Fundamentele limbajelor de programare
CURS 8

Traian Florin Șerbănuță și Andrei Sipoș

Facultatea de Matematică și Informatică, DL Info, Anul II
Semestrul II, 2024/2025

Motivație pentru tipuri

Am observat că λ-calculul fără tipuri are anumite „patologii", de exemplu termenul $(\lambda x.(xx))(\lambda x.(xx))$, care „se rescrie la infinit".

Ne putem întreba care este sursa acestui fenomen, pentru a ști cum l-am putea evita. De exemplu, am putea spune că xx sugerează că am avea o funcție care își aparține propriului său domeniu de definiție, lucru care, intuitiv vorbind, „nu prea ar avea sens"[1].

O cale spre a nu face asemenea amestecuri este, de pildă, ideea de a impune **tipuri** termenilor. Există mai multe moduri de a face aceasta. Noi îl vom studia pe cel mai simplu, care chiar poartă numele de „**tipuri simple".**

[1] De fapt, poate foarte bine să aibă sens, după cum am pomenit în treacăt cursul trecut.

Tipuri simple

Fixăm o mulțime de „tipuri de bază”, notată cu T_0.

Mulțimea tipurilor simple (numite **tipuri**, de acum încolo), notată cu T, va fi cea mai mică mulțime care:

- conține pe T_0;
- pentru orice $\rho, \tau \in T$, avem $\rho \to \tau \in T$ (cu semnificația: „tipul funcțiilor de la ρ la τ”).

Putem vedea, așadar, mulțimea tipurilor ca pe mulțimea termenilor peste o signatură de ordinul I unde avem un singur simbol, anume un simbol de operație de aritate 2, notat cu $\to$.

Tipurile, intuitiv

Cum funcționează, intuitiv, tipurile?

- În primul rând, variabilele „vor avea" tipuri (intenționat păstrăm ambiguitatea asupra sintagmei, vom vedea imediat de ce).
- Dacă avem o variabilă x de tip ρ și un termen M de tip τ, atunci $\lambda x.M$, cum reprezintă funcția care „duce pe x în M", va trebui să aibă tipul $\rho \to \tau$.
- Într-un termen de forma MN, dacă ρ este tipul lui N, atunci M va trebui să reprezinte o funcție care duce ceva de tip ρ în ceva de alt tip, să zicem τ. Deci M are tipul $\rho \to \tau$ și MN are tipul τ. Așadar, MN nu va avea sens decât atunci când tipurile lui M și N „se potrivesc".

Vedem acum că, oricum am defini riguros ideea de tip, dacă avem un termen de forma xx, x va trebui să aibă și un tip ρ, și un tip $\rho \to \tau$. Dar, pentru orice ρ și τ, $\rho \neq \rho \to \tau$, deci avem o contradicție. Așadar, xx nu va fi „bine format".

Tipuri à la Church

Am menținut ambiguitatea asupra definiției deoarece există două feluri de a defini modul cum termenii au tipuri. Primul este noțiunea de tip à la Church.

În acest sistem, termenii sunt din start definiți ca având tipuri:

- avem câte o mulțime numărabilă de variabile pentru fiecare tip (aceste mulțimi fiind disjuncte două câte două);
- dacă x este o variabilă de tip ρ și M este un termen de tip τ, atunci $\lambda x.M$ este un termen de tip $\rho \to \tau$;
- dacă M și N sunt termeni, iar ρ și τ sunt tipuri astfel încât M este de tip $\rho \to \tau$, iar N este de tip ρ, atunci MN este un termen de tip τ.

De exemplu, dacă α, β, $\gamma \in T$, iar x, y, z și u sunt variabile de tip $\alpha \to \alpha$, $(\alpha \to \alpha) \to \beta$, β și γ, respectiv, atunci $(\lambda z.(\lambda u.z))(yx)$ este un termen de tip $\gamma \to \beta$.

Substituție și reducție

Substituția se definește întocmai ca la λ-calculul fără tipuri, dar având grijă la tipuri. De exemplu, dacă $\sigma, \tau \in T$, t este un termen de tip τ, x este o variabilă de tip σ, iar u un termen de tip σ, atunci va avea sens să definim

$$t[x := u],$$

care va fi, apoi, un termen de tip τ.

Similar, relația de β-reducție se definește identic, având grijă, de exemplu, ca, în clauza fundamentală

$$(\lambda x.t)u \to t[x := u],$$

obiectele să aibă tipurile potrivite. (Care sunt acelea? De ce merge?)

Normalizare și confluență

Se poate arăta (și chiar vom arăta într-un curs viitor) că acest λ-calcul cu tipuri simple nu mai posedă rescrieri infinite (altfel spus, are proprietatea de **normalizare tare**), i.e. nu există o familie de termeni $(M_n)_{n\in\mathbb{N}}$ astfel încât pentru orice $n \in \mathbb{N}$, $M_n \to_\beta M_{n+1}$.

În plus, el păstrează proprietatea de **confluență** a λ-calculului fără tipuri.

Tipuri à la Curry

Un al doilea mod de a gândi tipurile este sistemul de tipuri à la Curry. Aici, termenii sunt fără tipuri, mai exact ei sunt exact termeni din λ-calculul fără tipuri, iar lor li se alocă tipuri de către un sistem de deducție. Mai precis, vom emite judecăți de forma

$$\Gamma \vdash M : \tau,$$

însemnând: în contextul Γ i se alocă termenului M tipul τ. Un context va fi o mulțime finită de obiecte de forma $x : \sigma$, cu x variabilă și $\sigma \in \mathcal{T}$ (simbolizând faptul că variabilei x i se alocă tipul σ), astfel încât orice variabilă apare cu cel mult un tip.

Reguli de deducție

Regulile de deducție vor fi următoarele:

$$\frac{}{\Gamma \cup \{x : \sigma\} \vdash x : \sigma}$$

$$\frac{\Gamma \cup \{x : \sigma\} \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

Încercați să deduceți tipul termenului „de mai devreme” $(\lambda z.(\lambda u.z))(yx)$ folosind aceste reguli. Acest exemplu ne arată că, într-un anume fel (în ce fel? detaliați!), cele două moduri de a gândi tipurile sunt echivalente.

Inferența tipurilor

Am dori, ca, dat un termen M în λ-calculul fără tipuri, să decidem **algoritmic** dacă există (și, atunci, să și găsim) Γ și τ cu $\Gamma \vdash M : \tau$. Aceasta se numește problema inferenței tipurilor. În continuare, vom prezenta un asemenea algoritm.

Algoritmul va avea nevoie, aproape la fiecare pas, de variabile (de tip) noi, nemaifolosite până atunci (și, deci, în particular, diferite între ele). Vom presupune tacit, în prezentarea sa, că variabilele de tip care apar au această proprietate. (Acesta va fi și punctul sensibil atunci când se încearcă implementarea lui.)

Introducem și noțiunea de **termen adnotat** ca fiind un termen cu tip undeva „între" Church și Curry, în sensul că termenii nu au tipuri, dar, la fiecare λ-abstracțiune, se adaugă și un tip pentru variabila abstractizată. Vom scrie un asemenea termen ca: $\lambda x : \sigma.M$.

Algoritmul

Definim, pentru orice termen M, contextul

$$\Gamma_M := \{x : X \mid x \in FV(M)\}.$$

De asemenea, pentru orice termen M, construim termenul adnotat M', unde toate λx-urile devin $\lambda x : X$.

În continuare, vom defini o funcție c care primește ca argumente: un termen adnotat, un context și o variabilă de tip și care returnează o mulțime de ecuații peste signatura de ordinul I care conține doar simbolul săgeată (vezi discuția de la început).

Finalizare

Definim:

$$c(x, \Gamma \cup \{x : \tau\}, Z) := \{\tau = Z\}$$
$$c(\lambda x : \sigma.M, \Gamma, Z) := c(M, \Gamma \cup \{x : \sigma\}, W) \cup \{Z = \sigma \to W\}$$
$$c(MN, \Gamma, Z) := c(M, \Gamma, W_1) \cup c(N, \Gamma, W_2) \cup \{W_1 = W_2 \to Z\}$$

Algoritmul va face apelul $c(M', \Gamma_M, Z)$ și va obține o mulțime de ecuații. Pentru ea, se caută un cgu θ. În caz că nu există, se returnează „eșec”. În caz de succes, rezultatul algoritmului va fi $\widetilde{\theta}(\Gamma_M) \vdash M : \widetilde{\theta}(Z)$ (unde $\widetilde{\theta}$ are semnificația firească).

Putem rula algoritmul pe termenii $(\lambda z.(\lambda u.z))(yx)$ și xx, evidențiați mai devreme.

Numere naturale

Să vedem acum cum putem vorbi despre λ-calculul cu tipuri simple ca model de calcul. Dacă ne uităm la un numeral Church oarecare, să zicem C_2 (care îl reprezintă pe 2):

$$\lambda f.\lambda x.(f(fx)),$$

vedem că, dacă încercăm să îi alocăm un tip, avem că, notând tipul variabilei x cu γ, tipul rezultant va fi $(\gamma \to \gamma) \to (\gamma \to \gamma)$. Vom fixa, așadar, un $\gamma \in T_0$ și vom defini **nat**, tipul numerelor naturale, ca fiind $(\gamma \to \gamma) \to (\gamma \to \gamma)$.

Vedem acum că, de exemplu, termenului introdus anterior pentru adunare,

$$\lambda n.\lambda m.\lambda f.\lambda x.((nf)((mf)x)),$$

i se poate aloca tipul $\mathbf{nat} \to (\mathbf{nat} \to \mathbf{nat})$. Ce va însemna, deci, acum că o funcție este reprezentabilă?

Funcții calculabile

Vom spune, pentru orice $k \in \mathbb{N}$, că o funcție $f : \mathbb{N}^k \to \mathbb{N}$ (observăm că, dată fiind proprietatea de normalizare, acum lucrăm doar cu funcții **totale**) este $\lambda_{\to}$-**reprezentabilă** dacă există un λ-termen M de tipul $\mathbf{nat}^k \to \mathbf{nat}$ astfel încât, pentru orice $n_1, \ldots, n_k \in \mathbb{N}$, avem

$$MC_{n_1} \ldots C_{n_k} \to_\beta^* C_{f(n_1,\ldots,n_k)}.$$

Apare întrebarea: orice funcție totală care este calculabilă (adică λ-reprezentabilă) va fi $\lambda_{\to}$-reprezentabilă? Răspunsul va fi unul negativ.

Caracterizare

Teoremă (Schwichtenberg, 1976)

Clasa funcțiilor $\lambda_{\rightarrow}$-reprezentabile este cea mai mică clasă de funcții totale care:

- conține constantele 0, 1, adunarea, înmulțirea și operațiile de proiecție pe o componentă;
- conține operația **cond** : $\mathbb{N}^3 \to \mathbb{N}$, definită, pentru orice n, m, $p \in \mathbb{N}$ prin

$$\mathbf{cond}(n, m, p) := \begin{cases} m, & \text{dacă } n = 0, \\ p, & \text{altfel.} \end{cases}$$

- este închisă la compunere, în sensul că, pentru orice k, $l \in \mathbb{N}$ și orice funcții $f : \mathbb{N}^l \to \mathbb{N}$, $g_1, \ldots, g_l : \mathbb{N}^k \to \mathbb{N}$, conține, odată cu f, $g_1, \ldots, g_l$, și funcția $h : \mathbb{N}^k \to \mathbb{N}$, definită, pentru orice $x_1, \ldots, x_k \in \mathbb{N}$, prin

$$h(x_1, \ldots, x_k) := f(g_1(x_1, \ldots, x_k), \ldots, g_l(x_1, \ldots, x_k)).$$

Funcții nereprezentabile

Următorul rezultat se arată imediat prin inducție.

Corolar

Pentru orice $k \in \mathbb{N}$, orice funcție $\lambda_{\to}$-reprezentabilă $f : \mathbb{N}^k \to \mathbb{N}$ există un polinom $p \in \mathbb{N}[X_1, \ldots, X_k]$ astfel încât, pentru orice $x_1, \ldots, x_k \in \mathbb{N}$,

$$f(x_1, \ldots, x_k) \leq p(x_1, \ldots, x_k).$$

Corolar

Funcția $n \mapsto 2^n$ nu este $\lambda_{\to}$-reprezentabilă.

Faptul că avem funcții calculabile care nu sunt $\lambda_{\to}$-reprezentabile nu provine dintr-o slăbiciune în definirea λ-calculului cu tipuri, ci este un fapt inerent restricționării la funcții totale, după cum vom vedea imediat.

Limitarea fundamentală

Teoremă

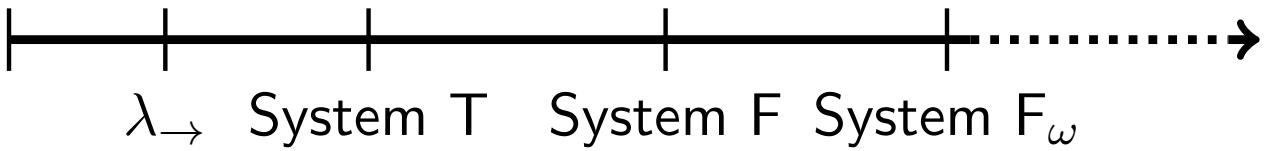
Orice limbaj de programare total este incomplet.

Demonstrație

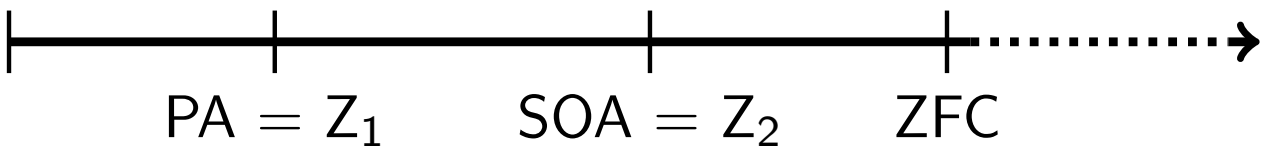
Gândim un limbaj de programare ca pe o funcție calculabilă $f : \mathbb{N}^2 \to \mathbb{N}$, unde cele două argumente ale lui f reprezintă un program și un input al său, iar f calculează output-ul programului relativ la acel input. Presupunem că el ar fi complet, în sensul că, pentru orice funcție calculabilă $h : \mathbb{N} \to \mathbb{N}$, ea este „$f$-reprezentabilă", i.e. există $n \in \mathbb{N}$ („programul" corespunzător) astfel încât, pentru orice $p \in \mathbb{N}$ („pentru orice input"), $f(n, p) = h(p)$. Încercăm să ajungem la o contradicție. Luăm $h : \mathbb{N} \to \mathbb{N}$, definită, pentru orice p, prin $h(p) := f(p, p) + 1$. Clar, h este calculabilă. Deci există n astfel încât, pentru orice p, $f(n, p) = h(p)$. Luând $p := n$, avem $f(n, n) = h(n) = f(n, n) + 1$, ceea ce este o contradicție.

Ierarhia Gödel

Așadar, nu putem avea un singur limbaj de programare total care să captureze toate funcțiile calculabile totale, ci o **ierarhie** de asemenea limbaje:



Aceasta seamănă, oarecum, cu ierarhia Gödel a sistemelor logice în care este fundamentată matematica (rezultatul limitativ, în acest caz, este prima teoremă de incompletitudine a lui Gödel):



Această asemănare nu este întâmplătoare.

Corespondența

Mai precis, există o legătură între sistemele în care este fundamentată matematica și limbajele de programare totale – avem în general, teoreme de genul următor: dacă într-un sistem S putem demonstra o propoziție de forma $\forall x \exists y \varphi(x, y)$ (unde φ este o formulă fără cuantificatori), atunci există un termen („program") t într-un limbaj total corespunzător T_S astfel încât este adevărat că, pentru orice $n \in \mathbb{N}$, $\varphi(n, t(n))$.

Această corespondență reprezintă unul dintre obiectele principale de studiu ale **teoriei demonstrațiilor** (**proof theory**).