Zippers, reprezentace textu, další rozšíření

IB016 Seminář z funkcionálního programování

Mnoho autorů napříč věky

Fakulta informatiky, Masarykova univerzita

Jaro 2021

Motivace: pracovní plochy

Chceme reprezentovat pracovní plochy v OS pomocí seznamu.

```
data WorkSpace = WS String deriving (Show, Eq)
```

```
let myWS = [ WS "Default",  WS "Work", WS "School",
↪  WS "Games", WS "Other" ]
```

- Po spuštění systému se ocitneme na `WS "Default"`.

Motivace: pracovní plochy

Chceme reprezentovat pracovní plochy v OS pomocí seznamu.

```
data WorkSpace = WS String deriving (Show, Eq)

let myWS = [ WS "Default",  WS "Work", WS "School",
↪  WS "Games", WS "Other" ]
```

- Po spuštění systému se ocitneme na `WS "Default"`.
- Na které ploše se nacházíme?

Motivace: pracovní plochy

Chceme reprezentovat pracovní plochy v OS pomocí seznamu.

```
data WorkSpace = WS String deriving (Show, Eq)

let myWS = [ WS "Default",  WS "Work", WS "School",
↪  WS "Games", WS "Other" ]
```

- Po spuštění systému se ocitneme na `WS "Default"`.
- Na které ploše se nacházíme?
  - Dvojice – `myWS2 :: [(WorkSpace, Bool)]`?

Motivace: pracovní plochy

Chceme reprezentovat pracovní plochy v OS pomocí seznamu.

```
data WorkSpace = WS String deriving (Show, Eq)

let myWS = [ WS "Default",  WS "Work", WS "School",
↪  WS "Games", WS "Other" ]
```

- Po spuštění systému se ocitneme na `WS "Default"`.
- Na které ploše se nacházíme?
  - Dvojice – `myWS2 :: [(WorkSpace, Bool)]`?
  - Dvojice vol. 2 – `myWS3 :: ([WorkSpace], Int)`?

Motivace: pracovní plochy

Chceme reprezentovat pracovní plochy v OS pomocí seznamu.

```
data WorkSpace = WS String deriving (Show, Eq)

let myWS = [ WS "Default",  WS "Work", WS "School",
↪  WS "Games", WS "Other" ]
```

- Po spuštění systému se ocitneme na `WS "Default"`.
- Na které ploše se nacházíme?
  - Dvojice – `myWS2 :: [(WorkSpace, Bool)]`?
  - Dvojice vol. 2 – `myWS3 :: ([WorkSpace], Int)`?
- Jak se mezi plochami přesouvat?
  - Když si po škole budeme chtít něco zahrát, musím projít celý seznam?

Motivace: pracovní plochy

Chceme reprezentovat pracovní plochy v OS pomocí seznamu.

```
data WorkSpace = WS String deriving (Show, Eq)

let myWS = [ WS "Default",  WS "Work", WS "School",
↪  WS "Games", WS "Other" ]
```

- Po spuštění systému se ocitneme na `WS "Default"`.
- Na které ploše se nacházíme?
  - Dvojice – `myWS2 :: [(WorkSpace, Bool)]`?
  - Dvojice vol. 2 – `myWS3 :: ([WorkSpace], Int)`?
- Jak se mezi plochami přesouvat?
  - Když si po škole budeme chtít něco zahrát, musím projít celý seznam?
  - Co když budu chtít nějakou plochu smazat nebo novou přidat?

Seznamy

Musíme si pamatovat *okolí* naší aktuální plochy.

Seznamy

Musíme si pamatovat *okolí* naší aktuální plochy.

Zevšeobecníme kontextové procházení seznamů:

```
data LZipper a = LZip [a] [a]
goForward  :: LZipper a -> Maybe (LZipper a)
goBackward :: LZipper a -> Maybe (LZipper a)
modifyLZip :: (a -> a) -> LZipper a -> LZipper a
listToZip :: [a] -> LZipper a
zipToList :: LZipper a -> [a]
```

Binární stromy I.

Jak si pamatovat pozici v binárním stromu, abychom mohli efektivně zpracovávat okolí aktuálního uzlu?

```
data BinTree a = Node (BinTree a) (BinTree a)
               | Empty
               deriving (Show, Eq)
```

Binární stromy I.

Jak si pamatovat pozici v binárním stromu, abychom mohli efektivně zpracovávat okolí aktuálního uzlu?

```
data BinTree a = Node (BinTree a) (BinTree a)
               | Empty
               deriving (Show, Eq)
```

Můžeme si pamatovat trasu k upravovanému uzlu

```
data Direction = L | R
modify :: BinTree a -> [Direction] -> a -> BinTree a
```

Neefektivní při opakovaných úpravách, úpravách blízkých uzlů!

Binární stromy II.

Ve stromě se budeme pohybovat, ale zároveň si budeme pamatovat i trasu zpět pro rekonstrukci stromu.

```
data BinTree a = Node (BinTree a) a (BinTree a)
               | Empty
               deriving (Show, Eq)

data TreeDir a = TLeft a (BinTree a)
               | TRight (BinTree a) a
               deriving (Show, Eq)
```

Binární stromy II.

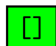
Ve stromě se budeme pohybovat, ale zároveň si budeme pamatovat i trasu zpět pro rekonstrukci stromu.

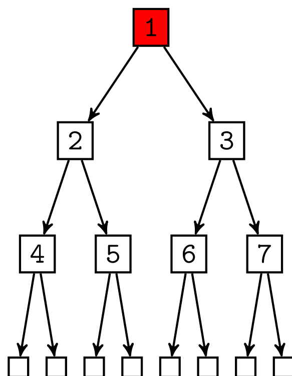
```
data BinTree a = Node (BinTree a) a (BinTree a)
               | Empty
               deriving (Show, Eq)

data TreeDir a = TLeft a (BinTree a)
               | TRight (BinTree a) a
               deriving (Show, Eq)

data TreeZipper a = TZip [TreeDir a] (BinTree a)
                  deriving (Show, Eq)
```

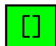
Binární stromy: příklad

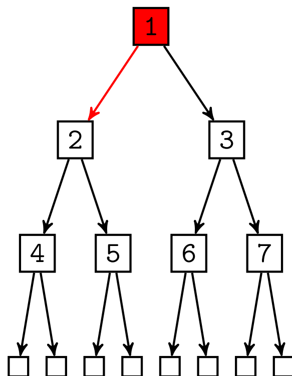





```
zipper = TZip [] (N (N (N E 4 E) 2 (N E 5 E) 1 (N (N
E 6 E) 3 (N E 7 E)))
```

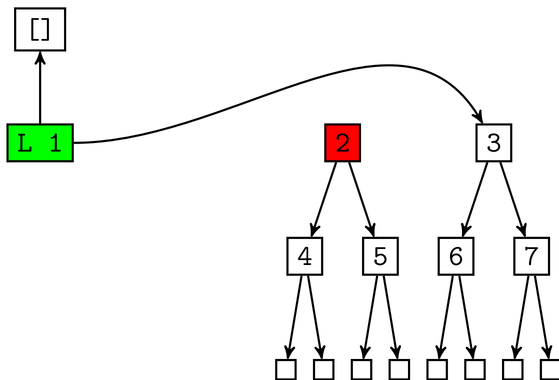
Binární stromy: příklad





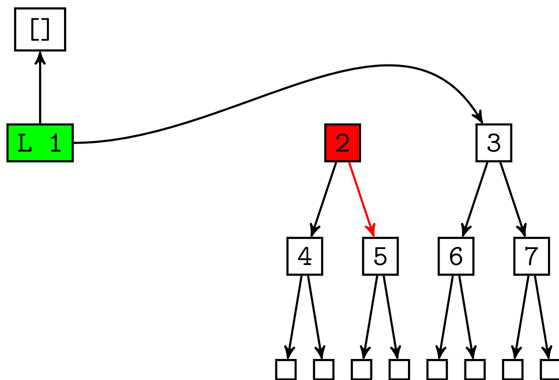

```
goLeft zipper
```

Binární stromy: příklad



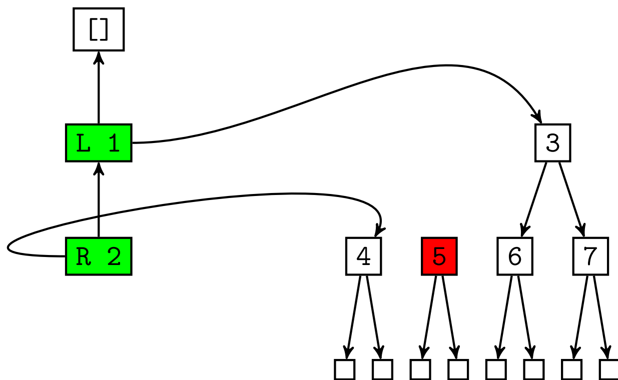

```
zipper TZip [L 1 (N (N E 6 E) 3 (N E 7 E))] (N (N E 4
 E) 2 (N E 5 E))
```

Binární stromy: příklad



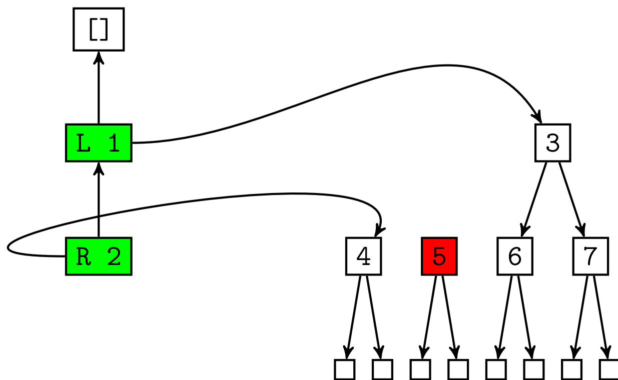

```
goRight zipper
```

Binární stromy: příklad



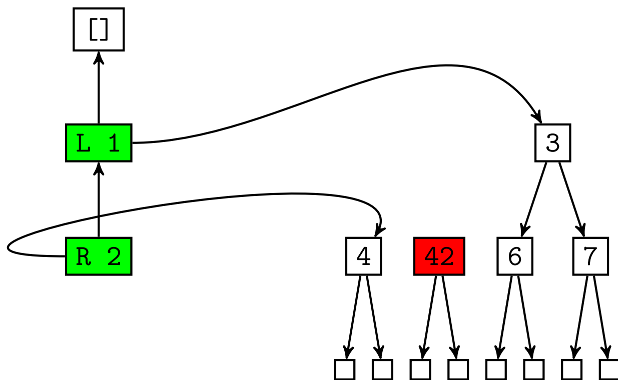

```
zipper TZip [R (N E 4 E) 2, L 1 (N (N E 6 E) 3 (N E 7
 E))] (N E 5 E)
```

Binární stromy: příklad



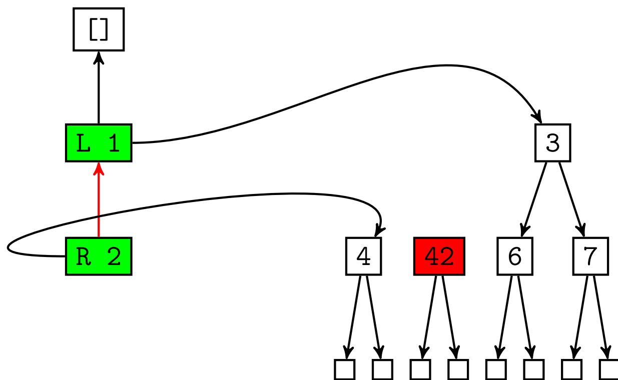

```
modify (+ 37) zipper
```

Binární stromy: příklad



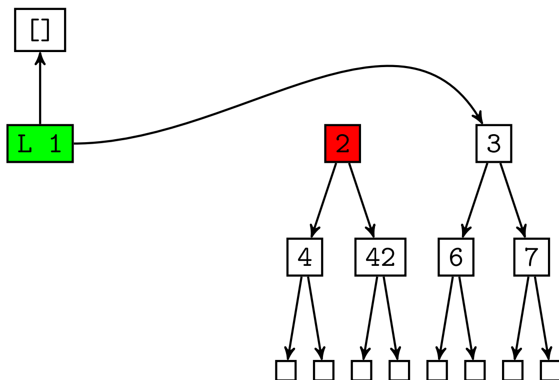

```
zipper = TZip [R (N E 4 E) 2, L 1 (N (N E 6 E) 3 (N
E 7 E))] (N E 42 E)
```

Binární stromy: příklad



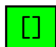

```
goUp zipper
```

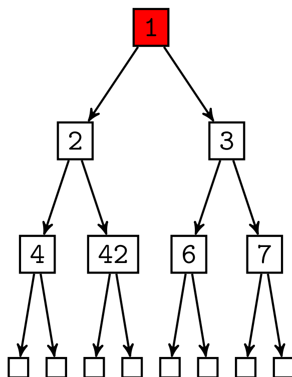
Binární stromy: příklad




```
zipper TZip [L 1 (N (N E 6 E) 3 (N E 7 E))] (N (N E 4
 E) 2 (N E 42 E))
```

Binární stromy: příklad






```
goUp zipper
```

Binární stromy: implementace

Manipulaci můžeme realizovat například pomocí:

```
goLeft  :: TZipper a -> TZipper a
goRight :: TZipper a -> TZipper a
goUp    :: TZipper a -> TZipper a
modify  :: (a -> a) -> TZipper a -> TZipper a
```

Zippers: Shrnutí

zipper pro danou datovou strukturu je datová struktura obohacená o „kontext“, který umožňuje efektivně manipulovat pozici ve struktuře

- **seznam**:
  zipper je dvojice seznamů: jeden obsahuje dosud neprojitý seznam, druhý prvky, které již byly projity v opačném pořadí
- **binární strom**:
  zipper je aktuální strom spolu se seznamem kroků zpět ke kořeni (vrchol + levý nebo pravý podstrom)
- obdobně pro složitější struktury
- Princip *zippers* můžeme zobecnit typovou (konstruktorovou) třídou. Zamyslete se, jak by deklarace/instance této třídy vypadaly a pak se podívejte na naši implementaci v ISu.

Další reprezentace textu

Typ *String*

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

- jaké jsou výhody?
- jaké jsou nevýhody?

Typ *String*

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

- jaké jsou výhody?
- jaké jsou nevýhody?

- Máme/známe lepší alternativy?

Řetězce v Haskellu

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

- výchozí, jednoduché, automaticky funguje se všemi seznamovými funkcemi
- pomalé a neefektivní: cca 4.4 GB na načtení 100MB souboru do paměti
- vhodné pro líné zpracování, vhodné pro krátké řetězce

Řetězce v Haskellu

`type String = [Char]`

- výchozí, jednoduché, automaticky funguje se všemi seznamovými funkcemi
- pomalé a neefektivní: cca 4.4 GB na načtení 100MB souboru do paměti
- vhodné pro líné zpracování, vhodné pro krátké řetězce

`Text` (`Data.Text`, balík *text*)

- pro unicode text, interně používá UTF-16 (2 B / znak)
- dostupné ve striktní i líné variantě
- cca 5.6 $\times$ více paměti pro načtení souvislého bloku

Řetězce v Haskellu

`type String = [Char]`

- výchozí, jednoduché, automaticky funguje se všemi seznamovými funkcemi
- pomalé a neefektivní: cca 4.4 GB na načtení 100MB souboru do paměti
- vhodné pro líné zpracování, vhodné pro krátké řetězce

`Text` (`Data.Text`, balík *text*)

- pro unicode text, interně používá UTF-16 (2 B / znak)
- dostupné ve striktní i líné variantě
- cca 5.6 $\times$ více paměti pro načtení souvislého bloku

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

- především pro binární data; síťová komunikace, volání do C
- striktní a líná varianta
- cca 2 $\times$ více paměti pro načtení souvislého bloku

Text

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

- definuje mnoho funkcí stejného jména jako `Prelude`
- líná varianta v `Data.Text.Lazy`
- mnoho operací je optimalizováno tak, že nevyžadují tvorbu mezilehlých hodnot (*fusion*)
- užitečné funkce
  - `pack :: String -> Text`
  - `unpack :: Text -> String`
  - `append :: Text -> Text -> Text`
  - `cons`, `snoc`, ...

IO v Text

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

- závislé na nastavení locale (v systému, nebo v GHC)
- `readFile :: FilePath -> IO Text`
- `appendFile`, `writeFile`, `getLine`, `putStr`...
- funguje i práce s `Handle`
- existuje i líná varianta `Data.Text.Lazy.IO`

Práce s Unicode v `Text`

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

Pokročilá práce s Unicode (balík *text-icu*)

- rozdělování na znaky, slova i věty
- porovnávání řetězcu s ohledem na místní zvyklosti (čeština!)
- konverze kódování
- normalizace
- umí i regulární výrazy

```
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 (resp. 8bitových znaků)
- striktní verze (v podstatě C-like $char*$ + hlavička)
- líná verze (seznam striktních bloků)

ByteString

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

- kompaktní reprezentace binárních dat (resp. 8bitových znaků)
- striktní verze (v podstatě C-like *char*∗ + hlavička)
- líná verze (seznam striktních bloků)

- binární `Data.ByteString` nebo
  znakový `Data.ByteString.Char8` pohled na data
  - `pack :: [Word8] -> ByteString`
  - `pack :: String -> ByteString`
  - stejný typ, jiná metoda přístupu
  - **znaková verze bere jen spodních 8 bitů z `Char`**

ByteString

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

- kompaktní reprezentace binárních dat (resp. 8bitových znaků)
- striktní verze (v podstatě C-like *char*∗ + hlavička)
- líná verze (seznam striktních bloků)

- binární `Data.ByteString` nebo
  znakový `Data.ByteString.Char8` pohled na data
  - `pack :: [Word8] -> ByteString`
  - `pack :: String -> ByteString`
  - stejný typ, jiná metoda přístupu
  - **znaková verze bere jen spodních 8 bitů z** `Char`
- opět (re)definuje vlastní `IO` funkce, podporuje *fusion*

Konverze mezi `Text` a `ByteString`

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

Vstup a výstup

- standardní IO (`System.IO`) má mnoho problémů
  - někdy neintuitivní líné vyhodnocování
  - závislost na nastaveném locale při práci se soubory
  - vyhazuje výjimky při nevalidním kódování (např. neplatný UTF-8 znak)
- typický bývá nejvhodnější použít striktní `ByteString` a dekódování do `Text`
- více třeba na www.snoyman.com/blog/2016/12/beware-of-readfile

Reprezentace řetězcových literálů

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
- typová 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

- → 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`

Problém s typem prvků

- ne všechny kontejnerové typy mají právě jeden typový parametr popisující typ obsahu
  - `Map k v` – inicializováno `[(k, v)]`
  - `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]` $\rightsquigarrow$

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]` $\rightsquigarrow$ `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]` $\leadsto$ `Integer`
  - `Item (Set Int)` $\leadsto$ `Int`
  - `Item (Map String Int)` $\leadsto$ `(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 lst] -> lst`
- `toList :: lst -> [Item lst]`

Kontejnerové typy – realizace

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

```
class IsList lst where
    type Item lst -- type function
    fromList :: [Item lst] -> lst
    toList   :: lst -> [Item lst]
    fromListN :: Int -> [Item lst] -> lst
    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 lst where
      type Item lst -- type function
      fromList :: [Item lst] -> lst
      toList   :: lst -> [Item lst]
      fromListN :: Int -> [Item lst] -> lst
      fromListN _ = fromList
  ```

- typové funkce fungují díky rozšíření **TypeFamilies**

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

Kontejnerové typy – realizace – pokračování

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

- `[]`

Kontejnerové typy – realizace – pokračování

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

- `[]` bude typu `IsList lst => lst`

Kontejnerové typy – realizace – pokračování

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

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

Kontejnerové typy – realizace – pokračování

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

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

Kontejnerové typy – realizace – pokračování

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

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

Kontejnerové typy – realizace – pokračování

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

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

Kontejnerové typy – realizace – pokračování

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

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

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

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 mají dostupnou definici

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 konstruktoru)

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 -
  ``action `catch` (\(x :: IOException) -> ...)``
- *PolyKinds*, *DataKinds* – závislé typy (relativně omezené)
  `Range 1 100, Vector 100`

Další rozšíření

- GeneralisedNewtypeDeriving – umožňuje derivovat pro `newtype` libovolnou instanci která je k dispozici pro zabalený typ
- Template Haskell – možnost generovat kód při kompilaci
- Safe Haskell – bezpečná podmnožina jazyka