

Úvod do síťové komunikace, pole v Haskellu

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Jaro 2015

Existuje mnoho různých balíčků, které se liší

- podporovanými schopnostmi (autentizace, šifrování, komprese, ...)
- závislostmi (třeba *zlib* je často vyžadováno kvůli kompresi)
- úrovní dokumentace

Asi nejjednodušší je balík HTTP

- základní HTTP funkcionalita obsažena v modulu `Network.HTTP`
- umožňuje provádět HTTP dotazy a dostávat odpovědi
- umí spravovat proxy, autentizaci připojení, cookies, ...
- neumí šifrování, kompresi dat, ...
- pracuje v monádě IO (jednoduché dotazy), nebo `BrowserAction` (více dotazů v rámci jednoho sezení)

Network.HTTP – základní funkce

```
simpleHTTP :: HStream ty =>  
           Request ty -> IO (Result (Response ty))
```

- otevře přímé jednorázové spojení na zadaný server
- zašle normalizovaný dotaz a vrátí odpověď

```
getRequest :: String -> Request_String
```

- ze zadaného URL vytvoří GET dotaz

```
getResponseBody :: Result (Response ty) -> IO ty  
getResponseCode ::  
    Result (Response ty) -> IO ResponseCode
```

- vrátí tělo/kód z výsledku HTTP dotazu

```
> simpleHTTP (getRequest "http://www.haskell.org/")  
>>= getResponseBody
```

```
"<html>\n<head><title>302 Found</title></head>\n  <body bgcolor=\"white\">\n    <center><h1>302 Found</h1></center>\n    <hr>\n    <center>nginx/1.6.2</center>\n  </body>\n</html>\n"
```

Motivace

- někdy chceme balíky z Hackage jen vyzkoušet
- odinstalace je obtížná
- můžeme mít různé verze závislostí v různých projektech
- to vše způsobuje v klasické instalaci v cabalu problémy

Motivace

- někdy chceme balíky z Hackage jen vyzkoušet
- odinstalace je obtížná
- můžeme mít různé verze závislostí v různých projektech
- to vše způsobuje v klasické instalaci v cabalu problémy

Sandboxy (cabal \geq 1.18)

- umožňuje vytvořit lokální instalaci balíčků ve složce
- `cabal sandbox init` – inicializuje sandbox v aktuální složce
- následné `cabal install BALÍK` probíhá do sandboxu
- binárky v `./cabal-sandbox/bin`
- ghc standardně nedokáže detekovat sandbox samo
→ `cabal repl` spustí ghci v sandboxu (cabal \geq 1.19)
- `cabal sandbox delete`

Proč zavádět datový typ pole? Jaké má vlastnosti?

- struktura mapující indexy na hodnoty
- struktura s konstantním náhodným přístupem
- časté náhodné čtení je efektivní
- čistá datová struktura (*pure*), takže při změně prvku musíme celé pole vybudovat znovu
- balík `array`, modul `Data.Array`

Typová třída Ix

```
class (Ord a) => Ix a where
  range      :: (a,a) -> [a]
  index      :: (a,a) -> a -> Int
  inRange    :: (a,a) -> a -> Bool
```

- typy, které mohou sloužit jako index
- knihovní instance pro Int, Integer, Char, Bool, uspořádané n-tice
- instance pro výčtové typy možno automaticky odvodit

Konstrukce pole I.

`array :: (Ix a) => (a,a) -> [(a,b)] -> Array a b`

- zkonstruuje pole z asociativního seznamu a hranic indexů
- asociativní pole musí obsahovat každý index nejvýše jednou
- index mimo mezí způsobí vytvoření nedefinovaného pole

Konstrukce pole I.

```
array :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
```

- zkonstruuje pole z asociativního seznamu a hranic indexů
- asociativní pole musí obsahovat každý index nejvýše jednou
- index mimo mezí způsobí vytvoření nedefinovaného pole

```
squares = array (1,100) [(i, i*i) | i <- [1..100]]
```

`(!) :: Ix i => Array i e -> i -> e`

`bounds :: Ix i => Array i e -> (i, i)`

`elems :: Ix i => Array i e -> [e]`

`range :: Ix a => (a, a) -> [a]`

`assocs :: Ix i => Array i e -> [(i, e)]`

Konstrukce pole II.

```
mkArray :: (Ix a) => (a -> b) -> (a,a) -> Array a b  
mkArray f bs = array bs [(i, f i) | i <- range bs]
```

- konstrukce pole zadáním funkce na indexech
- funkce může používat předešlé hodnoty pole

Konstrukce pole II.

```
mkArray :: (Ix a) => (a -> b) -> (a,a) -> Array a b
mkArray f bs = array bs [(i, f i) | i <- range bs]
```

- konstrukce pole zadáním funkce na indexech
- funkce může používat předešlé hodnoty pole

```
mkArray (\i -> i * i) (1,100)
```

```
fibs :: Int -> Array Int Int
```

```
fibs n = a
```

```
  where a = array (0,n) ([ (0, 1), (1, 1) ] ++
                        [(i, a!(i-2) + a!(i-1)) | i <- [2..n]])
```

Úpravy pole

```
(//) :: (Ix a) => Array a b -> [(a,b)] -> Array a b
```

```
accum :: Ix i =>
```

```
(e -> a -> e) -> Array i e -> [(i, a)] -> Array i e
```

- Pozor, konstruujeme nové pole!

Úpravy pole

```
(//) :: (Ix a) => Array a b -> [(a,b)] -> Array a b
```

```
accum :: Ix i =>
```

```
(e -> a -> e) -> Array i e -> [(i, a)] -> Array i e
```

- Pozor, konstruujeme nové pole!

```
m // [((i,i), 0) | i <- [1..n]]
```

```
swapRows :: (Ix a, Ix b, Enum b) =>
```

```
a -> a -> Array (a,b) c -> Array (a,b) c
```

```
swapRows i i' ar = ar //
```

```
(((i ,j), ar!(i',j)) | j <- [jLo..jHi]) ++
```

```
(((i',j), ar!(i ,j)) | j <- [jLo..jHi])
```

```
where ((iLo,jLo),(iHi,jHi)) = bounds ar
```


Jiné typy polí v Haskellu

	Immutable	IO monad	ST monad
Standard	Array DiffArray	IOArray	STArray
Unboxed	UArray DiffUArray	IOUArray, StorableArray	STUArray

- *Array* (klasické neměnitelné pole)
- *IOArray* (měnitelné pole v monádě IO)
- *STArray* (měnitelné pole v monádě ST)
- *DiffArray* (hybridní model: měnitelné pole s rozhraním jako neměnitelné pole)

Je to celé ještě složitější, více na wiki.haskell.org/Arrays

Samostatný úkol

Naprogramujte sadu základních funkcí pro práci s maticemi uložených v datové struktuře (neměnitelného) pole. Pro inspiraci můžete použít typové signatury níže.

```
type Matrix = Array (Int,Int) Int

identity :: Int -> Matrix
square  :: Matrix -> Bool
identical :: Matrix -> Matrix -> Bool
add     :: Matrix -> Matrix -> Matrix
trace  :: Matrix -> [Int]
transpose :: Matrix -> Matrix
conformable :: Matrix -> Matrix -> Bool
multiply :: Matrix -> Matrix -> Matrix
inversePair :: Matrix -> Matrix -> Bool
```