачи:
  - тогава `f` се замества с израза:
    `\`$x_1\ \ldots\ x_n$ `->` `if` g_1 `then` t_1 `else` `...` `if` g_k `then` t_k
    `else error "..."`

Кога се налага оценяване на израз?

Във всеки даден момент Haskell оценява някой израз s.

- ако $s \equiv$ `if` e `then` e_1 `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` $e_1\ e_2\ \ldots\ e_n$, за `f` — n-местна примитивна функция:
  - оценяват се последователно $e_1, \ldots, e_n$
  - 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` $x_1\ \ldots\ x_n$ `|` g_1 `=` $t_1\ \ldots$ `|` g_k `=` t_k е дефинирана чрез пазачи:
  - тогава `f` се замества с израза:
    `\`$x_1\ \ldots\ x_n$ `->` `if` g_1 `then` t_1 `else` $\ldots$ `if` g_k `then` t_k
    `else error "..."`
- ако `f` е конструктор (константа), оценката остава `f` e

Кога се налага оценяване на израз?

Във всеки даден момент Haskell оценява някой израз s.

- ако $s \equiv$ `if` e `then` e_1 `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` $e_1\ e_2\ \ldots\ e_n$, за `f` — n-местна примитивна функция:
  - оценяват се последователно $e_1, \ldots, e_n$
  - 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` $x_1\ \ldots\ x_n$ `|` g_1 `=` $t_1\ \ldots$ `|` g_k `=` t_k е дефинирана чрез пазачи:
  - тогава `f` се замества с израза:
    `\`$x_1\ \ldots\ x_n$ `->` `if` g_1 `then` t_1 `else` $\ldots$ `if` g_k `then` t_k
    `else error "..."`
- ако `f` е конструктор (константа), оценката остава `f` e
- ако `f = \`p `->` t, където p е образец, редът на оценяване зависи от образеца!

Кога се оценяват изразите при използване на образци?

Как се оценява `(\p -> t) e`?

Кога се оценяват изразите при използване на образци?

Как се оценява `(\p -> t) e`?

- ако $p \equiv c$ е константа

Кога се оценяват изразите при използване на образци?

Как се оценява `(\p -> t) e`?

- ако $p \equiv c$ е константа
  - преминава се към оценката на аргумента e

Кога се оценяват изразите при използване на образци?

Как се оценява `(\p -> 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

Кога се оценяват изразите при използване на образци?

Как се оценява `(\p -> 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

Кога се оценяват изразите при използване на образци?

Как се оценява `(\p -> 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

Кога се оценяват изразите при използване на образци?

Как се оценява `(\p -> 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 $p \equiv \mathtt{x}$ е променлива

Кога се оценяват изразите при използване на образци?

Как се оценява `(\p -> 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`

Кога се оценяват изразите при използване на образци?

Как се оценява `(\p -> 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 $p \equiv (p_1, p_2, \ldots, p_n)$

Кога се оценяват изразите при използване на образци?

Как се оценява `(\p -> 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 $p \equiv (p_1, p_2, \ldots, p_n)$
  - преминава се към оценката на e

Кога се оценяват изразите при използване на образци?

Как се оценява `(\p -> 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 $p \equiv (p_1, p_2, \ldots, p_n)$
  - преминава се към оценката на e
  - ако се установи, че тя е от вида $(e_1, e_2, \ldots, e_n)$, преминава се към оценката на израза `(\p1 p2 ... pn -> t) e1 e2 ... en`

Кога се оценяват изразите при използване на образци?

Как се оценява `(\p -> t) e`?

Кога се оценяват изразите при използване на образци?

Как се оценява `(\p -> t) e`?

- ако $p \equiv (p_h : p_t)$

Кога се оценяват изразите при използване на образци?

Как се оценява `(\p -> t) e`?

- ако $p \equiv (p_h : p_t)$
  - преминава се към оценката на e

Кога се оценяват изразите при използване на образци?

Как се оценява `(\p -> 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 `(\p_h p_t -> t) e_h e_t`

Кога се оценяват изразите при използване на образци?

Как се оценява `(\p -> 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 `(\p_h p_t -> t) e_h e_t`
- ако $p \equiv [p_1, p_2, \ldots, p_n]$

Кога се оценяват изразите при използване на образци?

Как се оценява `(\p -> 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 `(\p_h p_t -> t) e_h e_t`
- ако $p \equiv [p_1, p_2, \ldots, p_n]$
  - преминава се към оценката на e

Кога се оценяват изразите при използване на образци?

Как се оценява `(\p -> 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 `(\ph pt -> t) eh et`
- ако $p \equiv [p_1, p_2, \ldots, p_n]$
  - преминава се към оценката на e
  - ако се установи, че тя е от вида $[e_1, e_2, \ldots, e_n]$, преминава се към оценката на израза `(\p1 p2 ... pn -> t) e1 e2 ... en`

Кога се оценяват изразите при използване на образци?

Как се оценява `(\p -> 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 `(\ph pt -> t) eh et`
- ако $p \equiv [p_1, p_2, \ldots, p_n]$
  - преминава се към оценката на e
  - ако се установи, че тя е от вида $[e_1, e_2, \ldots, e_n]$, преминава се към оценката на израза `(\p1 p2 ... pn -> t) e1 e2 ... en`
  - всъщност е еквивалентно да разгледаме p като $p_1 : p_2 : \ldots : p_n : []$

Кога се оценяват изразите при използване на образци?

Как се оценява `(\p -> 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 `(\p_h p_t -> t) e_h e_t`
- ако $p \equiv [p_1, p_2, \ldots, p_n]$
  - преминава се към оценката на e
  - ако се установи, че тя е от вида $[e_1, e_2, \ldots, e_n]$, преминава се към оценката на израза `(\p_1 p_2 ... p_n -> t) e_1 e_2 ... e_n`
  - всъщност е еквивалентно да разгледаме p като $p_1 : p_2 : \ldots : p_n : []$
- 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
```

Оценяване в Haskell: пример 1

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

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

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]
```

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]
```

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]
```

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])
```

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
```

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
```

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
```