Pokročilá syntaxe, moduly a balíky, pokročilé datové struktury

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Jaro 2018

Úvod

Organizace předmětu, domácí úkoly, podmínky zápočtu

- Viz interaktivní osnova v ISu.

Vývojové prostředí – možnosti

- **samostatná instalace**
  - kompilátor GHC, balíčkovací systém cabal (viz dále)
  - možnost instalovat přímo z repozitářů distribuce

Vývojové prostředí – možnosti

- **samostatná instalace**
  - kompilátor GHC, balíčkovací systém cabal (viz dále)
  - možnost instalovat přímo z repozitářů distribuce
- **Haskell Platform**
  - instalace „všechno v jednom“
  - cross-platform
  - možná zbytečně velká

Vývojové prostředí – možnosti

- **samostatná instalace**
  - kompilátor GHC, balíčkovací systém cabal (viz dále)
  - možnost instalovat přímo z repozitářů distribuce
- **Haskell Platform**
  - instalace „všechno v jednom“
  - cross-platform
  - možná zbytečně velká
- **The Haskell Tool Stack**
  - cross-platform systém pro pokročilý vývoj
  - umožňuje vyvýjet různé projekty najednou
  - determinističnost
  - složitější prostředí

Pokročilejší syntax: case

Používání vzorů uvnitř funkce

- stejně jako u vzorů funkcí se prochází odshora

```
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
mapMaybe _ []     = []
mapMaybe f (x:xs) =
```

Pokročilejší syntax: case

Používání vzorů uvnitř funkce

- stejně jako u vzorů funkcí se prochází odshora

```
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
mapMaybe _ []     = []
mapMaybe f (x:xs) =
    case f x of
         Just v  -> v : mapMaybe f xs
         Nothing ->     mapMaybe f xs
```

Pokročilejší syntax: stráže (guards)

```
lookSTree :: Ord k => k -> BinTree k -> Bool
lookSTree _ BEmpty = False
lookSTree x (BNode v l r)
```

Pokročilejší syntax: stráže (guards)

```
lookSTree :: Ord k => k -> BinTree k -> Bool
lookSTree _ BEmpty = False
lookSTree x (BNode v l r)
    | x == v    = True
    | x < v     = lookSTree x l
    | otherwise = lookSTree x r
```

Definice pomocí alternativ

- použije se tělo u první podmínky, která se vyhodnotí na `True`
- v `Prelude`: `otherwise = True`

Pokročilejší syntax: $

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Operátor aplikace funkce (priorita 0, asociativita zprava)

- `f $ g $ x` ≡ `f (g x)`
- `f . g $ h x` ≡ `(f . g) (h x)`

Pokročilejší syntax: $

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Operátor aplikace funkce (priorita 0, asociativita zprava)

- `f $ g $ x` ≡ `f (g x)`
- `f . g $ h x` ≡ `(f . g) (h x)`

Zamyšlení:
mezera jako operátor aplikace (priorita 10, asociativita zleva)

- `f g x` ≡ `(f g) x`

Pokročilejší syntax: $

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Operátor aplikace funkce (priorita 0, asociativita zprava)

- `f $ g $ x` ≡ `f (g x)`
- `f . g $ h x` ≡ `(f . g) (h x)`

Zamyšlení:
mezera jako operátor aplikace (priorita 10, asociativita zleva)

- `f g x` ≡ `(f g) x`

```
filter (>10) ( map (^2)   filter even [1..10] )
```

Pokročilejší syntax: $

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Operátor aplikace funkce (priorita 0, asociativita zprava)

- `f $ g $ x` ≡ `f (g x)`
- `f . g $ h x` ≡ `(f . g) (h x)`

Zamyšlení:
mezera jako operátor aplikace (priorita 10, asociativita zleva)

- `f g x` ≡ `(f g) x`

```
  filter (>10) ( map (^2)   filter even [1..10] )
( filter (>10) . map (^2) ) filter even [1..10]
```

Pokročilejší syntax: $

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Operátor aplikace funkce (priorita 0, asociativita zprava)

- `f $ g $ x` ≡ `f (g x)`
- `f . g $ h x` ≡ `(f . g) (h x)`

Zamyšlení:
mezera jako operátor aplikace (priorita 10, asociativita zleva)

- `f g x` ≡ `(f g) x`

```
  filter (>10) ( map (^2)   filter even [1..10] )
( filter (>10) . map (^2) ) filter even [1..10]
  filter (>10) . map (^2) $ filter even [1..10]
```

Vlastní datové typy: opakování

```
data BinTree a = BEmpty
               | BNode a (BinTree a) (BinTree a)
```

Získávání přes vzory:

```
getValue (BNode v _ _) = v
getLeft  (BNode _ l _) = l
getRight (BNode _ _ r) = r
```

Vlastní datové typy: záznamy

Záznamy: pojmenování hodnot v definici

```
data BinTree a = BEmpty
               | BNode { btValue :: a
                       , btLeft  :: BinTree a
                       , btRight :: BinTree a
                       }
```

Vlastní datové typy: záznamy

Záznamy: pojmenování hodnot v definici

```
data BinTree a = BEmpty
               | BNode { btValue :: a
                       , btLeft  :: BinTree a
                       , btRight :: BinTree a
                       }
```

- získávání standardně přes vzory

```
getRight (BNode _ _ r) = r
```

Vlastní datové typy: záznamy

Záznamy: pojmenování hodnot v definici

```
data BinTree a = BEmpty
               | BNode { btValue :: a
                       , btLeft  :: BinTree a
                       , btRight :: BinTree a
                       }
```

- získávání standardně přes vzory

```
getRight (BNode _ _ r) = r
```

- názvy hodnot ve vzorech

```
getRight (BNode { btRight = r} ) = r
```

Vlastní datové typy: záznamy

Záznamy: pojmenování hodnot v definici

```
data BinTree a = BEmpty
               | BNode { btValue :: a
                       , btLeft  :: BinTree a
                       , btRight :: BinTree a
                       }
```

- získávání standardně přes vzory

```
getRight (BNode _ _ r) = r
```

- názvy hodnot ve vzorech

```
getRight (BNode { btRight = r} ) = r
```

- názvy hodnot jako funkce

```
getRight node = btRight node
```

Vlastní datové typy: záznamy

Záznamy: pojmenování hodnot v definici

```
data BinTree a = BEmpty
               | BNode { btValue :: a
                       , btLeft  :: BinTree a
                       , btRight :: BinTree a
                       }
```

- získávání standardně přes vzory

```
getRight (BNode _ _ r) = r
```

- názvy hodnot ve vzorech

```
getRight (BNode { btRight = r} ) = r
```

- názvy hodnot jako funkce

```
getRight node = btRight node
```

- modifikace podmnožiny atributů

```
setVal node = node { btValue = 2 }
```

Typové aliasy

```
type String = [Char]
type Matrix a = [[a]]
```

- jen nové pojmenování, zaměňitelné s původním typem
- výjimka: instance typových tříd
- bez `deriving`
- pouze pro přehlednost kódu, pro kompilátor transparentní

```
map (map (+4)) matrix
```

newtype

```
newtype Matrix a = M { unM :: [[a]] } deriving Show
```

- nový typ (jako `data`) ⇒ typová kontrola překladače
- musí mít právě jeden unární datový konstruktor
- nutnot „rozbalování“ a „balení“

```
M (map (map (+4)) (unM matrix))
```

- rychlejší než `data`, další rozdíly s rozšířeními GHC[1]

[1] nad rámec kurzu: `-XGeneralizedNewtypeDeriving`

Typové díry I.

Otypování samostatného výrazu:

- přes povel interpretru `:t`

Typové díry I.

Otypování samostatného výrazu:

- přes povel interpretru `:t`

Otypování výrazu v kódě:

- použijeme typovou díru: `[ "a", "b", _ ]`
- vygeneruje typovou chybu obsahující požadovaný typ

```
> [ "a", "b", _ ]
<interactive>:1:12: error:
    * Found hole: _ :: [Char]
    * In the expression: ["1", "2", _]
    ...
```

Typové díry II.

Typove díry:

- názvy začínají podtržítkem
- ladění typových chyb ve složitějším kódu
- prototypování programu (namísto `undefined`)
- chyby lze odložit až do okamžiku volání pomocí přepínače GHC/GHCi `-fdefer-typed-holes`
- pozor: i stejně pojmenované typové díry jsou *různé* typové díry
- více v dokumentaci GHC a na GHC wiki

Moduly a balíky

Moduly (`Data.Char`, `Data.Map.Lazy`, ...)

- skupina funkcí ve stejném „jmenném prostoru“
- jméno modulu vždy velkými písmeny, hierarchie (tečky)
- základní modul `Prelude`, další načítáme pomocí `import`
- nemusí exportovat všechni své funkce

Moduly a balíky

Moduly (`Data.Char`, `Data.Map.Lazy`, ...)

- skupina funkcí ve stejném „jmenném prostoru“
- jméno modulu vždy velkými písmeny, hierarchie (tečky)
- základní modul `Prelude`, další načítáme pomocí `import`
- nemusí exportovat všechni své funkce

Balíky (`containers-0.5.10.1`, `brainfuck-0.1.0.3`, ...)

- skupina modulů, která se instaluje spolu
- název většinou malými písmeny, nemá hierarchii, má verzi
- balík `base` (základní balík modulů)
- spravuje typicky balíčkovací systém `cabal`

Moduly v praxi

Informace o modulech/balících:

- databáze balíků Hackage
  - resp. Stackage („Stable Hackage“)
- vyhledávač Hayoo (hledání podle funkcí, typů, balíků, . . .)

Moduly v praxi

Informace o modulech/balících:

- databáze balíků Hackage
  - resp. Stackage („Stable Hackage“)
- vyhledávač Hayoo (hledání podle funkcí, typů, balíků, . . .)

Použití při programování

- nainstalovat balík přes cabal:
  `cabal update && cabal install <package>`
- import v kódu: `import <module>`
- přidání modulu v GHCi: `:m + <module>`

Cabal

Nástroj pro správu balíků

- aktualizace databáze balíků: `cabal update`
- instalace balíku z Hackage: `cabal install <package>`
  - Pozor, `cabal uninstall` neexistuje!
- všechno ukládá do `$HOME/.cabal/`
  - konfiguraci a instalace je možno snadno smazat
- možnost použít sandboxy
  - umožňuje mít víc verzí
  - vhodné pro pokusy

Cabal: sandboxy

Použití sandboxů:

- inicializace sandboxu: `cabal sandbox init`
  - balíky pak instaluje lokálně do této složky
- instalace balíků: standardní `cabal install <package>`
  - musí být voláno se zložky sandboxu!
  - binárky v `./.cabal-sandbox/bin`
- Spuštění GHCi v sandboxu: `cabal repl`
  - GHC standardně nedokáže detekovat sandbox samo
- smazání sandboxu: `cabal sandbox delete`

Datové typy `Set` a `Map`

Množiny a asociativní mapy:

- `Set a`
  - `a` je typ hodnoty, musí být `Ord`
- `Map k v`
  - `k` je typ klíče, musí být `Ord`
  - `v` je typ hodnoty

Datové typy `Set` a `Map`

Množiny a asociativní mapy:

- `Set a`
  - `a` je typ hodnoty, musí být `Ord`
- `Map k v`
  - `k` je typ klíče, musí být `Ord`
  - `v` je typ hodnoty
- moduly `Data.Set` a `Data.Map`
  - balík `containers`, součást běžné distribuce GHC
  - `Map` dvou typů (*lazy* a *strict*)
  - vhodný kvalifikovaný import
- logaritmické vkládání, odstraňování, zjišťování minima a maxima

Kvalifikovaný import: ukázka

- Selektivní import

```
import Data.Set (Set, empty, insert)

courses :: Set String
courses = insert "IB016" empty
```

Kvalifikovaný import: ukázka

- Selektivní import

```
import Data.Set (Set, empty, insert)

courses :: Set String
courses = insert "IB016" empty
```

- Kvalifikovaný import

```
import qualified Data.Set as S

courses :: S.Set String
courses = S.insert "IB016" S.empty
```

Úkol: Cabal, sandboxy, množiny

1. Vytvořte nový Cabal sandbox.
2. Nainstalujte do něj balík `hello` a spustěte jeho binárku.
3. Implementujte jednoduchý systém pro práci s geometrií.
   - `newtype Point = P { unP :: (Double, Double) }`
   - `type Canvas = Map Point Char`
   - `furthestFrom :: Point -> Canvas -> Maybe Char`
     (určí nejvzdálenější bod od zadaného bodu)
   - `linePoints :: Point -> Point -> Canvas -> [Char]`
     (vrátí všechny body na zadané přímce)
   - `reflectY :: Canvas -> Canvas`
     (přidá body vzniklé zrcadlením přes osu Y)
   - další funkce dle vaší invence ...

- Tip: Před implementací těla funkce se vždy zamyslete, jaké pomocné funkce chcete a napište si jejich typové signatury.