Paradigme de programare

2010-2011, semestrul 2

Curs 2

Cuprins

- Teza lui Church
- Calcul lambda – sintaxa si semantica operationala
- Functii curry/uncurry
- Forme normale
- Teorema Church – Rosser
- Strategii de evaluare

Teza lui Church

- Orice calcul efectiv poate fi modelat in calcul lambda (prin functii recursive)

Teza Church-Turing

- orice definitie adecvata a calculabilitatii efective se va dovedi echivalenta cu definitiile propuse de Church, respectiv Turing
- definitia lui Turing a avut un impact mai mare intrucat propunea un model de masina pe care sa se execute algoritmii
- in acelasi timp, reprezentarea numarului 2 in calcul lambda arata in felul urmator:

$$(((\lambda\, x\, (\lambda\, y\, (\lambda\, z\, ((z\, x)\, y))))(\lambda\, x\, (\lambda\, y\, y)))((\lambda\, x\, (\lambda\, y\, (\lambda\, z\, ((z\, x)\, y))))(\lambda\, x\, (\lambda\, y\, y)))(\lambda\, x\, x)))$$

Modele de calculabilitate

- Masina Turing (programare imperativa)
- Masina Lambda (programare functionala)
- Masina algoritmica Markov (programare asociativa)
- Masina de calcul cu FOL (programare logica)

Toate modelele sunt echivalente intre ele (aceleasi functii sunt calculabile in oricare din ele).

Calcul Lambda

- Inventat de Alonzo Church in 1932 ca un formalism matematic menit sa descrie comportamentul functiilor
- Nu a reusit sa inlocuiasca teoria multimilor ca fundament al matematicii, insa si-a gasit aplicatii semnificative in programare
- Limbaje bazate pe calcul lambda: Lisp, Scheme, Haskell, ML (intreaga programare functionala)

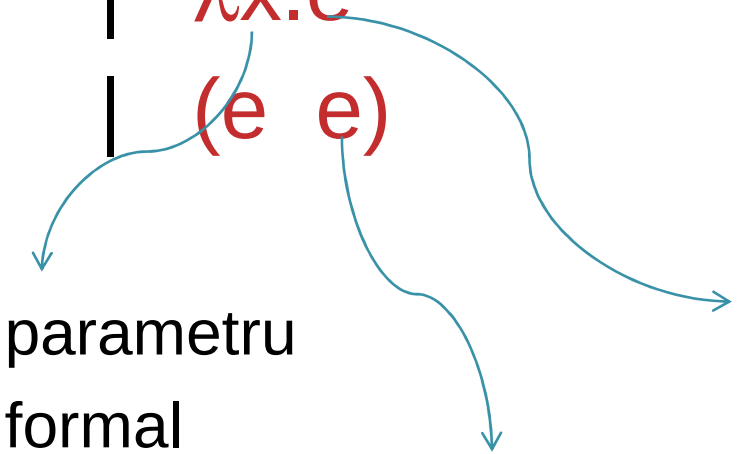
λ-expresia

Sintaxa:

e ≡ x — variabila

| λx.e — functie (unara, anonima)

| (e e) — aplicatie



parametru formal

corpul functiei

parametru efectiv

λ-expresia

Semantica:

$(\lambda x.e_1 \ \ e_2)$ = functia cu parametrul formal x si corpul e_1, aplicata asupra lui e_2

- pentru evaluare se substituie x cu e_2 in corpul e_1, se evalueaza e_1 si se intoarce rezultatul
- se noteaza $e_{1[e_2/x]}$

λ-expresii - exemple

λx.x

(x y)

λx. λy.(x y)

(λx.x a)

(λx.y a)

(λx.x λx.y)

λ-expresii - exemple

λx.x functia identitate

(x y)

λx. λy.(x y)

(λx.x a)

(λx.y a)

(λx.x λx.y)

λ-expresii - exemple

λx.x

(x y) aplicatia lui x asupra lui y

λx. λy.(x y)

(λx.x a)

(λx.y a)

(λx.x λx.y)

λ-expresii - exemple

λx.x

(x y)

λx. λy.(x y)

o functie de un parametru x care intoarce o alta functie de un parametru y care il aplica pe x asupra lui y

(λx.x a)

(λx.y a)

(λx.x λx.y)

λ-expresii - exemple

λx.x

(x y)

λx. λy.(x y)

(λx.x a) functia identitate aplicata asupra lui a (se va evalua la a)

(λx.y a)

(λx.x λx.y)

λ-expresii - exemple

λx.x

(x y)

λx. λy.(x y)

(λx.x a)

(λx.y a)

functia cu parametrul formal x si corpul y, aplicata asupra lui a (se va evalua la y)

(λx.x λx.y)

λ-expresii - exemple

λx.x

(x y)

λx. λy.(x y)

(λx.x a)

(λx.y a)

(λx.x λx.y)

functia identitate aplicata asupra functiei cu parametrul formal x si corpul y (=> λx.y)

Aparitii ale unei variabile intr-o λ-expresie

Fie o variabila x si o aparitie x_n a sa intr-o λ-expresie E.

Ex:

$\lambda x.(x \ \ \lambda y.x)$ s-ar putea rescrie ca

$\lambda x_1.(x_2 \ \ \lambda y.x_3)$ dpdv al aparitiilor lui x.

Aparitie  legata

x_n este legata in E daca:

- $E=...\lambda x_n.e\ldots$ x = variabila de legare
  x_n = aparitia care “leaga”
- $E=\ldots\lambda x.e\ldots$ si x_n apare in e

Ex: $\lambda x.(x\ \ \lambda y.x)$

Aici toate aparitiile lui x sunt legate.

Aparitie libera

O aparitie care nu este legata in E este libera in E.

Ex:

E = λx.(y λy.(x (y z)))

E = (x λx.(λx.y λy.(x z)))

Atentie!

Notiunea de aparitie libera/legata are sens in cadrul unei expresii.

Ex:

$E = \lambda x.(y\ \lambda y.(x\ (y\ z))) =_{notatie} \lambda x.e$

Orice aparitie a lui x e legata in E.

In schimb, in e, aparitia lui x este libera!

$e = (y\ \lambda y.(x\ (y\ z)))$

Exercitii

(x λy.x)

(y λy.x)

λz.((+ z) x)

(λx. λy.(x y) y)

Exercitii

(x λy.x)

(y λy.x)

λz.((+ z) x)

(λx. λy.(x y) y)

Exercitii

(x λy.x)

(y λy.x)

λz.((+ z) x)

(λx. λy.(x y) y)

Exercitii

(x λy.x)

(y λy.x)

λz.((+ z) x)

(λx. λy.(x y) y)

Exercitii

(x λy.x)

(y λy.x)

λz.((+ z) x)

(λx. λy.(x y) y)

Variabila legata/libera

O variabila x este legata intr-o λ-expresie E daca toate aparitiile ei sunt legate.

Variabilele legate sunt legate fedeles!

O variabila care nu e legata intr-o λ-expresie e libera in acea λ-expresie.

Exemple

(x λy.x)

(y λy.x)

λz.((+ z) x)

(λx.λy.(x y) y)

Obs: Si aici, notiunea are sens in cadrul unei expresii. Daca luam doar o parte din expresie, felul variabilei se poate schimba.

Exemple

$E = (\lambda x.\lambda y.(x\ \ y)\ \ y) =_{notatie} (\lambda x.e\ \ y)$

x e legata in E.

y e libera in E.

Dar in $e = \lambda y.(x\ \ y)$?

O imagine face cat 1000 de cuvinte.

E exact pe dos!

“Algoritm” de determinare a variabilelor legate/libere

BV(x) =

BV(λx.e) =

BV((e_1 e_2)) =

FV(x) =

FV(λx.e) =

FV((e_1 e_2)) =

BV = bound variables

FV = free variables

“Algoritm” de determinare a variabilelor legate/libere

$BV(x) = \emptyset$

$BV(\lambda x.e) = \{x\} \cup BV(e)$

$BV((e_1\ e_2)) =$

$BV(e_1) \cup BV(e_2) \setminus$

$(FV(e_1) \cap BV(e_2)) \setminus$

$(FV(e_2) \cap BV(e_1))$

$FV(x) =$

$FV(\lambda x.e) =$

$FV((e_1\ e_2)) =$

BV = bound variables

FV = free variables

“Algoritm” de determinare a variabilelor legate/libere

$BV(x) = \emptyset$

$BV(\lambda x.e) = \{x\} \cup BV(e)$

$BV((e_1\ e_2)) =$

$BV(e_1) \cup BV(e_2) \setminus$

$(FV(e_1) \cap BV(e_2)) \setminus$

$(FV(e_2) \cap BV(e_1))$

$FV(x) = \{x\}$

$FV(\lambda x.e) = FV(e) \setminus \{x\}$

$FV((e_1\ e_2)) =$

$FV(e_1) \cup FV(e_2)$

BV = bound variables

FV = free variables

β-redex si β-reducere

β-redex = λ-expresie de forma $(\lambda x.e_1 \ \ e_2)$

β-reducere = efectuarea calculului

$$(\lambda x.e_1 \ \ e_2) \rightarrow_\beta e_{1[e_2/x]}$$

Ex:

$(\lambda x.\lambda y.(x \ \ y) \ \ y)$ este un β-redex.

Problema cu β-reducerea

Ex1:

$(\lambda x.x\ \ t) \rightarrow_\beta x_{[t/x]} = t$ (perfect!)

Ex2:

$(\lambda x.\lambda y.(x\ \ y)\ \ y) \rightarrow_\beta$

$\lambda y.(x\ \ y)_{[y/x]} = \lambda y.(y\ \ y)$ (oooops!)

Nu asta voiam sa se intample! Voiam sa obtinem o functie care il aplica pe y asupra oricarui argument cu care va fi vreodata apelata functia. Acum orice argument este aplicat asupra lui insusi.

Mai clar, ce se intampla? De unde apare problema?

La o β-reducere $(\lambda x.e_1 \ e_2) \rightarrow_\beta e_{1[e_2/x]}$ inlocuiesc aparitiile libere are lui x in e_1 cu e_2.

Variabilele libere din e_2 se pot trezi legate in e_1!

Solutia: α-conversia

Ideea: numele parametrului formal nu conteaza, deci il putem reboteza.

α-conversie = rebotezarea sistematica a variabilelor legate dintr-o λ-expresie a.i. ele sa nu coincida cu variabilele libere din parametrul efectiv pe care aplicam expresia. ($\lambda x.e_1 \rightarrow_\alpha \lambda y.e_{1[y/x]}$, unde y nu era libera in e_1)

Ex:

$(\lambda x.\lambda y.(x\ \ y)\ \ y) \rightarrow_\alpha (\lambda x.\lambda z.(x\ \ z)\ \ y) \rightarrow_\beta$

$\lambda z.(x\ \ z)_{[y/x]} = \lambda z.(y\ \ z)$

Observatie

De ce la o β-reducere $(\lambda x.e_1\ \ e_2) \rightarrow_\beta e_{1[e_2/x]}$ inlocuiesc aparitiile libere are lui x in e_1 cu e_2?

Ex:

$(\lambda x.\lambda y.(x\ \ (y\ \ \lambda x.x))\ \ t)$ trebuie sa devina

$\lambda y.(t\ \ (y\ \ \lambda x.x))$.

Functia identitate ramane functia identitate oricum ii zicem parametrului formal, ceea ce trebuie sa devina t este primul x, iar acea aparitie (nu variabila!) e libera in e1, chiar daca e legata in $(\lambda x.e_1\ \ e_2)$.

Notatii

Pas de reducere:

$\rightarrow_\beta$ o $\rightarrow_\alpha$ se noteaza $\rightarrow$

Secventa de reducere:

$\rightarrow^*$

Exercitiu

$(\lambda x.(x\ z)\ \ \lambda z.\lambda x.(z\ x))$

Exercitiu

$(\lambda x.(x\ z)\ \ \lambda z.\lambda x.(z\ x)) \rightarrow_{\alpha}$

$(\lambda x.(x\ z)\ \ \lambda t.\lambda x.(t\ x))$

Exercitiu

$(\lambda x.(x\ z)\ \ \lambda z.\lambda x.(z\ x)) \rightarrow_{\alpha}$

$(\lambda x.(x\ z)\ \ \lambda t.\lambda x.(t\ x)) \rightarrow_{\beta}$

$(\lambda t.\lambda x.(t\ x)\ z)$

Exercitiu

$(\lambda x.(x\ z)\ \ \lambda z.\lambda x.(z\ x)) \rightarrow_{\alpha}$

$(\lambda x.(x\ z)\ \ \lambda t.\lambda x.(t\ x)) \rightarrow_{\beta}$

$(\lambda t.\lambda x.(t\ x)\ z) \rightarrow_{\beta}$

$\lambda x.(z\ x)$

Zaharel sintactic

- $\lambda x_1.\ \lambda x_2.\ \lambda x_3.\ \ldots\ \lambda x_n.e$ se noteaza
  $\lambda x_1\, x_2\, x_3\, \ldots\, x_n\, .e$

- $(\ldots(\lambda x_1.\ \lambda x_2.\ \ldots\ \lambda x_n.e\ \ p_1)\ \ p_2)\ \ldots\ p_m)$
  se noteaza
  $(\lambda x_1\, x_2\, \ldots\, x_n\, .e\ \ p_1\ \ p_2\, \ldots\, p_m)$

Noi stim ca e vorba de functii unare si de aplicatiile lor.

Functii curry/uncurry

Ati observat apelul:

$(\lambda x_1\, x_2 \ldots x_n.e \;\; p_1 \;\; p_2 \ldots p_m)$

Numarul de parametri formali nu coincide cu numarul de parametri efectivi. Se poate una ca asta?

DA! Se poate, iar rezultatul unui astfel de apel este o functie de restul de parametri (formali).

Functii curry/uncurry

Functia curry

- isi ia parametrii pe rand
- poate fi aplicata partial (doar pe o parte din parametri) – rezultand o noua functie
- $(\lambda x_1\, x_2 \ldots x_n.e\;\; p_1\;\; p_2 \ldots p_m) \rightarrow^*$
  $\lambda x_{m+1}\, x_{m+2} \ldots x_n.e_{[p_1/x_1,\, p_2/x_2, \ldots p_m/x_m]}$

Functia uncurry

- Isi ia obligatoriu toti parametrii deodata

Exemplu (in Scheme)

Sa urmarim impreuna cate o functie curry/uncurry pentru adunarea a 2 numere…

… si altele.

Functii curry - concluzii

- Functii curry $\Longrightarrow$ reutilizare de cod
- Sunt suportate de majoritatea limbajelor functionale
- Desi nu exista nici un motiv ca celelalte limbaje sa nu le aiba, in general nu le au

Forme normale

O λ-expresie e in forma normala = nu mai contine niciun β-redex.

- Are orice λ-expresie o forma normala?
- Daca o λ-expresie admite o forma normala, pot garanta gasirea ei?
- Secvente distincte de reducere pot duce la forme normale distincte?

Observatie

- raspunsurile la toate intrebarile de mai sus ar trebui sa devina evidente daca ne gandim ca avem de-a face cu un model de calculabilitate: lambda-expresiile sunt practic programe capabile sa ruleze pe o ipotetica masina Lambda

Are orice λ-expresie o forma normala?

NU.

Ex:

$(\lambda x.(x\ \ x)\ \ \lambda x.(x\ \ x)) \rightarrow$

?

λ-expresii (i)reductibile

$(\lambda x.(x\ \ x)\ \ \lambda x.(x\ \ x)) \rightarrow$

$(\lambda x.(x\ \ x)\ \ \lambda x.(x\ \ x)) \rightarrow$

$(\lambda x.(x\ \ x)\ \ \lambda x.(x\ \ x)) \rightarrow$

…

λ-expresie reductibila = admite o secventa finita de reducere care se termina cu o forma normala

Altfel: ireductibila

Daca o λ-expresie admite o forma normala, pot garanta gasirea ei?

DA.

Ex:

$E_1 = (\lambda x.(x\ \ x)\ \ \lambda x.(x\ \ x))$

$E_2 = (\lambda x.y\ E_1) \rightarrow y$

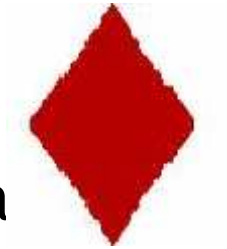
Daca incepeam sa reduc in interiorul E_1 nu mai terminam niciodata.

Teorema normalizarii

Daca o λ-expresie este reductibila, atunci voi ajunge la forma ei normala aplicand reducere stanga->dreapta (reducand mereu cel mai din stanga β-redex)

Secvente distincte de reducere pot duce la forme normale distincte?

NU.

Lema  caroului (the diamond lemma)

Daca $e \to a$ si $e \to b$ atunci $\exists$ d a.i. $a \to d$ si $b \to d$

Teorema Church-Rosser

Daca $e \rightarrow^* a$ si $e \rightarrow^* b$ atunci $\exists$ d a.i. $a \rightarrow^* d$ si $b \rightarrow^* d$

Se demonstreaza cu ajutorul lemei caroului.

Corolar: Daca o λ-expresie este reductibila, atunci forma normala este unica.

Strategii de evaluare

= reguli de evaluare a expresiilor intr-un limbaj de programare

2 mari categorii:

- Strategii stricte
- Strategii nestricte

Evaluare/functie stricta/nestricta

Evaluare stricta = argumentele unei functii sunt evaluate la apel (inainte ca functia sa fie aplicata)

Evaluare nestricta = argumentele unei functii nu sunt evaluate pana ce valoarea lor nu e efectiv necesara undeva in corpul functiei

Functie stricta/nestricta = functie care isi evalueaza strict/nestrict argumentele

In practica

- Limbajele care sunt stricte tind sa aiba si cateva functii cu evaluare nestricta
- Ex: Scheme (if, and, or)

Strategii stricte

- Evaluare aplicativa: la intalnirea unui redex ($\lambda x.\ e_1\ \ e_2$), mai intai se reduce e_2 cat de mult se poate
- Call by value: argumentul e evaluat inainte de a fi pasat functiei; functiei i se da o copie a valorii rezultate in urma evaluarii (Pascal, C, Java, Scheme, Ocaml etc)
- Call by reference: functiei i se paseaza o referinta la argument; in principiu asta inseamna ca il poate modifica (Perl, Visual Basic; C simuleaza cu ajutorul pointerilor)

Strategii nestricte

- Evaluare normala: evaluare stanga->dreapta; difera de call by name prin aceea ca face evaluari in corpul functiilor inca neaplicate
- Call by name: argumentele nu se evalueaza deloc, se transmit ca atare si, daca e nevoie de ele, se reevalueaza de fiecare data cand e nevoie (lent dar sigur)
- Call by need: un call by name in care prima evaluare stocheaza rezultatul intr-un cache (va fi luat de acolo cand va mai fi nevoie de el) (Haskell, R)