Fundamentele limbajelor de programare

Programare funcțională. λ-calcul.

Traian Florin Șerbănuță și Andrei Sipoș

Facultatea de Matematică și Informatică, DL Info

Anul II, Semestrul II, 2024/2025

Secțiunea 1

Lambda calcul - elemente de bază

Lambda calcul

- Un model de calculabilitate
- Limbajele de programare funcțională sunt extensii ale sale
- Un limbaj formal
  - Expresiile din acest limbaj se numesc lambda termeni
  - Vom defini reguli pentru a îi manipula

Lambda-termeni: Interpretarea informală

λ-termenii au fost gândiți ca reprezentând funcții. Mai exact (fixând x, y, $z \in V$ distincte două câte două):

- Un termen de forma $\lambda x.M$ este gândit ca reprezentând funcția care duce pe x în M (M fiind un termen în a cărui componență poate sau nu să apară variabila x).
  De exemplu: $\lambda x.x$ ar reprezenta funcția identitate, $\lambda x.y$ ar reprezenta o funcție constant egală cu y.
- Un termen de forma MN reprezintă rezultatul aplicării „funcției" M pe „argumentul" N.
  De exemplu: am vrea ca $(\lambda x.x)z$ să reprezinte z, iar $(\lambda x.y)z$ să reprezinte y.

Remarcăm că aceste interpretări sunt aici pur informale: a le face riguroase a fost mult timp o problemă aproape insurmontabilă. Situația devine mai ușoară dacă nu ne propunem să formalizăm termenii ca funcții (semantică denotațională), ci doar să stabilim regulile prin care ei sunt manipulați (semantică operațională).

Lambda termeni: Descrierea formală

Fie V o mulțime infinită de variabile, notate $x, y, z, \ldots$.

Mulțimea lambda termenilor este dată de următoarea formă BNF:

lambda termen	$=$	variabilă
	$\mid$	aplicare
	$\mid$	abstractizare

$M, N ::= x \mid (M\, N) \mid (\lambda x.\, M)$

Exemple

- x, y, z
- $(x\, y), (y\, x), (x\, (yx))$
- $(\lambda x.\, x), (\lambda x.\, (x\, y)), (\lambda z.\, (x\, y))$
- $((\lambda x.\, x)\, y), ((\lambda x.\, (x\, z))\, y)$
- $(\lambda f.\, (\lambda x.\, (f\, (f\, x))))$
- $((\lambda x.\, x)\, (\lambda x.\, x))$

Funcții anonime în Haskell

lambda termen	$=$	variabilă
	$\mid$	aplicare
	$\mid$	abstractizare

$M, N ::= x \mid (M\, N) \mid (\lambda x.\, M)$

În Haskell, \ e folosit în locul simbolului λ și `->` în locul punctului:

$\lambda x.x * x$	`\x -> x * x`
$\lambda x.x > 0$	`\x -> x > 0`

Convenții

- Se elimină parantezele exterioare
- Aplicarea este asociativă la stânga
  - $M\,N\,P$ înseamnă $(M\,N)\,P$
  - $f\,x\,y\,z$ înseamnă $((f\,x)\,y)\,z$
- Corpul abstractizării (partea de după punct) se extinde la dreapta cât se poate
  - $\lambda x.\,M\,N$ înseamnă $\lambda x.\,(M\,N)$, nu $(\lambda x.\,M)\,N$
- Mai mulți λ pot fi comprimați
  - $\lambda xyz.\,M$ este o abreviere pentru $\lambda x.\,\lambda y.\,\lambda z.\,M$

Aceste convenții nu afectează definiția lambda termenilor.

Exerciții

Exercițiu. Scrieți termenii de mai jos cu cât mai puține paranteze și folosind convențiile de mai sus, fără a schimba sensul termenilor:

1. $(\lambda x.(\lambda y.(\lambda z.((x\,z)(y\,z)))))$
2. $(((a\,b)\,(c\,d))\,((e\,f)\,(g\,h)))$

Exercițiu. Adăugați parantezele în termenii de mai jos astfel încât să nu le schimbați sensul:

1. $x\,x\,x\,x$
2. $\lambda x.\,x\,\lambda y.\,y$

Variabile libere și variabile legate

- $\lambda\_.\,\_$ se numește operator de legare (*binder*)
- x din $\lambda x.\,\_$ se numește variabilă de legare (*binding*)
- N din $\lambda x.\, N$ se numește domeniul (*scope*) de legare a lui x
- toate aparițiile lui x în N sunt legate
- O apariție care nu este legată se numește liberă.
- Un termen fără variable libere se numește închis (*closed*).
- Un termen închis se mai numește și combinator.

De exemplu, în termenul

$M \equiv (\lambda x.\, xy)\,(\lambda y.\, yz)$

- x este legată
- z este liberă
- y are și o apariție legată, și una liberă
- mulțimea variabilelor libere ale lui M este $\{y, z\}$

Variabile libere

Mulțimea variabilelor libere dintr-un termen M este notată $FV(M)$ și este definită formal prin:

$$
\begin{array}{rcl}
FV(x) & = & \{x\} \\
FV(M\,N) & = & FV(M) \cup FV(N) \\
FV(\lambda x.\, M) & = & FV(M) \setminus \{x\}
\end{array}
$$

Spre substituție

De exemplu, când spunem că $(\lambda x.x)z$ am vrea să reprezinte z, sugerăm că x-ul din corpul „funcției" am vrea să fie substituit cu z. Pentru aceasta, avem nevoie de o definiție a substituției. Nu putem substitui **naiv** variabilele cu λ-termeni din aceleași considerente ca la logica de ordinul I: am putea să ne trezim cu urmări nedorite, de exemplu, dacă în λ-termenul

$$\lambda x.y,$$

reprezentând funcția „constant egală cu y", substituim fără atenție y cu x, ajungem la λ-termenul

$$\lambda x.x,$$

care reprezintă o funcție identitate (variabila x fiind capturată „accidental" de către λx). Or, aceasta contravine intuiției care ne spune că o funcție compusă cu una constantă nu poate fi neconstantă.

Spre substituție

Observăm următorul fapt: termeni ca $\lambda x.x$ și $\lambda z.z$ am dori să denote aceeași funcție, funcția identitate; la fel și $\lambda x.y$ și $\lambda z.y$ aceeași funcție, funcția constant egală cu y. Așadar, vom transforma întâi $\lambda x.y$ în

$$\lambda z.y,$$

pentru a putea substitui apoi y cu x, obținând

$$\lambda z.x,$$

care este tot o funcție constantă.

Practic, ideea este că denumirile variabilelor legate nu contează, atâta timp cât ele sunt folosite consecvent: de aceea, ele se pot și substitui una cu alta atâta timp cât și substituția este consecventă. Dat fiind că în acest principiu se amintește de substituție, el se va putea formaliza abia după definirea substituției. Totuși, acea parte a sa care este relevantă pentru definirea substituției poate fi inclusă în definiție, folosind recursivitatea.

Definirea substituției

Pentru orice λ-termeni M, N și orice $x \in V$, vom defini termenul $M[x := N]$, reprezentând M în care x a fost înlocuit cu N. O vom face recursiv, în felul următor (unde x, $y \in V$, iar N, P, Q sunt λ-termeni):

- $x[x := N] := N$;
- $x[y := N] := x$, dacă $y \neq x$;
- $(PQ)[x := N] := (P[x := N])(Q[x := N])$;
- $(\lambda x.P)[x := N] := \lambda x.P$;
- $(\lambda y.P)[x := N] := \lambda y.(P[x := N])$, dacă $y \neq x$ și $y \notin FV(N)$;
- $(\lambda y.P)[x := N] := \lambda z.(P[y := z][x := N])$, dacă $y \neq x$ și $y \in FV(N)$, unde z este o variabilă „nouă”[1]

[1] variabila de indice minim diferită de x și care nu apare în N sau P, caz care corespunde fenomenului prezentat mai devreme.

Exemple

Care sunt următorii λ-termeni (presupunem u, v, w, x, y, $z \in V$, distincte două câte două)?

- $(\lambda y.(x(\lambda w.((vw)x))))[x := uv]$;
- $(\lambda y.(x(\lambda x.x)))[x := \lambda y.(xy)]$;
- $(y(\lambda v.(xv)))[x := \lambda y.(vy)]$;
- $(\lambda x.(zy))[x := uv]$.

Alpha-echivalență

În acest moment, putem formaliza intuiția de mai devreme legată de substituția variabilelor legate. Numim α-echivalență și o notăm cu $\equiv_\alpha$ cea mai mică relație de echivalență $\equiv$ pe λ-termeni care satisface următoarele:

- pentru orice x, $y \in V$ și orice λ-termen M cu $y \notin FV(M)$, $\lambda x.M \equiv \lambda y.(M[x := y])$;
- pentru orice $x \in V$ și orice λ-termeni M, N cu $M \equiv N$, avem $\lambda x.M \equiv \lambda x.N$;
- pentru orice λ-termeni M, N, P cu $M \equiv N$, avem $MP \equiv NP$ și $PM \equiv PN$.

Secțiunea 2

Lambda calcul - β-reducții

Spre reducție

Am spus mai devreme că $(\lambda x.x)z$ am vrea să reprezinte z, iar pentru aceasta am introdus o definiție a substituției astfel încât $x[x := z] = z$. Totuși, mai trebuie să spunem și de ce putem face trecerea

$$(\lambda x.x)z \to x[x := z]$$

sau, în celălalt exemplu,

$$(\lambda x.y)z \to y[x := z]$$

și, în general,

$$(\lambda x.M)N \to M[x := N].$$

Pentru aceasta, vom introduce o nouă relație pe λ-termeni, care va reprezenta această procedură de reducție.

Beta-reducție

Numim β-reducție și o notăm cu $\rightarrow_\beta$ cea mai mică relație $\rightarrow$ pe λ-termeni care satisface următoarele, pentru orice λ-termeni M, N, P și orice $x \in V$:

- $(\lambda x.M)N \rightarrow M[x := N]$;
- dacă $M \rightarrow N$, atunci $\lambda x.M \rightarrow \lambda x.N$, $MP \rightarrow NP$ și $PM \rightarrow PN$.

Notăm cu $\rightarrow_\beta^*$ închiderea reflexiv-tranzitivă a lui $\rightarrow_\beta$.

Un λ-termen M se numește **formă normală** dacă nu există N cu $M \rightarrow_\beta N$. Dacă M și N sunt λ-termeni, N se numește **formă normală a lui** M dacă $M \rightarrow_\beta^* N$ și N este formă normală.

β-reducții

La fiecare pas, subliniem redexul ales în procesul de β-reducție.

$$
\begin{array}{lcl}
(\lambda x.\, y)\,(\underline{(\lambda z.\, zz\,)\,(\lambda w.\, w)}) & \to_\beta & (\lambda x.\, y)\,((z\,z)[z := \lambda w.\, w]) \\
 & \equiv & (\lambda x.\, y)\,((z[z := \lambda w.\, w])\,(z[z := \lambda w.\, w]) \\
 & \equiv & (\lambda x.\, y)\,(\underline{(\lambda w.\, w)\,(\lambda w.\, w)}) \\
 & \to_\beta & \underline{(\lambda x.\, y)\,(\lambda w.\, w)} \\
 & \to_\beta & y
\end{array}
$$

Ultimul termen nu mai are redex-uri, deci este în formă normală.

β-reducții

$$
\begin{array}{lll}
(\lambda x.\, y)\, (\underline{(\lambda z.\, zz)\, (\lambda w.\, w)}) & \to_\beta & (\lambda x.\, y)\, (\underline{(\lambda w.\, w)\, (\lambda w.\, w)}) \\
 & \to_\beta & \underline{(\lambda x.\, y)\, (\lambda w.\, w)} \\
 & \to_\beta & y
\end{array}
$$

$$
\begin{array}{lll}
\underline{(\lambda x.\, y)\, ((\lambda z.\, zz)\, (\lambda w.\, w))} & \to_\beta & y[x := (\lambda z.\, zz)\, (\lambda w.\, w)] \\
 & \equiv & y
\end{array}
$$

Observăm că:

- reducerea unui redex poate crea noi redex-uri
- reducerea unui redex poate șterge alte redex-uri
- numărul de pași necesari până a atinge o formă normală poate varia, în funcție de ordinea în care sunt reduse redex-urile
- rezultatul final pare că nu a depins de alegerea redex-urilor

β-formă normală

Totuși, există lambda termeni care nu pot fi reduși la o β-formă normală (evaluarea nu se termină).

$$
\begin{array}{lcl}
\hline
\underline{(\lambda x.\, x\, x)\,(\lambda x.\, x\, x)} & \to_\beta & (\lambda x.\, x\, x)\,(\lambda x.\, x\, x) \\
 & \to_\beta & \ldots \\
\hline
\end{array}
$$

Observați că lungimea unui termen nu trebuie să scadă în procesul de β-reducție; poate crește sau rămâne neschimbată.

β-formă normală

Există lambda termeni care deși pot fi reduși la o formă normală, pot să nu o atingă niciodată.

$$
\begin{array}{lcl}
\underline{(\lambda xy.\, y)\,((\lambda x.\, x\,x)\,(\lambda x.\, x\,x))}\,(\lambda z.\, z) & \to_\beta & \underline{(\lambda y.\, y)\,(\lambda x.\, x)} \\
 & \to_\beta & \lambda x.\, x \\
(\lambda xy.\, y)\,(\underline{(\lambda x.\, x\,x)\,(\lambda x.\, x\,x)})\,(\lambda z.\, z) & \to_\beta & (\lambda xy.\, y)\,((\lambda x.\, x\,x)\,(\lambda x.\, x\,x))\,(\lambda z.
\end{array}
$$

Contează strategia de evaluare.

β-formă normală

Notăm cu $M \twoheadrightarrow_\beta M'$ faptul că M poate fi β-redus până la M' în 0 sau mai mulți pași (închiderea reflexivă și tranzitivă a relației $\rightarrow_\beta$).

M este slab normalizabil *(weakly normalising)* dacă există N în formă normală astfel încât $M \twoheadrightarrow_\beta N$.

M este puternic normalizabil *(strong normalising)* dacă nu există reduceri infinite care încep din M.

Orice termen puternic normalizabil este și slab normalizabil.

Exemplu

$(\lambda x.\, y)\,((\lambda z.\, zz\,)\,(\lambda w.\, w))$ este puternic normalizabil.
$(\lambda xy.\, y)\,((\lambda x.\, x\, x)\,(\lambda x.\, x\, x))\,(\lambda z.\, z)$ este slab normalizabil,
dar nu puternic normalizabil.

Confluența β-reducției

Teorema Church-Rosser. Dacă $a \twoheadrightarrow_\beta b$ și $a \twoheadrightarrow_\beta c$ atunci există d astfel încât $b \twoheadrightarrow_\beta d$ și $c \twoheadrightarrow_\beta d$.

$$
\begin{tikzcd}
\mathbf{a} \arrow[r, "*"] \arrow[d, "*"'] & \mathbf{b} \arrow[d, dashed, "*"] \\
\mathbf{c} \arrow[r, dashed, "*"] & \mathbf{d}
\end{tikzcd}
$$

Figure 1: Confluență

Consecință. Un lambda termen are cel mult o β-formă normală (modulo α-echivalență).

Exercițiu. Pot fi termenii de mai jos aduși la o β-formă normală?

1. $(\lambda x.\, x)\, M$
2. $(\lambda xy.\, x)\, M\, N$
3. $(\lambda x.\, x\, x)\, (\lambda y.\, y\, y\, y)$

Secțiunea 3

Strategii de evaluare

Strategii de evaluare

De cele mai multe ori, există mai mulți pași de β-reducție care pot fi aplicați unui termen. Cum alegem ordinea? Contează ordinea?

O strategie de evaluare ne spune în ce ordine să facem pașii de reducție.

Lambda calculul nu specifică o strategie de evaluare, fiind nedeterminist. O strategie de evaluare este necesară în limbaje de programare reale pentru a rezolva nedeterminismul.

Strategia normală (normal order)

Strategia normală $=$ *leftmost-outermost*
(alegem redex-ul cel mai din stânga și apoi cel mai din exterior)

- dacă M_1 și M_2 sunt redex-uri și M_1 este un subtermen al lui M_2, atunci M_1 nu va fi următorul redex ales
- printre redex-urile care nu sunt subtermeni ai altor redex-uri (și sunt incomparabili față de relația de subtermen), îl alegem pe cel mai din stânga.

Dacă un termen are o formă normală, atunci strategia normală va converge la ea.

$$\begin{array}{rcl} \underline{(\lambda xy.\,y)\,((\lambda x.\,x\,x)\,(\lambda x.\,x\,x))}\,(\lambda z.\,z) & \to_\beta & \underline{(\lambda y.\,y)\,(\lambda x.\,x)} \\ & \to_\beta & \lambda x.\,x \end{array}$$

Strategia aplicativă (applicative order)

Strategia aplicativă = *leftmost-innermost*
(alegem redex-ul cel mai din stânga și apoi cel mai din interior)

- dacă M_1 și M_2 sunt redex-uri și M_1 este un subtermen al lui M_2, atunci M_2 nu va fi următorul redex ales
- printre redex-urile care nu sunt subtermeni ai altor redex-uri (și sunt incomparabili față de relația de subtermen), îl alegem pe cel mai din stânga.

$$(\lambda xy.\, y)\,(\underline{(\lambda x.\, x\,x)\,(\lambda x.\, x\,x)})\,(\lambda z.\, z) \to_\beta$$
$$(\lambda xy.\, y)\,((\lambda x.\, x\,x)\,(\lambda x.\, x\,x))\,(\lambda z.\, z)$$

Strategii în programare funcțională

În limbaje de programare funcțională, în general, reducerile din corpul unei λ-abstractizări nu sunt efectuate (deși anumite compilatoare optimizate pot face astfel de reduceri în unele cazuri).

Strategia *call-by-name* (CBN) = strategia normală fără a face reduceri în corpul unei λ-abstractizări

Strategia *call-by-value* (CBV) = strategia aplicativă fără a face reduceri în corpul unei λ-abstractizări

Majoritatea limbajelor de programare funcțională folosesc CBV, excepție făcând Haskell.

CBN vs CBV

O valoare este un λ-term pentru care nu există β-reducții date de strategia de evaluare considerată.

De exemplu, $\lambda x.\, x$ este mereu o valoare, dar $(\lambda x.\, x)\, 1$ nu este.

Sub CBV, funcțille pot fi apelate doar prin valori (argumentele trebuie să fie complet evaluate). Astfel, putem face β-reducția $(\lambda x.\, M)\, N \twoheadrightarrow_\beta M[x := N]$ doar dacă N este valoare.

Sub CBN, amânăm evaluarea argumentelor cât mai mult posibil, făcând reducții de la stânga la dreapta în expresie. Aceasta este strategia folosită în Haskell.

CBN este o formă de evaluare leneșă *(lazy evaluation)*: argumentele funcțiilor sunt evaluate doar când sunt necesare.

CBN vs CBV

Exemplu

Considerăm 3 și *succ* primitive.
Strategia CBV:

$$
\begin{array}{lcl}
(\lambda x.\, succ\, x)\, ((\lambda y.\, succ\, y)\, 3) & \twoheadrightarrow_\beta & (\lambda x.\, succ\, x)\, (succ\, 3) \\
 & \to & (\lambda x.\, succ\, x)\, 4 \\
 & \twoheadrightarrow_\beta & succ\, 4 \\
 & \to & 5
\end{array}
$$

Strategia CBN:

$$
\begin{array}{lcl}
(\lambda x.\, succ\, x)\, ((\lambda y.\, succ\, y)\, 3) & \twoheadrightarrow_\beta & succ\, ((\lambda y.\, succ\, y)\, 3) \\
 & \twoheadrightarrow_\beta & succ\, (succ\, 3) \\
 & \to & succ\, 4 \\
 & \to & 5
\end{array}
$$