

# Typy pro práci s řetězci a další užitečná rozšíření

## IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

jaro 2016

# Reprezentace řetězců v Haskellu: přehled

```
type String = [Char]
```

- jednoduché, přímo funguje se všemi seznamovými funkcemi
- paměťově neefektivní: cca 4.4 GB na načtení 100 MB souboru do paměti (44×)
- vhodné pro líné zpracování, malé řetězce

---

<sup>1</sup>pravděpodobně zahrnuje postupné alokace a reálný overhead je až 2× menší

# Reprezentace řetězců v Haskellu: přehled

`type String = [Char]`

- jednoduché, přímo funguje se všemi seznamovými funkcemi
- paměťově neefektivní: cca 4.4 GB na načtení 100 MB souboru do paměti (44×)
- vhodné pro líné zpracování, malé řetězce

`Text` (`Data.Text`, balík `text`)

- pro unicode text, interně UTF-16 (2 B na znak)
- striktní a líná varianta
- cca 5.6× více paměti pro načtení souvislého bloku<sup>1</sup>

---

<sup>1</sup>pravděpodobně zahrnuje postupné alokace a reálný overhead je až 2× menší

# Reprezentace řetězců v Haskellu: přehled

`type String = [Char]`

- jednoduché, přímo funguje se všemi seznamovými funkcemi
- paměťově neefektivní: cca 4.4 GB na načtení 100 MB souboru do paměti (44×)
- vhodné pro líné zpracování, malé řetězce

`Text` (`Data.Text`, balík `text`)

- pro unicode text, interně UTF-16 (2 B na znak)
- striktní a líná varianta
- cca 5.6× více paměti pro načtení souvislého bloku<sup>1</sup>

`ByteString` (`Data.ByteString`, balík `bytestring`)

- především pro binární data, síťovou komunikaci, volání do C
- striktní a líná varianta
- cca 2× overhead pro načtení souvislého bloku<sup>1</sup>

---

<sup>1</sup>pravděpodobně zahrnuje postupné alokace a reálný overhead je až 2× menší

```
import Data.Text ( Text )  
import qualified Data.Text as T
```

- definuje mnoho funkcí stejného jména jako **Prelude**
- případně `Data.Text.Lazy`
  - v podstatě seznam striktních úseků
- mnoho operací je optimalizovaných tak, že nevyžadují tvorbu mezilehlých hodnot (fusion)
- `pack :: String -> Text`
- `unpack :: Text -> String`
- `append :: Text -> Text -> Text`
- `cons, snoc, ...`

```
import qualified Data.Text.IO as T
```

- závislé na nastavení locale (v systému, nebo v GHC)
- `readFile :: FilePath -> IO Text`
- `appendFile`, `writeFile`, `getLine`, `putStr`
- práce s handle (`System.IO`)
- případně `Data.Text.Lazy.IO`

- `toLower`, `toUpper` `:: Text -> Text`
- `justifyLeft`, `justifyRight`, `center`  
`:: Int -> Char -> Text -> Text`
  - padding na danou šířku daným znakem

# Pokročilá práce s Unicode (balík `text-icu`)

- rozdělování na znaky, slova, věty
- porovnávání řetězců s ohledem na místní zvyklosti (čeština!)
  - případně porovnání bez ohledu na velikost písmen
- konverze kódování
- normalizace
- regulární výrazy nad unicode



# Pokročilá práce s Unicode (balík text-icu)

- rozdělování na znaky, slova, věty
- porovnávání řetězců s ohledem na místní zvyklosti (čeština!)
  - případně porovnání bez ohledu na velikost písmen
- konverze kódování
- normalizace
- regulární výrazy nad unicode

```
import qualified Data.Text as T (pack)
import qualified Data.Text.ICU.Normalize as T (compare)
import Data.Function (on)
```

```
compareLocale :: String -> String -> Ordering
compareLocale = T.compare [] `on` T.pack
```

# ByteString

```
import Data.ByteString ( ByteString )
import qualified Data.ByteString as BS
```

- kompaktní reprezentace binárních dat (nebo 8b znaků)
- striktní verze (v podstatě C-style `char * + hlavička`) vs. líná verze (seznam striktních bloků)
  - různé typy
- binární (`Data.ByteString`) vs. znakový (`Data.ByteString.Char8`) pohled:
  - `pack :: [Word8] -> ByteString` vs. `pack :: String -> ByteString`
  - stený typ, jen jiná metoda přístupu
  - znaková verze bere jen spodních 8 bitů z `Char`!
- opět definuje vlastní `IO` funkce, podporuje fusion

# Konverze mezi Text a ByteString

- např. binární data obsahující UTF-8 znaky
- `decodeUtf8 :: ByteString -> Text`
  - může vyhodit výjimku pokud vstup není validní UTF-8
  - `decodeUtf8' :: ByteString -> Either UnicodeException Text`
- i varianty pro UTF-16, UTF-32 v big endian/little endian
- `encodeUtf8 :: Text -> ByteString`

Řetězcový literál je typu `String`

- `"ahoj" :: String`
- `Data.ByteString.Char8.pack "ahoj" :: ByteString`
- `Data.Text.pack "ahoj" :: Text`
- volat neustále `pack` je nepraktické

Řetězcový literál je typu `String`

- `"ahoj" :: String`
- `Data.ByteString.Char8.pack "ahoj" :: ByteString`
- `Data.Text.pack "ahoj" :: Text`
- volat neustále `pack` je nepraktické

## Jak je to u čísel?

- celočíselný literál je typu `Num` a `=>` a
- ve skutečnosti je výsledkem aplikace `fromInteger` na dané číslo reprezentované jako `Integer`
- vzory se překládají pomocí `==`

# Reprezentace řetězcových literálů s rozšířením GHC

- podobný princip jako u čísel
- datová třída `IsString` (modul `Data.String` v base)
- rozšíření `OverloadedStrings`

# Reprezentace řetězcových literálů s rozšířením GHC

- podobný princip jako u čísel
- datová třída `IsString` (modul `Data.String` v base)
- rozšíření `OverloadedStrings`

```
{-# LANGUAGE OverloadedStrings #-}
```

```
import Data.Text ( Text )
```

```
foo :: Text
```

```
foo = "This will be of the Text type"
```

- literály mají typ `IsString` a `=>` a
- možno používat i ve vzorech (překládá se na `==`)
- pozor, může být nutné přidat typovou deklaraci

# Kontejnerové typy

často pracujeme s typy, které představují kontejnery

- [], Map, Set, ... ale také ByteString, Text
- chtěli bychom některé typy práce nad nimi zobecnit
  - iterace/výpočet nad strukturou (Functor, Foldable, Monoid, Traversable)
  - syntax a vzory jako pro seznamy



# Kontejnerové typy

často pracujeme s typy, které představují kontejnery

- [], Map, Set, ... ale také ByteString, Text
- chtěli bychom některé typy práce nad nimi zobecnit
  - iterace/výpočet nad strukturou (Functor, Foldable, Monoid, Traversable)
  - syntax a vzory jako pro seznamy
- → rozšířit syntax tak aby se „seznamový literál“ [x, y, z] překládal na fromList [x, y, z] pro fromList z nějaké vhodné typové třídy
  - podobně jako inicializátory v mnohých jazycích

# Kontejnerové typy

často pracujeme s typy, které představují kontejnery

- [], Map, Set, ... ale také ByteString, Text
- chtěli bychom některé typy práce nad nimi zobecnit
  - iterace/výpočet nad strukturou (Functor, Foldable, Monoid, Traversable)
  - syntax a vzory jako pro seznamy
- → rozšířit syntax tak aby se „seznamový literál“ [x, y, z] překládal na fromList [x, y, z] pro fromList z nějaké vhodné typové třídy
  - podobně jako inicializátory v mnohých jazycích
- → mít univerzální funkci toList, která umožní (línou) konverzi na seznam
  - jistým způsobem odpovídá iterátorům

# Problém s typem prvků

- ne všechny kontejnerové typy mají právě jeden typový parametr popisující typ obsahu
  - `Map` `k v` inicializováno `[(k, v)]`
  - `Text` inicializován `[Char]`
- abstrakce by měla být dostatečně obecná aby to zvládla

# Problém s typem prvků

- ne všechny kontejnerové typy mají právě jeden typový parametr popisující typ obsahu
  - `Map` `k v` inicializováno `[(k, v)]`
  - `Text` inicializován `[Char]`
- abstrakce by měla být dostatečně obecná aby to zvládla

## myšlenka: typová funkce

- `Item` container – pro daný typ kontejneru vrátí typ prvků z nichž se dá zkonstruovat
  - `Item [Integer]`  $\rightsquigarrow$

# Problém s typem prvků

- ne všechny kontejnerové typy mají právě jeden typový parametr popisující typ obsahu
  - `Map` `k v` inicializováno `[(k, v)]`
  - `Text` inicializován `[Char]`
- abstrakce by měla být dostatečně obecná aby to zvládla

## myšlenka: typová funkce

- `Item` container – pro daný typ kontejneru vrátí typ prvků z nichž se dá zkonstruovat
  - `Item [Integer] ~> Integer`

# Problém s typem prvků

- ne všechny kontejnerové typy mají právě jeden typový parametr popisující typ obsahu
  - `Map` `k v` inicializováno `[(k, v)]`
  - `Text` inicializován `[Char]`
- abstrakce by měla být dostatečně obecná aby to zvládla

## myšlenka: typová funkce

- `Item` container – pro daný typ kontejneru vrátí typ prvků z nichž se dá zkonstruovat
  - `Item [Integer] ~> Integer`
  - `Item (Set Int) ~>`

# Problém s typem prvků

- ne všechny kontejnerové typy mají právě jeden typový parametr popisující typ obsahu
  - `Map` `k v` inicializováno `[(k, v)]`
  - `Text` inicializován `[Char]`
- abstrakce by měla být dostatečně obecná aby to zvládla

## myšlenka: typová funkce

- `Item` container – pro daný typ kontejneru vrátí typ prvků z nichž se dá zkonstruovat
  - `Item [Integer] ~> Integer`
  - `Item (Set Int) ~> Int`

# Problém s typem prvků

- ne všechny kontejnerové typy mají právě jeden typový parametr popisující typ obsahu
  - `Map` `k v` inicializováno `[(k, v)]`
  - `Text` inicializován `[Char]`
- abstrakce by měla být dostatečně obecná aby to zvládla

## myšlenka: typová funkce

- `Item` container – pro daný typ kontejneru vrátí typ prvků z nichž se dá zkonstruovat
  - `Item [Integer] ~> Integer`
  - `Item (Set Int) ~> Int`
  - `Item (Map String Int) ~>`



# Problém s typem prvků

- ne všechny kontejnerové typy mají právě jeden typový parametr popisující typ obsahu
  - `Map` `k v` inicializováno `[(k, v)]`
  - `Text` inicializován `[Char]`
- abstrakce by měla být dostatečně obecná aby to zvládla

## myšlenka: typová funkce

- `Item` container – pro daný typ kontejneru vrátí typ prvků z nichž se dá zkonstruovat
  - `Item [Integer] ~> Integer`
  - `Item (Set Int) ~> Int`
  - `Item (Map String Int) ~> (String, Int)`

# Problém s typem prvků

- ne všechny kontejnerové typy mají právě jeden typový parametr popisující typ obsahu
  - `Map` `k v` inicializováno `[(k, v)]`
  - `Text` inicializován `[Char]`
- abstrakce by měla být dostatečně obecná aby to zvládla

## myšlenka: typová funkce

- `Item` container – pro daný typ kontejneru vrátí typ prvků z nichž se dá zkonstruovat
  - `Item [Integer] ~> Integer`
  - `Item (Set Int) ~> Int`
  - `Item (Map String Int) ~> (String, Int)`
  - `Item Text ~>`

# Problém s typem prvků

- ne všechny kontejnerové typy mají právě jeden typový parametr popisující typ obsahu
  - `Map` `k v` inicializováno `[(k, v)]`
  - `Text` inicializován `[Char]`
- abstrakce by měla být dostatečně obecná aby to zvládla

## myšlenka: typová funkce

- `Item` container – pro daný typ kontejneru vrátí typ prvků z nichž se dá zkonstruovat
  - `Item [Integer] ~> Integer`
  - `Item (Set Int) ~> Int`
  - `Item (Map String Int) ~> (String, Int)`
  - `Item Text ~> Char`

# Problém s typem prvků

- ne všechny kontejnerové typy mají právě jeden typový parametr popisující typ obsahu
  - `Map` `k` `v` inicializováno `[(k, v)]`
  - `Text` inicializován `[Char]`
- abstrakce by měla být dostatečně obecná aby to zvládla

## myšlenka: typová funkce

- `Item` container – pro daný typ kontejneru vrátí typ prvků z nichž se dá zkonstruovat
  - `Item [Integer] ~> Integer`
  - `Item (Set Int) ~> Int`
  - `Item (Map String Int) ~> (String, Int)`
  - `Item Text ~> Char`
- `fromList :: [Item l] -> l`
- `toList :: l -> [Item l]`

# Kontejnerové typy – realizace

- nelze vyjádřit ve standardním Haskellu
- rozšíření `OverloadedLists` (od GHC 7.8), typová třída `IsList` (`GHC.Exts`)

```
class IsList l where
  type Item l -- type function

  fromList :: [Item l] -> l
  toList   :: l -> [Item l]
  fromListN :: Int -> [Item l] -> l
  fromListN _ = fromList
```

- typové funkce fungují díky rozšíření `TypeFamilies`

# Kontejnerové typy – realizace

- nelze vyjádřit ve standardním Haskellu
- rozšíření `OverloadedLists` (od GHC 7.8), typová třída `IsList` (`GHC.Exts`)

```
class IsList l where
  type Item l -- type function

  fromList :: [Item l] -> l
  toList   :: l -> [Item l]
  fromListN :: Int -> [Item l] -> l
  fromListN _ = fromList
```

- typové funkce fungují díky rozšíření `TypeFamilies`

```
instance IsList [a] where
  type (Item [a]) = a
  fromList = id
  toList = id
```

## Kontejnerové typy – realizace

```
{-# LANGUAGE OverloadedLists #-}  
foo :: Map String Int  
foo = [("ahoj", 1), ("?", 42)]
```

- []

# Kontejnerové typy – realizace

```
{-# LANGUAGE OverloadedLists #-}  
foo :: Map String Int  
foo = [("ahoj", 1), ("?", 42)]
```

- [] bude typu `IsList 1 => 1`
- `[1, 2, 3]`



# Kontejnerové typy – realizace

```
{-# LANGUAGE OverloadedLists #-}  
foo :: Map String Int  
foo = [("ahoj", 1), ("?", 42)]
```

- [] bude typu `IsList l => l`
- [1, 2, 3] bude typu `(IsList lst, Num (Item lst)) => lst`
- [True, False]

## Kontejnerové typy – realizace

```
{-# LANGUAGE OverloadedLists #-}  
foo :: Map String Int  
foo = [("ahoj", 1), ("?", 42)]
```

- [] bude typu `IsList l => l`
- [1, 2, 3] bude typu `(IsList lst, Num (Item lst)) => lst`
- [True, False] bude typu `(IsList lst, Item lst ~ Bool) => lst`
  - ~ je podmínka na rovnost typů (další rozšíření)

## Kontejnerové typy – realizace

```
{-# LANGUAGE OverloadedLists #-}  
foo :: Map String Int  
foo = [("ahoj", 1), ("?", 42)]
```

- [] bude typu `IsList l => l`
- [1, 2, 3] bude typu `(IsList lst, Num (Item lst)) => lst`
- [True, False] bude typu `(IsList lst, Item lst ~ Bool) => lst`
  - ~ je podmínka na rovnost typů (další rozšíření)
- fungují standardní seznamové zkratky: [1..10]
- fungují vzory **pro fixní počet parametrů**

```
foo :: IsList lst => lst -> String  
foo [] = "empty"  
foo [x] = "single"  
foo _ = "other"
```

# Rozšíření Pattern guards

- umožňuje používat vzory ve strážích
- není třeba zapínat

```
lookupDef :: a -> b -> [(a, b)] -> b
lookupDef k d l
  | Just v <- lookup k l = v
  | otherwise            = d
```

# Record puns, record wildcards

rozšíření NamedFieldPuns, RecordWildCards

```
data C = C { a :: Int, b :: Int }
```

```
f (C { a = a }) = a
```

- NamedFieldPuns umožňuje zkrátit na `f (C { a }) = a`
  - lze kombinovat: `f (C { a, b = 4 }) = a`

# Record puns, record wildcards

rozšíření NamedFieldPuns, RecordWildCards

```
data C = C { a :: Int, b :: Int }
```

```
f (C { a = a }) = a
```

- NamedFieldPuns umožňuje zkrátit na  $f (C \{ a \}) = a$ 
  - lze kombinovat:  $f (C \{ a, b = 4 \}) = a$
- RecordWildcards dále umožňuje zkrátit binding všech položek v záznamu
  - $f (C \{ .. \}) = a + b$
  - nevztahuje se na již explicitně navázané:  
 $f (C \{ a = 1, .. \}) = b$
  - i při vytváření  $f \ a \ b = C \{ .. \}$
  - vztahuje se jen na položky, které jsou definované

# Rozšíření typového systému

- spousta zajímavých rozšíření
- vesměs silně nad rámec předmětu
- s některými nefunguje dobře odvozování typů (je třeba psát typové signatury)

# Rozšíření typového systému

- spousta zajímavých rozšíření
- vesměs silně nad rámec předmětu
- s některými nefunguje dobře odvozování typů (je třeba psát typové signatury)
- některé naleznete v IA014 Advanced Functional Programming
  - Multi-parameter type classes, functional dependencies (typové třídy s více parametry)
  - Generalized Algebraic Data Types (GADTs, umožňuje explicitně specifikovat typ datového konstrukturu)



spousta dalších

- explicitní forall:

```
look :: forall a b. Eq a => [(a, b)] -> a -> b
```

# Rozšíření typového systému

spousta dalších

- explicitní forall:

```
look :: forall a b. Eq a => [(a, b)] -> a -> b
```

- Arbitrary-rank polymorphism:

```
foo :: (forall a. a -> a) -> Int -> Int
```

# Rozšíření typového systému

spousta dalších

- explicitní forall:

```
look :: forall a b. Eq a => [(a, b)] -> a -> b
```

- Arbitrary-rank polymorphism:

```
foo :: (forall a. a -> a) -> Int -> Int
```

- Type Families – typové funkce

spousta dalších

- explicitní forall:

```
look :: forall a b. Eq a => [(a, b)] -> a -> b
```

- Arbitrary-rank polymorphism:

```
foo :: (forall a. a -> a) -> Int -> Int
```

- Type Families – typové funkce

- Scoped Type Variables

- catch act (\(x :: IOException) -> ...)

spousta dalších

- explicitní forall:

```
look :: forall a b. Eq a => [(a, b)] -> a -> b
```

- Arbitrary-rank polymorphism:

```
foo :: (forall a. a -> a) -> Int -> Int
```

- Type Families – typové funkce

- Scoped Type Variables

- catch act (\(x :: IOException) -> ...)

- PolyKinds, DataKinds – závislé typy (relativně omezené)

- Range 1 100, Vector 100

# Rozšíření typového systému

spousta dalších

- explicitní forall:

```
look :: forall a b. Eq a => [(a, b)] -> a -> b
```

- Arbitrary-rank polymorphism:

```
foo :: (forall a. a -> a) -> Int -> Int
```

- Type Families – typové funkce

- Scoped Type Variables

- catch act (\(x :: IOException) -> ...)

- PolyKinds, DataKinds – závislé typy (relativně omezené)

- Range 1 100, Vector 100

- Template Haskell – možnost generovat kód při kompilaci

- Test.QuickCheck.All

spousta dalších

- explicitní forall:

```
look :: forall a b. Eq a => [(a, b)] -> a -> b
```

- Arbitrary-rank polymorphism:

```
foo :: (forall a. a -> a) -> Int -> Int
```

- Type Families – typové funkce

- Scoped Type Variables

- catch act (\(x :: IOException) -> ...)

- PolyKinds, DataKinds – závislé typy (relativně omezené)

- Range 1 100, Vector 100

- Template Haskell – možnost generovat kód při kompilaci

- Test.QuickCheck.All

- Safe Haskell – bezpečná podmnožina jazyka