

Úvod do síťové komunikace, pole v Haskellu, měnitelnost

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

jaro 2017

Existuje mnoho různých balíčků, které se liší

- podporovanými schopnostmi (autentizace, šifrování, komprese, . . .)
- závislostmi (třeba *zlib* je často vyžadováno kvůli kompresi)
 - tj. závislost na knihovně v C, která musí být nainstalovaná v systému
- úrovní dokumentace

Asi nejjednodušší je balík HTTP

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

```
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://example.com/")
  >>= getResponseBody >>= putStr
<!doctype html>
<html>
<head>
  <title>Example Domain</title>

  <meta charset="utf-8" />
  <meta http-equiv="Content-type" content="text/html;
    charset=utf-8" />
  <meta name="viewport"
    content="width=device-width, initial-scale=1" />
...

```

balík `network-uri` pro práci s URL

- modul `Network.URI`
- `parseURI :: String -> Maybe URI`
- `URI` je záznam umožňující přístup k jednotlivým částem
- převod zpět na `String` pomocí `show`

interakce s `Network.HTTP`:

- v modulu `Network.HTTP.Base` (exportováno i z `Network.HTTP`)
- `defaultGETRequest :: URI -> Request_String`
- `defaultGETRequest_ ::
 BufferType a => URI -> Request a`
- typová třída `BufferType` je nadtřídou `HStream`

Pole vs. seznamy

- seznam velmi dobře funguje jako neměnitelná (immutable) datová struktura ve funkcionálních programech

- seznam velmi dobře funguje jako neměnitelná (immutable) datová struktura ve funkcionálních programech
 - při modifikaci lze nezměněné konce seznamů znovu využívat
 - dobře se kombinuje s líným vyhodnocováním

Pole vs. seznamy

- seznam velmi dobře funguje jako neměnitelná (immutable) datová struktura ve funkcionálních programech
 - při modifikaci lze nezměněné konce seznamů znovu využívat
 - dobře se kombinuje s líným vyhodnocováním
- ale má i nevýhody

- seznam velmi dobře funguje jako neměnitelná (immutable) datová struktura ve funkcionálních programech
 - při modifikaci lze nezměněné konce seznamů znovu využívat
 - dobře se kombinuje s líným vyhodnocováním
- ale má i nevýhody
 - pomalá indexace
 - vysoký paměťový overhead
 - zřetězený seznam → mnoho ukazatelů
 - nelze použít při interakci s programy v jiném jazyce

- seznam velmi dobře funguje jako neměnitelná (immutable) datová struktura ve funkcionálních programech
 - při modifikaci lze nezměněné konce seznamů znovu využívat
 - dobře se kombinuje s líným vyhodnocováním
- ale má i nevýhody
 - pomalá indexace
 - vysoký paměťový overhead
 - zřetězený seznam → mnoho ukazatelů
 - nelze použít při interakci s programy v jiném jazyce
- pole nemá (nemusí mít) tyto nevýhody, ale má jiné

- seznam velmi dobře funguje jako neměnitelná (immutable) datová struktura ve funkcionálních programech
 - při modifikaci lze nezměněné konce seznamů znovu využívat
 - dobře se kombinuje s líným vyhodnocováním
- ale má i nevýhody
 - pomalá indexace
 - vysoký paměťový overhead
 - zřetězený seznam → mnoho ukazatelů
 - nelze použít při interakci s programy v jiném jazyce
- pole nemá (nemusí mít) tyto nevýhody, ale má jiné
 - pro zachování neměnitelnosti je třeba při modifikaci pole zkopírovat
 - měnitelná pole se nemohou vyskytovat v čistém kódu

Pole v Haskellu

máme několik balíčků pro reprezentaci polí v Haskellu

- `array`, hlavní modul `Data.Array`
 - odpovídá Haskell 2010 Language Report
 - indexace hodnotami libovolného typu v typové třídě `Ix`
 - rozsah indexů specifikován při vytváření pole (nemusíme indexovat od 0)
 - snadná práce s vícerozměrnými poly – indexace ntící

máme několik balíčků pro reprezentaci polí v Haskellu

- `array`, hlavní modul `Data.Array`
 - odpovídá Haskell 2010 Language Report
 - indexace hodnotami libovolného typu v typové třídě `Ix`
 - rozsah indexů specifikován při vytváření pole (nemusíme indexovat od 0)
 - snadná práce s vícerozměrnými poly – indexace ntací
- `vector`, hlavní modul `Data.Vector`
 - indexace nezápornými `Int` hodnotami
 - slicing polí
 - mnoho pomocných funkcí pro plnění polí, operace nad poli

máme několik balíčků pro reprezentaci polí v Haskellu

- `array`, hlavní modul `Data.Array`
 - odpovídá Haskell 2010 Language Report
 - indexace hodnotami libovolného typu v typové třídě `Ix`
 - rozsah indexů specifikován při vytváření pole (nemusíme indexovat od 0)
 - snadná práce s vícerozměrnými poli – indexace ntící
- `vector`, hlavní modul `Data.Vector`
 - indexace nezápornými `Int` hodnotami
 - slicing polí
 - mnoho pomocných funkcí pro plnění polí, operace nad poli
- obě verze mají příbuzné moduly pro práci s měnitelnými poli, poli která lze předávat do jiných programovacích jazyků...

Vector – základní použití

```
import Data.Vector ( Vector, (!), (!?) )
import qualified Data.Vector as V
```

- definuje mnoho funkcí stejného jména jako Prelude
 - vhodné importovat explicitně nekonfliktní funkce, nebo kvalifikovaně

Vector – základní použití

```
import Data.Vector ( Vector, (!), (!?) )
import qualified Data.Vector as V
```

- definuje mnoho funkcí stejného jména jako Prelude
 - vhodné importovat explicitně nekonfliktní funkce, nebo kvalifikovaně
- vytváření
 - `empty`, `singleton`, `replicate` – základní vytváření
 - `generate :: Int -> (Int -> a) -> Vector a` a vytváření pomocí funkce z indexu na hodnotu
 - `iterateN :: Int -> (a -> a) -> a -> Vector a` a obdobně jako seznamové `iterate`, ale dané délky

Vector – základní použití

```
import Data.Vector ( Vector, (!), (!?) )
import qualified Data.Vector as V
```

- definuje mnoho funkcí stejného jména jako Prelude
 - vhodné importovat explicitně nekonfliktní funkce, nebo kvalifikovaně
- vytváření
 - `empty`, `singleton`, `replicate` – základní vytváření
 - `generate :: Int -> (Int -> a) -> Vector a` a vytváření pomocí funkce z indexu na hodnotu
 - `iterateN :: Int -> (a -> a) -> a -> Vector a` a obdobně jako seznamové `iterate`, ale dané délky
- `length`, `null`, `head`, `last` – jako u seznamu, ale vše v $\mathcal{O}(1)$

Vector – základní použití

```
import Data.Vector ( Vector, (!), (!?) )
import qualified Data.Vector as V
```

- definuje mnoho funkcí stejného jména jako Prelude
 - vhodné importovat explicitně nekonfliktní funkce, nebo kvalifikovaně
- vytváření
 - `empty`, `singleton`, `replicate` – základní vytváření
 - `generate :: Int -> (Int -> a) -> Vector a` a vytváření pomocí funkce z indexu na hodnotu
 - `iterateN :: Int -> (a -> a) -> a -> Vector a` obdobně jako seznamové `iterate`, ale dané délky
- `length`, `null`, `head`, `last` – jako u seznamu, ale vše v $\mathcal{O}(1)$
- `indexace: (!), (!?)` (v `Maybe`, vrací `Nothing` mimo rozsah), $\mathcal{O}(1)$

Vector – slicing

slicing: získání podrozsahu vektoru, bez kopírování (vektor je neměnný), $O(1)$

- `slice :: Int -> Int -> Vector a -> Vector a`
 - `slice i l v` vytvoří slice vektoru `v` začínající na indexu `i` délky `l`
- `tail :: Vector a -> Vector a`
- `init :: Vector a -> Vector a`
- `drop :: Int -> Vector a -> Vector a`
- `splitAt :: Int -> Vector a -> (Vector a, Vector a)`
- v případě přístupu mimo rozsah vyhazují výjimky, existují `unsafe` varianty

Vector – slicing

slicing: získání podrozsahu vektoru, bez kopírování (vektor je neměnný), $\mathcal{O}(1)$

- `slice :: Int -> Int -> Vector a -> Vector a`
 - `slice i l v` vytvoří slice vektoru `v` začínající na indexu `i` délky `l`
- `tail :: Vector a -> Vector a`
- `init :: Vector a -> Vector a`
- `drop :: Int -> Vector a -> Vector a`
- `splitAt :: Int -> Vector a -> (Vector a, Vector a)`
- v případě přístupu mimo rozsah vyhazují výjimky, existují `unsafe` varianty

pozor, `slice` není kopie, takže drží celou původní paměť, i kdyby ta už nikdy nebyla použita

- `force :: Vector a -> Vector a` – v případě potřeby zkopíruje obsah do vektoru nezbytné velikosti, $\mathcal{O}(n)$

- `(//)` `:: Vector a -> [(Int, a)] -> Vector a`
zapiše na dané indexy dané hodnoty, $\mathcal{O}(n + m)$
- `update` `:: Vector a -> Vector (Int, a) -> Vector a`
obdobné, jen změny jsou kódované jako vektor
- `reverse`, `map`, `filter`, `dropWhile`, `foldy`...
- `imap` `:: (Int -> a -> b) -> Vector a -> Vector b`
- `zip`, `zipWith` – pro 2 až 6 vektorů
- + monadické varianty

Vector – další operace

- `toList`, `fromList`, `fromListN`
- spojování: `(++)`, nebo pomocí monoidového `(<>)`
- instance `Functor`, `Foldable`, `Traversable`, `Monoid`, `Show`, `Eq`, `Ord` ...
- také `Applicative`, `Monad` – obdobné jako pro seznamy

Vector – další operace

- `toList`, `fromList`, `fromListN`
- spojování: `(++)`, nebo pomocí monoidového `(<>)`
- instance `Functor`, `Foldable`, `Traversable`, `Monoid`, `Show`, `Eq`, `Ord` ...
- také `Applicative`, `Monad` – obdobné jako pro seznamy
- operace využívající měnitelných (mutable) vektorů
 - vektor je možné dočasně převést na měnitelný, modifikovat a následně převést zpět

Teoretická vsuvka – měnitelný stav výpočtu

někdy je vhodné moci do funkcionálního výpočtu vložit výpočet, který může mít stav (alespoň dočasně)

Teoretická vsuvka – měnitelný stav výpočtu

někdy je vhodné moci do funkcionálního výpočtu vložit výpočet, který může mít stav (alespoň dočasně)

- myšlenka: použijeme vhodnou monádu pro reprezentaci výpočtu se stavem

Teoretická vsuvka – měnitelný stav výpočtu

někdy je vhodné moci do funkcionálního výpočtu vložit výpočet, který může mít stav (alespoň dočasně)

- myšlenka: použijeme vhodnou monádu pro reprezentaci výpočtu se stavem
- **IO** – umí stav, ale nelze z něj utéct způsobem, který by nepodřýval typový systém a optimalizace

Teoretická vsuvka – měnitelný stav výpočtu

někdy je vhodné moci do funkcionálního výpočtu vložit výpočet, který může mít stav (alespoň dočasně)

- myšlenka: použijeme vhodnou monádu pro reprezentaci výpočtu se stavem
- **IO** – umí stav, ale nelze z něj utéct způsobem, který by nepodřýval typový systém a optimalizace
 - protože může interagovat s vnějším světem, to ale není vždy potřeba

Teoretická vsuvka – měnitelný stav výpočtu

někdy je vhodné moci do funkcionálního výpočtu vložit výpočet, který může mít stav (alespoň dočasně)

- myšlenka: použijeme vhodnou monádu pro reprezentaci výpočtu se stavem
- **IO** – umí stav, ale nelze z něj utéct způsobem, který by nepodřýval typový systém a optimalizace
 - protože může interagovat s vnějším světem, to ale není vždy potřeba
- **ST** – umí stav, ale ne interakci s vnějším světem

State Transformers (ST)

State Threads/State Transformers: `Control.Monad.ST`

- `ST :: * -> * -> *`, tedy má dva argumenty (`ST s a`)
 - `s` – typ stavu, volná typová proměnná
 - `a` – typ výsledku akce
- stav je mapování referencí na hodnoty
- `ST` akce lze spustit a získat čistou hodnotu

Spouštění ST

- spuštění musí začít ve stavu v němž nejsou alokovány žádné proměnné
- nesmí být možné použít proměnnou z jiné sekvence akcí

Spouštění ST

- spuštění musí začít ve stavu v němž nejsou alokovány žádné proměnné
- nesmí být možné použít proměnnou z jiné sekvence akcí
- myšlenka: ST je indexován typem stavu, bezpečně spustit lze, pokud může být stav libovolný
 - kompilátor si při spuštění vnitřně vytvoří nový čistý stav

Spouštění ST

- spuštění musí začít ve stavu v němž nejsou alokovány žádné proměnné
- nesmí být možné použít proměnnou z jiné sekvence akcí
- myšlenka: ST je indexován typem stavu, bezpečně spustit lze, pokud může být stav libovolný
 - kompilátor si při spuštění vnitřně vytvoří nový čistý stav

```
runST :: (forall s. ST s a) -> a
```

Spouštění ST

- spuštění musí začít ve stavu v němž nejsou alokovány žádné proměnné
- nesmí být možné použít proměnnou z jiné sekvence akcí
- myšlenka: ST je indexován typem stavu, bezpečně spustit lze, pokud může být stav libovolný
 - kompilátor si při spuštění vnitřně vytvoří nový čistý stav

```
runST :: (forall s. ST s a) -> a
```

- forall uvnitř závorky zajišťuje, že s je volná typová proměnná
- parametr runST je „něco, co dokáže být typu ST s a pro libovolné s“
- výsledek nemůže být závislý na s, protože s je vázané v typu prvního argumentu

Vztah IO a ST

- speciální typ stavu `RealWorld`
- `IO` je v podstatě `newtype` nad `ST RealWorld`
 - `ST RealWorld` nelze spustit pomocí `runST`, protože typ stavu není volná proměnná
- `stToIO :: ST RealWorld a -> IO a`

Vztah IO a ST

- speciální typ stavu `RealWorld`
- `IO` je v podstatě `newtype` nad `ST RealWorld`
 - `ST RealWorld` nelze spustit pomocí `runST`, protože typ stavu není volná proměnná
- `stToIO :: ST RealWorld a -> IO a`

`Control.Monad.Primitive` definuje typovou třídu `PrimMonad` stavových monád do které patří `IO` i `ST`

- definuje typovou funkci `PrimState`, která pro stavovou monádu vrací typ stavu
 - `PrimState (ST s) = s`
 - `PrimState IO = RealWorld`
- umožňuje uniformně pracovat se stavovými monádami

Měnitelné vektory

```
import Data.Vector (Vector, freeze, thaw, unsafeFreeze, (!))
import Data.Vector.Mutable ( MVector, STVector, IOVector
                             , new, write, read, modify )
import qualified Data.Vector.Mutable as MV
```

- `MVector s` a – měnitelný vektor hodnot typu `a` nad stavovou monádou se stavem typu `s`
 - `type IOVector = MVector RealWorld`
 - `type STVector s = MVector s`

Měnitelné vektory

```
import Data.Vector (Vector, freeze, thaw, unsafeFreeze, (!))
import Data.Vector.Mutable ( MVector, STVector, IOVector
                             , new, write, read, modify )
import qualified Data.Vector.Mutable as MV
```

- `MVector s a` – měnitelný vektor hodnot typu `a` nad stavovou monádou se stavem typu `s`
 - `type IOVector = MVector RealWorld`
 - `type STVector s = MVector s`
- `new :: PrimMonad m`
`=> Int -> m (MVector (PrimState m) a)`
 - vytvoří nový vektor dané délky v požadované monádě (neinicializovaný)
- `replicate :: PrimMonad m`
`=> Int -> a -> m (MVector (PrimState m) a)`

Měnitelné vektory – příklad

```
> runST $ MV.replicate 16 0 >>= \vec ->
  write vec 0 1 >> write vec 8 2 >>
  unsafeFreeze vec
[1,0,0,0,0,0,0,0,2,0,0,0,0,0,0,0]
```

výsledek je typu `Num a => Vector a`

- `freeze :: PrimMonad m => MVector (PrimState m) a -> m (Vector a)`
 - převod vektoru na neměnitelný, $\mathcal{O}(n)$
 - `unsafeFreeze` má stejný typ, ale neprovádí kopii, tedy je bezpečný jen pokud už následně není měnitelný vektor použitý, $\mathcal{O}(1)$
- `thaw :: PrimMonad m => Vector a -> m (MVector (PrimState m) a)`
 - konverze na měnitelný vektor, $\mathcal{O}(n)$

Modifikace neměnného vektoru

akci nad `STVector` lze převést na akci nad `Vector`:

```
modify :: (forall s. MVector s a -> ST s ())  
       -> Vector a -> Vector a
```


Modifikace neměnného vektoru

akci nad `STVector` lze převést na akci nad `Vector`:

```
modify :: (forall s. MVector s a -> ST s ())  
       -> Vector a -> Vector a
```

příklad:

```
> let v = V.replicate 8 1
```

```
> V.modify (\mv -> MV.write mv 4 16) v
```

```
[1,1,1,1,16,1,1,1]
```

```
> V.modify (\mv -> forM_ [0,2 .. MV.length mv - 1] $  
              \i -> MV.modify mv (*2) i) v
```

```
[2,1,2,1,2,1,2,1]
```

- storable vektory – pro komunikaci s jinými programovacími jazyky (primárně C)
 - vektor hodnot, které patří do typové třídy `Storable` – mají definovanou serializaci do paměti čitelnou z C
 - pointer na paměť lze převést na storable vektor (a naopak)¹
- unboxed vektory
 - běžné hodnoty jsou v Haskellu uloženy za pointrem (boxed), což usnadňuje implementaci lenosti, garbage collection. . .
 - unboxed vektory umožňují ukládat *některé datové typy* přímo

¹Ano, v Haskellu lze pracovat s pointery

Samostatný úkol I

Naprogramujte sadu základních funkcí pro práci s maticemi uloženými ve vektoru. Pro inspiraci můžete použít datový typ a typové signatury uvedené níže. Matice lze reprezentovat například vektorem, který obsahuje postupně řádky matice a rozměrem matice. Jako bonus můžete využít `modify/thaw/freeze` v **ST** k zefektivnění implementace.

```
data Matrix a = Matrix
  { rows :: Int -- ^ number of rows
  , cols :: Int -- ^ number of columns
  , vector :: Vector a -- ^ storage of size rows * cols
  } deriving ( Show, Eq, Ord )
```

- `fromList :: [[a]] -> Matrix a`
`fromList lst` zkonstruuje matici ze seznamu řádků (všechny řádky musí mít stejnou délku).

Samostatný úkol II

- `identity :: Num a => Int -> Matrix a`
Konstruuje jednotkovou matici o daném rozměru.
- `square :: Matrix a -> Bool`
Zjistí, je-li matice čtvercová.
- `identical :: Eq a => Matrix a -> Matrix a -> Bool`
Zjistí, jsou-li matice stejné.
- `add :: Num a => Matrix a -> Matrix a -> Matrix a`
Provede standardní součet matic.
- `trace :: Num a => Matrix a -> a`
Vrátí stopu matice (součet prvků na hlavní diagonále).
- `transpose :: Matrix a -> Matrix a`
Transponuje matici.
- `multipliable :: Num a => Matrix a -> Matrix a -> Bool`
Zjistí, jestli je možné dané matice násobit.

Samostatný úkol III

- `multiply :: Num a => Matrix a -> Matrix a -> Matrix a`
Provede standardní násobení matic.

- `inversePair :: (Eq a, Num a) => Matrix a -> Matrix a -> I`
Zjistí, jsou-li zadané matice k sobě inverzní.