Deklaratívne programovanie webových aplikácií

Poznámky k prednáškam

Ján Kľuka

Katedra aplikovanej informatiky
FMFI UK Bratislava

Zimný semester 2008/09

Obsah

O predmete

- Úvod do jazyka Haskell
- Opakovanie techník deklaratívneho programovania v Haskelli (rekurzia, zoznamy, stromové štruktúry)
- XML ako stromová štruktúra
- Zobrazovanie a spracovanie XML
- SVG
- Vstup/výstup v Haskelli
- Interaktívne webové aplikácie
  - formulárové
  - s trochou AJAXu

Organizácia a pravidlá

- Učíme prvý rok
- Voľnejšia forma
- Prvých 8 týždňov: domáce úlohy po 5 bodov
- Zvyšok semestra a skúškové: projekt, priebežná kontrola (60 bodov)

Haskell

- Funkcionálny programovací jazyk
- Čisto deklaratívny
- Typovaný
- Prakticky orientovaný
- Imperatívne črty (polia, vstup/výstup) bez narušenia deklaratívnosti (vďaka funkcionálom)
- Viacero implementácií, interpretery aj kompilátory
- Veľká knižnica modulov
- http://haskell.org
- Začneme interpreterom Hugs, neskôr prejdeme na kompilátor GHC
- http://haskell.org/hugs

Jednoduché výrazy

Výrazy možno vyhodnocovať priamo v príkazovom riadku intepretera Hugs:

- Konštanty: celé čísla (`123`, `-5`), znaky (`'c'`), reťazce (`"abcd"`)
- Infixové aritmetické operátory so zvyčajnou prioritou:
  `2+(9-4)^2*21/7`
- Porovnania sú boolovské funkcie (vrátia True alebo False), nízka priorita:
  `11+2 < 55`
- Mená funkcií (a premenných):
  prvý znak: `a-z` alebo `_`, ďalšie znaky: `a-z`, `A-Z`, `0-9`, `_`
- Aplikácia funkcie: `fun arg`; vyššia priorita ako infixové operátory
  `sqrt 2+2 = (sqrt 2)+2`,
  zátvorkuje sa doľava
  `sqrt sqrt 2 = (sqrt sqrt) 2`,
  preto treba písať
  `sqrt (sqrt 2)`

Definície funkcií

Definície funkcií musia byť v súbore.

```
Hugs> :edit lec1.hs
```

Funkcie sa definujú rovnicami:

$$five = 5$$

$$sq\ x = x * x$$

$$isZero\ n = n == 0$$

```
Hugs> :load lec1.hs.
```

Definované funkcie potom môžeme používať vo výrazoch:
five, *sq five*, *isZero* (*sq* 20)

Komentáre:

-- jednoriadkové (do konca riadka)

{- viacriadkové -}

Zložené výrazy

if-then-else

$$sgn\ x = \textbf{if}\ x > 0\ \textbf{then}\ 1\ \textbf{else}\ 0$$

case-of

$$sgn_1\ x = \textbf{case}\ x\ \textbf{of}\ \{\ 0 \to 0;\quad \_ \to 1\ \}$$

„_“ znamená „čokoľvek“

Prehľadnejší zápis s odsadením:

$$sgn_2\ x = \textbf{case}\ x\ \textbf{of}$$
$$0 \to 0$$
$$\_ \to 1$$

Podobne ako v CL môžeme dosadiť možnosti za premennú x:

$$sgn_3\ 0 = 0$$
$$sgn_3\ \_ = 1$$

Výlučnosť

Možnosti v **case** nemusia byť výlučné (aj 0 je „čokoľvek“).
Použije sa prvá vyhovujúca:

$$sgn_4\ 0 = 0$$
$$sgn_4\ \_ = 1$$
$$sgn_4\ 9 = 999$$

```
Main> sgn4 9
1
```

Rekurzia

Nič prekvapujúce:

$$\begin{aligned} &\mathit{factorial}\ n = \textbf{case}\ n\ \textbf{of} \\ &\qquad\qquad 0 \to 1 \\ &\qquad\qquad k+1 \to n * \mathit{factorial}\ k \end{aligned}$$

alebo dvomi rovnicami

$$\begin{aligned} &\mathit{sumFirst}\ 0 = 0 \\ &\mathit{sumFirst}\ (n+1) = (n+1) + \mathit{sumFirst}\ n \end{aligned}$$

Notoricky známa funkcia počítajúca množenie králikov

$$\begin{aligned} &\mathit{fib}\ 0 = 1 \\ &\mathit{fib}\ 1 = 1 \\ &\mathit{fib}\ (x+2) = \mathit{fib}\ (x+1) + \mathit{fib}\ x \end{aligned}$$

Viac argumentov, pomocné premenné

Ďalší argument sa pripája medzerou

$$max\ x\ y = \textbf{if}\ x >= y\ \textbf{then}\ x\ \textbf{else}\ y$$

Rekurzia s akumulátorom

$$facta\ 0\ a = a$$
$$facta\ (n+1)\ a = facta\ n\ ((n+1)*a)$$

$$factorial_1\ n = facta\ n\ 0$$

Pomocné premenné – zložený výraz **let-in**

$$facta_1\ 0\ a = a$$
$$facta_1\ (n+1)\ a = \textbf{let}\ b = (n+1)*a\ \textbf{in}$$
$$facta_1\ n\ b$$

Viac a podrobnejšie o Haskelli

Hal Daume III et al.: *Yet Another Haskell Tutorial*

http://www.haskell.org/haskellwiki/Tutorials

2. prednáška

Dátové štruktúry

CL:

- Všetky dátové štruktúry sa tvoria (konštruujú) párovaním.
- Každá dátová štruktúra je číslo.
- Každé číslo je zoznam.

Haskell:

- Viacero rôznych konštruktorov (osobitné pre *n*-tice, zoznamy, stromy atď.).
- Čísla a dátové štruktúry sú rozličné objekty.

n-tice

Konštruktory *n*-tíc: (x, y) (x, y, z) (w, x, y, z) atď.

n-tice pre rôzne *n* sú samostatné typy, nie sú navzájom porovnateľné.

Špeciálne, $(x, (y, z)) \neq (x, y, z) \neq ((x, y), z)$.

Funkcia nad dvojicami (napr. fst $(x, y) = x$) nefunguje na trojiciach.

Pattern matching na *n*-ticiach:

case p **of**

$(x, y) \rightarrow \cdots$

let $(x, y) = p$ **in** $\cdots$

n-tice — príklad

Úloha 7 z prvého cvičenia: Funkcia $evenodd_{10}$, ktorá pre číslo x vráti *dvojicu* čísel y a z:

- y obsahuje všetky párne číslice desiatkového zápisu x v rovnakom poradí ako v x;
- z obsahuje všetky nepárne číslice desiatkového zápisu x v rovnakom poradí.

Riešenie bez pomocnej funkcie:

$$\begin{aligned}
&\text{evenodd}_{10} :: \text{Int} \to (\text{Int}, \text{Int}) \\
&\text{evenodd}_{10}\ 0 = (0,0) \\
&\text{evenodd}_{10}\ x = \textbf{let}\ (y,z) = \text{evenodd}_{10}\ (x\ \text{`div`}\ 10) \\
&\qquad\qquad\qquad\qquad n = x\ \text{`mod`}\ 10\ \textbf{in} \\
&\qquad\qquad \textbf{case}\ n\ \text{`mod`}\ 2\ \textbf{of} \\
&\qquad\qquad\quad 0 \to (y*10+n, z) \\
&\qquad\qquad\quad 1 \to (y, z*10+n)
\end{aligned}$$

Zoznamy

Konštruktory zoznamov: $[]$ (CL: 0) $x : xs$ (CL: x, xs)

$x : y : xs = x : (y : xs)$

Skratka pre zoznamy dĺžky n: $[x_1, x_2, \ldots, x_n] = x_1 : x_2 : \cdots : x_n : []$

Pattern matching na zoznamoch: **case** *xs* **of**

$[] \longrightarrow \cdots$

$y : ys \longrightarrow \cdots$

Konvencia: premenné označujúce zoznamy majú príponu *-s*

conc *xs ys* = **case** *xs* **of**

$[] \longrightarrow ys$

$z : zs \longrightarrow z :$ conc *zs ys*

conc $[]$ *ys* = *ys*

conc $(x : xs)$ *ys* = $x :$ conc *xs ys*

Pattern matching na zoznamoch

Konštruktory [] a : sa dajú kombinovať so skratkami a vnárať:

$$
\begin{aligned}
&\text{maxl } xs = \textbf{case } xs \textbf{ of} \\
&\qquad\qquad [] \to 0 \\
&\qquad\qquad [y] \to y \\
&\qquad\qquad y : ys \to \textbf{let } m = \text{maxl } ys \textbf{ in} \\
&\qquad\qquad\qquad\qquad \textbf{if } m < y \textbf{ then } y \textbf{ else } m \\
&\text{maxl}_1\ [] = 0 \\
&\text{maxl}_1\ [x] = x \\
&\text{maxl}_1\ (x : y : xs) = \textbf{if } x < y \textbf{ then } \text{maxl}_1\ (y : xs) \textbf{ else } \text{maxl}_1\ (x : xs)
\end{aligned}
$$

Napriek tomuto príkladu,

jednoprvkové zoznamy $[x]$ sú zriedka špeciálnym prípadom!

2. prednáška

Základné typy Haskellu

Mená typov začínajú veľkým písmenom

Typy čísel:

Int, Float, Double, Integer, ...

Znaky: Char

Pravdivostné hodnoty: Bool

Zistenie typu výrazu v Hugs:

```
Hugs> :t 'a'
'a' :: Char
```

Zložené typy

Typy n-tíc sa zapisujú rovnako ako ich konštruktory:

(Char, Int) (Float, Int, Char)

Typy zoznamov sa zapisujú do []:

[Int] [(Int, Char)]

Typy funkcií:

$T_1 \to T_2 \to \cdots \to T_n \to T$

$T_1, \ldots, T_n$ – typy argumentov T – typ výsledku

$\max_3$ :: Int → Int → Int → Int

Skratky zložených typov

type String = [Char] (zabudovaná)

type StringPairs = [(String,String)]

Polymorfizmus 1

Funkcia „prvá zložka dvojice“

```
fst (x,y) = x
```

funguje na dvojiciach *bez ohľadu na to, akého typu sú zložky*:

```
fst (1, 'x') = 1        fst ('x', "yz") = 'x'        ···
```

Aký je typ fst?

```
Int → Char → Int?        Char → String → Char?        ···
```

fst je **polymorfná** funkcia, má (zdanlivo) nekonečne veľa typov.

Haskell dovoľuje typy s **typovými premennými**.
fst má potom jediný typ

```
fst :: (a, b) → a
```

a a *b* sú typové premenné (začínajú malým písmenom)

Polymorfizmus 2

Zreťazenie

$$\text{conc}\ []\ ys = ys$$
$$\text{conc}\ (x : xs)\ ys = x : \text{conc}\ xs\ ys$$

funguje na zoznamoch *bez ohľadu typ prvkov*:

$$\text{conc}\ [1,2,3]\ [4,5] = [1,2,3,4,5] \qquad \text{conc "foo" "bar"} = \text{"foobar"}$$

no typy prvkov prvého a druhého zoznamu musia byť rovnaké:

$$\text{conc "foo"}\ [3,5] \Longrightarrow \text{ERROR}$$

Typ zreťazenia:

$$\text{conc} :: [a] \to [a] \to [a]$$

Typové chyby

```
Hugs> let b = "a" == 'a' in b
ERROR - Type error in application
*** Expression     : "a" == 'a'
*** Term           : "a"
*** Type           : String
*** Does not match : Char

Hugs> case [1,2,3] of 'a' -> True
ERROR - Type error in case pattern
*** Term           : 'a'
*** Type           : Char
*** Does not match : [a]
```

Prečo typovať funkcie ručne

Haskell odvodzuje typy funkcií sám.
Niekedy sa aj funkcia s preklepom

$$\text{bconc}\ [\,]\ ys = ys$$
$$\text{bconc}\ (x : xs)\ ys = \boxed{xs} : \text{bconc}\ xs\ ys$$

dá dobre otypovať:

```
Hugs> :load lec02.hs
Lec02>
```

Nie je to však zamýšľaný typ:

```
Lec02> :t bconc
bconc :: [a] -> [[a]] -> [[a]]
```

Lepšie je explicitne uviesť v zdrojovom kóde zamýšľaný typ funkcie.

Prečo typovať funkcie ručne

Lepšie je explicitne uviesť v zdrojovom kóde zamýšľaný typ funkcie:

$$\text{bconc} :: [a] \to [a] \to [a]$$

$$\text{bconc}\ []\ ys = ys$$

$$\text{bconc}\ (x : xs)\ ys = \boxed{xs} : \text{bconc}\ xs\ ys$$

```
Lec02> :reload
ERROR "lec02.hs":37 - Type error in application
*** Expression     : xs : bug xs ys
*** Term           : xs
*** Type           : [a]
*** Does not match : a
*** Because        : unification would give infinite type
```