Lenost a striktnost

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Jaro 2015

Líné vyhodnocování: příklad

`fst (1, undefined)` $\rightsquigarrow^*$ `1`

- druhá složka dvojice se vůbec nevyhodnotí

Líné vyhodnocování: příklad

`fst (1, undefined)` $\rightsquigarrow^*$ `1`

- druhá složka dvojice se vůbec nevyhodnotí

```
let fact n = product [1..n]
in map fact [1000000,999999..0] !! 999995
```
$\rightsquigarrow^*$ `120`

- jednotlivé prvky jsou vypočítány jen když jsou vyžádány

Líné vyhodnocování: příklad

```
fst (1, undefined) ⇝* 1
```

- druhá složka dvojice se vůbec nevyhodnotí

```
let fact n = product [1..n]
in map fact [1000000,999999..0] !! 999995 ⇝* 120
```

- jednotlivé prvky jsou vypočítány jen když jsou vyžádány

```
let fibs = 0:1:zipWith (+) fibs (tail fibs)
in fibs !! 100 ⇝* 354224848179261915075
```

- seznam je konstruován líně, tedy jen prvky, které jsou potřeba

Líné vyhodnocování

- každá hodnota v Haskellu může být
  a) nevyhodnocená – funkce počítající hodnotu
  b) vyhodnocená
- pokud se snažíme nevypočítanou hodnotu přečíst, vypočítá se
- při příštím čtení je již vypočítaná
- vyhodnocuje se vždy jen to co je nezbytně nutné

Líné vyhodnocování

```
fact n = product [1..n]
facts = map fact [0..]
[]     !! _ = error "(!!): index too large"
(x:_)  !! 0 = x
(_:xs) !! n = xs !! (n - 1)
```

```
facts !! 2 ⇝ (map fact [0..]) !! 2
     ⇝ (map fact (0:[1..])) !! 2
     ⇝ (fact 0 : map fact [1..]) !! 2
     ⇝ (map fact [1..]) !! (2 - 1)
     ⇝ (map fact (1:[2..])) !! (2 - 1)
     ⇝ (fact 1 : map fact [2..]) !! (2 - 1)
     ⇝ (fact 1 : map fact [2..]) !! 1
     ⇝ (map fact [2..]) !! (1 - 1)
    ⇝* (fact 2 : map [3..]) !! 0
     ⇝ fact 2 ⇝ product [1..2] ⇝* 2
```

Striktnost a lenost

```
selectsort :: Ord a => [a] -> [a]
selectsort [] = []
selectsort xs = min : selectsort (withoutMin xs)
  where
    min = minimum xs
    withoutMin (y:ys)
        | y == min  = ys
        | otherwise = y : withoutMin ys
```

- striktní vzhledem ke struktuře seznamu (vyžaduje vyhodnotit seznam až po `[]`)
- vyhodnotí všechny prvky až do té míry jak vyžaduje `(<)`
- ale produkuje výstup líně!

Striktnost a lenost

```
selectsort :: Ord a => [a] -> [a]
selectsort [] = []
selectsort xs = min : selectsort (withoutMin xs)
  where
    min = minimum xs
    withoutMin (y:ys)
        | y == min  = ys
        | otherwise = y : withoutMin ys
```

- striktní vzhledem ke struktuře seznamu (vyžaduje vyhodnotit seznam až po `[]`)
- vyhodnotí všechny prvky až do té míry jak vyžaduje `(<)`
- ale produkuje výstup líně!
  - jaká je složitost `head (selectsort [10000,9999..0])`?

Striktnost a lenost

```
selectsort :: Ord a => [a] -> [a]
selectsort [] = []
selectsort xs = min : selectsort (withoutMin xs)
  where
    min = minimum xs
    withoutMin (y:ys)
        | y == min  = ys
        | otherwise = y : withoutMin ys
```

- striktní vzhledem ke struktuře seznamu (vyžaduje vyhodnotit seznam až po `[]`)
- vyhodnotí všechny prvky až do té míry jak vyžaduje `(<)`
- ale produkuje výstup líně!
  - jaká je složitost `head (selectsort [10000,9999..0])`?
  - $\mathcal{O}(n)$

Co vynucuje vyhodnocení?

Vzory

```
f1, f2, f3 :: [a] -> ()
f1 _ = ()
f2 (_:_) = ()
f3 (_:_:[]) = ()

f1 undefined ⇝* ()
f2 undefined ⇝* *** Exception: Prelude.undefined
f1 [] ⇝* ()
f2 [] ⇝* *** Exception: ...
f2 [undefined,undefined] ⇝* ()
f3 [undefined,undefined] ⇝* ()
f2 (undefined:undefined) ⇝* ()
f3 (undefined:undefined) ⇝* *** Exception: ...
```

- jen co je nezbytně nutné k rozhodnutí, zda hodnota odpovídá vzoru

Co vynucuje vyhodnocení?

Funkce `seq :: a -> b -> b`

- `a `seq` b` zaručuje, že `a` bude vyhodnoceno (= jeden výpočetní krok), vrátí `b`
- k vyhodnocení dojde až když něco vynutí výpočet výrazu `a `seq` b`
- nevyhodnocuje `a` plně:
  `(undefined, undefined) `seq` 1` $\rightsquigarrow^*$ `1`
  `undefined `seq` 1` $\rightsquigarrow^*$ `*** Exception: ...`

Co vynucuje vyhodnocení?

Funkce `seq :: a -> b -> b`

- `a `seq` b` zaručuje, že `a` bude vyhodnoceno (= jeden výpočetní krok), vrátí `b`
- k vyhodnocení dojde až když něco vynutí výpočet výrazu `a `seq` b`
- nevyhodnocuje `a` plně:
  `(undefined, undefined) `seq` 1` $\rightsquigarrow^*$ `1`
  `undefined `seq` 1` $\rightsquigarrow^*$ `*** Exception: ...`

Striktní aplikace `($!) :: (a -> b) -> a -> b`

- obdoba `($)`, ale vynutí vyhodnocení argumentu před aplikací
- `f1 $! undefined` $\rightsquigarrow^*$ `*** Exception: ...`

Proč striktnost?

- striktní výpočet může být rychlejší, paměťově efektivnější

```
>λ= :set +s
(0.18 secs, 74075592 bytes)
>λ= const () $! product [1..100000]
()
(18.32 secs, 9052106864 bytes)
>λ= :m + Data.List
>λ= const () $! foldl' (*) 1 [1..100000]
()
(2.73 secs, 9045968328 bytes)
```

- `foldl'` je striktní verze `foldl`

Proč striktnost?

```
readFiles :: [FilePath] -> IO [String]
readFiles paths = mapM readFile paths
```

- `readFile` vrací obsah souboru líně
- soubor je zavřen až když je dočten na konec!
- lehce můžeme překročit limit na množství otevřených souborů
- `seq` nepomůže

Control.DeepSeq

Úplné/hluboké vyhodnocení.

```
import Control.DeepSeq

readFiles :: [FilePath] -> IO String
readFiles paths = forM paths $ \p ->
    withFile p ReadMode $ \h -> do
        c <- hGetContents h
        return $!! c
```

- `($!!) :: NFData a => (a -> b) -> a -> b`
  plně vyhodnotí argument, pak aplikuje funkci
- `deepseq :: NFData a => a -> b -> b`
- ```
  class NFData a where
      rnf :: a -> ()
  ```
  plně vyhodnotí argument
  ```

Control.DeepSeq

```
import Control.DeepSeq

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

instance NFData a => NFData (BinTree a) where
    rnf Empty = ()
    rnf (Node v t1 t2) = rnf v `seq` rnf t1 `seq`
                         rnf t2
```

Rozšíření GHC

- GHC má různá rozšíření Haskellu
- rozšíření zapínáme specielním pragma komentářem na prvním řádku souboru
  {-# LANGUAGE <rozsireni> #-}
  nebo na příkazové řádce pomocí `-X<rozšíření>`, z ghci `:set -X<rozšíření>`
- různá rozšíření syntaxe, návrhy budoucích standardů, apod.
- `https://downloads.haskell.org/~ghc/7.8.3/docs/html/users_guide/ghc-language-features.html`

BangPatterns

```
{-# LANGUAGE BangPatterns #-}
```

Umožňuje snadno vynutit striktnost u vzoru, `let` výrazu, hodnoty v typu.

```
data Tuple a b = Tuple !a !b

lazy, strict :: a -> ()
lazy _ = ()
strict !_ = ()
```

- taková hodnota je pak **na nejvyšší úrovni** vyhodnocena před voláním funkce/konstruktoru
- v `let` způsobí vyhodnocení příslušného výrazu před přejitím do `in` podvýrazu (striktní `let`)

```
($!) :: (a -> b) -> a -> b
f $! x = let !ex = x in f ex
```

Samostatná práce

Implementujte datový typ `SortedSet a`, který bude reprezentovat množiny (pomocí binárního vyhledávacího stromu, *bonus*: vyvážený strom).

- tato datová struktura bude striktní a to jak strukturou stromu, tak i hodnotami v něm obsaženými
- bude dokonce hluboce striktní – každá hodnota bude před vložením vyhodnocena (pomocí `Control.DeepSeq`)
- napište rovněž instanci `NFData` pro `SortedSet`

Implementujte následující funkce:

- `emptySet :: SortedSet a`
- `insertSet :: (NFData a, Ord a) => a -> SortedSet a -> SortedSet a`
  plně vyhodnotí a vloží hodnotu do stromu
- `setElem :: Ord a => a -> SortedSet a -> Bool`
  zjistí, zda je prvek obsažen v množině, pokud je množina prázdná, nevyhodnotí prvek
- `getMin :: Ord a => SortedSet a -> Maybe a`
- `splitMin :: Ord a => SortedSet a -> Maybe (a, SortedSet a)`
  extrahuje minimum z množiny (výsledek vrací striktně)