Datové typy, seznamy, vzory, funkce na seznamech, typové třídy

IB015 Neimperativní programování

Kolektiv cvičících IB015

Fakulta informatiky, Masarykova univerzita

podzim 2018

Opakování

Příklad 2.1.1: Napište rekurzivní funkci `divBy3`, která celočíselně vydělí zadaný argument číslem 3 bez použití funkce `div`.

Základní datové typy

Numerické

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

Char

- znaky jako `'a'`, `'z'`
- seznam znaků ([Char]) je totéž co textový řetězec (String)

Logické hodnoty

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

Funkční typ

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

Žádné implicitní konverze typů

(nelze `Int` namísto `Integer`, `Char` namísto `[Char]`, ...)

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)`
- ne vždy polymorfismus stačí, např. `(+) :: a -> a -> a` nebude fungovat (toto řeší typové třídy, které uvidíme za chvíli)

Datové typy

Příklad 2.1.4: Určete typy následujících výrazů, zkontrolujte si řešení pomocí interpretu.

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 8`, kde `g` je definovaná jako

```
g :: Int -> Int -> Int
g x y = x * y - 6
```

Otypování funkcí

Otypujte následující funkce z minula:

- `isSmallVowel`, která rozhodne, jestli je dané ASCII písmeno malou samohláskou.
- `mid`, která pro tři celá čísla `z1`, `z2` a `z3` vrátí to *prostřední* z nich (t.j. to druhé v jejich uspořádané trojici podle $\leq$).
- `power`, která bude brát dva argumenty `z` a `n` a vrátí n-tou mocninu čísla `z`.

První pohled na typové třídy

Typové třídy umožňují omezit polymorfismus funkcí:

- `(+) :: Num a => a -> a -> a`
  `Num a` představuje numerické typy
- `(==) :: Eq a => a -> a -> Bool`
  `Eq a` představuje typy porovnatelné na shodu
- `(<) :: Ord a => a -> a -> Bool`
  `Ord a` představuje uspořádatelné typy
- `show :: Show a => a -> String`
  `Show a` představuje typy, kterých hodnoty mají smysluplnou textovou reprezentaci

- u některých funkcí se také setkáte se třídou `Foldable t`, kterou si dočasně můžete představit jako třídu představující seznamy

První pohled na typové třídy

Část typu před `=>` označujeme jako **typový kontext**.
Omezení může být i více:
`notElem :: (Eq a, Foldable t) => a -> t a -> Bool`.

Při typování výrazů, které obsahují takto kvantifikovaně polymorfní typy, je nutné uvádět i přesný typový kontext.

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:[]`

Hodnotové konstruktory:

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

Seznamy

Příklad 2.2.1: 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) `[]:[]`

Vzory seznamů

Příklad 2.2.5: 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]]
```

Vzory seznamů

Příklad 2.2.4: Definujte funkce `myHead :: [a] -> a` (která vrátí první prvek seznamu) a `myTail :: [a] -> [a]` (která vrátí seznam bez prvního prvku). Nepoužívejte knihovní funkce `head, tail`.

Rekurzivní funkce na seznamech I.

Příklad 2.2.6: Napište následující rekurzivní funkce za pomoci vzorů:

a) `oddLength :: [a] -> Bool`, která vrátí `True`, pokud je seznam liché délky, jinak `False` (pomocí vzorů, bez použití funkce `length`).

b) `listSum :: [Int] -> Int`, která dostane seznam čísel a vrátí součet všech jeho prvků.

c) `deleteElem :: Eq a => a -> [a] -> [a]`, která ze seznamu odstraní všechny výskyty prvku zadaného prvním argumentem (nepoužívejte funkce `map`, `filter`).

d) `listsEqual :: Eq a => [a] -> [a] -> Bool`, která dostane na vstup dva seznamy a vrátí `True` právě tehdy, když se rovnají (bez použití funkce `==` na celé seznamy).

Rekurzivní funkce na seznamech II.

Příklad 2.2.7: Definujte funkci `getLast :: [a] -> a`, která vrátí poslední prvek neprázdného seznamu. Nesmíte použít funkci `last`.

Příklad 2.2.8: Definujte funkci `stripLast :: [a] -> [a]`, která pro neprázdný seznam vrátí tentýž seznam bez posledního prvku. Nesmíte použít funkci `init`.

Příklad 2.2.9: Pomocí rekurze napište funkci `elem' :: Eq a => a -> [a] -> Bool`, která vrací True, pokud je první argument obsažen v seznamu zadaném druhým argumentem, jinak vrací False.

Příklad 2.2.10: 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 III.

Příklad 2.2.14: 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 2.2.15: 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], []]` $\rightsquigarrow^*$ `[6, 1, 100, 0]`

Lokální definice

Lokální definice jsou pojmenované výrazy s lokální platností, jejich primárním účelem je zpřehlednit kód (a nezavádět nová globální jména). Rozeznáváme 2 druhy:

1. s definicí před výrazem (**let-in**)
   Syntax: `let var = subexpr in expr`
2. s definicí za výrazem (**where**)
   Syntax: `expr where var = subexpr`

- Je možné definovat i více výrazů v jedné definici:

```
let a = 25 + b
    b = 5
    in (a + b) * b - 1
```

- Na zarovnání záleží!
- V lokálních definicích je možné používat vzory.

Lokální definice

Příklad 2.3.1: S využitím lokálních definic upravte následující funkci, která z celočíselných koeficientů kvadratické rovnice spočítá počet reálných kořenů.

```
numRoots :: Int -> Int -> Int -> Int
numRoots a b c = if b^2 - 4 * a * c < 0
                 then 0
                 else if b^2 - 4 * a * c == 0
                      then 1 else 2
```

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

Příklad 2.2.20: 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"`.

Funkce na seznamech: `map, filter`

Příklad 2.2.21: 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 2.2.26: 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"]`.