Datové typy, seznamy, lambda-abstrakce, funkce na seznamech

IB015 Neimperativní programování

Tomáš Szaniszlo, Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Podzim 2017

Základní datové typy

Numerické

- celočíselné: `Int` (na platformě závislá velikost) a `Integer` (neomezený)
- s pohyblivou desetinnou čárkou: `Float`, `Double`

Logické hodnoty

- typ `Bool`
- `True` nebo `False`

Funkční typ

- `Parameter -> ... -> Parameter -> ReturnValue`

Nultice

- typ `()`
- jediná hodnota je `()`

Žádné implicitní konverze typů (nelze `Int` namísto `Integer`, ...)

Složené datové typy, n-tice, polymorfismus

Komplikovanější datové typy lze získat skládáním jednodušších:

- n-tice: `(Int, Bool)`, `(Double, Int, Int)`
- seznamy: `[Integer]`, `[Char]`
- vlastní datové typy

Často nezávislé na typu vnitřních hodnot (polymorfismus):

- polymorfní datové typy: `(a, b, c)` (trojice je definovaná pro libovolné typy `a`, `b`, `c`)
- polymorfní funkce: `fst :: (a, b) -> a`
- lze i kombinovat: `(Int, [(a, a)], Bool)`

Datové typy

Příklad 2.1.2: Nalezněte příklady hodnot následujících typů:

a) `Bool`
b) `Integer`
c) `Double`
d) `False`
e) `()`
f) `(Int, Integer)`
g) `(Integer, Double, Bool)`
h) `((), (), ())`

Datové typy

Příklad 2.1.3: Určete typy následujících výrazů, zkontrolujte si řešení pomocí interpretru.

a) `True`
b) `"True"`
c) `not True`
d) `True || False`
e) `True && "1"`
f) `f 1`, kde funkce `f` je definovaná jako

```
f :: Integer -> Integer
f x = x * x + 2
```

g) `f 3.14`, kde `f` je definovaná stejně jako v části f
h) `g 3.14`, kde `g` je definovaná jako

```
g :: Double -> Double
g x = x * x + 2
```

Seznamy

- homogenní kolekce
- seznam může obsahovat libovolný počet prvků daného typu
- typ seznamu zapisujeme jako `[ElementType]`, je to tedy parametrizovaný typ
- má sekvenční přístup k prvkům – k prvkům lze přistupovat pouze od začátku seznamu

Příklady:

- `[True, False]`
- `[1, 2, 42, 128]`
- `1:2:3:[]`

Datové konstruktory:

- `[] :: [a]` prázdný seznam (nad libovolným typem)
- `(:) :: a -> [a] -> [a]` připojí prvek na začátek seznamu

Seznamy

Příklad 2.3.1: Rozhodněte, které z následujících seznamů jsou správně utvořené. U nesprávných rozhodněte proč, u správně utvořených určete typ. Konzultujte své řešení s interpretrem.

a) `[1, 2, 3]`
b) `(1:2):3:[]`
c) `1:2:3:[]`
d) `1:(2:(3:[]))`
e) `[1, 'a', 2]`
f) `[ [], [1, 2], 1:[] ]`
g) `[ 1, [1, 2], 1:[] ]`
h) `[]:[]`

Seznamy

Příklad 2.3.2: Určete typy seznamů:

a) `["a", "b", "c"]`
b) `['a', 'b', 'c']`
c) `"abc"`
d) `[(True, ()), (False, ())]`
e) `[(++) "abc" "def", "X" ++ "Y" ++ "Z"]`
f) `[(&&), (||)]`
g) `[]`
h) `[[]]`
i) `[[], [""]]`

Vzory seznamů

Příklad 2.3.3: Pro následující vzory a seznamy určete, které vzory mohou reprezentovat které seznamy. Stanovte, jak se navážou proměnné ze vzoru.

vzory:

```
[], x, [x], [x,y], (x:s), (x:y:s), [x:s], (x:y):s
```

seznamy: `[1]`, `[1,2]`, `[1,2,3]`, `[[]]`, `[[1]]`, `[[1],[2,3]]`

Rekurzivní funkce na seznamech I.

Příklad 2.3.5: Definujte funkci `getLast :: [a] -> a`, která vrátí poslední prvek neprázdného seznamu. Nesmíte použít funkci `last`.

Příklad 2.3.7: Pomocí funkce `init` definujte funkci `median`, která vrátí medián konečného uspořádaného neprázdného seznamu. Medián seznamu je jeho v pořadí prostřední prvek. Pro seznam se sudým počtem prvků vraťte levý z dvojice ve středu.

Rekurzivní funkce na seznamech II.

Příklad 2.3.11: Definujte rekurzivní funkci `multiplyN :: Integer -> [Integer] -> [Integer]`, která vrátí seznam, v němž je každý prvek v druhém seznamovém parametru vynásoben číslem, které je prvním parametrem funkce.

Příklad 2.3.12: Definujte funkci `sums :: [[Int]] -> [Int]`, která ze seznamu seznamů čísel získá seznam součtů vnitřních seznamů. Funkci zadefinujte bez použití knihovních funkcí `map` a `sum`. Příklad použití funkce:

```
sums [[1,2,3], [0,1,0], [100], []] ⇝* [6, 1, 100, 0]
```

Funkce na seznamech: `filter`

```
filter :: (a -> Bool) -> [a] -> [a]
```

Vybere ze seznamu ty prvky, které splňují danou podmínku.

`filter odd [1,2,3,4,5,6]` $\rightsquigarrow^*$ `[1,3,5]`

Funkce na seznamech: `map`

```
map :: (a -> b) -> [a] -> [b]
```

Funkce `map` aplikuje zadanou funkci na každý prvek seznamu zvlášť a vrátí seznam výsledků aplikací.

```
map negate [1,-2,3] ⇝* [-1,2,-3]
```

Funkce na seznamech: `map`, `filter`

Příklad 2.3.16: Definujte funkci `evens :: [Integer] -> [Integer]`, která ze seznamu vybere sudá čísla. Použijte funkci `filter`.

Příklad 2.3.17: S využitím funkce `map` a knihovní funkce `toUpper :: Char -> Char` z modulu `Data.Char` (tj. je třeba použít `import Data.Char`, na začátku souboru, nebo `:m + Data.Char` v interpretru) definujte novou funkci `toUpperStr`, která převádí řetězec písmen na řetězec velkých písmen, tj. `toUpperStr "bob"` $\rightsquigarrow^*$ `"BOB"`.

Lambda-abstrakce

Anonymní funkce, umožňuje zapsat funkci přímo v místě volání.

- vhodné pro funkce jako `map`, `filter`
- `map (\x -> x ^ 2 + 2 * x + 1) [1, 2, 3]`
- obecně zapisujeme `\<parameters> -> function_body`
- parametrů může být více a oddělují se mezerami
- parametry můžou obsahovat vzory: `\(x, y) -> x + y`
- nelze zapsat ekvivalent víceřádkové definice nebo vyjádřit rekurzi

Lambda-abstrakce

Příklad 2.2.1: Které z následujících výrazů jsou korektní?

a) `\x y -> 0`
b) `\f -> f 0`
c) `(\s -> "ahoj, " ++ s, "to: " ++ s)`
d) `\x -> x . \y -> y x`
e) `\[(x, y)] z -> testIt x y z`
f) `\(_) [_] -> ()`
g) `\x y x -> y + 2 * x`
h) `(\x y -> x y) (\x y -> x y) (\x y -> x y)`
i) `\a b -> a (\c d e -> b c (d e))`
j) `\[] -> ()`

Funkce na seznamech: `map, filter`

Příklad 2.3.18: Definujte funkci `multiplyEven :: [Integer] -> [Integer]`, která vezme seznam čísel a vrátí seznam, který bude obsahovat všechna sudá čísla původního seznamu vynásobená 2. Nepoužívejte rekurzi explicitně.

Příklad: `multiplyEven [2,3,4]` $\rightsquigarrow^*$ `[4,8]`, `multiplyEven [6,6,3]` $\rightsquigarrow^*$ `[12,12]`.

Funkce na seznamech: `map, filter`

Příklad 2.3.23: Napište funkci `vowels`, která dostane seznam řetězců a vrátí seznam řetězců takových, že v každém řetězci ponechá jenom samohlásky (ale zachová jejich pořadí). Například `vowels ["ABC","DEF"]` se vyhodnotí na `["A","E"]`.