Eta-redukce, pointfree vs. pointwise zápis, líné vyhodnocování, intenzionální seznamy

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Podzim 2017

Úvodní příklad

Příklad 4.1.1: Uvažme funkci `negp :: (a -> Bool) -> a -> Bool`, která neguje výsledek unárních predikátů (funkcí typu `a -> Bool`). Tj. funkce `negp` vrátí opačnou logickou hodnotu, než by vrátil zadaný predikát na zadané hodnotě.

a) Definujte funkci `negp` (můžete využít třeba funkci `not`).
b) Definujte funkci `negp` jako unární funkci (s použitím pouze jedného formálního parametru).
c) Definujte funkci `negp` bez použití formálních parametrů.

Eta-redukce (η-redukce)

Odstraňování formálních parametrů z definice funkce:

```
multiplyN n xs = map (n *) xs
multiplyN n    = map (n *)
multiplyN      = map . (*)
```

- někdy umožňuje funkci zapsat elegantněji
- často umožňuje zbavit se nutnosti použít λ-funkci
- nebezpečí nečitelného kódu:
  `(.) . (.)` vs. `\f g x y -> f (g x y)`

Eta-redukce (η-redukce)

Odstraňování formálních parametrů z definice funkce:

```
multiplyN n xs = map (n *) xs
multiplyN n    = map (n *)
multiplyN      = map . (*)
```

- někdy umožňuje funkci zapsat elegantněji
- často umožňuje zbavit se nutnosti použít λ-funkci
- nebezpečí nečitelného kódu:
  `(.) . (.)` vs. `\f g x y -> f (g x y)`

Pointfree vs. pointwise

- pointfree tvar je tvar bez formálních parametrů
- pointwise tvar je tvar se všemi formálními parametry
- oba tvary jsou ekvivalentní (tj. mají stejný typ i chování)

Pomocné funkce `flip` a `const`

Některé funkce nelze přímo přepsat do pointfree tvaru, potřebujeme pomocné funkce:

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f x y = f y x
```

- `flip` předá 2 parametry funkci v opačném pořadí

Pomocné funkce `flip` a `const`

Některé funkce nelze přímo přepsat do pointfree tvaru, potřebujeme pomocné funkce:

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f x y = f y x
```

- `flip` předá 2 parametry funkci v opačném pořadí

```
const :: a -> b -> a
const x y = x
```

- `const` zapomíná svůj druhý parametr a vrací první

Pomocné funkce `flip` a `const`

Některé funkce nelze přímo přepsat do pointfree tvaru, potřebujeme pomocné funkce:

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f x y = f y x
```

- `flip` předá 2 parametry funkci v opačném pořadí

```
const :: a -> b -> a
const x y = x
```

- `const` zapomíná svůj druhý parametr a vrací první

Pozor, vyhodnocení `const` může změnit typ!

```
\x -> const x (x True) :: (Bool -> b) -> Bool -> b
\x -> x                :: a -> a
```

Převody pointfree $\leftrightarrow$ pointwise I.

Příklad 4.1.2: Následující výrazy použijte v lokální definici a vyhodnoťte v interpretru jazyka Haskell na vhodných parametrech. Po úspěšné aplikaci výrazy upravujte tak, abyste se při jejich definici vyhnuli použití λ-abstrakce a formálních parametrů.

a) `\x -> 3 * x`
b) `\x -> x ^ 3`
c) `\x -> 3 + 60 `div` x ^ 2 > 0`
d) `\x -> [x]`
e) `\s -> "<" ++ s ++ ">"`
f) `\x -> 0 < 35 - 3 * 2 ^ x`
g) `\x y -> x ^ y`
h) `\x y -> y ^ x`
i) `\x y -> 2 * x + y`

Převody pointfree $\leftrightarrow$ pointwise II.

Příklad 4.1.4: Převeďte následující výrazy do pointwise tvaru:

a) `(^2) . mod 4 . (+1)`

b) `(+) . sum . take 10`

c) `map f . flip zip [1, 2, 3]` (funkce `f` je definována externě)

d) `(.)`

e) `flip flip 0`

f) `(.) (+) . (+)`

g) `(.(.))`

Líné vyhodnocování – teorie

„Vyhodnotíme jenom potřebné a jenom jednou.“

```
head [sum [1,2] + sum [1,2], answer lifeQuestions ]
⇝ sum [1,2] + sum [1,2]
⇝ 3 + 3
⇝ 6
```

Líné vyhodnocování – teorie

„Vyhodnotíme jenom potřebné a jenom jednou.“

```
head [sum [1,2] + sum [1,2], answer lifeQuestions ]
⇝ sum [1,2] + sum [1,2]
⇝ 3 + 3
⇝ 6
```

- Spoléháme se na referenční transparentnost.
- V obecnosti můžeme uvažovat, že každý podvýraz se vyhodnotí jenom jednou, ve skutečnosti však GHC všechny optimalizace nedělá (pokud si chcete být jisti, použijte definici přes `let` nebo `where`).
- Ovlivňuje časovou a paměťovou složitost vyhodnocení.

Líné vyhodnocování – příklad

Příklad 4.2.1: Uvažte význam líného vyhodnocování v následujících výrazech:

a) `take 10 [1..]`

b) `let f = f in fst (2, f)`

c) `let f [] = 3 in const True (f [1])`

d) `0 * div 2 0`

e) `snd ("a" * 10, id)`

Nekonečné seznamy

Užitečné seznamové funkce:

- notace `[a..c]`, `[a,b..c]`, `[a..]`, `[a,b..]` (i nečíselné seznamy)
- `take`, `(!!)`, `repeat`, `replicate`, `cycle`, `iterate`

Nekonečné seznamy

Užitečné seznamové funkce:

- notace `[a..c]`, `[a,b..c]`, `[a..]`, `[a,b..]` (i nečíselné seznamy)
- `take`, `(!!)`, `repeat`, `replicate`, `cycle`, `iterate`

Příklad 4.2.3: Pomocí některé z funkcí `iterate`, `repeat`, `replicate`, `cycle` vyjádřete nekonečné seznamy:

a) Seznam sestávající z hodnot `True`.
b) Rostoucí seznam všech mocnin čísla 2.
c) Rostoucí seznam všech mocnin čísla 3 se sudým exponentem.
d) Rostoucí seznam všech mocnin čísla 3 s lichým exponentem.
e) Alternující seznam `-1` a `1`: `[1,-1,1,-1, ...]`.
f) Seznam řetězců `["", "*", "**", "***", "****", ...]`.
g) Seznam zbytků po dělení `4` pro seznam `[1..]`:
   `[1,2,3,0,1,2,3,0,...]`.

Intenzionální definice seznamů

```
[ 2*n | n <- [10..15], odd n ]
```
$\rightsquigarrow^*$ `[22, 26, 30]`

Prvky seznamu generovány společnými pravidly:

- zkouší se všechny prvky z generátoru
- pro prvky vyhovující kvalifikátoru (podmínce) se do výsledného seznamu vloží výraz na levé straně

Intenzionální definice seznamů

Generátory:

- může jich být i více
- vyhodnocují se zleva doprava
- můžou používat vzory

Kvalifikátory:

- musí být výrazy typu `Bool`
- může jich být i více
- výraz na levé straně se přidá, pouze pokud platí všechny podmínky zároveň

Lokální definice:

- ve tvaru `let x = ...`, může jich být i více

Intenzionální seznamy

Příklad 4.3.1: Pomocí intensionálních seznamů definujte funkci `divisors`, která k zadanému přirozenému číslu vrátí seznam jeho kladných dělitelů.

Příklad 4.3.2: Intensionálním způsobem zapište výrazy, které se chovají stejně jako následující (předpokládejte externě definované funkce/hodnoty `f`, `p`, `s`, `x`):

a) `map f s`
b) `filter p s`
c) `map f (filter p s)`
d) `repeat x`
e) `replicate n x`
f) `filter p (map f s)`

Intenzionální seznamy

Příklad 4.3.4: Intensionálním způsobem zapište následující seznamy nebo funkce:

a) `[1,4,9,...,k^2]` (pro pevně dané externě definované `k`)
b) funkci `f`, která ze seznamu seznamů vybere jenom ty delší než 3 prvky
c) `"*****"`
d) `["","*","**","***",...]`
e) seznam seznamů `[[1],[1,2],[1,2,3],...]`
f) seznam všech dvojic přirozených čísel (zadefinujte tak, aby se ke každému prvku dalo dopočítat v konečném čase)