Vyhodnocovací strategie (lenost vs. striktnost)

IB016 Seminář z funkcionálního programování

Mnoho autorů napříč věky

Fakulta informatiky, Masarykova univerzita

Jaro 2021

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 (počítá se jen faktoriál pětky)

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

- seznam je konstruován líně, tedy jen prvky, které jsou potřeba
- už jednou vypočítané prvky seznamu se nepočítají znovu

Líné vyhodnocování

- každá hodnota v Haskellu může být
  a) vyhodnocená
  b) nevyhodnocená – tzv. *thunk* – výraz, který lze vyhodnotit
- pokud se snažíme nevypočítanou hodnotu přečíst, vypočítá se
- při příštím čtení je již vypočítaná
- vyhodnocuje se vždy jen to, co je nezbytně nutné
  - jen ty části výrazů, jejichž výsledek je potřeba
  - jen do té hloubky, dokud je to nutné
    (vzpomeňte na `take` na nekonečných seznamech)

Líné vyhodnocování

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

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

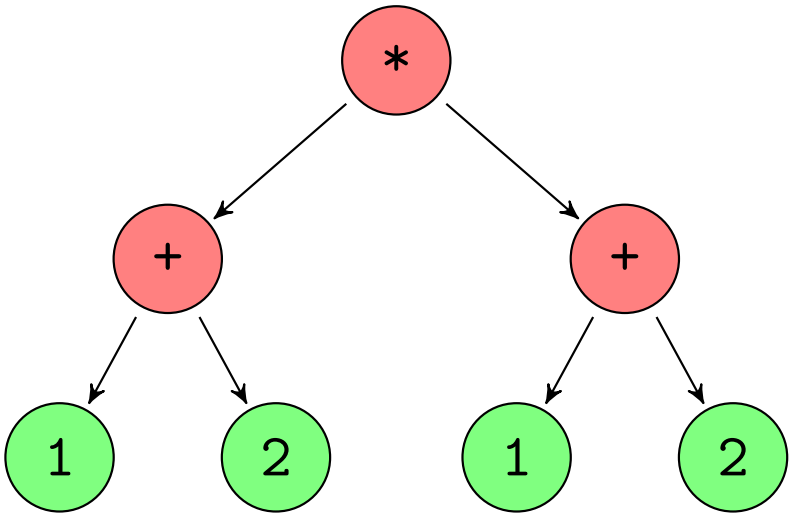
Striktnost a lenost

```haskell
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
```

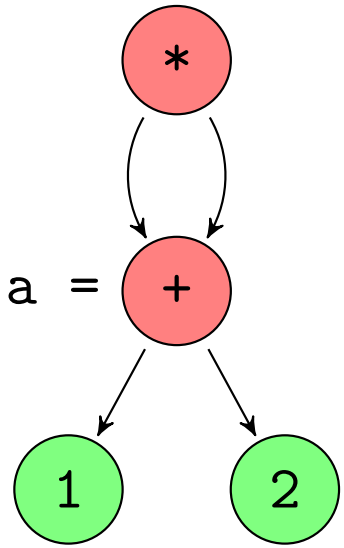
- striktní vzhledem ke struktuře seznamu (vyžaduje vyhodnotit seznam až po `[]`)
- vyhodnotí všechny prvky až do té míry jak vyžaduje `(<)`
- ale produkuje výstup líně!
  - jaká je složitost `head (selectsort [10000,9999..0])`?
  - $\mathcal{O}(n)$

Sdílení podvýrazů

```
(1 + 2) * (1 + 2)
 ⇝  3 * (1 + 2)
 ⇝  3 * 3   ⇝  9
```




```
square a = a * a
square (1 + 2)
 ⇝  (1 + 2) * (1 + 2)
 ⇝  3 * 3   ⇝  9
```



= grafová redukce (graph reduction)

Sdílení v GHCi

- standardně GHCi vypisuje hodnoty pomocí `print`, což obvykle způsobuje plné vyhodnocení
- pomocí `:sprint` lze vypsat bez vynucení vyhodnocení
- pozor, polymorfní výrazy se vyhodnocují vždy znovu

```
> f a = [a, 1, 2] ++ [3, a, 4]
> xs = f (5 + 2 :: Int)
> :sprint xs
xs = _                -- xs je thunk (nevyhodnocený výraz)
> head xs
7
> :sprint xs
xs = 7 : _            -- ocásek xs se zatím nevyhodnotil
> length xs
6
> :sprint xs
xs = [7, _, _, _, 7, _]
```

Teoretická vsuvka: vyhodnocovací strategie

Striktní vyhodnocovací strategie

- výrazy vyhodnoceny při přiřazení do proměnných
- argumenty vyhodnoceny před voláním
- pořadí vyhodnocování obvykle do značné míry jasné ze zdrojového kódu
- typické pro imperativní jazyky, ale i mnohé funkcionální (*ML)

```
head (take (2 + 2) [1..10])
    ⇝* head (take 4 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
    ⇝* head [1, 2, 3, 4]
    ⇝* 1
```

Teoretická vsuvka: vyhodnocovací strategie

Normální/nestriktní vyhodnocovací strategie

- výrazy se vyhodnocují, až když jsou potřeba
- umožňuje existenci potenciálně nekonečných datových struktur
- pořadí vyhodnocení méně viditelné
- může mít větší spotřebu paměti (nevyhodnocené výrazy)

```
head (take (2 + 2) [1..10])
    ⇝* head (take 4 (1 : [2..10]))
    ⇝* head (1 : take 3 [2..10])
    ⇝* 1
```

Líná vyhodnocovací strategie

- normální + sdílení podvýrazů

Teoretická vsuvka: vyhodnocovací strategie

Churchova-Rosserova věta
Výsledná hodnota ukončeného výpočtu výrazu nezáleží na redukční strategii: pokud výpočet skončí, je jeho výsledek vždy stejný.

Věta o normalizace
Jestliže pro nějaký výraz M existuje redukční strategie, s jejímž použitím se úprava výrazu M nezacyklí, pak se tento výpočet nezacyklí ani s použitím normální (či líné) redukční strategie.

Teoretická vsuvka: produktivní algoritmy

- u algoritmů, které vracejí konečné datové struktury nás zajímá, zda skonční
- pokud ale produkují nekonečné datové struktury, tak jistě neskončí
- neskončení ale může být různé:
  - `repeat x = x : repeat x`
  - `inf x = inf x`
- výzraz, je produktivní pokud
  - ho lze plně vyhodnotit v konečném čase
  - je možné výsledek získaný po libovolném konečném počtu kroků rozebrat pomocí vzorů (provést pattern-mathing) a takto získané podvýrazy jsou opět produktivní
- funkce je produktivní, jestliže každý výraz, který vznikne její aplikací na produktivní výrazy je produktivní
  - `inf 42` není produktivní
  - `repeat 42` je produktivní
  - `[1..10] ++ inf 42` není produktivní

Vzory, ovlivňování lenosti

Vzory vynucují vyhodnocení

```
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: ...
```

- Argument se vyhodnocuje jen do hloubky nezbytně nutné k rozhodnutí, zda hodnota odpovídá vzoru.

Líné vzory

- Vlnovkou před vzorem lze odložit pattern matching až do doby, kdy se nějaký argument navázaný vzorem použije.
- Takový vzor nevynucuje vyhodnocení a vždy uspěje.
- Srovnejte:

```
strict, lazy :: Bool -> Maybe Int -> Int
strict b (Just x) = if b then x else 0
lazy b ~(Just x) = if b then x else 0
```

```
strict False Nothing ⇝* ???
lazy False Nothing ⇝* ???
lazy True Nothing ⇝* ???
```

Líné vzory

- `f ~(x, y) = g x y` se přeloží na (něco jako)
  `f p = g (fst p) (snd p)`

- v `do`-notaci potlačuje navíc požadavek na `MonadFail`:
  `~(x:xs) <- listOf1 arbitrary`
  (fragment generátoru pro QuickCheck, `listOf1` generuje neprázdné seznamy)

seq vynucuje vyhodnocení

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

- `a `seq` b` zaručuje, že `a` bude vyhodnoceno, a vrátí `b`
- k vyhodnocení dojde až když něco vynutí výpočet výrazu `a `seq` b`
- nevyhodnocuje `a` plně, nýbrž jen po nejvyšší konstruktor:
  `(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: ...`

★ Weak head normal form

`seq` vynucuje vyhodnocení pouze do tzv. *weak head normal form*:

- po nejvyšší (i část. aplik.) hodnotový konstruktor, nebo
- dokud výraz není λ-abstrakce, nebo
- dokud výraz není nedostatečně aplikovanou vestavěnou funkcí.

Příklady výrazů a jejich WHNF:

- `replicate 3 'A'` $\rightsquigarrow$ `(:) 'A' (replicate (3-1) 'A')`
- `replicate 3` $\rightsquigarrow$ `\x -> x : replicate (3-1) x`
- `head` $\rightsquigarrow$ `\(x:xs) -> x`
- `(\x -> if x then Right else Left) (4 < 3)` $\rightsquigarrow$ `Left`
- `Just (replicate 3 'A')` už je ve WHNF
- `(+) (4 * 5)` už je ve WHNF (uvažujeme-li vestavěné plus)

Proč striktnost?

- striktní výpočet může být rychlejší, paměťově efektivnější

```
> :set +s
(0.18 secs, 74075592 bytes)
> const () $! product [1..100000]
()
(18.32 secs, 9052106864 bytes)
> :m + Data.List
> 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

```haskell
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 BangPatterns

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

Umožňuje snadno vynutit striktnost u vzoru, výrazu `let`, argumentu hodnotového konstruktoru.

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

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

- taková hodnota je pak před aplikací funkce/konstruktoru vyhodnocena do WHNF (po první konstruktor)
- v `let` způsobí vyhodnocení příslušného výrazu před započetím vyhodnocování části `in` (striktní `let`)

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

Rozšíření StrictData a Strict

`{-# LANGUAGE StrictData #-}`

- jako by všechna pole v datových definicích *v daném modulu* byla definována pomocí `!` (z rozšíření BangPatterns)
- lenost může být vynucena vlnovkou (`data T = D ~Int`)

`{-# LANGUAGE Strict #-}`

- obsahuje StrictData
- argumenty funkcí, definice `where`/`let`, vazby v `case`/`do` jsou také striktní
- vnořené vzory a top-level definice nadále striktní nejsou

https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#extension-StrictData

Striktnost v knihovně

- `Data.List`: `foldl'`, `foldl1'` – akumulátorem je jediná, vždy vyhodnocená hodnota namísto potenciálně obrovského thunku (nevyhodnoceného výrazu).
- `Data.Map.Lazy` – *klíče* se vyhodnocují striktně (ukládají se už vyhodnocené pro rychlejší vyhledávání).
- `Data.Map.Strict` – *klíče i hodnoty* se ukládají vyhodnocené.
- ★ Pokročilé věci související s transformátory monád a `ST`.

Lenost v jiných jazycích

- zkrácené vyhonocování logických výrazů ve většině jazyků (včetně vedlejších efektů!)
- iterátory se mohou chovat líně (map/filter/... v Pythonu, C++ ranges, C# LINQ, ...)
- future/promise – často mohou běžet paralelně, či podle potřeby (líně)

Procvičení

Naprogramujte variantu ohodnocených binárních stromů se striktními hodnotami (`StrictValTree`), ale línou reprezentací stromové struktury, a variantu, která je plně striktní jak ve struktuře tak v hodnotách (`StrictTree`).

- rozmyslete si, pro jaké použití by se mohly jednotlivé varianty hodit a kdy byste naopak použili línou variantu
- rozmyslete si, jaké prostředky z tohoto cvičení chcete použít pro vynucení striktnosti v jednotlivých případech a proč
- pro obě verze stromů implementujte instanci `Functor`