Programowanie Funkcyjne

lato 2013/2014

Jakub Kozik

Informatyka Analityczna
tcs@jagiellonian

Zasady

Punkty

- zadania programistyczne submitowane przez Satori (zadania różnie punktowane, w sumie 80 punktów)
- aby uzyskać pozytywną ocenę należy zaliczyć wszystkie zadania oznaczone jako obowiązkowe
- egzamin (20 punktów)
- aby uzyskać pozytywną ocenę należy z egzaminu uzyskać przynajmniej 10 punktów

Ostateczna ocena

0-50	ndst
50-60	dst
60-70	+dst
70-80	db
80-90	+db
90-100	bdb

Program Wykładu

Część 1: Lambda rachunek i podstawy SML

- podstawowe konstrukcje SML'a i ich odpowiedniki w lambda rachunku
- strategie ewaluacji termów/programów
- system typów (Hindley-Milner)
- podstawowe techniki programowania funkcyjnego

Część 2: Funkcyjne struktury danych

- ocena wydajności programów funkcyjnych
- amortyzowana analiza persystentnych struktur danych
- eliminacja amortyzacji
- implementacje z uleniwianiem/wymuszaniem obliczeń

Część 3: Haskell

- leniwa ewaluacja
- monady
- ...

Literatura

Paweł Urzyczyn, Rachunek Lambda, skrypt dostępny na stronie autora

Robert Harper, Programming in Standard ML, Working Draft dostępny na stronie autora

Chris Okasaki, Purely Functional Data Structures, Cambridge University Press 1999 (wczesna wersja książki (rozprawa doktorska) dostępna na stronie autora)

Simon L. Peyton Jones, The Implementation of Functional Programming Languages, Prentice Hall 1987 (książka w wersji elektronicznej jest udostępniana przez autora)

Bryan O'Sullivan, Don Stewart, John Goerzen, Real World Haskell, O'Reilly Media, 2008 (książka w wersji elektronicznej jest udostępniana przez autorów)

Lambda rachunek - przypomnienie

Składnia

- zmienne są termami (przeliczalny zbiór zmiennych Var)
- jeśli T jest termem a x jest zmienną to $\lambda x.T$ jest termem (abstrakcja)
- jeśli T oraz S są termami to $(T \cdot S)$ jest termem (aplikacja)

Konwencje

- pomijanie $\cdot$ przy aplikacji $(T \cdot S) \equiv (TS)$
- pomijanie nawiasów – domyślne nawiasowanie do lewej $RST \equiv ((RS)T)$
- grupowanie abstrahowanych zmiennych $\lambda xy.T \equiv \lambda x.(\lambda y.T)$

Redukcja

β redukcja

$$(\lambda x.T)S \ \rightarrow_\beta \ T[x \leftarrow S]$$

(pod warunkiem że żadne wolne wystąpienie zmiennej w S nie zostaje związane w $T[x \leftarrow S]$)

α równoważność

Termy, które różnią się tylko nazwami zmiennych związanych, są równoważne i można je sobą zastępować.

$$(\lambda s\ z.s(s(z)))(\lambda s\ z.s(s(z)))$$

Obliczenia w lambda rachunku

Postać normalna

Term jest w *postaci (beta) normalnej* jeśli nie zawiera β-redex'u.

program	$\leftrightarrow$	term
ewaluacja programu	$\leftrightarrow$	wykonywanie β redukcji
wynik obliczenia	$\leftrightarrow$	term w postaci normalnej

Theorem (Church-Rosser)

Jeśli $P \,{}_\beta\!\!\twoheadleftarrow M \twoheadrightarrow_\beta Q$ *to istnieje* M' *taki że* $P \twoheadrightarrow_\beta M' \,{}_\beta\!\!\twoheadleftarrow Q$.

Wniosek

Każdy term ma co najwyżej jedną postać normalną.

W stronę języków funkcyjnych

Wzbogacamy λ rachunek o:

1. wyrażenia `let` i `letrec`
2. pattern-matching lambda abstractions
3. operator []
4. wyrażenia `case`
5. stałe: małe liczby, znaki, funkcje na małych liczbach itp.

let

Składnia (bez pattern-matchingu)

let v = B **in** E

let
    w = A
    v = B
in E

$=$

let w = A **in**
    (**let** v = B **in** E)

Tłumaczenie

$$(\textbf{let } v = B \textbf{ in } E) \quad \equiv \quad ((\lambda v.E)B)$$

letrec

Składnia (bez pattern-matchingu)

letrec

a = A

b = B

...

n = N

in E

Tłumaczenie (dla jednej zmiennej)

(**letrec** v = B **in** E) $\equiv$ (**let** v = Y (λ v.B) **in** E)

(*Y* jest kombinatorem punktu stałego)

letrec

```
letrec f= \n. IF (n=1) THEN 1 ELSE (n * (f (n-1))) in f 4
```

$$F = \lambda f\ n.\textbf{if}(n = 1)\textbf{then}\ 1\ \textbf{else}(n * f(n-1))$$

$$\begin{aligned}
& (Y\ F)\ 4 \twoheadrightarrow F\ (Y\ F)\ 4 \\
= \quad & (\lambda f\ n.\textbf{if}(n = 1)\textbf{then}\ 1\ \textbf{else}(n * f(n-1)))\ (Y\ F)\ 4 \\
\twoheadrightarrow \quad & \textbf{if}(4 = 1)\textbf{then}\ 1\ \textbf{else}(4 * ((Y\ F)(4-1))) \\
\twoheadrightarrow \quad & 4 * ((Y\ F)(4-1)) \twoheadrightarrow 4 * (F(Y\ F)(4-1)) \\
\twoheadrightarrow \quad & 4 * (\textbf{if}(4-1 = 1)\textbf{then}\ 1\ \textbf{else}((4-1) * ((Y\ F)(4-1-1)))) \\
\twoheadrightarrow \quad & 4 * (\textbf{if}(3 = 1)\textbf{then}\ 1\ \textbf{else}(3 * ((Y\ F)(3-1)))) \\
\twoheadrightarrow \quad & 4 * (3 * ((Y\ F)\ (3-1))) \twoheadrightarrow 4 * (3 * (F\ (Y\ F)\ (3-1))) \\
\twoheadrightarrow \quad & 4 * (3 * (2 * (F\ (Y\ F)\ (2-1)))) \twoheadrightarrow 4 * (3 * (2 * 1)) \\
\twoheadrightarrow \quad & 4 * (3 * 2) \twoheadrightarrow 4 * 6 \twoheadrightarrow 24
\end{aligned}$$

`let` ?

Tłumaczenie

$$(\textbf{let } \mathrm{v} = \mathrm{B} \textbf{ in } \mathrm{E}) \quad \equiv \quad ((\lambda v.E)B)$$

1. polimorfizm - system typowania inaczej traktuje konstrukcje `let` i λ abstrakcje
2. wydajność - specyficzna aplikacja $(\lambda v.E)$ jest aplikowana tylko do konkretnego argumentu B
3. `letrec` generuje Y
   1. można wydajniej obliczać bezpośrednio na termach z `letrec`
   2. termy z Y nie mogą być ewaluowane gorliwie