zip(With), η-redukce, pointfree vs. pointwise zápis, líné vyhodnocování, intensionální seznamy

IB015 Neimperativní programování

Kolektiv cvičících IB015

Fakulta informatiky, Masarykova univerzita

podzim 2018

Opakování

Naprogramujte funkci `longStringInitials :: [String] -> [Char]`, která vezme seznam řetězců a vrátí seznam prvních znaků z těch, které obsahují alespoň 5 znaků.

```
longStringInitials [] ⇝* []
longStringInitials ["", "vratv", "ne", "alejiste"]
⇝* ['v', 'a']
```

Další funkce na seznamech

Funkcí na seznamech je velké množství, mnohé užitečné lze nalézt v modulu `Data.List`[1], základní i přímo v `Prelude`[2].

Funkce `zip` a `unzip` umožňují převod mezi dvěma seznamy a seznamem dvojic:

```
zip [1,2,3] ['a','b','c'] ⇝*
    [(1,'a'),(2,'b'),(3,'c')]
unzip [(1,'a'),(2,'b'),(3,'c')] ⇝*
    ([1,2,3],['a','b','c'])
```

Délka výsledku funkce `zip` je určena délkou kratšího seznamu:

```
zip [1,2,3,4] ['a','b'] ⇝* [(1,'a'),(2,'b')]
```

[1] `http://hackage.haskell.org/package/base/docs/Data-List.html`
[2] `http://hackage.haskell.org/package/base/docs/Prelude.html#g:13`

Funkce `zip`

Příklad 4.1.3: Funkci `zip :: [a] -> [b] -> [(a,b)]` lze definovat následovně:

```
zip (x:s) (y:t) = (x,y) : zip s t
zip _     _     = []
```

a) Které dvojice parametrů vyhovují prvnímu řádku definice?
b) Přepište definici tak, aby první klauzule definice (první řádek) byla použita jako poslední klauzule definice.

Funkce `zipWith`: intuice

Funkce `zipWith` je zobecněním funkce `zip`:

- funkce `zip` spojuje prvky seznamů pomocí hodnotového konstruktoru uspořádané dvojice `(,)`
- funkce `zipWith` má navíc jako argument funkci, kterou určíme způsob spojení prvků ze seznamů

```
zipWith f [x,y] [z,q] ⇝* [f x z, f y q]
zipWith (,) [1,2] ['a','b'] ⇝* [(1,'a'),(2,'b')]
zipWith (*) [5,10,11] [3,4] ⇝* [15,40]
```

Funkce zipWith I

Příklad 4.1.6: Jaká je hodnota následujících výrazů?

a) `zipWith (^) [1..5] [1..5]`

b) `zipWith (:) "MF" ["axipes", "ík"]`

c)
```
let fibs = [0,1,1,2,3,5,8,13] in zipWith (+) fibs
  (tail fibs)
```

d)
```
let fibs = [0,1,1,2,3,5] in zipWith (/) (tail
  (tail fibs)) (tail fibs)
```

Funkce `zipWith` II

Příklad 4.1.8: Naprogramujte funkci
`connectEven :: Integral b => [a] -> [b] -> [(a, b)]`,
která dostane dva seznamy a vrátí jeden seznam dvojic `(x, y)`, kde `x` je prvek z prvního seznamu na *i*-té pozici, právě když `y` je sudý prvek z druhého seznamu na *i*-té pozici.
Pokud je jeden ze seznamů kratší, přesahující prvky z delšího seznamu ignorujte.
`connectEven [] [9, 10]` $\rightsquigarrow^*$ `[]`
`connectEven ['b', 'c', 'd'] [2, 3]` $\rightsquigarrow^*$ `[('b', 2)]`

Příklad 4.1.9: Napište funkci, která zjistí, jestli jsou v seznamu typu `Eq a => [a]` některé dva sousední prvky stejné. Úlohu zkuste vyřešit pomocí funkce `zipWith`.

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
- volba mezi nimi bývá věcí vkusu

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 `f` v opačném pořadí
- arita funkce `f` může být *vyšší* než 2!

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 `f` v opačném pořadí
- arita funkce `f` může být *vyšší* než 2!

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

- `const` zapomíná svůj druhý parametr a vrací první

Převody pointfree $\leftrightarrow$ pointwise

Příklad 4.2.2: 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 -> [x]`
d) `\x y -> x ^ y`
e) `\x y -> y ^ x`

Převody pointfree $\leftrightarrow$ pointwise

Příklad 4.2.2: 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 -> [x]`
d) `\x y -> x ^ y`
e) `\x y -> y ^ x`

Příklad 4.2.4: 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) `(.)`

Líné vyhodnocování

Příklad 4.3.1: Naprogramujte funkci `naturals`, která bude generovat seznam všech přirozených čísel.

Příklad 4.3.2: Uvažte význam líného vyhodnocování v následujících výrazech:

a) `take 10 naturals`
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 4.3.4: 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,...]`.

Intensionální definice seznamů

Příklad 4.4.1: Pomocí znalostí z minulých cvičení definujte funkci `fstOdd2s`, která pro daný seznam vygeneruje seznam všech dvojic prvků z tohoto seznamu takových, kde první člen je lichý.

Intensionální definice seznamů

Příklad 4.4.1: Pomocí znalostí z minulých cvičení definujte funkci `fstOdd2s`, která pro daný seznam vygeneruje seznam všech dvojic prvků z tohoto seznamu takových, kde první člen je lichý.

```
fstOdd2s xs = [(x1, x2) | x1 <- xs, odd x1, x2 <- xs]
```

Prvky seznamu generovány společnými pravidly:

- **generátory**: zkouší *všechny* n-tice nad danými seznamy zleva doprava, mohou používat vzory
- **kvalifikátory**: filtrují prvky vložené do výsledného seznamu
- můžeme využívat vnořené lokální definice pomocí `let`

Intensionální seznamy I

Příklad 4.4.2: Pomocí intensionálních seznamů definujte funkci `divisors`, která k zadanému přirozenému číslu vrátí seznam jeho kladných dělitelů.

Příklad 4.4.3: 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)

Intensionální seznamy II

Příklad 4.4.5: 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)`