Fundamentele limbajelor de programare

C01

Denisa Diaconescu
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Organizare

Instructori

- Curs
  - Seria 23: Traian Șerbănuță
  - Seria 24: Denisa Diaconescu
  - Seria 25: Traian Șerbănuță
- Laborator
  - 231: Horațiu Cheval
  - 232: Horațiu Cheval/Bogdan Macovei
  - 233: Andrei Văcaru
  - 234: Horațiu Cheval/Bogdan Macovei
  - 241: Natalia Ozunu
  - 242: Bogdan Macovei
  - 243: Bogdan Macovei
  - 244: Bogdan Macovei
  - 251: Mihai Calancea
  - 252: Andrei Burdușa

Resurse

- Moodle
- Teams
  https://tinyurl.com/FLP2023-Teams
- Suporturile de curs si laborator
  https://tinyurl.com/FLP2023-Materials

Prezență

Prezența la curs sau la laboratoare nu este obligatorie,
dar extrem de încurajată.

Evaluare

Notare

- Nota finală: 1 (oficiu) + nota laborator + parțial + examen
- Restanță: 1 (oficiu) + examen
  (nota de la laborator si parțialul nu se iau în calcul la restanță)

Condiție de promovabilitate

- cel puțin 5 > 4.99

Nota laborator

- valorează 2 puncte din nota finală
- se notează activitatea din cadrul laboratorului

Examen parțial

- valorează 3 puncte din nota finală
- durata 30 min
- în săptămâna 7, în cadrul cursului
- nu este obligatoriu și nu se poate reface
- întrebări grilă asemănatoare cu cele din quiz-urile de la curs
- materiale ajutătoare: suporturile de curs și de laborator

Examen final

- valorează 4 puncte din nota finală
- durata 1 oră
- în sesiune, fizic
- acoperă toată materia
- exerciții asemănătoare cu exemplele de la curs (nu grile)
- materiale ajutătoare: suporturile de curs și de laborator

Imagine de ansamblu asupra materiei

Curs

- Partea I
  - Lambda calcul
  - Deducție naturală
  - Corespondența Curry-Howard
- Partea II
  - Puncte fixe/recursivitate
  - Semantica limbajelor de programare
  - Elemente de programare logică*

Laborator

- Limbajul suport: Haskell
- Parsere
- Type-checking
- Implementarea unor semantici de limbaje

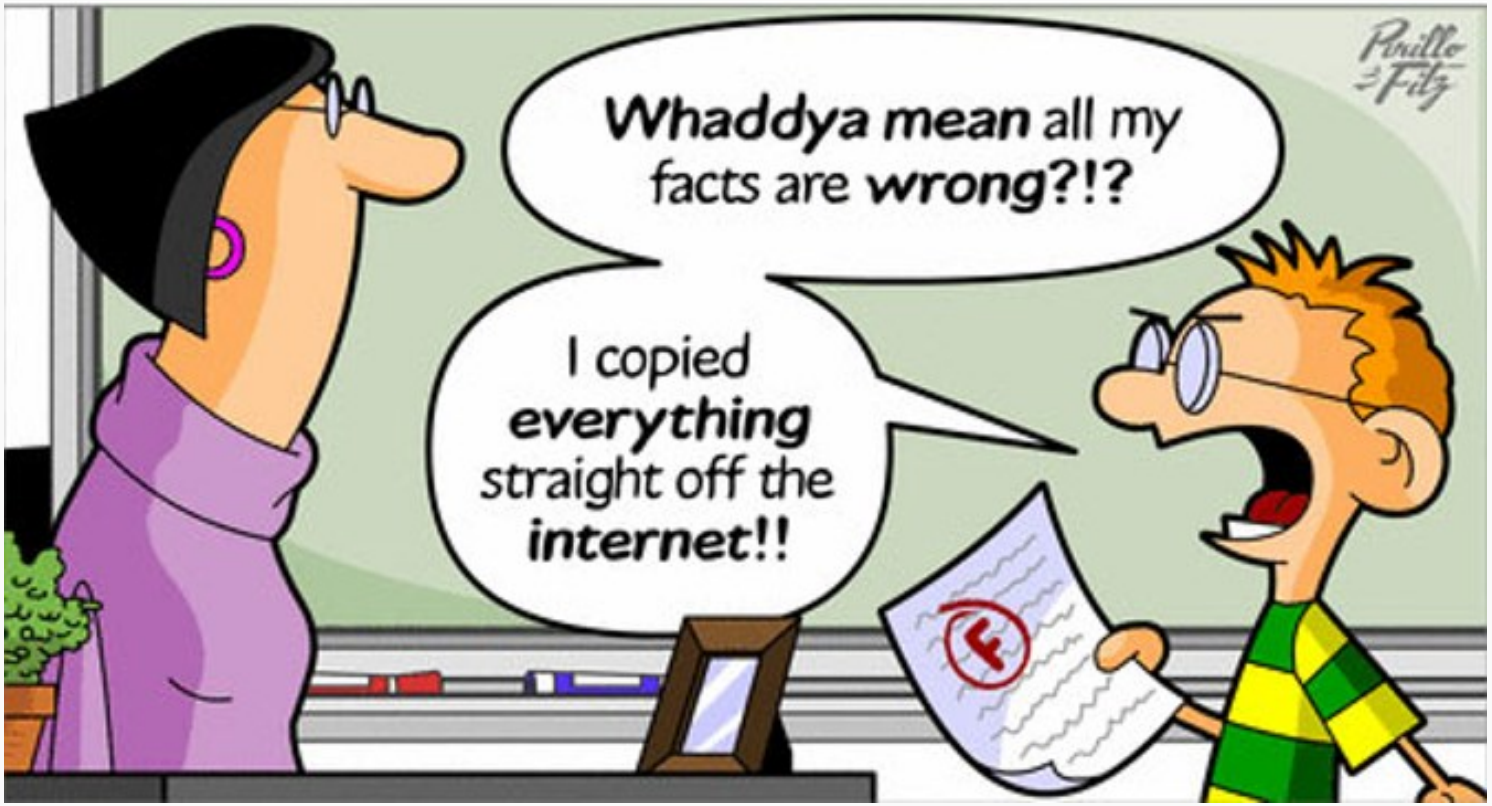
Bibliografie

- H. Barendregt, E. Barendsen, **Introduction to Lambda Calculus**, 2000.
- R. Nederpelt, H. Geuvers , **Type Theory and Formal Proof**. Cambridge University Press, 2014.
- B.C. Pierce, **Types and programming languages**. MIT Press, 2002
- P. Selinger,**Lecture Notes on the Lambda Calculus**. Dep. of Mathematics and Statistics, Dalhousie University, Canada.
- P. Blackburn, J. Bos, and K. Striegnitz, **Learn Prolog Now!** (Texts in Computing, Vol. 7),College Publications, 2006
- M. Huth, M. Ryan, **Logic in Computer Science (Modelling and Reasoning about Systems)**, Cambridge University Press, 2004.
- J. Lloyd. **Foundations of Logic Programming**, second edition. Springer, 1987.

La acest curs vom folosi destul de mult literele grecești

Αα	Ββ	Γγ	Δδ	Εε	Ζζ
ALPHA [a] *ἄλφα*	**BETA** [b] *βῆτα*	**GAMMA** [g] *γάμμα*	**DELTA** [d] *δέλτα*	**EPSILON** [e] *ἒ ψιλόν*	**ZETA** [dz] *ζῆτα*
Ηη	Θθ	Ιι	Κκ	Λλ	Μμ
ETA [ɛː] *ἦτα*	**THETA** [tʰ] *θῆτα*	**IOTA** [i] *ἰῶτα*	**KAPPA** [k] *κάππα*	**LAMBDA** [l] *λάμβδα*	**MU** [m] *μῦ*
Νν	Ξξ	Οο	Ππ	Ρρ	Σσς
NU [n] *νῦ*	**XI** [ks] *ξεῖ*	**OMICRON** [o] *ὂ μικρόν*	**PI** [p] *πεῖ*	**RHO** [r] *ῥῶ*	**SIGMA** [s] *σῖγμα*
Ττ	Υυ	Φφ	Χχ	Ψψ	Ωω
TAU [t] *ταῦ*	**UPSILON** [ʉ] *ὗ ψιλόν*	**PHI** [pʰ] *φεῖ*	**CHI** [kʰ] *χεῖ*	**PSI** [ps] *ψεῖ*	**OMEGA** [ɔː] *ὦ μέγα*

Nu trișați, cereți-ne ajutorul!



Ce și de ce lambda calcul?

Ce este o funcție în matematică?

- In matematica modernă, avem "funcții prin grafice":
  - orice funcție f are un domeniu X și un codomeniu Y fixate, și
  - orice funcție $f : X \to Y$ este o mulțime de perechi $f \subseteq X \times Y$ a.î. pentru orice $x \in X$, există exact un $y \in Y$ astfel încât $(x, y) \in f$.

- Acesta este un punct de vedere *extensional*, singurul lucru pe care îl putem observa despre funcție este cum duce intrările în ieșiri.

- Două funcții $f, g : X \to Y$ sunt considerate ca fiind *extensional* egale dacă pentru aceeași intrare obțin aceeași ieșire,

$$f(x) = g(x), \text{ pentru orice } x \in X.$$

Ce este o funcție în matematică?

- Înainte de secolul 20, funcțiile erau privite ca "reguli/formule".
- A defini o funcție înseamnă să dai o regulă/formulă pentru a o calcula. De exemplu,
$$f(x) = x^2 - 1.$$
- Doua funcții sunt *intensional* egale dacă sunt definite de aceeași formulă. De exemplu, este f de mai sus intensional egala cu g de mai jos?
$$g(x) = (x - 1)(x + 1).$$
- Daca privim o funcție ca o formulă, nu este mereu necesar să știm domeniul și codomeniul ei. De exemplu, funcția identitate
$$h(x) = x$$
poate fi privită ca o funcție $h : X \to X$, pentru orice mulțime X.

Extensional vs. intensional

- Paradigma "funcții prin grafice" este foarte elegantă și definește o clasă mai largă de funcții, deoarece cuprinde și funcții care nu pot fi definite prin formule.
- Paradigma "funcții ca formule" este utilă de multe ori în informatică. De exemplu, putem privi un program ca o funcție de la intrări la ieșiri. De cele mai multe ori, nu ne interesează doar cum sunt duse intrările în ieșiri, ci și cum o putem implementa, cum a fost calculată ieșirea, diverse informații suplimentare etc.
  - Cât a durat să o calculăm?
  - Câtă memorie a folosit?
  - Cu cine a comunicat?

O paranteză: expresii aritmetice

- Expresiile aritmetice sunt construite din
  - variabile ($x, y, z, \ldots$)
  - numere ($1, 2, 3, \ldots$)
  - operatori ("$+$", "$-$", "$\times$" etc)

- Gândim o expresie de forma $x + y$ ca rezultatul adunării lui x cu y, nu ca instrucțiunea/declarația de a aduna x cu y.

- Expresiile aritmetice pot fi combinate, fără a menționa în mod explicit rezultatele intermediare. De exemplu, scriem

$$A = (x + y) \times z^2$$

  în loc de

$$\text{fie } w = x + y, \text{ apoi fie } u = z^2, \text{ apoi fie } A = w \times u.$$

Lambda calcul

- Lambda calculul este o teorie a funcțiilor ca formule.
- Este un sistem care permite manipularea funcțiilor ca expresii.
  Extindem intuiția de la expresii aritmetice pentru funcții.
- De exemplu, dacă în mod normal am scrie

  $$\text{Fie } f \text{ funcția } x \mapsto x^2. \text{ Atunci } A = f(5),$$

  în lambda calcul scriem doar

  $$A = (\lambda x.x^2)(5).$$

- Expresia $\lambda x.x^2$ reprezintă funcția care duce x în x^2
  (nu instrucțiunea/declarația că x este dus în x^2).
- Variabila x este locală/legată în termenul $\lambda x.x^2$
  De aceea, nu contează dacă am fi scris $\lambda y.y^2$

Funcții de nivel înalt

- Lambda calculul ne permite să lucrăm ușor cu funcții de nivel înalt (funcții ale căror intrări/ieșiri sunt tot funcții).
- De exemplu, operația $f \circ f$ este exprimată în lambda calcul prin

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

iar operația $f \mapsto f \circ f$ prin

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

- Evaluarea funcțiilor de nivel înalt poate deveni complexă. De exemplu, expresia

$$((\lambda f.\lambda x.f(f(x)))(\lambda y.y^2))(5)$$

se evalueaza la 625.

Lambda calcul fără tipuri vs cu tipuri

Cateva exemple:

- Funcția identitate $f = \lambda x.x$ are tipul $X \to X$.
  - X poate să fie orice multime
  - contează doar ca domeniul și codomeniul să coincidă
- Funcția $g = \lambda f.\lambda x.f(f(x))$ are tipul $(X \to X) \to (X \to X)$
  - g duce orice funcție $f : X \to X$ intr-o funcție $g(f) : X \to X$

Lambda calcul fără tipuri vs cu tipuri

Permițând flexibilitate în alegerea domeniilor și a codomeniilor, putem manipula funcții în moduri surprinzătoare. De exemplu,

- Pentru funcția identitate $f = \lambda x.x$ avem $f(x) = x$, pentru orice x. În particular, putem lua $x = f$ si obținem
$$f(f) \simeq (\lambda x.x)(\lambda x.x) \simeq \lambda x.x \simeq f.$$

- Combinatorul $\omega = \lambda x.xx$ care pentru un x reprezintă funcția care aplică x lui x
$$\omega(\lambda y.y) \simeq (\lambda x.xx)(\lambda y.y) \simeq (\lambda y.y)(\lambda y.y) \simeq (\lambda y.y)$$
Ce reprezintă $\omega(\omega)$?

Lambda calcul fără tipuri vs cu tipuri

Permițând flexibilitate în alegerea domeniilor și a codomeniilor, putem manipula funcții în moduri surprinzătoare. De exemplu,

- Pentru funcția identitate $f = \lambda x.x$ avem $f(x) = x$, pentru orice x. În particular, putem lua $x = f$ si obținem

$$f(f) \simeq (\lambda x.x)(\lambda x.x) \simeq \lambda x.x \simeq f.$$

- Combinatorul $\omega = \lambda x.xx$ care pentru un x reprezintă funcția care aplică x lui x

$$\omega(\lambda y.y) \simeq (\lambda x.xx)(\lambda y.y) \simeq (\lambda y.y)(\lambda y.y) \simeq (\lambda y.y)$$

Ce reprezintă $\omega(\omega)$?

$$\omega(\omega) \simeq (\lambda x.xx)(\lambda x.xx) \simeq (\lambda x.xx)(\lambda x.xx)$$

Lambda calcul

- Lambda calcul fără tipuri
  - nu specificăm tipul niciunei expresii
  - nu specificăm domeniul/codomeniul funcțiilor
  - flexibilitate maximă, dar riscant deoarece putem ajunge în situații în care încercăm să aplicăm o funcție unui argument pe care nu îl poate procesa
- Lambda calcul cu tipuri simple
  - specificăm mereu tipul oricărei expresii
  - nu putem aplica funcții unui argument care are alt tip față de domeniul funcției
  - expresiile de forma $f(f)$ sunt eliminate, chiar dacă f este funcția identitate
- Lambda calcul cu tipuri polimorfice
  - o situație intermediară între cele două de mai sus
  - de exemplu, putem specifica că o expresie are tipul $X \to X$, dar fără a specifica cine este X

Calculabilitate

- Una din marile întrebări din anii 1930:

  Ce înseamnă că o funcție $f : \mathbb{N} \to \mathbb{N}$ este calculabilă?

- O definiție informală:

  ar trebui să existe o "metodă pe foaie" *(pen-and-paper)*
  care să îi permită unei persoane cu experiență
  să calculeze $f(n)$, pentru orice n.

- Conceptul de metodă "pen-and-paper" nu este ușor de formalizat

Definiții pentru Calculabilitate

1. Turing – a definit un calculator ideal numit mașina Turing și a postulat că o funcție este calculabilă ddacă poate fi calculată de o astfel de mașină.

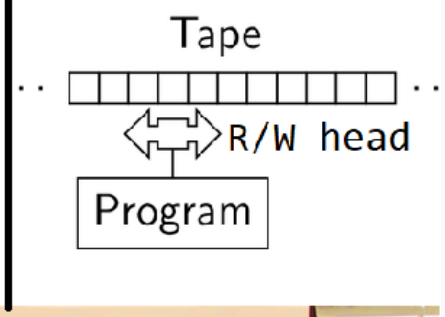
2. Gödel – a definit clasa funcțiilor recursive și a postulat că o funcție este calculabilă ddacă este o funcție recursivă.

3. Church – a definit un limbaj de programare ideal numit lambda calcul și a postulat că o funcție este calculabilă ddacă poate fi scrisă ca un lambda termen.

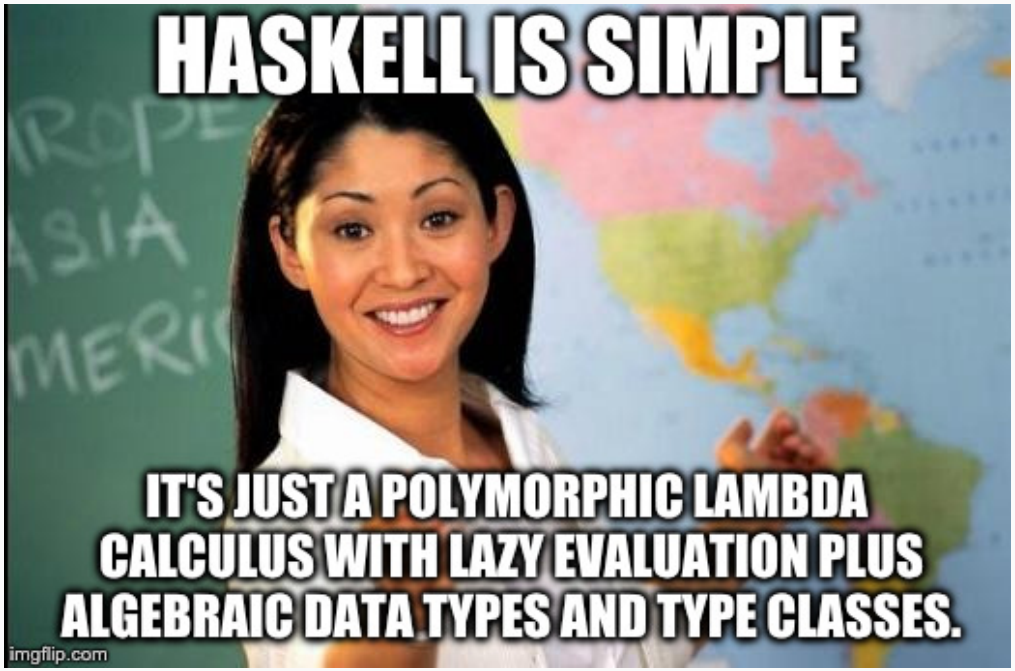
Teza Church-Turing

- Church, Kleene, Rosser și Turing au arătat că cele trei modele de calculabilitate sunt echivalente (definesc aceeași clasă de funcții calculabile).
- Dacă sunt sau nu echivalente cu noțiunea "intuitivă" de calculabilitate este o întrebare la care nu se poate răspunde, deoarece nu avem o definiție pentru "calculabilitate intuitivă".
- Faptul că cele trei modele coincid cu noțiunea intuitivă de calculabilitate se numește teza Church-Turing.

Syntax	Name
x	Variable
(λx.M)	Abstraction
(M N)	Application



Lambda calcul în informatică



- Lambda calcul este un limbaj de programare ideal.
- Probabil cel mai simplu limbaj de programare Turing complet.
- Toate limbajele de programare funcțională sunt extensii ale lambda calculului cu diferite caracteristici (tipuri de date, efecte laterale etc)

Proofs-as-programs

- Ce este o demonstrație?
  - Logica clasică: plecând de la niște presupuneri, este suficient să ajungi la o contradicție
  - Logica constructivistă: pentru a arata ca un obiect există, trebuie să îl construim explicit.

- Legătura dintre lambda calcul și logica constructivistă este dată de paradigma *proofs-as-programs*.
  - o demonstrație trebuie să fie o "construcție", un program
  - lambda calculul este o notație pentru astfel de programe

Quiz time!

https://tinyurl.com/C01-Quiz1

Pe săptămâna viitoare!

Fundamentele limbajelor de programare

C02

Denisa Diaconescu
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

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

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

Mulțimea lambda termenilor este dată de următoarea formă BNF:

lambda termen	=	variabilă
	|	aplicare
	|	abstractizare

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

Example

- 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
	|	aplicare
	|	abstractizare

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

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

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

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

Lambda termeni - definiție alternativă

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

Fie A un alfabet format din elementele din V, și simbolurile speciale $"("$, $")"$, $"\lambda"$ si $"."$

Fie A^* mulțimea tuturor cuvintelor finite pentru alfabetul A.

Mulțimea lambda termenilor este cea mai mică submulțime $\Lambda \subseteq A^*$ astfel încât:

[Variabilă] $V \subseteq \Lambda$

[Aplicare] dacă $M, N \in \Lambda$ atunci $(M\,N) \in \Lambda$

[Abstractizare] dacă $x \in V$ și $M \in \Lambda$ atunci $(\lambda x.M) \in \Lambda$

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. $\lambda x.x\,\lambda y.y$

Exerciții

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

1. $(\lambda x.(\lambda y.(\lambda z.((x\,z)(y\,z)))))$ Corect: $\lambda xyz.x\,z\,(y\,z)$
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. $\lambda x.x\,\lambda y.y$

Exerciții

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

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

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

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

Exerciții

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

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

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

1. $x\,x\,x\,x$ Corect: $(((x\,x)\,x)\,x)$
2. $\lambda x.x\,\lambda y.y$

Exerciții

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

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

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

1. $x\,x\,x\,x$ Corect: $(((x\,x)\,x)\,x)$
2. $\lambda x.x\,\lambda y.y$ Corect: $(\lambda x.(x\,(\lambda y.y)))$

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:

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

Exemplu de definiție recursivă pe termeni. Adică în definiția lui $FV(M)$ am presupus că am definit deja $FV(N)$ pentru toți subtermenii lui M.

Example

- $FV(\lambda x.x\,y) = FV(x\,y) \setminus \{x\} = (FV(x) \cup FV(y)) \setminus \{x\}$
  $= (\{x\} \cup \{y\}) \setminus \{x\} = \{y\}$
- $FV(x\,\lambda x.x\,y) = \{x, y\}$

Redenumire de variabile

Ce înseamnă să redenumim o variabilă într-un termen?

Dacă x, y sunt variabile și M este un termen, $M\langle y/x\rangle$ este rezultatul obținut după redenumirea lui x cu y în M.

$$
\begin{aligned}
x\langle y/x\rangle &\equiv y, \\
z\langle y/x\rangle &\equiv z, && \text{dacă } x \neq z \\
(M\,N)\langle y/x\rangle &\equiv (M\langle y/x\rangle)\,(N\langle y/x\rangle) \\
(\lambda x.M)\langle y/x\rangle &\equiv \lambda y.(M\langle y/x\rangle) \\
(\lambda z.M)\langle y/x\rangle &\equiv \lambda z.(M\langle y/x\rangle), && \text{dacă } x \neq z
\end{aligned}
$$

Observați că acest tip de redenumire înlocuiește toate aparițiile lui x cu y, indiferent dacă este liberă, legată, sau de legare.

Se folosește doar în cazuri în care y nu apare deja în M.

α-echivalență

Ce înseamnă că doi termeni sunt egali,
modulo redenumire de variabile legate?

Definim α-echivalența ca fiind cea mai mică relație de congruență $=_\alpha$ pe mulțimea lambda termenilor, astfel încât pentru orice termen M și orice variabilă y care nu apare în M, avem

$$\lambda x.M \;=_\alpha\; \lambda y.(M\langle y/x\rangle)$$

α-echivalență

α-echivalența $=_\alpha$ este cea mai mică relație pe lambda termeni care satisface regulile:

$$(refl)\quad \frac{}{M = M} \qquad (cong)\quad \frac{M = M' \qquad N = N'}{MN = M'N'}$$

$$(symm)\quad \frac{M = N}{N = M} \qquad (\xi)\quad \frac{M = M'}{\lambda x.M = \lambda x.M'}$$

$$(trans)\quad \frac{M = N \qquad N = P}{M = P} \qquad (\alpha)\quad \frac{y \notin M}{\lambda x.M = \lambda y.(M\{y/x\})}$$

Convenția Barendregt:

variabilele legate sunt redenumite pentru a fi distincte.

Substituții

Vrem să substituim variabile cu lambda termeni.

$M[N/x]$ este rezultatul obținut după înlocuirea lui x cu N în M.

Trebuie să fim atenți la următoarele cazuri:

1. Vrem să înlocuim doar variabile libere.

Numele variabilelor legate este considerat imaterial, și nu ar trebui să afecteze rezultatul substituției.
De exemplu, $x\,(\lambda xy.x)[N/x]$ ar trebui să fie $N\,(\lambda xy.x)$,
nu $N\,(\lambda xy.N)$ sau $N\,(\lambda Ny.N)$.

Substituții

2. Nu vrem să legăm variabile libere neintenționat.

De exemplu, fie $M \equiv \lambda x.y\ x$ și $N \equiv \lambda z.x\ z$.

Variabila x este legată în M și liberă în N.

Ce ar trebui să obținem dacă am substitui y cu N în M?
Naiv, ne-am gândi la

$$M[N/y] = (\lambda x.y\ x)[N/y] = \lambda x.N\ x = \lambda x.(\lambda z.x\ z)\ x.$$

Totuși, nu este ceea ce am vrea să obținem, deoarece x este liber în N, iar în timpul "substituției" a devenit legată.

Trebuie să luăm în calcul că x-ul legat din M nu este x-ul liber din N, și de aceea redenumim variabilele legate înainte de substituție.

$$M[N/y] = (\lambda x'.y\ x')[N/y] = \lambda x'.N\ x' = \lambda x'.(\lambda z.x\ z)\ x'.$$

Substituții

Substituția aparițiilor libere ale lui x cu N în M, notată cu $M[N/x]$, este definită prin:

$$
\begin{array}{lcll}
x[N/x] & \equiv & N & \\
y[N/x] & \equiv & \mathsf{y} & \text{dacă } x \neq y \\
(M\,P)[N/x] & \equiv & (M[N/x])\,(P[N/x]) & \\
(\lambda x.M)[N/x] & \equiv & \lambda x.M & \\
(\lambda y.M)[N/x] & \equiv & \lambda y.(M[N/x]) & \text{dacă } x \neq y \text{ și } y \notin FV(N) \\
(\lambda y.M)[N/x] & \equiv & \lambda y'.(M\langle y'/y\rangle[N/x]) & \text{dacă } x \neq y,\ y \in FV(N) \\
 & & & \text{și } y' \text{ variabilă nouă}
\end{array}
$$

Deaorece nu specificăm ce variabilă nouă alegem,
spunem că substituția este bine-definită modulo α-echivalențe.

Exerciții

Exercițiu. Calculați următoarele substituții:

1. $(\lambda z.x)[y/x]$
2. $(\lambda y.x)[y/x]$
3. $(\lambda y.x)[(\lambda z.z\, w)/x]$

Exerciții

Exercițiu. Calculați următoarele substituții:

1. $(\lambda z.x)[y/x]$ Corect: $\lambda z.y$
2. $(\lambda y.x)[y/x]$
3. $(\lambda y.x)[(\lambda z.z\, w)/x]$

Exerciții

Exercițiu. Calculați următoarele substituții:

1. $(\lambda z.x)[y/x]$ — Corect: $\lambda z.y$
2. $(\lambda y.x)[y/x]$ — Corect: $\lambda y'.y$, Greșit: $\lambda y.y$
3. $(\lambda y.x)[(\lambda z.z\ w)/x]$

Exerciții

Exercițiu. Calculați următoarele substituții:

1. $(\lambda z.x)[y/x]$ — Corect: $\lambda z.y$
2. $(\lambda y.x)[y/x]$ — Corect: $\lambda y'.y$, Greșit: $\lambda y.y$
3. $(\lambda y.x)[(\lambda z.z\ w)/x]$ — Corect: $\lambda yz.zw$

Quiz time!

https://tinyurl.com/C02-Quiz1

Pe săptămâna viitoare!

Fundamentele limbajelor de programare

C03

Denisa Diaconescu
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Lambda calcul - β-reducții

β-reducții

Convenție. Spunem că doi termeni sunt egali, notat $M = N$, dacă sunt α-echivalenți.

- β-reducție = procesul de a evalua lambda termeni prin "pasarea de argumente funcțiilor"
- β-redex = un termen de forma $(\lambda x.M)\ N$
- redusul unui redex $(\lambda x.M)\ N$ este $M[N/x]$
- reducem lambda termeni prin găsirea unui subtermen care este redex, și apoi înlocuirea acelui redex cu redusul său
- repetăm acest proces de câte ori putem, până nu mai sunt redex-uri
- formă normală = un lambda termen fără redex-uri

β-reducții

Un pas de β-reducție $\rightarrow_\beta$ este cea mai mică relație pe lambda termeni care satisface regulile:

$$(\beta) \quad \frac{}{(\lambda x.M)N \rightarrow_\beta M[N/x]}$$

$$(cong_1) \quad \frac{M \rightarrow_\beta M'}{MN \rightarrow_\beta M'N}$$

$$(cong_2) \quad \frac{N \rightarrow_\beta N'}{MN \rightarrow_\beta MN'}$$

$$(\xi) \quad \frac{M \rightarrow_\beta M'}{\lambda x.M \rightarrow_\beta \lambda x.M'}$$

β-reducții

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

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

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

β-reducții

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

$$
\begin{aligned}
\underline{(\lambda x.y)\,((\lambda z.zz)\,(\lambda w.w))} &\to_\beta y[(\lambda z.zz)\,(\lambda w.w)/x] \\
&\equiv y
\end{aligned}
$$

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).

$$\begin{aligned} \underline{(\lambda x.x\ x)\,(\lambda x.x\ x)} \quad &\rightarrow_\beta \quad (\lambda x.x\ x)\,(\lambda x.x\ x) \\ &\rightarrow_\beta \quad \ldots \end{aligned}$$

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

β-formă normală

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

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

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

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.

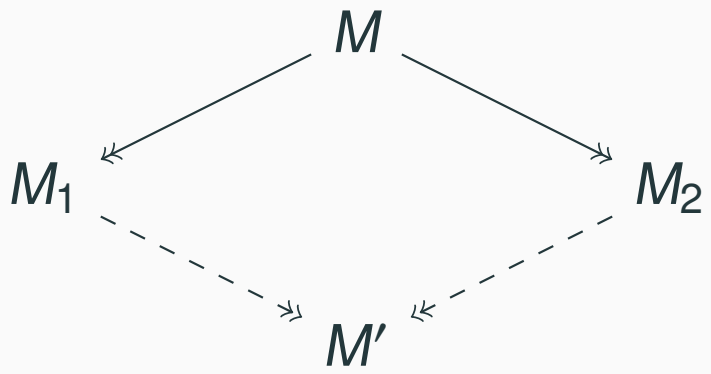
Example

$(\lambda x.y)\,((\lambda z.zz)\,(\lambda w.w))$ 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 $M \twoheadrightarrow_\beta M_1$ și $M \twoheadrightarrow_\beta M_2$ atunci există M' astfel încât $M_1 \twoheadrightarrow_\beta M'$ și $M_2 \twoheadrightarrow_\beta M'$.



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

Exerciții

Exercițiu. Verificați dacă termenii de mai jos pot fi aduși la o β-formă normală:

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

Exerciții

Exercițiu. Verificați dacă termenii de mai jos pot fi aduși la o β-formă normală:

1. $(\lambda x.x)\, M$ — Corect: M
2. $(\lambda xy.x)\, M\, N$ — Corect: M
3. $(\lambda x.x\ x)\, (\lambda y.y\ y\ y)$ — Corect: $(\lambda y.y\ y\ y)\, (\lambda y.y\ y\ y)\, (\lambda y.y\ y\ y) \ldots$

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.

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

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.

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

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 $(\lambda x.M)\ N \twoheadrightarrow_\beta M[N/x]$ 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 lenește *(lazy evaluation)*: argumentele funcțiilor sunt evaluate doar când sunt necesare.

CBN vs CBV

Example

Considerăm 3 și *succ* primitive.

Strategia CBV:

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

Strategia CBN:

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

Quiz time!

https://tinyurl.com/C03-Quiz1

Pe săptămâna viitoare!

Fundamentele limbajelor de programare

C04

Denisa Diaconescu
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Expresivitatea λ-calculului

Expresivitatea λ-calculului

Deși lambda calculul constă doar în λ-termeni, putem reprezenta și manipula tipuri de date comune.

Vom vedea cum putem reprezenta:

- valori booleene
- numere naturale

Booleeni

Booleeni

Vrem să definim λ-termeni care să reprezinte constantele booleene.

Sunt mai multe modalități, una dintre ele fiind:

- $\mathbf{T} \triangleq \lambda xy.x$ (dintre cele două alternative o alege pe prima)
- $\mathbf{F} \triangleq \lambda xy.y$ (dintre cele două alternative o alege pe a doua)

Booleeni

$$\mathbf{T} \triangleq \lambda xy.x \qquad \mathbf{F} \triangleq \lambda xy.y$$

Acum am vrea să definim un test condiționat **if**.

Am vrea ca **if** să ia trei argumente b, t, f, unde b este o valoare booleană, iar t, f sunt λ-termeni oarecare.

Funcția ar trebui să returneze t dacă $b = \mathit{true}$ și f dacă $b = \mathit{false}$

$$\mathbf{if} = \lambda btf. \begin{cases} t, & \text{if } b = \mathit{true}, \\ f, & \text{if } b = \mathit{false}. \end{cases}$$

Deoarece $\mathbf{T}\ t\ f \twoheadrightarrow_\beta t$ și $\mathbf{F}\ t\ f \twoheadrightarrow_\beta f$, **if** trebuie doar să aplice argumentul său boolean celorlalte argumente:

$$\mathbf{if} \triangleq \lambda btf.b\ t\ f$$

Booleeni

$$\mathbf{T} \triangleq \lambda xy.x \qquad \mathbf{F} \triangleq \lambda xy.y \qquad \mathbf{if} \triangleq \lambda btf.b\ t\ f$$

Celelalte operații booleene pot fi definite folosind **if**:

$$\begin{aligned} \mathbf{and} &\triangleq \lambda b_1 b_2.\mathbf{if}\ b_1\ b_2\ \mathbf{F} \\ \mathbf{or} &\triangleq \lambda b_1 b_2.\mathbf{if}\ b_1\ \mathbf{T}\ b_2 \\ \mathbf{not} &\triangleq \lambda b_1.\mathbf{if}\ b_1\ \mathbf{F}\ \mathbf{T} \end{aligned}$$

Observați că aceste operații lucrează corect doar dacă primesc ca argumente valori booleene.

Nu există nicio garanție să se comporte rezonabil pe orice alți λ-termeni.

Folosind lambda calcul fără tipuri, avem *garbage in, garbage out*.

Codările nu sunt unice. De exemplu, pentru **and** am fi putut folosi codarile $\lambda b_1 b_2.b_2\ b_1\ b_2$ sau $\lambda b_1 b_2.b_1\ b_2\ \mathbf{F}$.

Booleeni

$$\mathbf{T} \triangleq \lambda xy.x \qquad \mathbf{F} \triangleq \lambda xy.y \qquad \mathbf{if} \triangleq \lambda btf.b\ t\ f$$

$$\begin{aligned} \mathbf{and} &\triangleq \lambda b_1 b_2.\mathbf{if}\ b_1\ b_2\ \mathbf{F} \\ \mathbf{or} &\triangleq \lambda b_1 b_2.\mathbf{if}\ b_1\ \mathbf{T}\ b_2 \\ \mathbf{not} &\triangleq \lambda b_1.\mathbf{if}\ b_1\ \mathbf{F}\ \mathbf{T} \end{aligned}$$

Exercițiu. Aduceți la o formă normală următorii termenii:

- **and TF**
- **or FT**
- **not T**

Booleeni

$$\mathbf{T} \triangleq \lambda xy.x \qquad \mathbf{F} \triangleq \lambda xy.y \qquad \mathbf{if} \triangleq \lambda btf.b\ t\ f$$

$$\begin{aligned} \mathbf{and} &\triangleq \lambda b_1 b_2.\mathbf{if}\ b_1\ b_2\ \mathbf{F} \\ \mathbf{or} &\triangleq \lambda b_1 b_2.\mathbf{if}\ b_1\ \mathbf{T}\ b_2 \\ \mathbf{not} &\triangleq \lambda b_1.\mathbf{if}\ b_1\ \mathbf{F}\ \mathbf{T} \end{aligned}$$

Soluții:

$$\begin{aligned} \mathbf{and}\,\mathbf{T}\,\mathbf{F} &= (\lambda b_1 b_2.\mathbf{if}\ b_1\ b_2\ \mathbf{F})\,\mathbf{T}\,\mathbf{F} \twoheadrightarrow_\beta \mathbf{if}\,\mathbf{T}\,\mathbf{F}\,\mathbf{F} = (\lambda btf.b\ t\ f)\,\mathbf{T}\,\mathbf{F}\,\mathbf{F} \\ &\twoheadrightarrow_\beta \mathbf{T}\,\mathbf{F}\,\mathbf{F} = (\lambda xy.x)\,\mathbf{F}\,\mathbf{F} \twoheadrightarrow_\beta \mathbf{F} \end{aligned}$$

$$\begin{aligned} \mathbf{or}\,\mathbf{F}\,\mathbf{T} &= (\lambda b_1 b_2.\mathbf{if}\ b_1\ \mathbf{T}\ b_2)\,\mathbf{F}\,\mathbf{T} \twoheadrightarrow_\beta \mathbf{if}\,\mathbf{F}\,\mathbf{T}\,\mathbf{T} = (\lambda btf.b\ t\ f)\,\mathbf{F}\,\mathbf{T}\,\mathbf{T} \\ &\twoheadrightarrow_\beta \mathbf{F}\,\mathbf{T}\,\mathbf{T} = (\lambda xy.y)\,\mathbf{T}\,\mathbf{T} \twoheadrightarrow_\beta \mathbf{T} \end{aligned}$$

$$\begin{aligned} \mathbf{not}\,\mathbf{T} &= (\lambda b_1.\mathbf{if}\ b_1\ \mathbf{F}\,\mathbf{T})\,\mathbf{T} \twoheadrightarrow_\beta \mathbf{if}\,\mathbf{T}\,\mathbf{F}\,\mathbf{T} = (\lambda btf.b\ t\ f)\,\mathbf{T}\,\mathbf{F}\,\mathbf{T} \\ &\twoheadrightarrow_\beta \mathbf{T}\,\mathbf{F}\,\mathbf{T} = (\lambda xy.x)\,\mathbf{F}\,\mathbf{T} \twoheadrightarrow_\beta \mathbf{F} \end{aligned}$$

Numere naturale

Numere naturale

Vom reprezenta numerele naturale $\mathbb{N}$ folosind numeralii Church.

Numeralul Church pentru numărul $n \in \mathbb{N}$ este notat $\overline{n}$.

Numeralul Church $\overline{n}$ este λ-termenul $\lambda fx.f^n\ x$, unde f^n reprezintă compunerea lui f cu ea însăși de n ori:

$$
\begin{array}{lcll}
\overline{0} & \triangleq & \lambda fx.f^0\ x & = \lambda fx.x \\
\overline{1} & \triangleq & \lambda fx.f^1\ x & = \lambda fx.f\ x \\
\overline{2} & \triangleq & \lambda fx.f^2\ x & = \lambda fx.f\ (f\ x) \\
\overline{3} & \triangleq & \lambda fx.f^3\ x & = \lambda fx.f\ (f\ (f\ x)) \\
 & \vdots & & \\
\overline{n} & \triangleq & \lambda fx.f^n\ x & = \lambda fx.\underbrace{f(f(\ldots(f}_{n}\ x)\ldots))
\end{array}
$$

Numere naturale

$$\overline{n} \triangleq \lambda fx.f^n\ x$$

Acum putem defini funcția succesor prin

$$\mathbf{Succ} \triangleq \lambda nfx.f\ (n\ f\ x)$$

Observați că **Succ** pe argumentul $\overline{n}$ returnează o funcție care primește ca argument o funcție f, îi aplică $\overline{n}$ pentru a obține compunerea de n ori a lui f cu ea însăși, apoi aplică iar f pentru a obține compunerea de $n+1$ ori a lui f cu ea însăși.

$$\begin{array}{rcl}
\mathbf{Succ}\,\overline{n} & = & (\lambda nfx.f\ (n\ f\ x))\,\overline{n} \\
& \twoheadrightarrow_\beta & \lambda fx.f\ (\overline{n}\ f\ x) \\
& \twoheadrightarrow_\beta & \lambda fx.f\ (f^n\ x) \\
& = & \lambda fx.f^{n+1}\ x \\
& = & \overline{n+1}
\end{array}$$

Numere naturale

$$\overline{n} \triangleq \lambda fx.f^n\ x \qquad \textbf{Succ} \triangleq \lambda nfx.f\ (n\ f\ x)$$

Putem face operații aritmetice de bază cu numeralii Church.

Pentru adunare, putem defini

$$\textbf{add} \triangleq \lambda mnfx.m\ f\ (n\ f\ x)$$

Pentru argumentele $\overline{m}$ și $\overline{n}$, obținem:

$$\begin{array}{rcl} \textbf{add}\ \overline{m}\ \overline{n} & = & (\lambda mnfx.m\ f\ (n\ f\ x))\ \overline{m}\ \overline{n} \\ & \twoheadrightarrow_\beta & \lambda fx.\overline{m}\ f\ (\overline{n}\ f\ x) \\ & \twoheadrightarrow_\beta & \lambda fx.f^m\ (f^n\ x) \\ & = & \lambda fx.f^{m+n}\ x \\ & = & \overline{m+n} \end{array}$$

Am folosit compunerea lui f^m cu f^n pentru a obține f^{m+n}.

Numere naturale

$$\overline{n} \triangleq \lambda fx.f^n\, x \qquad \mathbf{Succ} \triangleq \lambda nfx.f\,(n\,f\,x)$$

Putem defini adunarea și ca aplicarea repetată a funcției succesor:

$$\mathbf{add'} \triangleq \lambda mn.m\,\mathbf{Succ}\,n$$

$$\begin{aligned}
\mathbf{add'}\,\overline{m}\,\overline{n} &= (\lambda mn.m\,\mathbf{Succ}\,n)\,\overline{m}\,\overline{n} \\
&\twoheadrightarrow_\beta \overline{m}\,\mathbf{Succ}\,\overline{n} \\
&= (\lambda fx.f^m\,x)\,\mathbf{Succ}\,\overline{n} \\
&\twoheadrightarrow_\beta \mathbf{Succ}^m\,\overline{n} \\
&= \underbrace{\mathbf{Succ}(\mathbf{Succ}(\ldots(\mathbf{Succ}}_{m}\,\overline{n})\ldots)) \\
&\twoheadrightarrow_\beta \underbrace{\mathbf{Succ}(\mathbf{Succ}(\ldots(\mathbf{Succ}}_{m-1}\,\overline{n+1})\ldots)) \\
&\twoheadrightarrow_\beta \overline{m+n}
\end{aligned}$$

Numere naturale

$$\overline{n} \triangleq \lambda fx.f^n\, x \qquad \textbf{Succ} \triangleq \lambda nfx.f\,(n\, f\, x)$$

$$\textbf{add}' \triangleq \lambda mn.m\,\textbf{Succ}\,n$$

Similar înmulțirea este adunare repetată, iar ridicarea la putere este înmulțire repetată:

$$\begin{aligned} \textbf{mul} &\triangleq \lambda mn.m\,(\textbf{add}\,n)\,\overline{0} \\ \textbf{exp} &\triangleq \lambda mn.m\,(\textbf{mul}\,n)\,\overline{1} \end{aligned}$$

Numere naturale

Putem defini o funcție de la numere naturale la booleeni care verifică dacă un număr natural este 0 sau nu

$$\begin{aligned} \textbf{isZero}(0) &= \mathit{true} \\ \textbf{isZero}(n) &= \mathit{false} \qquad \text{dacă } n \neq 0 \end{aligned}$$

O codare în lambda calcul a unei astfel de funcții este

$$\textbf{isZero} \triangleq \lambda nxy.n\,(\lambda z.y)\,x$$

Exercițiu. Verificați afirmația de mai sus.

Putem să definim și codarea **pred** pentru predecesorul unui număr natural. Această codare nu este deloc ușoară și alegem să lucrăm cu ea ca cu o cutie neagră.

Putem exprima mai mult?

Avem văzut codari simple pentru booleeni și numere naturale.

Totuși nu avem o metodă pentru a construi astfel de λ-termeni.

Ne trebuie un mecanism care să ne permită să construim funcții mai complicate din funcții mai simple.

De exemplu, să considerăm funcția factorial

$$\begin{aligned} 0! &= 1 \\ n! &= n \cdot (n-1)!, \qquad \text{dacă } n \neq 0 \end{aligned}$$

Puncte fixe

Puncte fixe

Fie f o funcție. Spunem că x este un punct fix al lui f dacă $f(x) = x$.

În matematică, unele funcții au puncte fixe, altele nu au.

De exemplu, $f(x) = x^2$ are două puncte fixe 0 și 1, dar $f(x) = x + 1$ nu are puncte fixe.

Mai mult, unele funcții au o infinitate de puncte fixe, cum ar fi $f(x) = x$.

β-echivalență

Am notat cu $M \twoheadrightarrow_\beta M'$ faptul că M poate fi β-redus până la M' în 0 sau mai mulți pași de β-reducție.

$\twoheadrightarrow_\beta$ este închiderea reflexivă și tranzitivă a relației $\rightarrow_\beta$.

Notăm cu $M =_\beta M'$ faptul că M poate fi transformat în M' în 0 sau mai mulți pași de β-reducție, transformare în care pașii de reducție pot fi și întorși.

$=_\beta$ este închiderea reflexivă, simetrică și tranzitivă a relației $\rightarrow_\beta$.

De exemplu, avem $(\lambda y.y\ v)\ z =_\beta (\lambda x.z\ x)\ v$ deoarece avem

$$(\lambda y.y\ v)\ z \rightarrow_\beta z\ v \leftarrow_\beta (\lambda x.z\ x)\ v$$

Notăm cu $\leftarrow_\beta$ inversul relației $\rightarrow_\beta$.

Puncte fixe în lambda-calcul

Dacă F și M sunt λ-termeni, spunem că M este un punct fix al lui F dacă $F\,M =_\beta M$.

Thm. În lambda calculul fără tipuri, orice termen are un punct fix.

Puncte fixe în lambda-calcul

Dacă F și M sunt λ-termeni, spunem că M este un punct fix al lui F dacă $F\,M =_\beta M$.

Thm. În lambda calculul fără tipuri, orice termen are un punct fix.

Dem. Vrem să arătăm că pentru orice termen F există un termen M astfel încât $F\,M =_\beta M$.

Fie F un termen. Considerăm $M \triangleq (\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x))$. Avem

$$\begin{array}{rcl} M & = & (\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x)) \\ & \to_\beta & F\,((\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x))) \\ & = & F\,M \end{array}$$

Deci avem $FM =_\beta M$.

Combinatori de punct fix

Combinatorii de puncte fixe sunt termeni închiși care "construiesc" un punct fix pentru un termen arbitrar.

Câteva exemple:

- Combinatorul de punct fix al lui Curry
  $\mathbf{Y} \triangleq \lambda y.(\lambda x.y\,(x\,x))\,(\lambda x.y\,(x\,x))$
  Pentru orice termen F, $\mathbf{Y}F$ este un punct fix al lui F deoarece $\mathbf{Y}F \twoheadrightarrow_\beta F\,(\mathbf{Y}\,F)$.
- Combinatorul de punct fix al lui Turing
  $\Theta \triangleq (\lambda xy.y\,(x\,x\,y))\,(\lambda xy.y\,(x\,x\,y))$
  Pentru orice termen F, ΘF este un punct fix al lui F deoarece $\Theta F \twoheadrightarrow_\beta F\,(\Theta\,F)$.

Rezolvarea de ecuații în lambda calcul

Punctele fixe ne permit să rezolvăm ecuații. A găsi un punct fix pentru f este același lucru cu a rezolva o ecuație de forma

$$x = f(x)$$

Am văzut că în lambda calcul există mereu o soluție pentru astfel de ecuații.

Rezolvarea de ecuații în lambda calcul

Să aplicăm această idee pentru funcția factorial.

Cea mai naturală definiție a funcției factorial este cea recursivă și o putem scrie în lambda calcul prin

$$\textbf{fact}\, n = \textbf{if}\, (\textbf{isZero}\, n)\, (\overline{1})\, (\textbf{mul}\, n\, (\textbf{fact}(\textbf{pred}\, n))\,)$$

În ecuația de mai sus, **fact** apare și în stânga, și în dreapta. Pentru a găsi cine este **fact**, trebuie să rezolvăm o ecuație.

Rezolvarea de ecuații în lambda calcul

Să rezolvăm ecuația de mai sus. Rescriem problema puțin

$$\begin{aligned} \mathbf{fact} &= \lambda n.\mathbf{if}\,(\mathbf{isZero}\,n)\,(\overline{1})\,(\mathbf{mul}\,n\,(\mathbf{fact}(\mathbf{pred}\,n)))\\ \mathbf{fact} &= (\lambda fn.\mathbf{if}\,(\mathbf{isZero}\,n)\,(\overline{1})\,(\mathbf{mul}\,n\,(f(\mathbf{pred}\,n))))\,\mathbf{fact} \end{aligned}$$

Notăm termenul $\lambda fn.\mathbf{if}\,(\mathbf{isZero}\,n)\,(\overline{1})\,(\mathbf{mul}\,n\,(f(\mathbf{pred}\,n)))$ cu F.

Ultima ecuație devine $\mathbf{fact} = F\,\mathbf{fact}$, o ecuație de punct fix.

Am văzut că $\mathbf{Y}\,F$ este un punct fix pentru F (adică $\mathbf{Y}\,F \twoheadrightarrow_\beta F\,(\mathbf{Y}\,F)$), de aceea putem rezolva ecuația de mai sus luând

$$\begin{aligned} \mathbf{fact} &\triangleq \mathbf{Y}\,F\\ \mathbf{fact} &\triangleq \mathbf{Y}\,(\lambda fn.\mathbf{if}\,(\mathbf{isZero}\,n)\,(\overline{1})\,(\mathbf{mul}\,n\,(f(\mathbf{pred}\,n)))) \end{aligned}$$

Observați că **fact** a dispărut din partea dreaptă.

Exercițiu. Evaluați $\mathbf{fact}\,\overline{2}$ ținând cont că $\mathbf{fact} \twoheadrightarrow_\beta F\,\mathbf{fact}$.

Quiz time!

https://tinyurl.com/C04-Quiz1

Pe săptămâna viitoare!

Fundamentele limbajelor de programare

C05

Denisa Diaconescu
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Lambda calcul cu tipuri simple

Probleme cu lambda calculul fără tipuri

Proprietăți negative ale lambda calculului fără tipuri:

- Aplicări de forma $x\ x$ sau $M\ M$ sunt pemise,
  deși sunt contraintuitive.
- Existența formelor normale pentru λ-termeni nu este garantată
  și putem avea "calcule infinite" nedorite
- Orice λ-termen are un punct fix ceea ce nu este în armonie cu
  ceea ce știam despre funcții oarecare

Vrem să eliminăm aceste proprietăți negative, păstrându-le pe cele pozitive.

Proprietățile negative sunt eliminate prin adăugarea de tipuri ceea ce induce restricțiile dorite pe termeni.

Tipuri simple

Fie $\mathbb{V} = \{\alpha, \beta, \gamma, \ldots\}$ o mulțime infinită de tipuri variabilă.

Mulțimea tuturor tipurilor simple $\mathbb{T}$ este definită prin

$$\mathbb{T} = \mathbb{V} \mid \mathbb{T} \to \mathbb{T}$$

- (Tipul variabilă) Dacă $\alpha \in \mathbb{V}$, atunci $\alpha \in \mathbb{T}$.
- (Tipul săgeată) Dacă $\sigma, \tau \in \mathbb{T}$, atunci $(\sigma \to \tau) \in \mathbb{T}$.

Câteodată vom nota tipurile simple și cu litere $A, B, \ldots$.

Tipurile variabilă sunt reprezentări abstracte pentru tipuri de bază cum ar fi *Nat* pentru numere naturale, *List* pentru liste etc.

Tipurile săgeată reprezintă tipuri pentru funcții cum ar fi

- $Nat \to Real$, mulțimea tuturor funcțiilor de la numere naturale la numere reale
- $(Nat \to Int) \to (Int \to Nat)$, mulțimea tuturor funcțiilor care au ca intrare o funcție de la numere naturale la întregi și produce o funcție de la întregi la numere naturale.

Tipuri simple

Mulțimea tipurilor simple $\mathbb{T} = \mathbb{V} \mid \mathbb{T} \to \mathbb{T}$

Exemple de tipuri simple:

- γ
- $(\beta \to \gamma)$
- $((\gamma \to \alpha) \to (\alpha \to (\beta \to \gamma)))$

În tipurile săgeată, parantezele exterioare pot fi omise.

Parantezele în tipurile săgeată sunt asociative la dreapta.

De exemplu,

- $\alpha_1 \to \alpha_2 \to \alpha_3 \to \alpha_4$ este abreviere pentru $(\alpha_1 \to (\alpha_2 \to (\alpha_3 \to \alpha_4)))$
- $x_1\ x_2\ x_3\ x_4$ este abreviere pentru $(((x_1\ x_2)\ x_3)\ x_4)$

Termeni și tipuri

Ce înseamnă că un termen M are un tip σ?

Vom nota acest lucru cu $M : \sigma$.

Termeni și tipuri

Ce înseamnă că un termen M are un tip σ?

Vom nota acest lucru cu $M : \sigma$.

Variabilă. Dacă o variabilă x are un tip σ, notăm cu $x : \sigma$.

Convenția Barendregt: variabilele legate sunt distincte.

Presupunem că orice variabilă din M are un unic tip.

Dacă $x : \sigma$ și $x : \tau$, atunci $\sigma \equiv \tau$.

Termeni și tipuri

Ce înseamnă că un termen M are un tip σ?

Vom nota acest lucru cu $M : \sigma$.

Variabilă. Dacă o variabilă x are un tip σ, notăm cu $x : \sigma$.

Convenția Barendregt: variabilele legate sunt distincte.

Presupunem că orice variabilă din M are un unic tip.

Dacă $x : \sigma$ și $x : \tau$, atunci $\sigma \equiv \tau$.

Aplicare. Pentru $M\,N$ este clar că vrem să știm tipurile lui M și N.

Intuitiv, $M\,N$ înseamnă că ("funcția") M este aplicată ("intrării") N.

Atunci M trebuie să aibă un tip funcție, adică $M : \sigma \to \tau$, iar N trebuie să fie "adecvat" pentru această funcție, adică $N : \sigma$.

Dacă $M : \sigma \to \tau$ și $N : \sigma$, atunci $M\,N : \tau$.

Termeni și tipuri

Ce înseamnă că un termen M are un tip σ?

Vom nota acest lucru cu $M : \sigma$.

Variabilă. Dacă o variabilă x are un tip σ, notăm cu $x : \sigma$.

Convenția Barendregt: variabilele legate sunt distincte.

Presupunem că orice variabilă din M are un unic tip.

Dacă $x : \sigma$ și $x : \tau$, atunci $\sigma \equiv \tau$.

Aplicare. Pentru $M\,N$ este clar că vrem să știm tipurile lui M și N.

Intuitiv, $M\,N$ înseamnă că ("funcția") M este aplicată ("intrării") N.

Atunci M trebuie să aibă un tip funcție, adică $M : \sigma \to \tau$, iar N trebuie să fie "adecvat" pentru această funcție, adică $N : \sigma$.

Dacă $M : \sigma \to \tau$ și $N : \sigma$, atunci $M\,N : \tau$.

Abstractizare. Dacă $M : \tau$, ce tip trebuie sa aibă $\lambda x.\, M$?

Dacă $x : \sigma$ și $M : \tau$, atunci $\lambda x.\, M : \sigma \to \tau$.

Termeni și tipuri

Variabilă. $x : \sigma$.

Aplicare. Dacă $M : \sigma \to \tau$ și $N : \sigma$, atunci $M\,N : \tau$.

Abstractizare. Dacă $x : \sigma$ și $M : \tau$, atunci $\lambda x.\, M : \sigma \to \tau$.

M are tip (este *typeable*) dacă există un tip σ astfel încât $M : \sigma$.

Termeni și tipuri

Variabilă. $x : \sigma$.

Aplicare. Dacă $M : \sigma \to \tau$ și $N : \sigma$, atunci $M\,N : \tau$.

Abstractizare. Dacă $x : \sigma$ și $M : \tau$, atunci $\lambda x.\, M : \sigma \to \tau$.

M are tip (este *typeable*) dacă există un tip σ astfel încât $M : \sigma$.

Exemple.

- Dacă $x : \sigma$, atunci funcția identitate are tipul $\lambda x.\, x : \sigma \to \sigma$.

Termeni și tipuri

Variabilă. $x : \sigma$.

Aplicare. Dacă $M : \sigma \to \tau$ și $N : \sigma$, atunci $M\,N : \tau$.

Abstractizare. Dacă $x : \sigma$ și $M : \tau$, atunci $\lambda x.\, M : \sigma \to \tau$.

M are tip (este *typeable*) dacă există un tip σ astfel încât $M : \sigma$.

Exemple.

- Dacă $x : \sigma$, atunci funcția identitate are tipul $\lambda x.\, x : \sigma \to \sigma$.
- Conform convențiilor de la aplicare, $y\,x$ poate avea un tip doar dacă y are un tip săgeată de forma $\sigma \to \tau$ și tipul lui x se potrivește cu tipul domeniu σ. În acest caz, tipul lui $y\,x : \tau$.

Termeni și tipuri

Variabilă. $x : \sigma$.

Aplicare. Dacă $M : \sigma \to \tau$ și $N : \sigma$, atunci $M\,N : \tau$.

Abstractizare. Dacă $x : \sigma$ și $M : \tau$, atunci $\lambda x.\, M : \sigma \to \tau$.

M are tip (este *typeable*) dacă există un tip σ astfel încât $M : \sigma$.

Exemple.

- Dacă $x : \sigma$, atunci funcția identitate are tipul $\lambda x.\, x : \sigma \to \sigma$.
- Conform convențiilor de la aplicare, $y\,x$ poate avea un tip doar dacă y are un tip săgeată de forma $\sigma \to \tau$ și tipul lui x se potrivește cu tipul domeniu σ. În acest caz, tipul lui $y\,x : \tau$.
- Termenul $x\,x$ nu poate avea nici un tip (nu este typeable). Pe de o parte, x ar trebui să aibă tipul $\sigma \to \tau$ (pentru prima apariție), pe de altă ar trebui să aibă tipul σ (pentru a doua apariție). Cum am stabilit că orice variabilă are un unic tip, obținem $\sigma \to \tau \equiv \sigma$, ceea ce este imposibil.

Discuție despre asocitativitate

Asociativitatea la dreapta pentru tipurile săgeată vs. asociativitatea la stânga pentru aplicare:

- Să presupunem că $f : \rho \to (\sigma \to \tau)$, $x : \rho$ și $y : \sigma$.
- Atunci $f\ x : \sigma \to \tau$ și $(f\ x)\ y : \tau$.

Discuție despre asocitativitate

Asociativitatea la dreapta pentru tipurile săgeată vs. asociativitatea la stânga pentru aplicare:

- Să presupunem că $f : \rho \to (\sigma \to \tau)$, $x : \rho$ și $y : \sigma$.
- Atunci $f\ x : \sigma \to \tau$ și $(f\ x)\ y : \tau$.
- Folosind ambele convenții pentru asociativitate pentru a elimina parantezele, avem

$$f : \rho \to \sigma \to \tau$$
$$f\ x\ y : \tau$$

Convențiile pentru asociativitate sunt în armonie una cu cealaltă.

Church-typing vs. Curry-typing

A găsi tipul unui termen începe cu a găsi tipurile pentru variabile. Există două metode prin care putem asocia tipuri variabilelor.

Asociere explicită *(Church-typing)*.

- Constă în prescrierea unui unic tip pentru fiecare variabilă, la introducerea acesteia.
- Presupune că tipurile variabilelor sunt explicit stabilite.
- Tipurile termenilor mai complecși se obțin natural, ținând cont de convențiile pentru aplicare și abstractizare.

Asociere implicită *(Curry-typing)*.

- Constă în a nu prescriere un tip pentru fiecare variabilă, ci în a le lăsa "deschise" (implicite).
- În acest caz, termenii *typeable* sunt descoperiți printr-un proces de căutare, care poate presupune "ghicirea" anumitor tipuri.

Church-typing vs. Curry-typing

Exemplu. Asociere explicită *(Church-typing)*.

Vrem să calculăm tipul expresiei $(\lambda zu.\, z)\,(y\,x)$ știind că

1. $x : \alpha \to \alpha$
2. $y : (\alpha \to \alpha) \to \beta$
3. $z : \beta$
4. $u : \gamma$

Aplicare. Dacă $M : \sigma \to \tau$ și $N : \sigma$, atunci $M\,N : \tau$.

Abstractizare. Dacă $x : \sigma$ și $M : \tau$, atunci $\lambda x.\, M : \sigma \to \tau$.

Church-typing vs. Curry-typing

Exemplu. Asociere explicită *(Church-typing)*.

Vrem să calculăm tipul expresiei $(\lambda zu.\, z)\,(y\,x)$ știind că

1. $x : \alpha \to \alpha$
2. $y : (\alpha \to \alpha) \to \beta$
3. $z : \beta$
4. $u : \gamma$

Aplicare. Dacă $M : \sigma \to \tau$ și $N : \sigma$, atunci $M\,N : \tau$.

Abstractizare. Dacă $x : \sigma$ și $M : \tau$, atunci $\lambda x.\, M : \sigma \to \tau$.

Din (2) și (1), prin aplicare obținem (5): $y\,x : \beta$.

Din (4) și (3), prin abstractizare obținem (6): $\lambda u.\, z : \gamma \to \beta$.

Din (3) și (6), prin abstractizare obținem (7): $\lambda zu.\, z : \beta \to \gamma \to \beta$.
Nu uitați că $\beta \to \gamma \to \beta$ înseamnă $\beta \to (\gamma \to \beta)$.

Atunci, din (7) și (5), prin aplicare, avem $(\lambda zu.\, z)\,(y\,x) : \gamma \to \beta$.

Church-typing vs. Curry-typing

Exemplu. Asociere implicită *(Curry-typing)*.

Considerăm termenul de mai devreme $M = (\lambda zu.\, z)\,(y\, x)$.

Putem să "ghicim" tipurile variabilelor astfel încât M să *aibă tip*?

Aplicare. Dacă $M : \sigma \to \tau$ și $N : \sigma$, atunci $M\,N : \tau$.

Abstractizare. Dacă $x : \sigma$ și $M : \tau$, atunci $\lambda x.\, M : \sigma \to \tau$.

Church-typing vs. Curry-typing

Exemplu. Asociere implicită *(Curry-typing)*.

Considerăm termenul de mai devreme $M = (\lambda zu.\, z)\,(y\, x)$.

Putem să "ghicim" tipurile variabilelor astfel încât M să *aibă tip*?

Aplicare. Dacă $M : \sigma \to \tau$ și $N : \sigma$, atunci $M\,N : \tau$.

Abstractizare. Dacă $x : \sigma$ și $M : \tau$, atunci $\lambda x.\, M : \sigma \to \tau$.

- Observăm că M este o aplicare a lui $\lambda zu.\, z$ termenului $y\, x$.
- Atunci $\lambda zu.\, z$ trebuie să aibă un tip săgeată, de exemplu $\lambda zu.\, z : A \to B$, și $y\, x$ să se potrivească, adică $y\, x : A$.
- În acest caz, avem $M : B$.

Church-typing vs. Curry-typing

Exemplu. Asociere implicită *(Curry-typing)* (cont.)

Știm $M = (\lambda zu.\, z)\,(y\, x)$ și am dedus până acum:

$$\lambda zu.\, z : A \to B \qquad y\, x : A \qquad M : B$$

- Faptul că $\lambda zu.\, z : A \to B$ implică că $z : A$ și $\lambda u.\, z : B$.
- Deducem că B este tipul unei abstractizări, deci $B \equiv C \to D$, și obținem că $u : C$ și $z : D$.
- Pe de altă parte, $y\, x$ este o aplicare, deci trebuie să existe E și F astfel încât $y : E \to F$ și $x : E$. Atunci $y\, x : F$.

Church-typing vs. Curry-typing

Exemplu. Asociere implicită *(Curry-typing)* (cont.)

Știm $M = (\lambda zu.\, z)\,(y\, x)$. Am dedus următoarele:

- $x : E$
- $y : E \to F$
- $z : A$ și $z : D$, deci $A \equiv D$
- $u : C$
- $B \equiv C \to D$
- $y\, x : A$ și $y\, x : F$, deci $A \equiv F$.

În concluzie, $A \equiv D \equiv F$, și eliminând redundanțele obținem

$$(*) \qquad x : E \qquad y : E \to A \qquad z : A \qquad u : C$$

Reamintim că aveam $M : B$, adică $M : C \to A$.

Am obținut o schemă generală $(*)$ pentru tipurile lui x, y, z, u care induc un tip pentru M.

Church-typing vs. Curry-typing

Exemplu. Asociere implicită *(Curry-typing)* (cont.)

Știm $M = (\lambda zu.\, z)\,(y\, x)$. Am obținut schema generală

$$(*) \quad x : E \quad y : E \to A \quad z : A \quad u : C \quad M : C \to A$$

În schema de mai sus, putem considera tipuri "reale":

- $x : \beta$, $y : \beta \to \alpha$, $z : \alpha$, $u : \delta$, $M : \delta \to \alpha$

Church-typing vs. Curry-typing

Exemplu. Asociere implicită *(Curry-typing)* (cont.)

Știm $M = (\lambda zu.\, z)\,(y\, x)$. Am obținut schema generală

$$(*) \quad x : E \quad y : E \to A \quad z : A \quad u : C \quad M : C \to A$$

În schema de mai sus, putem considera tipuri "reale":

- $x : \beta$, $y : \beta \to \alpha$, $z : \alpha$, $u : \delta$, $M : \delta \to \alpha$
- $x : \alpha \to \alpha$, $y : (\alpha \to \alpha) \to \beta$, $z : \beta$, $u : \gamma$, $M : \gamma \to \beta$
  (soluția discutată la Church-typing)

Church-typing vs. Curry-typing

Exemplu. Asociere implicită *(Curry-typing)* (cont.)

Știm $M = (\lambda zu.\, z)\,(y\, x)$. Am obținut schema generală

$$(*) \quad x : E \quad y : E \to A \quad z : A \quad u : C \quad M : C \to A$$

În schema de mai sus, putem considera tipuri "reale":

- $x : \beta$, $y : \beta \to \alpha$, $z : \alpha$, $u : \delta$, $M : \delta \to \alpha$
- $x : \alpha \to \alpha$, $y : (\alpha \to \alpha) \to \beta$, $z : \beta$, $u : \gamma$, $M : \gamma \to \beta$
  (soluția discutată la Church-typing)
- $x : \alpha$, $y : \alpha \to \alpha \to \beta$, $z : \alpha \to \beta$, $u : \alpha \to \alpha$,
  $M : (\alpha \to \alpha) \to \alpha \to \beta$

Church-typing vs. Curry-typing

Asocierea implicită de tipuri *(Curry-typing)* are proprietăți interesante, cum am văzut în exemplul anterior.

Totuși, în continuare vom folosi asocierea explicită *(Church-typing)* deoarece de obicei tipurile sunt cunoscute dinainte (și declararea tipurilor pentru argumentele unei funcții este o bună-practică).

Marcăm tipurile variabilelor legate imediat după introducerea lor cu o abstractizare. Tipurile variabilelor libere sunt date de un context.

Church-typing

Exemplu. Să considerăm exemplul anterior $(\lambda zu.\, z)\,(y\, x)$.

Observați că z și u sunt legate, iar x și y sunt libere.

Presupunând că $z : \beta$ și $u : \gamma$, scriem termenul astfel

$$(\lambda z : \beta.\, \lambda u : \gamma.\, z)\,(y\, x)$$

Church-typing

Exemplu. Să considerăm exemplul anterior $(\lambda zu.\, z)\,(y\, x)$.

Observați că z și u sunt legate, iar x și y sunt libere.

Presupunând că $z : \beta$ și $u : \gamma$, scriem termenul astfel

$$(\lambda z : \beta.\, \lambda u : \gamma.\, z)\,(y\, x)$$

Dacă presupunem un context în care despre variabilele libere știm, de exemplu, că $x : \alpha \to \alpha$ și $y : (\alpha \to \alpha) \to \beta$, atunci folosim notația:

$$x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta \vdash (\lambda z : \beta.\, \lambda u : \gamma.\, z)\,(y\, x)$$

Church-typing

Exemplu. Să considerăm exemplul anterior $(\lambda zu.\, z)\,(y\, x)$.

Observați că z și u sunt legate, iar x și y sunt libere.

Presupunând că $z : \beta$ și $u : \gamma$, scriem termenul astfel

$$(\lambda z : \beta.\, \lambda u : \gamma.\, z)\,(y\, x)$$

Dacă presupunem un context în care despre variabilele libere știm, de exemplu, că $x : \alpha \to \alpha$ și $y : (\alpha \to \alpha) \to \beta$, atunci folosim notația:

$$x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta \vdash (\lambda z : \beta.\, \lambda u : \gamma.\, z)\,(y\, x)$$

Încă nu avem o noțiune de β-reducție pentru termeni cu tipuri, dar ne-am putea gândi că am avea:

$$(\lambda z : \beta.\, \lambda u : \gamma.\, z)\,(y\, x) \to_\beta \lambda u : \gamma.\, y\, x.$$

Observați că am dori să deducem că $(\lambda u : \gamma.\, y\, x) : \gamma \to \beta$.

Sistem de deducție pentru Church $\lambda\!\!\rightarrow$

Deoarece am convenit cum să decorăm cu informații despre tipuri variabilele legate, trebuie să actualizăm definiția λ-termenilor.

Mulțimea λ-termenilor cu pre-tipuri $\Lambda_{\mathbb{T}}$ este

$$\Lambda_{\mathbb{T}} = x \mid \Lambda_{\mathbb{T}}\,\Lambda_{\mathbb{T}} \mid \lambda x : \mathbb{T}.\,\Lambda_{\mathbb{T}}$$

O afirmație este o expresie de forma $M : \sigma$, unde $M \in \Lambda_{\mathbb{T}}$ și $\sigma \in \mathbb{T}$. Într-o astfel de afirmație, M se numește subiect și σ tip.

O declarație este o afirmație în care subiectul este o variabilă ($x : \sigma$).

Un context este o listă de declarații cu subiecți diferiți.

O judecată este o expresie de forma $\Gamma \vdash M : \sigma$, unde Γ este context și $M : \sigma$ este o afirmație.

Sistem de deducție pentru Church $\lambda\!\!\rightarrow$

Deoarece suntem în general interesați de termeni *typeable*, am dori să avem o metodă prin care să putem stabili dacă un termen $t \in \Lambda_{\mathbb{T}}$ este *typeable* și dacă da, să calculăm un tip pentru t.

Vom da niște reguli care să ne permită să stabilim dacă o judecată $\Gamma \vdash M : \sigma$ poate fi dedusă, adică dacă M are tipul σ în contextul Γ.

Sistem de deducție pentru calculul Church $\lambda\!\rightarrow$

$$\frac{}{\Gamma \vdash x : \sigma} \text{ dacă } x : \sigma \in \Gamma \ (var)$$

$$\frac{\Gamma \vdash M : \sigma \rightarrow \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau} \ (app)$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma.\, M) : \sigma \rightarrow \tau} \ (abs)$$

Un termen M în calculul $\lambda\!\rightarrow$ este legal dacă există un context Γ și un tip ρ astfel încât $\Gamma \vdash M : \rho$.

Sistem de deducție pentru calculul Church $\lambda\!\rightarrow$

$$\frac{}{\Gamma \vdash x : \sigma}\ (var) \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}\ (app) \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma.\, M) : \sigma \to \tau}\ (abs)$$

dacă $x : \sigma \in \Gamma$

Exemplu. Să arătăm că termenul $\lambda y : \alpha \to \beta.\, \lambda z : \alpha.\, yz$ are tipul $(\alpha \to \beta) \to \alpha \to \beta$ în contextul vid.

$$\emptyset \vdash (\lambda y : \alpha \to \beta.\, \lambda z : \alpha.\, yz\,) : (\alpha \to \beta) \to \alpha \to \beta$$

Sistem de deducție pentru calculul Church $\lambda\!\rightarrow$

$$\frac{}{\Gamma \vdash x : \sigma}\ (var) \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}\ (app) \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma.\, M) : \sigma \to \tau}\ (abs)$$

dacă $x : \sigma \in \Gamma$

Exemplu. Să arătăm că termenul $\lambda y : \alpha \to \beta.\, \lambda z : \alpha.\, yz$ are tipul $(\alpha \to \beta) \to \alpha \to \beta$ în contextul vid.

$$\emptyset \vdash (\lambda y : \alpha \to \beta.\, \lambda z : \alpha.\, yz\,) : (\alpha \to \beta) \to \alpha \to \beta$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{y : \alpha \to \beta, z : \alpha \vdash y : \alpha \to \beta}\ (var) \qquad \dfrac{}{y : \alpha \to \beta, z : \alpha \vdash z : \alpha}\ (var)}{y : \alpha \to \beta, z : \alpha \vdash (yz\,) : \beta}\ (app)}{y : \alpha \to \beta \vdash (\lambda z : \alpha.\, yz\,) : \alpha \to \beta}\ (abs)}{\emptyset \vdash (\lambda y : \alpha \to \beta.\, \lambda z : \alpha.\, yz\,) : (\alpha \to \beta) \to \alpha \to \beta}\ (abs)$$

Diferite stiluri pentru a scrie deducții

În exemplul anterior, am scris derivarea în stilul arbore.

În stilul liniar, derivarea precedentă ar arăta astfel:

1.	$y : \alpha \to \beta, z : \alpha \vdash y : \alpha \to \beta$	(var)
2.	$y : \alpha \to \beta, z : \alpha \vdash z : \alpha$	(var)
3.	$y : \alpha \to \beta, z : \alpha \vdash (yz) : \beta$	(app) cu 1 și 2
4.	$y : \alpha \to \beta \vdash (\lambda z : \alpha . yz) : \alpha \to \beta$	(abs) cu 3
5.	$\emptyset \vdash (\lambda y : \alpha \to \beta . \lambda z : \alpha . yz) : (\alpha \to \beta) \to \alpha \to \beta$	(abs) cu 4

Diferite stiluri pentru a scrie deducții

În stilul cu cutii, afișăm fiecare declarație la începutul unei cutii și considerăm că declarația respectivă face parte din contextul pentru toate afirmațiile din cutia respectivă.

Când închidem o cutie, abstractizăm după variabila din declarația de la începutul cutiei.

$$(\lambda y : \alpha \to \beta.\, \lambda z : \alpha.\, yz\,) : (\alpha \to \beta) \to \alpha \to \beta$$

Diferite stiluri pentru a scrie deducții

În stilul cu cutii, afișăm fiecare declarație la începutul unei cutii și considerăm că declarația respectivă face parte din contextul pentru toate afirmațiile din cutia respectivă.

Când închidem o cutie, abstractizăm după variabila din declarația de la începutul cutiei.

$$\dfrac{\boxed{\begin{array}{ll} y : \alpha \to \beta & (\mathit{context}) \\ (\lambda z : \alpha . \, yz) : \alpha \to \beta & \end{array}}}{(\lambda y : \alpha \to \beta . \, \lambda z : \alpha . \, yz) : (\alpha \to \beta) \to \alpha \to \beta \quad (\mathit{abs})}$$

Diferite stiluri pentru a scrie deducții

În stilul cu cutii, afișăm fiecare declarație la începutul unei cutii și considerăm că declarația respectivă face parte din contextul pentru toate afirmațiile din cutia respectivă.

Când închidem o cutie, abstractizăm după variabila din declarația de la începutul cutiei.

$$\dfrac{\dfrac{\dfrac{y : \alpha \to \beta \quad (context)}{\dfrac{z : \alpha \quad (context)}{(yz) : \beta}}}{(\lambda z : \alpha . yz) : \alpha \to \beta \quad (abs)}}{(\lambda y : \alpha \to \beta . \lambda z : \alpha . yz) : (\alpha \to \beta) \to \alpha \to \beta \quad (abs)}$$

Diferite stiluri pentru a scrie deducții

În stilul cu cutii, afișăm fiecare declarație la începutul unei cutii și considerăm că declarația respectivă face parte din contextul pentru toate afirmațiile din cutia respectivă.

Când închidem o cutie, abstractizăm după variabila din declarația de la începutul cutiei.

1.	$y : \alpha \to \beta$	(*context*)
2.	$z : \alpha$	(*context*)
3.	$(yz) : \beta$	(*app*) cu 1 și 2
4.	$(\lambda z : \alpha . yz) : \alpha \to \beta$	(*abs*) cu 3
5.	$(\lambda y : \alpha \to \beta . \lambda z : \alpha . yz) : (\alpha \to \beta) \to \alpha \to \beta$	(*abs*) cu 4

Sistem de deducție pentru Church $\lambda\!\rightarrow$

Exercițiu. Arătăți că termenul $\lambda x : ((\alpha \to \beta) \to \alpha).\, x\,(\lambda z : \alpha.\, y)$ are tipul $((\alpha \to \beta) \to \alpha) \to \alpha$ în contextul $y : \beta$.

Quiz time!

https://tinyurl.com/2p9xf67e

Pe săptămâna viitoare!

Fundamentele limbajelor de programare

C06

Denisa Diaconescu
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Lambda calcul cu tipuri simple (cont.)

Tipuri simple

Mulțimea tipurilor simple $\mathbb{T} = \mathbb{V} \mid \mathbb{T} \to \mathbb{T}$

- (Tipul variabilă) Dacă $\alpha \in \mathbb{V}$, atunci $\alpha \in \mathbb{T}$.
- (Tipul săgeată) Dacă $\sigma, \tau \in \mathbb{T}$, atunci $(\sigma \to \tau) \in \mathbb{T}$.

Mulțimea λ-termenilor cu pre-tipuri $\Lambda_{\mathbb{T}}$ $\Lambda_{\mathbb{T}} = x \mid \Lambda_{\mathbb{T}}\, \Lambda_{\mathbb{T}} \mid \lambda x : \mathbb{T}.\, \Lambda_{\mathbb{T}}$

- O afirmație este o expresie de forma $M : \sigma$, unde $M \in \Lambda_{\mathbb{T}}$ și $\sigma \in \mathbb{T}$.
- O declarație este o afirmație de forma $x : \sigma$.
- Un context Γ este o listă de declarații cu subiecți diferiți.
- O judecată este o expresie de forma $\Gamma \vdash M : \sigma$.

Sistem de deducție pentru calculul Church $\lambda\!\to$

$$\frac{}{\Gamma \vdash x : \sigma}\ (var) \quad \text{dacă } x : \sigma \in \Gamma \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}\ (\to_E) \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma.\, M) : \sigma \to \tau}\ (\to_I)$$

Un termen M în calculul $\lambda\!\to$ este legal dacă $\Gamma \vdash M : \rho$.

Ce probleme putem să rezolvăm în teoria tipurilor?

Type Checking

Se reduce la a verifica că putem găsi o derivare pentru

$$context \vdash term : type$$

Ce probleme putem să rezolvăm în teoria tipurilor?

Well-typedness (Typability)

Se reduce la a verifica dacă un termen este legal. Concret, trebuie să găsim un context și un tip dacă termenul este legal, altfel să arătăm de ce nu se poate.

$$? \vdash \mathit{term} : ?$$

O variațiune a problemei este *Type Assignment* în care contextul este dat și trebuie să găsim tipul.

$$\mathit{context} \vdash \mathit{term} : ?$$

Ce probleme putem să rezolvăm în teoria tipurilor?

Term Finding (Inhabitation)

Dându-se un context și un tip, să stabilim dacă există un termen cu acel tip, în contextul dat.

$$\mathit{context} \vdash ? : \mathit{type}$$

Toate aceste probleme sunt decidabile pentru calculul Church $\lambda\!\!\rightarrow$!

Limitări ale lambda-calculului cu tipuri simple

Nu mai avem recursie nelimitată deoarece combinatorii de punct fix nu sunt *typeable*.

De exemplu, $\mathbf{Y} \triangleq \lambda y.\,(\lambda x.\,y\,(x\,x))\,(\lambda x.\,y\,(x\,x))$ nu este typeable.

Dar avem recursie primitivă (recursie care permite doar *looping* în care numărul de iterații este cunoscut dinainte).

De exemplu, $\mathbf{add} \triangleq \lambda mnfx.\,m\,f\,(n\,f\,x)$ este o funcție primitiv recursivă.

Faptul că orice evaluare se termină este important pentru implementări ale logicilor folosind lambda-calculul.

Limitări ale lambda-calculului cu tipuri simple

Tipurile pot fi prea restrictive.

De exemplu, am putea gândi că termenul $(\lambda f.\, \textbf{if}\, (f\, \textbf{T})\, (f\, 3)\, (f\, 5))\, (\lambda x.\, x)$ ar trebui să aibă un tip. Dar nu are!

Soluții posibile:

- Let-polymorphism unde variabilele libere din tipul lui f se redenumesc la fiecare folosire. De exemplu, am putea scrie

$$\textbf{let}\, f = \lambda x.\, x\, \textbf{in}$$
$$\textbf{if}\, (f\, \textbf{T})\, (f\, 3)\, (f\, 5)$$

- Cuantificatori de tipuri. De exemplu, am avea

$$\lambda x.\, x : \Pi\alpha\, .\, \alpha \to \alpha$$

  Operatorul de legare Π face explicit faptul că variabila de tip α nu este rigidă.

Alte tipuri

Tipul Unit și constructorul unit

Mulțimea tipurilor

$$\mathbb{T} = \mathbb{V} \mid \mathbb{T} \to \mathbb{T} \mid \texttt{Unit}$$

Mulțimea λ-termenilor cu pre-tipuri $\Lambda_{\mathbb{T}}$

$$\Lambda_{\mathbb{T}} = x \mid \Lambda_{\mathbb{T}}\, \Lambda_{\mathbb{T}} \mid \lambda x : \mathbb{T}.\, \Lambda_{\mathbb{T}} \mid \text{unit}$$

$$\frac{}{\Gamma \vdash \text{unit} : \texttt{Unit}}\ (\mathit{unit})$$

Tipul Void

Mulțimea tipurilor

$$\mathbb{T} = \mathbb{V} \mid \mathbb{T} \to \mathbb{T} \mid \texttt{Unit} \mid \texttt{Void}$$

Mulțimea λ-termenilor cu pre-tipuri $\Lambda_{\mathbb{T}}$

$$\Lambda_{\mathbb{T}} = x \mid \Lambda_{\mathbb{T}}\, \Lambda_{\mathbb{T}} \mid \lambda x : \mathbb{T}.\, \Lambda_{\mathbb{T}} \mid \text{unit}$$

Nu există regulă de tipuri pentru deoarece tipul `Void` nu are inhabitant.

Tipul produs și constructorul pereche

Mulțimea tipurilor

$$\mathbb{T} = \mathbb{V} \mid \mathbb{T} \to \mathbb{T} \mid \texttt{Unit} \mid \texttt{Void} \mid \mathbb{T} \times \mathbb{T}$$

Mulțimea λ-termenilor cu pre-tipuri $\Lambda_{\mathbb{T}}$

$$\Lambda_{\mathbb{T}} = x \mid \Lambda_{\mathbb{T}}\, \Lambda_{\mathbb{T}} \mid \lambda x : \mathbb{T}.\, \Lambda_{\mathbb{T}} \mid \text{unit} \mid \langle \Lambda_{\mathbb{T}}, \Lambda_{\mathbb{T}} \rangle$$

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \tau}{\Gamma \vdash \langle M, N \rangle : \sigma \times \tau}\ (\times_I)$$

Tipul produs și constructorul pereche

Mulțimea tipurilor

$$\mathbb{T} = \mathbb{V} \mid \mathbb{T} \to \mathbb{T} \mid \texttt{Unit} \mid \texttt{Void} \mid \mathbb{T} \times \mathbb{T}$$

Mulțimea λ-termenilor cu pre-tipuri $\Lambda_{\mathbb{T}}$

$$\Lambda_{\mathbb{T}} = x \mid \Lambda_{\mathbb{T}}\, \Lambda_{\mathbb{T}} \mid \lambda x : \mathbb{T}.\, \Lambda_{\mathbb{T}} \mid \text{unit} \mid \langle \Lambda_{\mathbb{T}}, \Lambda_{\mathbb{T}} \rangle \mid \mathit{fst}\ \Lambda_{\mathbb{T}} \mid \mathit{snd}\ \Lambda_{\mathbb{T}}$$

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \tau}{\Gamma \vdash \langle M, N \rangle : \sigma \times \tau}\ (\times_I)$$

$$\frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash \mathit{fst}\ M : \sigma}\ (\times_{E_1}) \qquad \frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash \mathit{snd}\ M : \tau}\ (\times_{E_2})$$

Tipul sumă și constructorii Left/Right

Mulțimea tipurilor

$$\mathbb{T} = \mathbb{V} \mid \mathbb{T} \to \mathbb{T} \mid \texttt{Unit} \mid \texttt{Void} \mid \mathbb{T} \times \mathbb{T} \mid \mathbb{T} + \mathbb{T}$$

Mulțimea λ-termenilor cu pre-tipuri $\Lambda_{\mathbb{T}}$

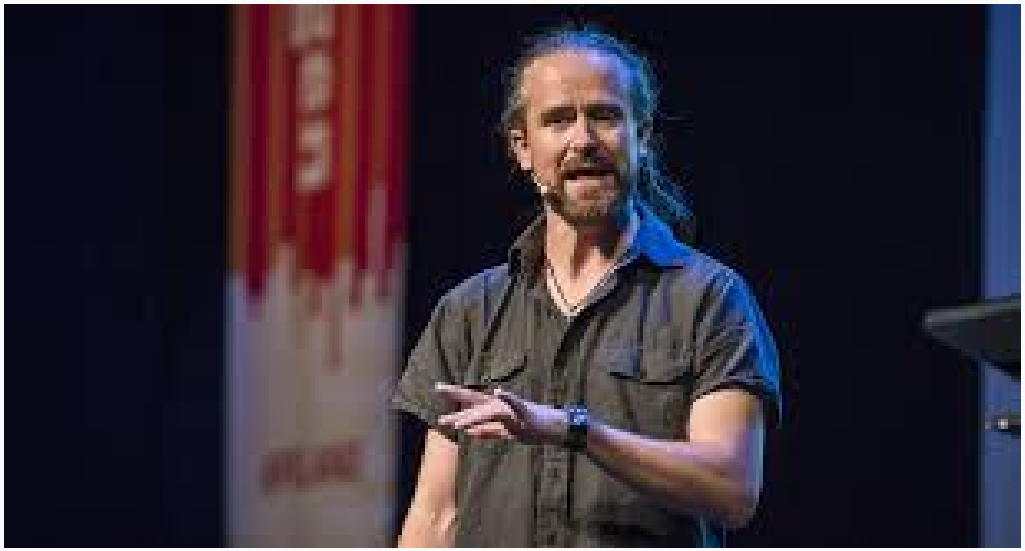
$$\Lambda_{\mathbb{T}} = x \mid \Lambda_{\mathbb{T}}\, \Lambda_{\mathbb{T}} \mid \lambda x : \mathbb{T}.\, \Lambda_{\mathbb{T}} \mid \text{unit} \mid \langle \Lambda_{\mathbb{T}}, \Lambda_{\mathbb{T}} \rangle \mid \mathit{fst}\ \Lambda_{\mathbb{T}} \mid \mathit{snd}\ \Lambda_{\mathbb{T}}$$
$$\mid \text{Left}\ \Lambda_{\mathbb{T}} \mid \text{Right}\ \Lambda_{\mathbb{T}} \mid \text{case}\ \Lambda_{\mathbb{T}}\ \text{of}\ \Lambda_{\mathbb{T}}\ ;\ \Lambda_{\mathbb{T}}$$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \text{Left}\ M : \sigma + \tau}\ (+_{I_1}) \qquad \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \text{Right}\ M : \sigma + \tau}\ (+_{I_2})$$

$$\frac{\Gamma \vdash M : \sigma + \tau \qquad \Gamma \vdash M_1 : \sigma \to \gamma \qquad \Gamma \vdash M_2 : \tau \to \gamma}{\Gamma \vdash \text{case}\ M\ \text{of}\ M_1\ ;\ M_2 : \gamma}\ (+_E)$$

Corespondența Curry-Howard

Schimbați perspectiva



Roger Antonsen

Universitatea din Oslo

TED Talk: Math is the hidden secret to understanding the world

"... înțelegerea constă în abilitatea de a-ți schimba perspectiva"

https://www.ted.com/talks/roger_antonsen_math_is_the_hidden_secret_to_understanding_the_world

Un program simplu în Haskell

```haskell
data Point = Point Int Int

makePoint :: Int -> Int -> Point
makePoint x y = Point x y

getX :: Point -> Int
getX (Point x y) = x

getY :: Point -> Int
getY (Point x y) = y

origin :: Point
origin = makePoint 0 0
```

Un program simplu în Haskell

Hai să schimbăm perspectiva!

```
data Point = Point Int Int

makePoint :: Int -> Int -> Point
makePoint x y = Point x y

getX :: Point -> Int
getX (Point x y) = x

getY :: Point -> Int
getY (Point x y) = y
```

$$\frac{x : Int \quad y : Int}{makePoint\ x\ y : Point}\ (Point_I)$$

$$\frac{p : Point}{getX\ p : Int}\ (Point_{E_1})$$

$$\frac{p : Point}{getY\ p : Int}\ (Point_{E_2})$$

Generalizare

$$\frac{x : \mathit{Int} \quad y : \mathit{Int}}{\mathit{makePoint}\ x\ y : \mathit{Point}}\ (\mathit{Point}_I) \qquad \frac{M : \sigma \quad N : \tau}{\langle M, N \rangle : \sigma \times \tau}\ (\times_I)$$

$$\frac{p : \mathit{Point}}{\mathit{getX}\ p : \mathit{Int}}\ (\mathit{Point}_{E_1}) \qquad \frac{M : \sigma \times \tau}{\mathit{fst}\ M : \sigma}\ (\times_{E_1})$$

$$\frac{p : \mathit{Point}}{\mathit{getY}\ p : \mathit{Int}}\ (\mathit{Point}_{E_2}) \qquad \frac{M : \sigma \times \tau}{\mathit{snd}\ M : \tau}\ (\times_{E_2})$$

Generalizare

$$\frac{x : Int \quad y : Int}{makePoint\ x\ y : Point}\ (Point_I)$$

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash \langle M, N \rangle : \sigma \times \tau}\ (\times_I)$$

$$\frac{M : Point}{getX\ M : Int}\ (Point_{E_1})$$

$$\frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash fst\ M : \sigma}\ (\times_{E_1})$$

$$\frac{M : Point}{getY\ M : Int}\ (Point_{E_2})$$

$$\frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash snd\ M : \tau}\ (\times_{E_2})$$

Alt exemplu simplu

```
f = (\x -> x * 3) :: Int -> Int
```

$$\frac{\{x : Int\} \vdash x * 3 : Int}{\lambda x.x * 3 : Int \to Int}\ (fun_I)$$

```
> f 5
15
```

$$\frac{f : Int \to Int \quad 5 : Int}{f\ 5 : Int}\ (fun_E)$$

Generalizare

$$\frac{\{x : Int\} \vdash x * 3 : Int}{\lambda x.x * 3 : Int \to Int}\ (fun_I)$$

$$\frac{\{x : \sigma\} \vdash M : \tau}{\lambda x.M : \sigma \to \tau}\ (\to_I)$$

$$\frac{f : Int \to Int \quad 5 : Int}{f\ 5 : Int}\ (fun_E)$$

$$\frac{M : \sigma \to \tau \quad N : \sigma}{M\ N : \tau}\ (\to_E)$$

Generalizare

$$\frac{\{x : Int\} \vdash x * 3 : Int}{\lambda x.x * 3 : Int \to Int}\ (fun_I)$$

$$\frac{\Gamma \cup \{x : \sigma\} \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}\ (\to_I)$$

$$\frac{f : Int \to Int \quad 5 : Int}{f\ 5 : Int}\ (fun_E)$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}\ (\to_E)$$

Logica. Ce este adevăt și ce este fals?

Hai să schimbăm perspectiva iar!

Logica. Ce este adevăt și ce este fals?

Hai să schimbăm perspectiva iar!

Dacă afară este întuneric atunci,
dacă porcii zboară atunci este întuneric afară.

σ = afară este întuneric
τ = porcii zboară

$$\sigma \supset (\tau \supset \sigma)$$

Logica. Ce este adevăt și ce este fals?

Hai să schimbăm perspectiva iar!

Dacă afară este întuneric atunci,
dacă porcii zboară atunci este întuneric afară.

σ = afară este întuneric
τ = porcii zboară

$$\sigma \supset (\tau \supset \sigma)$$

Este adevărată această afirmație? Da!

σ	τ	$\tau \supset \sigma$	$\sigma \supset (\tau \supset \sigma)$
false	false	true	true
false	true	false	true
true	false	true	true
true	true	true	true

Semantica unei logici

Dăm valori variabilelor în mulțimea $\{0, 1\}$,
definim o evaluare $e : V \to \{0, 1\}$.

Putem să o extindem o evaluare la formule:

$\wedge : \{0, 1\} \times \{0, 1\} \to \{0, 1\}$

σ	τ	$\sigma \wedge \tau$
0	0	0
0	1	0
1	0	0
1	1	1

$\supset : \{0, 1\} \times \{0, 1\} \to \{0, 1\}$

σ	τ	$\sigma \supset \tau$
0	0	1
0	1	1
1	0	0
1	1	1

Dacă pentru toate evaluările posibile, o formulă are valoarea 1,
atunci spunem că este o tautologie.

Sintaxa unei logici

Dăm metode pentru a manipula simbolurile din logică (i.e., $\supset$, $\wedge$) pentru a stabili când o formulă este demonstrabilă/teoremă .

Corectitudine	=	sintaxa implică semantica
Completitudine	=	sintaxa și semantica coincid

Un sistem de deducție naturală

Reguli pentru a manevra fiecare conector logic (introducerea si eliminarea conectorilor).

$$\frac{\Gamma \vdash \sigma \quad \Gamma \vdash \tau}{\Gamma \vdash \sigma \wedge \tau}\ (\wedge_I) \qquad \frac{\Gamma \vdash \sigma \wedge \tau}{\Gamma \vdash \sigma}\ (\wedge_{E_1}) \qquad \frac{\Gamma \vdash \sigma \wedge \tau}{\Gamma \vdash \tau}\ (\wedge_{E_2})$$

$$\frac{\Gamma \cup \{\sigma\} \vdash \tau}{\Gamma \vdash \sigma \supset \tau}\ (\supset_I) \qquad \frac{\Gamma \vdash \sigma \supset \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash \tau}\ (\supset_E)$$

Arată cunoscut?

Corespondența Curry-Howard

λ-calcul cu tipuri	Deducție naturală
$\dfrac{\Gamma \vdash M : \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash \langle M, N \rangle : \sigma \times \tau}\ (\times_I)$	$\dfrac{\Gamma \vdash \sigma \quad \Gamma \vdash \tau}{\Gamma \vdash \sigma \wedge \tau}\ (\wedge_I)$
$\dfrac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash \mathit{fst}\ M : \sigma}\ (\times_{E_1})$	$\dfrac{\Gamma \vdash \sigma \wedge \tau}{\Gamma \vdash \sigma}\ (\wedge_{E_1})$
$\dfrac{\Gamma \vdash p : \sigma \times \tau}{\Gamma \vdash \mathit{snd}\ p : \tau}\ (\times_{E_2})$	$\dfrac{\Gamma \vdash \sigma \wedge \tau}{\Gamma \vdash \tau}\ (\wedge_{E_2})$
$\dfrac{\Gamma \cup \{x : \sigma\} \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}\ (\to_I)$	$\dfrac{\Gamma \cup \{\sigma\} \vdash \tau}{\Gamma \vdash \sigma \supset \tau}\ (\supset_I)$
$\dfrac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}\ (\to_E)$	$\dfrac{\Gamma \vdash \sigma \supset \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash \tau}\ (\supset_E)$

Propositions are types! ♡

Să analizăm mai atent

λ-calcul cu tipuri	Deducție naturală
$\Gamma \vdash M : \sigma$	$\Gamma \vdash \sigma$

Faptul că există un termen de tip σ *(inhabitation of type σ)*
înseamnă că σ este teoremă/are o demonstrație în logică! $\heartsuit$

Să analizăm mai atent

λ-calcul cu tipuri

$$\frac{\{x : \sigma\} \vdash x : \sigma}{\vdash \lambda x.\, x : \sigma \to \sigma}\ (\to_I)$$

Deducție naturală

$$\frac{\{\sigma\} \vdash \sigma}{\vdash \sigma \supset \sigma}\ (\supset_I)$$

Să analizăm mai atent

λ-calcul cu tipuri

$$\frac{\{x : \sigma\} \vdash x : \sigma}{\vdash \lambda x.\, x : \sigma \to \sigma}\ (\to_I)$$

$$\frac{\dfrac{\overline{\{x : \sigma, y : \tau\} \vdash x : \sigma}}{\{x : \sigma\} \vdash \lambda y.\, x : \tau \to \sigma}\ (\to_I)}{\vdash \lambda x.\, (\lambda y.\, x) : \sigma \to (\tau \to \sigma)}\ (\to_I)$$

Deducție naturală

$$\frac{\{\sigma\} \vdash \sigma}{\vdash \sigma \supset \sigma}\ (\supset_I)$$

$$\frac{\dfrac{\overline{\{\sigma, \tau\} \vdash \sigma}}{\{\sigma\} \vdash \tau \to \sigma}\ (\supset_I)}{\vdash \sigma \to (\tau \to \sigma)}\ (\supset_I)$$

Să analizăm mai atent

λ-calcul cu tipuri

$$\frac{\{x : \sigma\} \vdash x : \sigma}{\vdash \lambda x.\, x : \sigma \to \sigma}\ (\to_I)$$

$$\frac{\dfrac{\overline{\{x : \sigma, y : \tau\} \vdash x : \sigma}}{\{x : \sigma\} \vdash \lambda y.\, x : \tau \to \sigma}\ (\to_I)}{\vdash \lambda x.\, (\lambda y.\, x) : \sigma \to (\tau \to \sigma)}\ (\to_I)$$

Deducție naturală

$$\frac{\{\sigma\} \vdash \sigma}{\vdash \sigma \supset \sigma}\ (\supset_I)$$

$$\frac{\dfrac{\overline{\{\sigma, \tau\} \vdash \sigma}}{\{\sigma\} \vdash \tau \to \sigma}\ (\supset_I)}{\vdash \sigma \to (\tau \to \sigma)}\ (\supset_I)$$

Proofs are Terms! ♡

Demonstrațiile sunt termeni!

Corespondența Curry-Howard

Teoria Tipurilor	Logică
tipuri	formule
termeni	demonstrații
inhabitation a tipului σ	demonstrație a lui σ

Corespondența Curry-Howard

Teoria Tipurilor	Logică
tipuri	formule
termeni	demonstrații
inhabitation a tipului σ	demonstrație a lui σ
tip produs	conjuncție
tip funcție	implicație

Corespondența Curry-Howard

Teoria Tipurilor	Logică
tipuri	formule
termeni	demonstrații
inhabitation a tipului σ	demonstrație a lui σ
tip produs	conjuncție
tip funcție	implicație
tip sumă	disjuncție
tipul void	false
tipul unit	true

Logica intuiționistă

- Logică constructivistă
- Bazată pe noțiunea de demonstrație
- Utilă deoarece demonstrațiile sunt executabile și produc exemple
  Permite "extragererea" de programe demonstrate a fi corecte.
- Baza pentru *proof assistants* (e.g., Coq, Agda, Idris)
- Următoarele formule echivalente nu sunt demonstrabile în logica intuiționistă!
  - dubla negație: $\neg\neg\varphi \supset \varphi$
  - excluded middle: $\varphi \vee \neg\varphi$
  - legea lui Pierce: $((\varphi \supset \tau) \supset \varphi) \supset \varphi$
- Nu există semantică cu tabele de adevăr pentru logica intuiționistă! Semantici alternative (e.g., semantica de tip Kripke)

Corespondența Curry-Howard

Inițial, corespondența Curry-Howard a fost între

Calculul Church $\lambda \rightarrow$	Sistemul de deducție naturală al lui Gentzen pentru logica intuiționistă

De ce?

- Este pur si simplu fascinant
- Nu gândiți logica și informatica ca domenii diferite.
- Gândind din perspective diferite ne poate ajuta să știm ce este posibil/imposibil.
- Teoria tipurilor nu ar trebui să fie o adunătură *ad hoc* de reguli!

Quiz time!

https://tinyurl.com/C06-Quiz1

Pe săptămâna viitoare!

Fundamentele limbajelor de programare

C07

Denisa Diaconescu
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Lambda calcul cu tipuri simple (cont.)

Ce problemă vrem să rezolvăm în cursul de astăzi?

Type Inference

Pentru un lambda termen M fără tipuri, vrem să adnotăm termenul M cu tipuri obținând $\overline{M}$ și să rezolvăm problema

$$? \vdash \overline{M} : ?$$

(să găsim un context și un tip, pentru a avea o judecată legală).

Exemple:

- Pentru termenul $(\lambda z.\, \lambda u.\, z)\,(y\, x)$, am putea obține

$$\{x : \alpha, y : \alpha \to \beta\} \vdash (\lambda z : \beta.\, \lambda u : \gamma.\, z)\,(y\, x) : \gamma \to \beta$$

- Pentru termenul $x\, x$ nu putem să rezolvăm problema.

Tipuri simple

Sistemul $\lambda\!\!\rightarrow$

$$\Gamma \vdash M : \sigma$$

$$\frac{}{\Gamma \vdash x : \sigma}\ (var) \text{ dacă } x : \sigma \in \Gamma$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma.\, M) : \sigma \to \tau}\ (\to_I)$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}\ (\to_E)$$

Tipuri simple

Sistemul $\lambda\!\!\rightarrow$

$$\Gamma \vdash M : \sigma$$

$$\frac{}{\Gamma \vdash x : \sigma}\ (var) \text{ dacă } x : \sigma \in \Gamma$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma.\, M) : \sigma \to \tau}\ (\to_I)$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}\ (\to_E)$$

Sistemul $\lambda\!\!\rightarrow$ cu constrângeri

Tipuri simple

Sistemul $\lambda\!\rightarrow$

$$\Gamma \vdash M : \sigma$$

$$\frac{}{\Gamma \vdash x : \sigma}\ (var) \quad \text{dacă } x : \sigma \in \Gamma$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma.\, M) : \sigma \rightarrow \tau}\ (\rightarrow_I)$$

$$\frac{\Gamma \vdash M : \sigma \rightarrow \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}\ (\rightarrow_E)$$

Sistemul $\lambda\!\rightarrow$ cu constrângeri

$$\Gamma \vdash M : \sigma \triangleright C$$

Tipuri simple

Sistemul $\lambda\!\rightarrow$

$$\Gamma \vdash M : \sigma$$

$$\frac{}{\Gamma \vdash x : \sigma}\ (var) \quad \text{dacă } x : \sigma \in \Gamma$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma.\, M) : \sigma \rightarrow \tau}\ (\rightarrow_I)$$

$$\frac{\Gamma \vdash M : \sigma \rightarrow \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}\ (\rightarrow_E)$$

Sistemul $\lambda\!\rightarrow$ cu constrângeri

$$\Gamma \vdash M : \sigma \triangleright C$$

$$\frac{}{\Gamma \cup \{x : \tau\} \vdash x : \sigma \triangleright \{\sigma \doteq \tau\}}\ (var^*)$$

Tipuri simple

Sistemul $\lambda\!\rightarrow$

$$\Gamma \vdash M : \sigma$$

$$\frac{}{\Gamma \vdash x : \sigma}\ (var) \quad \text{dacă } x : \sigma \in \Gamma$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma.\, M) : \sigma \rightarrow \tau}\ (\rightarrow_I)$$

$$\frac{\Gamma \vdash M : \sigma \rightarrow \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}\ (\rightarrow_E)$$

Sistemul $\lambda\!\rightarrow$ cu constrângeri

$$\Gamma \vdash M : \sigma \triangleright C$$

$$\frac{}{\Gamma \cup \{x : \tau\} \vdash x : \sigma \triangleright \{\sigma \doteq \tau\}}\ (var^*)$$

$$\frac{\begin{array}{c}\Gamma, x : \sigma \vdash M : \tau' \triangleright C' \\ C = C' \cup \{\tau \doteq \sigma \rightarrow \tau'\}\end{array}}{\Gamma \vdash (\lambda x : \sigma.\, M) : \tau \triangleright C}\ (\rightarrow_I^*)$$

Tipuri simple

Sistemul $\lambda\!\!\rightarrow$

$$\Gamma \vdash M : \sigma$$

$$\frac{}{\Gamma \vdash x : \sigma}\ (var) \quad \text{dacă } x : \sigma \in \Gamma$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma.\, M) : \sigma \to \tau}\ (\to_I)$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}\ (\to_E)$$

Sistemul $\lambda\!\!\rightarrow$ cu constrângeri

$$\Gamma \vdash M : \sigma \triangleright C$$

$$\frac{}{\Gamma \cup \{x : \tau\} \vdash x : \sigma \triangleright \{\sigma \doteq \tau\}}\ (var^*)$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau' \triangleright C' \qquad C = C' \cup \{\tau \doteq \sigma \to \tau'\}}{\Gamma \vdash (\lambda x : \sigma.\, M) : \tau \triangleright C}\ (\to_I^*)$$

$$\frac{\Gamma \vdash M : \tau_1 \triangleright C_1 \qquad \Gamma \vdash N : \tau_2 \triangleright C_2 \qquad C = C_1 \cup C_2 \cup \{\tau_1 \doteq \tau_2 \to \tau\}}{\Gamma \vdash M\,N : \tau \triangleright C}\ (\to_E^*)$$

Tipuri simple

Sistemul $\lambda\!\!\rightarrow$

$$\Gamma \vdash M : \sigma$$

$$\frac{}{\Gamma \vdash x : \sigma}\ (var) \quad \text{dacă } x : \sigma \in \Gamma$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma.\, M) : \sigma \to \tau}\ (\to_I)$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}\ (\to_E)$$

σ, τ variabile de tip

Sistemul $\lambda\!\!\rightarrow$ cu constrângeri

$$\Gamma \vdash M : \sigma \triangleright C$$

$$\frac{}{\Gamma \cup \{x : \tau\} \vdash x : \sigma \triangleright \{\sigma \doteq \tau\}}\ (var^*)$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau' \triangleright C' \qquad C = C' \cup \{\tau \doteq \sigma \to \tau'\}}{\Gamma \vdash (\lambda x : \sigma.\, M) : \tau \triangleright C}\ (\to_I^*)$$

$$\frac{\Gamma \vdash M : \tau_1 \triangleright C_1 \qquad \Gamma \vdash N : \tau_2 \triangleright C_2 \qquad C = C_1 \cup C_2 \cup \{\tau_1 \doteq \tau_2 \to \tau\}}{\Gamma \vdash M\,N : \tau \triangleright C}\ (\to_E^*)$$

$\sigma, \tau, \tau', \tau_1, \tau_2$ variabile de tip

Sistemul $\lambda\!\rightarrow$ cu constrângeri

O judecată de forma $\Gamma \vdash M : \sigma \triangleright C$ este legală dacă constrângerile din C au o "soluție".

De exemplu, judecata de mai jos este legală

$$\{x : \alpha, y : \alpha \to \beta\} \vdash (\lambda z : \beta.\, \lambda u : \gamma.\, z)\,(y\,x) : \gamma \to \beta \triangleright C, \text{ unde}$$
$$C = \{\delta \doteq \beta, \tau_1' \doteq (\gamma \to \delta), \tau_1 \doteq (\beta \to \tau_1'), \sigma_1 \doteq \alpha \to \beta, \sigma_2 \doteq \alpha,$$
$$\sigma_1 \doteq \sigma_2 \to \tau_2, \tau_1 \doteq (\tau_2 \to (\gamma \to \beta))\}$$

C are "soluție" și spune, de exemplu, că ar trebui ca $\alpha = \sigma_2$ și $\beta = \delta$.

În slide-urile următoare dăm mai multe detalii despre ce înseamnă acest lucru.

Type Inference

Fie M un lambda termen fără tipuri.

Construim un context Γ_M pentru M:

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

(toate variabilele de tip X introduse mai sus sunt noi și distincte)

Adnotăm M cu tipuri pentru variabilele legate obținând $\overline{M}$ prin inducție după structura lui M astfel:

- dacă $M = x$, atunci $\overline{M} = M$
- dacă $M = M_1\ N_1$, atunci $\overline{M} = \overline{M_1}\ \overline{N_1}$
- dacă $M = \lambda x.\ N$, atunci $\overline{M} = \lambda x : X.\ \overline{N}$, unde X este o variabilă de tip nouă

Type Inference

Fie M un lambda termen fără tipuri.

Dacă există o constrângere de tipuri C_M și o variabilă de tip nouă V astfel încât

$$\Gamma_M \vdash \overline{M} : V \triangleright C_M$$

este o judecată legală, atunci M este *typable*.
(soluția o găsim prin C_M)

Type Inference - Exemplul 1

Fie $M_1 = (\lambda z.\, \lambda u.\, z)\,(y\, x)$.

Obținem $\Gamma_{M_1} = \{x : X, y : Y\}$ și $\overline{M_1} = (\lambda z : Z.\, \lambda u : U.\, z)\,(y\, x)$.

$$\frac{\Gamma_{M_1} \cup \{z : Z, u : U\} \vdash z : \delta \triangleright D \qquad C_1' = D \cup \{\tau_1' \doteq (U \to \delta)\}}{\Gamma_{M_1} \cup \{z : Z\} \vdash \lambda u : U.\, z : \tau_1' \triangleright C_1'}\ (\to_I^*)$$

$$\frac{\Gamma_{M_1} \cup \{z : Z\} \vdash \lambda u : U.\, z : \tau_1' \triangleright C_1' \qquad C_1 = C_1' \cup \{\tau_1 \doteq (Z \to \tau_1')\}}{\Gamma_{M_1} \vdash \lambda z : Z.\, \lambda u : U.\, z : \tau_1 \triangleright C_1}\ (\to_I^*)$$

$$\frac{\Gamma_{M_1} \vdash y : \sigma_1 \triangleright C_2' \qquad \Gamma_{M_1} \vdash x : \sigma_2 \triangleright C_2'' \qquad C_2 = C_2' \cup C_2'' \cup \{\sigma_1 \doteq \sigma_2 \to \tau_2\}}{\Gamma_{M_1} \vdash y\, x : \tau_2 \triangleright C_2}\ (\to_E^*)$$

$$\frac{C_{M_1} = C_1 \cup C_2 \cup \{\tau_1 \doteq (\tau_2 \to V)\}}{\Gamma_{M_1} \vdash (\lambda z : Z.\, \lambda u : U.\, z)\,(y\, x) : V \triangleright C_{M_1}}\ ($$

$D = \{\delta \doteq Z\}$

$C_1' = \{\delta \doteq Z, \tau_1' \doteq (U \to \delta)\}$

$C_1 = \{\delta \doteq Z, \tau_1' \doteq (U \to \delta), \tau_1 \doteq (Z \to \tau_1')\}$

$C_2' = \{\sigma_1 \doteq Y\}$

$C_2'' = \{\sigma_2 \doteq X\}$

$C_2 = \{\sigma_1 \doteq Y, \sigma_2 \doteq X, \sigma_1 \doteq \sigma_2 \to \tau_2\}$

$$C_{M_1} = \{\delta \doteq Z, \tau_1' \doteq (U \to \delta), \tau_1 \doteq (Z \to \tau_1'), \sigma_1 \doteq Y, \sigma_2 \doteq X, \sigma_1 \doteq \sigma_2 \to \tau_2, \tau_1 \doteq (\tau_2 \to V)\}$$

Constrângerile C_{M_1} au "soluție". Ce înseamnă asta?

Type Inference - Exemplul 2

Fie $M_2 = x\ x$.

Obținem $\Gamma_{M_2} = \{x : X\}$ și $\overline{M_2} = M_2$.

$$\frac{\{x : X\} \vdash x : \tau_1 \triangleright C_1 \quad \{x : X\} \vdash x : \tau_2 \triangleright C_2 \qquad C_M = C_1 \cup C_2 \cup \{\tau_1 \doteq \tau_2 \to V\}}{\{x : X\} \vdash (x\ x) : V \triangleright C_{M_2}} \ (\to_E^*)$$

$C_1 = \{\tau_1 \doteq X\}$

$C_2 = \{\tau_2 \doteq X\}$

$C_{M_2} = \{\tau_1 \doteq X, \tau_2 \doteq X, \tau_1 \doteq \tau_2 \to V\}$

Constrângerile C_{M_2} nu au "soluție". Ce înseamnă asta?

Type Inference - Exemplul 2

Fie $M_2 = x\ x$.
Obținem $\Gamma_{M_2} = \{x : X\}$ și $\overline{M_2} = M_2$.

$$\frac{\{x : X\} \vdash x : \tau_1 \triangleright C_1 \quad \{x : X\} \vdash x : \tau_2 \triangleright C_2 \qquad C_M = C_1 \cup C_2 \cup \{\tau_1 \doteq \tau_2 \to V\}}{\{x : X\} \vdash (x\ x) : V \triangleright C_{M_2}} \ (\to_E^*)$$

$C_1 = \{\tau_1 \doteq X\}$
$C_2 = \{\tau_2 \doteq X\}$
$C_{M_2} = \{\tau_1 \doteq X, \tau_2 \doteq X, \tau_1 \doteq \tau_2 \to V\}$

Constrângerile C_{M_2} nu au "soluție". Ce înseamnă asta?

Constrângerile au "soluție" dacă se pot unifica.

Unificare

Termeni

Alfabet:

- $\mathcal{F}$ o mulțime de simboluri de funcții de aritate cunoscută
- $\mathcal{V}$ o mulțime numărabilă de variabile
- $\mathcal{F}$ și $\mathcal{V}$ sunt disjuncte

Termeni peste $\mathcal{F}$ si $\mathcal{V}$:

$$t \ ::= \ x \mid f(t_1, \ldots, t_n)$$

- $n \geq 0$
- x este o variabilă
- f este un simbol de funcție de aritate n

Pentru ușurință, considerăm funcțiile în forma prefixată.

Termeni

Notații:

- constante: simboluri de funcții de aritate 0
- $x, y, z, \ldots$ pentru variabile
- $a, b, c, \ldots$ pentru constante
- $f, g, h, \ldots$ pentru simboluri de funcții arbitrare
- $s, t, u, \ldots$ pentru termeni
- $var(t)$ mulțimea variabilelor care apar în t
- ecuații $s \doteq t$ pentru o pereche de termeni
- $Trm_{\mathcal{F},\mathcal{V}}$ mulțimea termenilor peste $\mathcal{F}$ și $\mathcal{V}$

Termeni

Exemple:

- $f(x, g(x, a), y)$ este un termen, unde f are aritate 3, g are aritate 2, a este o constanta
- $var(f(x, g(x, a), y)) = \{x, y\}$

Legătura cu teoria tipurilor

Mulțimea tipurilor simple $\mathbb{T} = \mathbb{V} \mid \mathbb{T} \to \mathbb{T}$

În acest caz, avem alfabetul:

- $\mathcal{F} = \{\to\}$, iar aritatea lui $\to$ este 2
- $\mathcal{V} = \mathbb{V}$

Dacă avem și alte tipuri, extindem $\mathcal{F}$ cu noi simboluri. De exemplu,

- `Unit`, `Void` cu aritate 0 (deci constante)
- `Bool`, `Nat` cu aritate 0 (deci constante)
- `Maybe`, `List` cu aritate 1
- $\times$ cu aritate 2
- ...

Substituții

O substituție Θ este o funcție (parțială) de la variabile la termeni,

$$\Theta : \mathcal{V} \to Trm_{\mathcal{F},\mathcal{V}}$$

Exemplu:

În notația uzuală, $\Theta = \{a/x,\ g(w)/y, b/z\}$.

Substituția Θ este identitate pe restul variabilelor.

Notație alternativă $\Theta = \{x \mapsto a, y \mapsto g(w), z \mapsto b\}$.

Substituții

Substituțiile sunt o modalitate de a înlocui variabile cu alți termeni.

Substituțiile se aplică simultan pe toate variabilele.

Aplicarea unei substituții Θ unui termen t:

$$\Theta(t) = \begin{cases} \Theta(x), \text{ dacă } t = x \\ f(\Theta(t_1), \ldots, \Theta(t_n)), \text{ dacă } t = f(t_1, \ldots, t_n) \end{cases}$$

Exemplu:

- $\Theta = \{x \mapsto f(x, y), y \mapsto g(a)\}$
- $t = f(x, g(f(x, f(y, z))))$
- $\Theta(t) = f(f(x, y), g(f(f(x, y), f(g(a), z))))$

Substituții

Două substituții Θ_1 și Θ_2 se pot compune

$$\Theta_1; \Theta_2$$

(aplicăm întâi Θ_1, apoi Θ_2).

Exemplu:

- $t = h(u, v, x, y, z)$
- $\Theta_1 = \{x \mapsto f(y),\ y \mapsto f(a),\ z \mapsto u\}$
- $\Theta_2 = \{y \mapsto g(a),\ u \mapsto z,\ v \mapsto f(f(a))\}$
- $(\Theta_1; \Theta_2)(t) = \Theta_2(\Theta_1(t)) = \Theta_2(h(u, v, f(y), f(a), u)) =$
  $= h(z, f(f(a)), f(g(a)), f(a), z)$
- $(\Theta_2; \Theta_1)(t) = \Theta_1(\Theta_2(t)) = \Theta_1(h(z, f(f(a)), x, g(a), z))$
  $= h(u, f(f(a)), f(y), g(a), u)$

Unificare

Doi termeni t_1 și t_2 se unifică dacă există o substituție Θ astfel încât

$$\Theta(t_1) = \Theta(t_2).$$

În acest caz, Θ se numește un unificator al termenilor t_1 și t_2.

Un unificator Θ pentru t_1 și t_2 este cel mai general unificator (cmgu,mgu) dacă pentru orice alt unificator Θ' pentru t_1 și t_2, există o substituție Δ astfel încât

$$\Theta' = \Theta; \Delta.$$

Unificatori

Exemplu:

- $t = x + (y * y) = +(x, *(y, y))$
- $t' = x + (y * x) = +(x, *(y, x))$
- $\Theta = \{x \mapsto y\}$
  - $\Theta(t) = y + (y * y)$
  - $\Theta(t') = y + (y * y)$
  - Θ este cmgu
- $\Theta' = \{x \mapsto 0, y \mapsto 0\}$
  - $\Theta'(t) = 0 + (0 * 0)$
  - $\Theta'(t') = 0 + (0 * 0)$
  - $\Theta' = \Theta; \{y \mapsto 0\}$
  - Θ' este unificator, dar nu este cmgu

Quiz time!

https://tinyurl.com/C07-Quiz1

Vacanță plăcută!

Fundamentele limbajelor de programare

C08

Denisa Diaconescu
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Lambda calcul cu tipuri simple și unificare (recap)

Ce problemă am rezolvat în cursul trecut?

Type Inference

Pentru un lambda termen M fără tipuri, am adnotat termenul M cu tipuri obținând $\overline{M}$ și am rezolvat (parțial) problema

$$? \vdash \overline{M} : ?$$

(am găsit un context și un tip, pentru a avea o judecată legală).

Type Inference

Fie M un lambda termen fără tipuri.

Construim un context Γ_M pentru M:

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

(toate variabilele de tip X introduse mai sus sunt noi și distincte)

Adnotăm M cu tipuri pentru variabilele legate obținând $\overline{M}$ prin inducție după structura lui M astfel:

- dacă $M = x$, atunci $\overline{M} = M$
- dacă $M = M_1\ N_1$, atunci $\overline{M} = \overline{M_1}\ \overline{N_1}$
- dacă $M = \lambda x.\ N$, atunci $\overline{M} = \lambda x : X.\ \overline{N}$, unde X este o variabilă de tip nouă

Sistemul $\lambda\!\rightarrow$ cu constrângeri

$$\Gamma \vdash M : \sigma \triangleright C$$

$$\frac{}{\Gamma \cup \{x : \tau\} \vdash x : \sigma \triangleright \{\sigma \doteq \tau\}} \ (var^*)$$

$$\frac{\begin{array}{c}\Gamma, x : \sigma \quad \vdash \quad M : \tau' \ \triangleright \ C' \\ C = C' \cup \{\tau \doteq \sigma \rightarrow \tau'\}\end{array}}{\Gamma \vdash (\lambda x : \sigma.\, M) : \tau \triangleright C} \ (\rightarrow_I^*)$$

$$\frac{\begin{array}{c}\Gamma \vdash M : \tau_1 \triangleright C_1 \qquad \Gamma \vdash N : \tau_2 \triangleright C_2 \\ C = C_1 \cup C_2 \cup \{\tau_1 \doteq \tau_2 \rightarrow \tau\}\end{array}}{\Gamma \vdash M\,N : \tau \triangleright C} \ (\rightarrow_E^*)$$

$\sigma, \tau, \tau', \tau_1, \tau_2$ variabile de tip

Sistemul $\lambda\!\!\rightarrow$ cu constrângeri

O judecată de forma $\Gamma \vdash M : \sigma \triangleright C$ este legală dacă constrângerile din C au o "soluție".

Fie M un lambda termen fără tipuri. Dacă există o constrângere de tipuri C_M și o variabilă de tip nouă V astfel încât

$$\Gamma_M \vdash \overline{M} : V \triangleright C_M$$

este o judecată legală, atunci M este *typable*.
(soluția o găsim prin C_M)

Type Inference - Exemplul 1

Fie $M_1 = (\lambda z.\, \lambda u.\, z)\,(y\, x)$.

Obținem $\Gamma_{M_1} = \{x : X, y : Y\}$ și $\overline{M_1} = (\lambda z : Z.\, \lambda u : U.\, z)\,(y\, x)$.

$$\frac{\Gamma_{M_1} \cup \{z : Z, u : U\} \vdash z : \delta \triangleright D \qquad C_1' = D \cup \{\tau_1' \doteq (U \to \delta)\}}{\Gamma_{M_1} \cup \{z : Z\} \vdash \lambda u : U.\, z : \tau_1' \triangleright C_1'}\ (\to_I^*)$$

$$\frac{\Gamma_{M_1} \cup \{z : Z\} \vdash \lambda u : U.\, z : \tau_1' \triangleright C_1' \qquad C_1 = C_1' \cup \{\tau_1 \doteq (Z \to \tau_1')\}}{\Gamma_{M_1} \vdash \lambda z : Z.\, \lambda u : U.\, z : \tau_1 \triangleright C_1}\ (\to_I^*)$$

$$\frac{\Gamma_{M_1} \vdash y : \sigma_1 \triangleright C_2' \qquad \Gamma_{M_1} \vdash x : \sigma_2 \triangleright C_2'' \qquad C_2 = C_2' \cup C_2'' \cup \{\sigma_1 \doteq \sigma_2 \to \tau_2\}}{\Gamma_{M_1} \vdash y\, x : \tau_2 \triangleright C_2}\ (\to_E^*)$$

$$\frac{C_{M_1} = C_1 \cup C_2 \cup \{\tau_1 \doteq (\tau_2 \to V)\}}{\Gamma_{M_1} \vdash (\lambda z : Z.\, \lambda u : U.\, z)\,(y\, x) : V \triangleright C_{M_1}}\ ($$

$D = \{\delta \doteq Z\}$

$C_1' = \{\delta \doteq Z, \tau_1' \doteq (U \to \delta)\}$

$C_1 = \{\delta \doteq Z, \tau_1' \doteq (U \to \delta), \tau_1 \doteq (Z \to \tau_1')\}$

$C_2' = \{\sigma_1 \doteq Y\}$

$C_2'' = \{\sigma_2 \doteq X\}$

$C_2 = \{\sigma_1 \doteq Y, \sigma_2 \doteq X, \sigma_1 \doteq \sigma_2 \to \tau_2\}$

$$C_{M_1} = \{\delta \doteq Z, \tau_1' \doteq (U \to \delta), \tau_1 \doteq (Z \to \tau_1'), \sigma_1 \doteq Y, \sigma_2 \doteq X, \sigma_1 \doteq \sigma_2 \to \tau_2, \tau_1 \doteq (\tau_2 \to V)\}$$

Constrângerile C_{M_1} au "soluție". Ce înseamnă asta?

Type Inference - Exemplul 2

Fie $M_2 = x\,x$.
Obținem $\Gamma_{M_2} = \{x : X\}$ și $\overline{M_2} = M_2$.

$$\frac{\{x : X\} \vdash x : \tau_1 \triangleright C_1 \quad \{x : X\} \vdash x : \tau_2 \triangleright C_2 \qquad C_M = C_1 \cup C_2 \cup \{\tau_1 \doteq \tau_2 \to V\}}{\{x : X\} \vdash (x\,x) : V \triangleright C_{M_2}} \; (\to^*_E)$$

$C_1 = \{\tau_1 \doteq X\}$
$C_2 = \{\tau_2 \doteq X\}$
$C_{M_2} = \{\tau_1 \doteq X, \tau_2 \doteq X, \tau_1 \doteq \tau_2 \to V\}$

Constrângerile C_{M_2} nu au "soluție". Ce înseamnă asta?

Constrângerile au "soluție" dacă se pot unifica.

Termeni

Alfabet:

- $\mathcal{F}$ o mulțime de simboluri de funcții de aritate cunoscută
- $\mathcal{V}$ o mulțime numărabilă de variabile
- $\mathcal{F}$ și $\mathcal{V}$ sunt disjuncte

Termeni peste $\mathcal{F}$ si $\mathcal{V}$:

$$t \ ::= \ x \mid f(t_1, \ldots, t_n)$$

- $n \geq 0$
- x este o variabilă
- f este un simbol de funcție de aritate n

Termeni

Notații:

- constante: simboluri de funcții de aritate 0
- $x, y, z, \ldots$ pentru variabile
- $a, b, c, \ldots$ pentru constante
- $f, g, h, \ldots$ pentru simboluri de funcții arbitrare
- $s, t, u, \ldots$ pentru termeni
- $var(t)$ mulțimea variabilelor care apar în t
- ecuații $s \doteq t$ pentru o pereche de termeni
- $Trm_{\mathcal{F},\mathcal{V}}$ mulțimea termenilor peste $\mathcal{F}$ și $\mathcal{V}$

Legătura cu teoria tipurilor

Mulțimea tipurilor simple $\mathbb{T} = \mathbb{V} \mid \mathbb{T} \to \mathbb{T}$

În acest caz, avem alfabetul:

- $\mathcal{F} = \{\to\}$, iar aritatea lui $\to$ este 2
- $\mathcal{V} = \mathbb{V}$

Dacă avem și alte tipuri, extindem $\mathcal{F}$ cu noi simboluri. De exemplu,

- `Unit`, `Void` cu aritate 0 (deci constante)
- `Bool`, `Nat` cu aritate 0 (deci constante)
- `Maybe`, `List` cu aritate 1
- $\times$ cu aritate 2
- ...

Substituții

O substituție Θ este o funcție de la variabile la termeni,

$$\Theta : \mathcal{V} \to Trm_{\mathcal{F},\mathcal{V}}$$

Notație pentru substituții care schimbă un număr finit de variabile:

$$[u_1/x_1, u_2/x_2, \ldots, u_n/x_n]$$

Aplicarea unei substituții Θ unui termen t:

$$\Theta(t) = \begin{cases} \Theta(x), \text{ dacă } t = x \\ f(\Theta(t_1), \ldots, \Theta(t_n)), \text{ dacă } t = f(t_1, \ldots, t_n) \end{cases}$$

Notație pentru $\Theta(t)$: $t[u_1/x_1, u_2/x_2, \ldots, u_n/x_n]$

Unificare

Doi termeni t_1 și t_2 se unifică dacă există o substituție Θ astfel încât

$$\Theta(t_1) = \Theta(t_2).$$

În acest caz, Θ se numește un unificator al termenilor t_1 și t_2.

Un unificator Θ pentru t_1 și t_2 este cel mai general unificator (cmgu,mgu) dacă pentru orice alt unificator Θ' pentru t_1 și t_2, există o substituție Δ astfel încât

$$\Theta' = \Theta; \Delta.$$

De exemplu, dacă Θ este $[u_1/x_1, u_2/x_2, \ldots, u_n/x_n]$, atunci Δ este de forma $[v_1/y_1, v_2/y_2, \ldots, v_n/y_m]$ cu $x_i \neq y_j$ pentru orice alegere a lui i și j, și

$$\Theta' = [\Delta(u_1)/x_1, \ldots \Delta(u_n)/x_n, v_1/y_1, \ldots, v_m/y_m]$$

Unificatori

Exemplu:

- $t = x + (y * y) = +(x, *(y, y))$
- $t' = x + (y * x) = +(x, *(y, x))$
- $\Theta = \{x \mapsto y\}$
  - $\Theta(t) = y + (y * y)$
  - $\Theta(t') = y + (y * y)$
  - Θ este cmgu
- $\Theta' = \{x \mapsto 0, y \mapsto 0\}$
  - $\Theta'(t) = 0 + (0 * 0)$
  - $\Theta'(t') = 0 + (0 * 0)$
  - $\Theta' = \Theta; \{y \mapsto 0\}$
  - Θ' este unificator, dar nu este cmgu

Algoritmul de unificare

Algoritmul de unificare

- Pentru o mulțime finită de termeni $\{t_1, \ldots, t_n\}$, $n \geq 2$,
  algoritmul de unificare stabilește dacă există un cmgu.
- Există algoritmi mai eficienți,
  dar îl alegem pe acesta pentru simplitatea sa.
- Algoritmul lucrează cu două liste:
  - Lista soluție: S
  - Lista de rezolvat: R
- Inițial:
  - Lista soluție: $S = \emptyset$
  - Lista de rezolvat: $R = \{t_1 \doteq t_2, \ldots, t_{n-1} \doteq t_n\}$
    $\doteq$ este un simbol nou care ne ajută să formăm perechi de termeni ("ecuații")

Algoritmul de unificare

Algoritmul constă în aplicarea regulilor de mai jos:

- SCOATE
  - orice ecuație de forma $t \doteq t$ din R este eliminată.
- DESCOMPUNE
  - orice ecuație de forma $f(t_1, \ldots, t_n) \doteq f(t'_1, \ldots, t'_n)$ din R este înlocuită cu ecuațiile $t_1 \doteq t'_1, \ldots, t_n \doteq t'_n$.
- REZOLVĂ
  - orice ecuație de forma $x \doteq t$ sau $t \doteq x$ din R, unde variabila x nu apare în termenul t, este mutată sub forma $x \doteq t$ în S.
    În toate celelalte ecuații (din R și S), x este înlocuit cu t.

Algoritmul de unificare

Algoritmul se termină normal dacă $R = \emptyset$.
În acest caz, S conține cmgu.

Algoritmul este oprit cu concluzia inexistenței unui unificator dacă:

1. În R există o ecuație de forma
$$f(t_1, \ldots, t_n) \doteq g(t'_1, \ldots, t'_k) \text{ cu } f \neq g.$$
2. În R există o ecuație de forma $x \doteq t$ sau $t \doteq x$ și variabila x apare în termenul t.

Algoritmul de unificare - schemă

	Lista soluție S	Lista de rezolvat R
Inițial	$\emptyset$	$t_1 \doteq t'_1, \ldots, t_n \doteq t'_n$
SCOATE	S	$R', t \doteq t$
	S	R'
DESCOMPUNE	S	$R', f(t_1, \ldots, t_n) \doteq f(t'_1, \ldots, t'_n)$
	S	$R', t_1 \doteq t'_1, \ldots t_n \doteq t'_n$
REZOLVĂ	S	$R', x \doteq t$ sau $t \doteq x$, x nu apare în t
	$x \doteq t, S[t/x]$	$R'[t/x]$
Final	S	$\emptyset$

$S[t/x]$: în toate ecuațiile din S, x este înlocuit cu t

Exemplul 1

Ecuațiile $\{g(y) \doteq x,\ f(x, h(x), y) \doteq f(g(z), w, z)\}$ au cmgu?

Exemplul 1

Ecuațiile $\{g(y) \doteq x,\ f(x, h(x), y) \doteq f(g(z), w, z)\}$ au cmgu?

S	R	
$\emptyset$	$g(y) \doteq x,\ f(x, h(x), y) \doteq f(g(z), w, z)$	REZOLVĂ
$x \doteq g(y)$	$f(g(y), h(g(y)), y) \doteq f(g(z), w, z)$	DESCOMPUNE
$x \doteq g(y)$	$g(y) \doteq g(z),\ h(g(y)) \doteq w,\ y \doteq z$	REZOLVĂ
$w \doteq h(g(y))$, $x \doteq g(y)$	$g(y) \doteq g(z),\ y \doteq z$	REZOLVĂ
$y \doteq z, x \doteq g(z)$, $w \doteq h(g(z))$	$g(z) \doteq g(z)$	SCOATE
$y \doteq z, x \doteq g(z)$, $w \doteq h(g(z))$	$\emptyset$	

$\Theta = \{y \mapsto z,\ x \mapsto g(z),\ w \mapsto h(g(z))\}$ este cmgu.

Exemplul 2

Ecuațiile $\{g(y) \doteq x,\ f(x, h(y), y) \doteq f(g(z), b, z)\}$ au cmgu?

Exemplul 2

Ecuațiile $\{g(y) \doteq x,\ f(x, h(y), y) \doteq f(g(z), b, z)\}$ au cmgu?

S	R	
$\emptyset$	$g(y) \doteq x,\ f(x, h(y), y) \doteq f(g(z), b, z)$	REZOLVĂ
$x \doteq g(y)$	$f(g(y), h(y), y) \doteq f(g(z), b, z)$	DESCOMPUNE
$x \doteq g(y)$	$g(y) \doteq g(z),\ h(y) \doteq b,\ y \doteq z$	- EȘEC -

- h și b sunt simboluri de funcții diferite!
- Nu există unificator pentru acești termeni.

Exemplul 3

Ecuațiile $\{g(y) \doteq x,\ f(x, h(x), y) \doteq f(y, w, z)\}$ au cmgu?

Exemplul 3

Ecuațiile $\{g(y) \doteq x,\ f(x, h(x), y) \doteq f(y, w, z)\}$ au cmgu?

S	R	
$\emptyset$	$g(y) \doteq x,\ f(x, h(x), y) \doteq f(y, w, z)$	REZOLVĂ
$x \doteq g(y)$	$f(g(y), h(g(y)), y) \doteq f(y, w, z)$	DESCOMPUNE
$x \doteq g(y)$	$g(y) \doteq y,\ h(g(y)) \doteq w,\ y \doteq z$	- EȘEC -

- În ecuația $g(y) \doteq y$, variabila y apare în termenul $g(y)$.
- Nu există unificator pentru aceste ecuații.

Exemplul 4

Înapoi la constrângerea obținută când am vorbit de *type inference* pentru termenul $M_1 = (\lambda z.\, \lambda u.\, z)\,(y\, x)$.

Am obținut constrângerile

$$C_{M_1} = \{\delta \doteq Z, \tau_1' \doteq (U \to \delta), \tau_1 \doteq (Z \to \tau_1'), \sigma_1 \doteq Y, \sigma_2 \doteq X,$$
$$\sigma_1 \doteq \sigma_2 \to \tau_2, \tau_1 \doteq (\tau_2 \to V)\}$$

- $\to$ simbol de funcție de aritate 2
- $\delta, \tau_1, \tau_1', \tau_2, \sigma_1, \sigma_2, X, Y, Z, U, V$ variabile

Exemplul 4 (cont.)

S	R	
$\emptyset$	$\delta \doteq Z,\ \tau_1' \doteq (U \to \delta),\ \tau_1 \doteq (Z \to \tau_1'),\ \sigma_1 \doteq Y$ $\sigma_2 \doteq X,\ \sigma_1 \doteq \sigma_2 \to \tau_2,\ \tau_1 \doteq (\tau_2 \to V)$	REZ.
$\delta \doteq Z$	$\tau_1' \doteq (U \to Z),\ \tau_1 \doteq (Z \to \tau_1'),\ \sigma_1 \doteq Y$ $\sigma_2 \doteq X,\ \sigma_1 \doteq \sigma_2 \to \tau_2,\ \tau_1 \doteq (\tau_2 \to V)$	REZ.
$\delta \doteq Z,\ \tau_1' \doteq (U \to Z)$	$\tau_1 \doteq (Z \to (U \to Z)),\ \sigma_1 \doteq Y$ $\sigma_2 \doteq X,\ \sigma_1 \doteq \sigma_2 \to \tau_2,\ \tau_1 \doteq (\tau_2 \to V)$	REZ.
$\delta \doteq Z,\ \tau_1' \doteq (U \to Z),$ $\tau_1 \doteq (Z \to (U \to Z))$	$\sigma_1 \doteq Y,\ \sigma_2 \doteq X,\ \sigma_1 \doteq \sigma_2 \to \tau_2,$ $(Z \to (U \to Z)) \doteq (\tau_2 \to V)$	DESC.
$\delta \doteq Z,\ \tau_1' \doteq (U \to Z),$ $\tau_1 \doteq (Z \to (U \to Z))$	$\sigma_1 \doteq Y,\ \sigma_2 \doteq X,\ \sigma_1 \doteq \sigma_2 \to \tau_2,$ $Z \doteq \tau_2,\ U \to Z \doteq V$	REZ.
$\delta \doteq Z,\ \tau_1' \doteq (U \to Z),$ $\tau_1 \doteq (Z \to (U \to Z)),$ $\sigma_1 \doteq Y$	$\sigma_2 \doteq X,\ Y \doteq \sigma_2 \to \tau_2,$ $Z \doteq \tau_2,\ U \to Z \doteq V$	REZ.
$\delta \doteq Z,\ \tau_1' \doteq (U \to Z),$ $\tau_1 \doteq (Z \to (U \to Z)),$ $\sigma_1 \doteq Y, \sigma_2 \doteq X$	$Y \doteq X \to \tau_2,$ $Z \doteq \tau_2,\ U \to Z \doteq V$	REZ.

Exemplul 4 (cont.)

S	R	
$\delta \doteq Z,\ \tau_1' \doteq (U \to Z),$ $\tau_1 \doteq (Z \to (U \to Z)),$ $\sigma_1 \doteq Y, \sigma_2 \doteq X,\ \tau_2 \doteq Z$	$Y \doteq X \to Z,\ U \to Z \doteq V$	REZ.
$\delta \doteq Z,\ \tau_1' \doteq (U \to Z),$ $\tau_1 \doteq (Z \to (U \to Z)),$ $\sigma_1 \doteq X \to Z, \sigma_2 \doteq X,\ \tau_2 \doteq Z$ $Y \doteq X \to Z$	$U \to Z \doteq V$	REZ.
$\delta \doteq Z,\ \tau_1' \doteq (U \to Z),$ $\tau_1 \doteq (Z \to (U \to Z)),$ $\sigma_1 \doteq X \to Z, \sigma_2 \doteq X,\ \tau_2 \doteq Z$ $Y \doteq X \to Z,\ V \doteq U \to Z$		

Constrângerile se pot unifica!

Exemplul 5

Înapoi la constrângerea obținută când am vorbit de *type inference* pentru termenul $M_2 = x\ x$.

Am obținut constrângerile

$$C_{M_2} = \{\tau_1 \doteq X, \tau_2 \doteq X, \tau_1 \doteq \tau_2 \to V\}$$

- $\to$ simbol de funcție de aritate 2
- τ_1, τ_2, V variabile

Exemplul 5 (cont.)

S	R	
$\emptyset$	$\tau_1 \doteq X,\ \tau_2 \doteq X,\ \tau_1 \doteq \tau_2 \to V$	REZ.
$\tau_1 \doteq X$	$\tau_2 \doteq X,\ X \doteq \tau_2 \to V$	REZ.
$\tau_1 \doteq X,\ \tau_2 \doteq X$	$X \doteq X \to V$	- EȘEC -

- În ecuația $X \doteq X \to V$, variabila X apare în termenul $X \to V$.
- Nu există unificator pentru aceste ecuații.

Exerciții

Considerăm

- x, y, z, u, v variabile,
- a, b, c simboluri de constantă,
- h, g simboluri de funcție de aritate 1,
- f simbol de funcție de aritate 2,
- p simbol de funcție de aritate 3.

Aplicați algoritmul de unificare de mai sus pentru termenii:

1. $p(a, x, h(g(y)))$ și $p(z, h(z), h(u))$
2. $f(h(a), g(x))$ și $f(y, y)$
3. $p(a, x, g(x))$ și $p(a, y, y)$
4. $p(x, y, z)$ și $p(u, f(v, v), u)$

Exerciții - rezolvări

1.

S	R	
$\emptyset$	$p(a, x, h(g(y))) = p(z, h(z), h(u))$	DESCOMPUNE
$\emptyset$	$a \doteq z, x \doteq h(z), h(g(y)) \doteq h(u)$	REZOLVĂ
$z \doteq a$	$x \doteq h(a), h(g(y)) \doteq h(u)$	REZOLVĂ
$z \doteq a, x \doteq h(a)$	$h(g(y)) \doteq h(u)$	DESCOMPUNE
$z \doteq a, x \doteq h(a)$	$g(y) \doteq u$	REZOLVĂ
$z \doteq a, x \doteq h(a), u \doteq g(y)$	$\emptyset$	

$\Theta = \{z/a,\ x/h(a),\ u/g(y)\}$ este cmgu.

Exerciții - rezolvări

2.

S	R	
$\emptyset$	$f(h(a), g(x)) \doteq f(y, y)$	DESCOMPUNE
$\emptyset$	$y \doteq h(a), y \doteq g(x)$	REZOLVĂ
$y \doteq h(a)$	$g(x) \doteq h(a)$	EȘEC

Nu există unificator!

Exerciții - rezolvări

3.

S	R	
$\emptyset$	$p(a, x, g(x)) \dot{=} p(a, y, y)$	DESCOMPUNE
$\emptyset$	$a \dot{=} a, x \dot{=} y, y \dot{=} g(x)$	SCOATE
$\emptyset$	$x \dot{=} y, y \dot{=} g(x)$	REZOLVĂ
$x \dot{=} y$	$y \dot{=} g(y)$	EȘEC

Nu există unificator!

Exerciții - rezolvări

4.

S	R	
$\emptyset$	$p(x,y,z) \doteq p(u,f(v,v),u)$	DESCOMPUNE
$\emptyset$	$x \doteq u, y \doteq f(v,v), z \doteq u$	REZOLVĂ
$x \doteq u$	$y \doteq f(v,v), z \doteq u$	REZOLVĂ
$y \doteq f(v,v), x \doteq u$	$z \doteq u$	REZOLVĂ
$z \doteq u, y \doteq f(v,v), x \doteq u$		

$\Theta = \{z/u,\ y/f(v,v),\ x/u\}$ este cmgu.

Pe data viitoare!

Fundamentele limbajelor de programare

C09

Denisa Diaconescu
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Programare logică & Prolog

Programare logică

Programarea logică este o paradigmă de programare bazată pe logică.

Unul din sloganurile programării logice:

Program = Logică + Control (R. Kowalski)

Programarea logică poate fi privită ca o deducție controlată.

Un program scris într-un limbaj de programare logică este

o listă de formule într-o logică

ce exprimă fapte și reguli despre o problemă.

Programare logică

Exemple de limbaje de programare logică:

- Prolog
- Answer set programming (ASP)
- Datalog

Programare logică - în mod idealist

- Un "program logic" este o colecție de proprietăți presupuse (sub formă de formule logice) despre lumea programului.
- Programatorul furnizează și o proprietate (o formula logică) care poate să fie sau nu adevărată în lumea respectivă (întrebare, query).
- Sistemul determină dacă proprietatea aflată sub semnul întrebării este o consecință a proprietăților presupuse în program.
- Programatorul nu specifică metoda prin care sistemul verifică dacă întrebarea este sau nu consecință a programului.

Exemplu de program logic

$$\begin{aligned} \texttt{oslo} &\rightarrow \texttt{windy} \\ \texttt{oslo} &\rightarrow \texttt{norway} \\ \texttt{norway} &\rightarrow \texttt{cold} \\ \texttt{cold} \wedge \texttt{windy} &\rightarrow \texttt{winterIsComing} \\ & \quad\ \texttt{oslo} \end{aligned}$$

Exemplu de întrebare. Este adevărat `winterIsComing`?

Prolog

- bazat pe logica clauzelor Horn
- semantica operațională este bazată pe rezoluție
- este Turing complet

Program:

```
windy :- oslo.
norway :- oslo.
cold :- norway.
winterIsComing :- windy, cold.
oslo.
```

Intrebare:

```
?- winterIsComing.
true
```

http://swish.swi-prolog.org/

Sintaxă: constante, variabile, termeni compuși

- Atomi: `brian, 'Brian Griffin', brian_griffin`
- Numere: `23, 23.03,-1`

  Atomii și numerele sunt constante.
- Variabile: `X, Griffin, _family`
- Termeni compuși: `father(peter, stewie_griffin)`,

  `and(son(stewie,peter), daughter(meg,peter))`
  - forma generală: atom(termen,. . ., termen)
  - atom-ul care denumește termenul se numește functor
  - numărul de argumente se numește aritate

Sintaxă Prolog

Exercițiu. Care din următoarele șiruri de caractere sunt constante și care sunt variabile în Prolog?

- vINCENT
- Footmassage
- variable23
- Variable2000
- big_kahuna_burger
- 'big kahuna burger'
- big kahuna burger
- 'Jules'
- _Jules
- '_Jules'

Sintaxă Prolog

Exercițiu. Care din următoarele șiruri de caractere sunt constante și care sunt variabile în Prolog?

- vINCENT – constantă
- Footmassage – variabilă
- variable23 – constantă
- Variable2000 – variabilă
- big_kahuna_burger – constantă
- 'big kahuna burger' – constantă
- big kahuna burger – nici una, nici alta
- 'Jules' – constantă
- _Jules – variabilă
- '_Jules' – constantă

Program în Prolog = bază de cunoștințe

Exemplu. Un program în Prolog:

```
father(peter,meg).
father(peter,stewie).

mother(lois,meg).
mother(lois,stewie).

griffin(peter).
griffin(lois).

griffin(X) :- father(Y,X),  griffin(Y).
```

Un program în Prolog este o bază de cunoștințe (Knowledge Base).

Program în Prolog = mulțime de predicate

Practic, gândim un program în Prolog ca o mulțime de predicate cu ajutorul cărora descriem *lumea* (*universul*) programului respectiv.

Exemplu.

```
father(peter,meg).
father(peter,stewie).

mother(lois,meg).
mother(lois,stewie).

griffin(peter).
griffin(lois).

griffin(X) :- father(Y,X),  griffin(Y).
```

Predicate:

father/2

mother/2

griffin/1

Un program în Prolog

Program

- Un program în Prolog este format din reguli de forma

  Head :- Body.

- Head este un predicat, iar Body este o secvență de predicate separate prin virgulă.

- Regulile fără `Body` se numesc fapte.

Exemple.

- Exemplu de regulă:
  ```
  griffin(X) :- father(Y,X), griffin(Y).
  ```
- Exemplu de fapt:
  ```
  father(peter,meg).
  ```

Interpretarea din punctul de vedere al logicii

Operatorul **:-** este implicația logică $\leftarrow$.

Exemplu. `comedy(X) :- griffin(X).`

dacă `griffin(X)` este adevărat, atunci `comedy(X)` este adevărat.

Virgula , este conjuncția $\wedge$.

Exemplu. `griffin(X) :- father(Y,X), griffin(Y).`

dacă `father(Y,X)` și `griffin(Y)` sunt adevărate,

atunci `griffin(X)` este adevărat.

Interpretarea din punctul de vedere al logicii

Mai multe reguli cu același Head definesc același predicat, între definiții fiind un sau logic.

Exemplu.

```
comedy(X) :- family_guy(X).
comedy(X) :- south_park(X).
comedy(X) :- disenchantment(X).
```

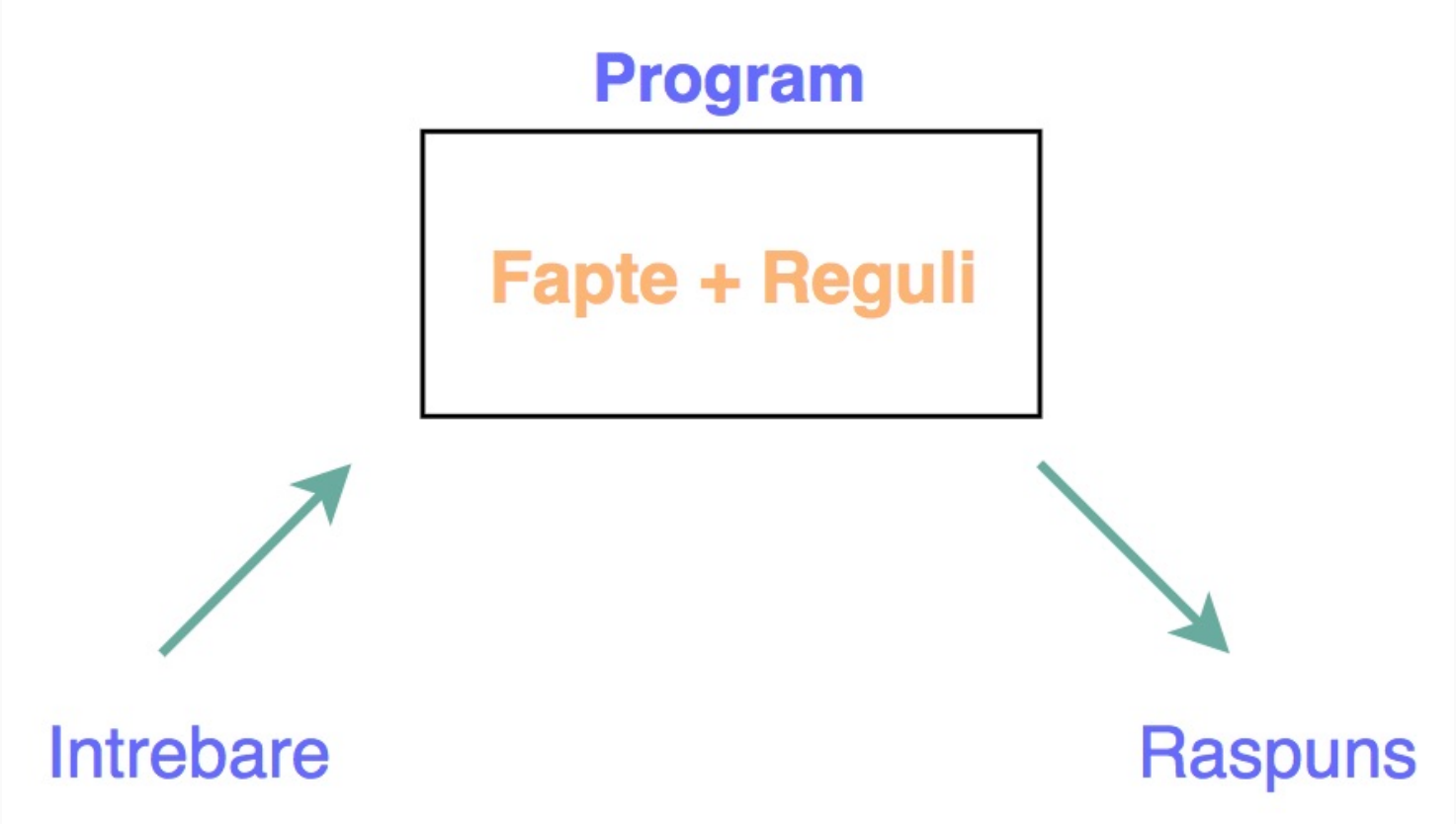
dacă `family_guy(X)` este adevărat sau `south_park(X)` este adevărat sau `disenchantment(X)` este adevărat,

atunci `comedy(X)` este adevărat.

Un program în Prolog

Cum folosim un program în Prolog?

Întrebări în Prolog



Întrebări și ținte în Prolog

- Prolog poate răspunde la întrebări legate de consecințele relațiilor descrise într-un program în Prolog.
- Întrebările sunt de forma:

$$\texttt{?- predicat}_1\texttt{(...),...,predicat}_n\texttt{(...).}$$

- Prolog verifică dacă întrebarea este o consecință a relațiilor definite în program.
- Dacă este cazul, Prolog caută valori pentru variabilele care apar în întrebare astfel încât întrebarea să fie o consecință a relațiilor din program.
- Un predicat care este analizat pentru a răspunde la o întrebare se numește țintă (goal).

Întrebări în Prolog

Prolog poate da 2 tipuri de răspunsuri:

- `false` – dacă întrebarea nu este o consecință a programului.
- `true` sau valori pentru variabilele din întrebare dacă întrebarea este o consecință a programului.

Example

```
?- griffin(meg)
true
?- griffin(glenn)
false
```

```
?- griffin(X)
X = petr ;
X = lois ;
X = meg ;
X = stewie ;
false
```

Cum găsește Prolog răspunsul

Pentru a găsi un răspuns,
Prolog încearcă regulile în ordinea apariției lor.

Exemplu. Să presupunem că avem programul:

```
foo(a).  foo(b).  foo(c).
```

și că punem întrebarea:

```
?- foo(X).
X = a.
```

Pentru a răspunde la întrebare se caută o potrivire (unificator) între scopul `foo(X)` și baza de cunoștințe. Răspunsul este substituția care realizează unificarea, în cazul nostru `X = a`.

Răspunsul la întrebare este găsit prin unificare!

Cum găsește Prolog răspunsul

Exemplu. Să presupunem că avem programul:

```
foo(a).  foo(b).  foo(c).
```

și că punem întrebările:

```
?- foo(X).
X = a.

?- foo(d).
false
```

Dacă nu se poate face unificarea, răspunsul este false.

Cum găsește Prolog răspunsul

Exemplu. Să presupunem că avem programul:

```
foo(a).  foo(b).  foo(c).
```

și că punem întrebarea:

```
?- foo(X).
X = a.
```

Dacă dorim mai multe răspunsuri, tastăm `;`

```
?- foo(X).
X = a ;
X = b ;
X = c.
```

Cum găsește Prolog răspunsul

Exemplu.

Să presupunem că avem programul:

```
foo(a).
foo(b).
foo(c).
```

și că punem întrebarea:

```
?- foo(X).
```

```
?- trace.
true.

[trace]  ?- foo(X).
   Call: (8) foo(_4556) ? creep
   Exit: (8) foo(a) ? creep
X = a ;
   Redo: (8) foo(_4556) ? creep
   Exit: (8) foo(b) ? creep
X = b ;
   Redo: (8) foo(_4556) ? creep
   Exit: (8) foo(c) ? creep
X = c.
```

Cum găsește Prolog răspunsul

Pentru a găsi un raspuns, Prolog redenumește variabilele.

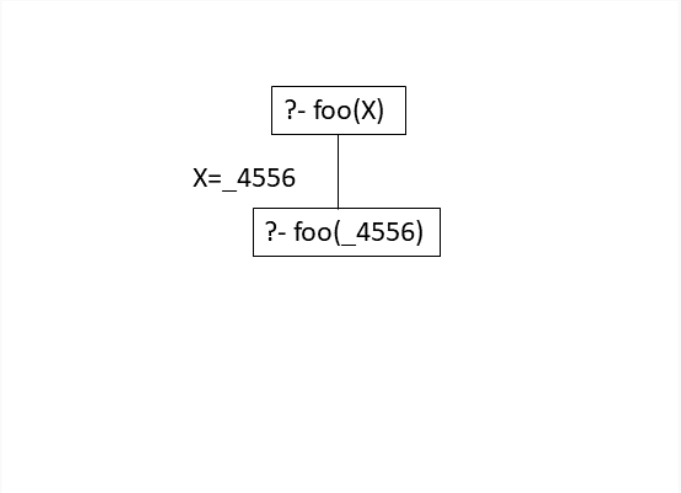
Exemplu.

Să presupunem că avem programul:

```
foo(a).
foo(b).
foo(c).
```

și că punem întrebarea:

```
?- foo(X).
```

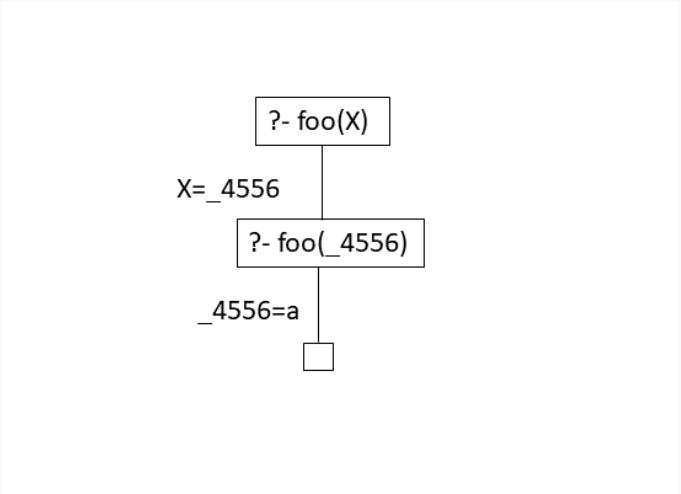


Cum găsește Prolog răspunsul

Exemplu.

Să presupunem că avem programul:

```
foo(a).
foo(b).
foo(c).
```



și că punem întrebarea:

```
?- foo(X).
```

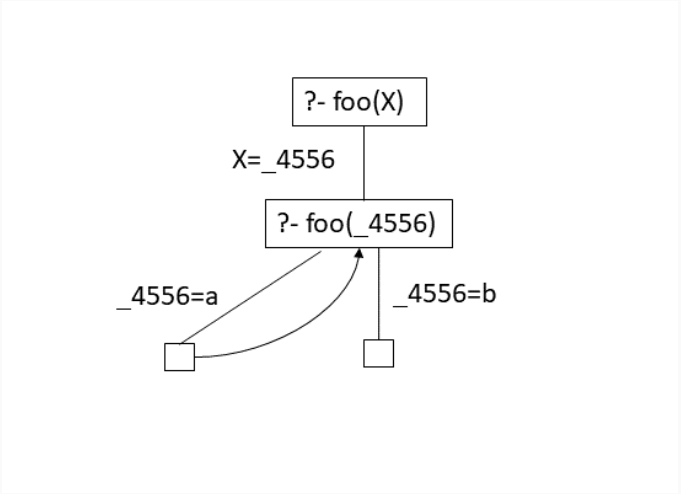
În acest moment, a fost găsită prima soluție: `X=_4556=a`.

Cum găsește Prolog răspunsul

Exemplu.

Să presupunem că avem programul:

```
foo(a).
foo(b).
foo(c).
```



și că punem următoarea întrebare:

```
?- foo(X).
```

Dacă se dorește încă un răspuns, atunci se face un pas înapoi în arborele de căutare și se încearcă satisfacerea țintei cu o nouă valoare.

Cum găsește Prolog răspunsul

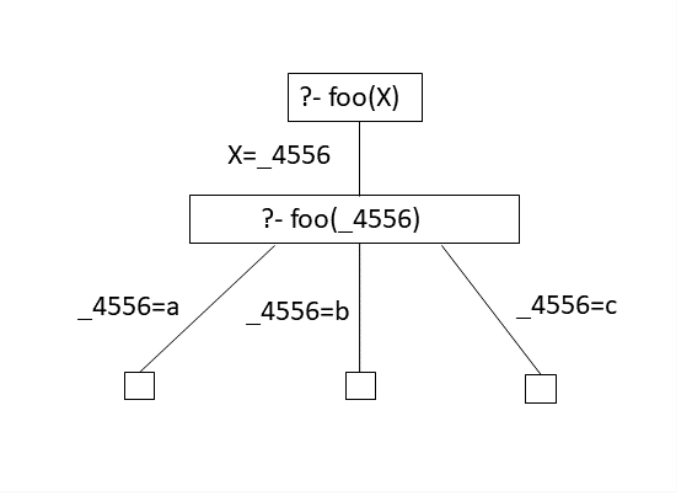
Exemplu.

Să presupunem că avem programul:

```
foo(a).
foo(b).
foo(c).
```

și că punem întrebarea:

```
?- foo(X).
```



arborele de căutare

Cum găsește Prolog răspunsul

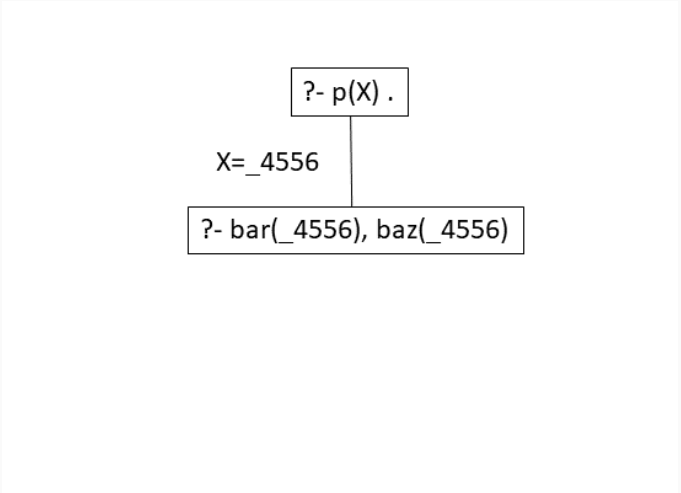
Exemplu.

Să presupunem că avem programul:

```
bar(b).
bar(c).
baz(c).
```

și că punem întrebarea:

```
?- bar(X),baz(X).
```



Cum găsește Prolog răspunsul

Prolog se întoarce la ultima alegere dacă o subțintă eșuează.

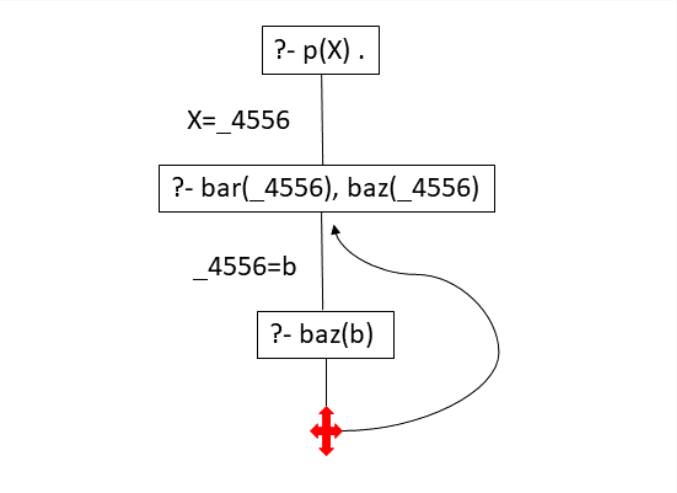
Exemplu.

Să presupunem că avem programul:

```
bar(b).
bar(c).
baz(c).
```

și că punem întrebarea:

```
?- bar(X),baz(X).
```



Cum găsește Prolog răspunsul

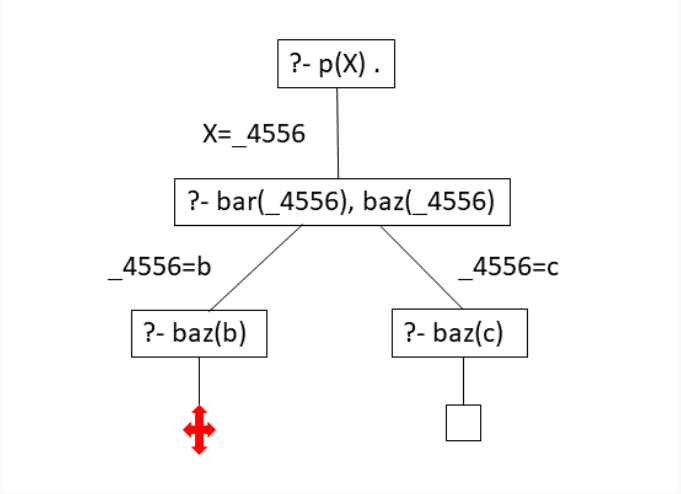
Exemplu.

Să presupunem că avem programul:

```
bar(b).
bar(c).
baz(c).
```

și că punem întrebarea:

```
?- bar(X),baz(X).
```



Soluția găsită este: `X=_4556=c`.

Cum găsește Prolog răspunsul

Ce se întâmplă dacă schimbăm ordinea regulilor?

Exemplu.

Să presupunem că avem programul:

```
bar(c).
bar(b).
baz(c).
```

și că punem întrebarea:

```
?- bar(X),baz(X).
```

Cum găsește Prolog răspunsul

Ce se întâmplă dacă schimbăm ordinea regulilor?

Exemplu.

Să presupunem că avem programul:

```
bar(c).
bar(b).
baz(c).
```

și că punem întrebarea:

```
?- bar(X),baz(X).
X = c ;
false
```

Vă explicați ce s-a întâmplat? Desenați arborele de căutare!

Problema colorării hărților

Să se coloreze o hartă dată cu o mulțime de culori dată astfel încât oricare două țări vecine să fie colorate diferit.

Cum modelăm această problemă în Prolog?

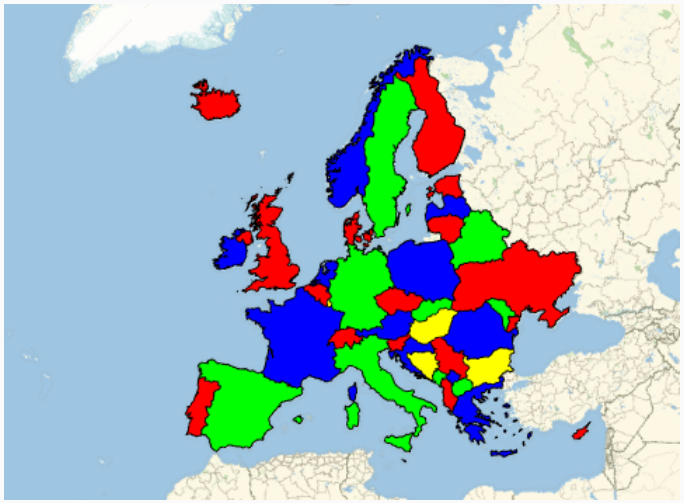
Problema colorării hărților

Să se coloreze o hartă dată cu o mulțime de culori dată astfel încât oricare două țări vecine să fie colorate diferit.

Cum modelăm această problemă în Prolog?

Trebuie să definim:

- culorile
- harta
- constrângerile



Sursa imaginii

Problema colorării hărților

Definim culorile, harta și constrângerile.

```
culoare(albastru).
culoare(rosu).
culoare(verde).
culoare(galben).

harta(RO,SE,MD,UA,BG,HU) :- vecin(RO,SE), vecin(RO,UA),
                            vecin(RO,MD), vecin(RO,BG),
                            vecin(RO,HU), vecin(UA,MD),
                            vecin(BG,SE), vecin(SE,HU).

vecin(X,Y) :- culoare(X), culoare(Y), X \== Y.
```

Problema colorării hărților

Definim culorile, harta și constrângerile. Cum punem întrebarea?

```
culoare(albastru).
culoare(rosu).
culoare(verde).
culoare(galben).

harta(RO,SE,MD,UA,BG,HU) :- vecin(RO,SE), vecin(RO,UA),
                            vecin(RO,MD), vecin(RO,BG),
                            vecin(RO,HU), vecin(UA,MD),
                            vecin(BG,SE), vecin(SE,HU).

vecin(X,Y) :- culoare(X), culoare(Y), X \== Y.
```

Problema colorării hărților

Definim culorile, harta și constrângerile. Cum punem întrebarea?

```
culoare(albastru).
culoare(rosu).
culoare(verde).
culoare(galben).

harta(RO,SE,MD,UA,BG,HU) :- vecin(RO,SE), vecin(RO,UA),
                            vecin(RO,MD), vecin(RO,BG),
                            vecin(RO,HU), vecin(UA,MD),
                            vecin(BG,SE), vecin(SE,HU).

vecin(X,Y) :- culoare(X), culoare(Y), X \== Y.

?- harta(RO,SE,MD,UA,BG,HU).
```

Problema colorării hărților

Ce răspuns primim?

```
?- harta(RO,SE,MD,UA,BG,HU).
RO = albastru,
SE = UA, UA = rosu,
MD = BG, BG = HU, HU = verde
```

Liste în Prolog

- O listă în Prolog este un șir de elemente, separate prin virgulă, între paranteze drepte:

  ```
  [1,cold, parent(jon),[winter,is,coming],X]
  ```

- O listă poate conține termeni de orice fel.
- Ordinea termenilor din listă are importanță:

  ```
  ?- [1,2] == [2,1] .
  false
  ```

- Lista vidă se notează `[]`.
- Simbolul `|` desemnează coada listei:

  ```
  ?- [1,2,3,4,5,6] = [X|T].
  X = 1, T = [2, 3, 4, 5, 6].

  ?- [1,2,3|[4,5,6]] == [1,2,3,4,5,6].
  true.
  ```

Liste în Prolog

Exerciții.

1. Definiți un predicat care verifică că un termen este lista.

Liste în Prolog

Exerciții.

1. Definiți un predicat care verifică că un termen este lista.

```
is_list([]).
is_list([_ | T]) :- is_list(T).
```

Liste în Prolog

Exerciții.

1. Definiți un predicat care verifică că un termen este lista.

```
is_list([]).
is_list([_ | T]) :- is_list(T).
```

2. Definiți predicate care verifică dacă un termen este primul element, ultimul element sau coada unei liste.

Liste în Prolog

Exerciții.

1. Definiți un predicat care verifică că un termen este lista.

```
is_list([]).
is_list([_ | T]) :- is_list(T).
```

2. Definiți predicate care verifică dacă un termen este primul element, ultimul element sau coada unei liste.

```
head([X|_],X).

last([X],X).
last([_|T],Y):- last(T,Y).

tail([],[]).
tail([_|T],T).
```

Pe data viitoare!

Fundamentele limbajelor de programare

C10

Denisa Diaconescu
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Programare Logică
Logica Clauzelor Horn

Program în Prolog = mulțime de predicate

Un exemplu de program în Prolog din cursul trecut:

```
father(peter,meg).
father(peter,stewie).

mother(lois,meg).
mother(lois,stewie).

griffin(peter).
griffin(lois).

griffin(X) :- father(Y,X),  griffin(Y).
```

Predicate:

father/2

mother/2

griffin/1

Spre logica din spatele Prologului

Pentru a putea modela universul programului, fixăm un alfabet/signatură/vocabular:

- o mulțime $\mathbf{R}$ de simboluri de relații/predicate
- o mulțime $\mathbf{F}$ de simboluri de operații/funcții
- o funcție aritate $ar : \mathbf{F} \cup \mathbf{R} \to \mathbb{N}$
- Fie $\mathbf{C} \subseteq \mathbf{F}$ mulțimea simbolurilor de operații de aritate 0, numite și simboluri de constante.

Notăm cu $\mathbf{R}_n/\mathbf{F}_n$ mulțimea simbolurilor de relații/operații de aritate n.
Observație: $\mathbf{F}_0 = C$

Exemplu.

- $\mathbf{R} = \mathbf{R}_1 \uplus \mathbf{R}_2$, unde $\mathbf{R}_1 = \{P\}$ și $\mathbf{R}_2 = \{R\}$
- $\mathbf{F} = \mathbf{F}_0 \uplus \mathbf{F}_2$, unde $\mathbf{F}_0 = C = \{c\}$ și $\mathbf{F}_2 = \{f\}$

Sintaxa Prolog

- Sintaxa Prolog nu face diferență între simboluri de operații și simboluri de predicate!

- Dar este important când ne uităm la teoria corespunzătoare programului în logică să facem această distincție.

- În sintaxa Prolog
  - termenii compuși sunt predicate: `father(peter, meg)`
  - operatorii sunt funcții: `+, *, mod`

Termeni

Fixăm o mulțime numărabilă de variabile V.

Definim termenii inductiv astfel:

- orice variabilă este un termen
- orice simbol de constantă este un termen
- dacă $f \in \mathbf{F}_n$, $n > 0$ și $t_1, \ldots, t_n$ sunt termeni, atunci $f(t_1, \ldots, t_n)$ este termen.

Exemple: $c,\ x_1,\ f(x_1, c),\ f(f(x_2, x_2), c)$
unde $c \in \mathbf{C}$ reprezintă o constantă, $x_1, x_2 \in V$ sunt variabile, iar $f \in \mathbf{F}_2$ reprezintă un simbol de funcție de aritate 2.

Formule atomice

Formulele atomice sunt definite astfel:

dacă $R \in \mathbf{R}_0$, atunci R este formulă atomică

dacă $R \in \mathbf{R}_n$, $n > 0$ și $t_1, \ldots, t_n$ sunt termeni,
atunci $R(t_1, \ldots, t_n)$ este formulă atomică.

Exemple: $P(f(x_1, c)),\ R(c, x_2)$
unde $c \in \mathbf{C}$, $x_1, x_2 \in V$, $f \in \mathbf{F}_2$, $P \in \mathbf{R}_1$, iar $R \in \mathbf{R}_2$.

Formulele logicii de ordinul I

Definim formulele astfel:

- orice formulă atomică este o formulă
- dacă φ este o formulă, atunci $\neg\varphi$ este o formulă
- dacă φ și ψ sunt formule, atunci $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \to \psi$ sunt formule
- dacă φ este o formulă și x este o variabilă, atunci $\forall x\, \varphi$, $\exists x\, \varphi$ sunt formule

Formulele logicii de ordinul I

Definim formulele astfel:

- orice formulă atomică este o formulă
- dacă φ este o formulă, atunci $\neg\varphi$ este o formulă
- dacă φ și ψ sunt formule, atunci $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \to \psi$ sunt formule
- dacă φ este o formulă și x este o variabilă, atunci $\forall x\, \varphi$, $\exists x\, \varphi$ sunt formule

Exemplu:

$$P(f(x_1, c)), \quad P(x_1) \vee P(c), \quad \forall x_1\, P(x_1), \quad \forall x_2\, R(x_2, x_1)$$

unde $c \in \mathbf{C}$, $x_1, x_2 \in V$, $f \in \mathbf{F}_2$, $P \in \mathbf{R}_1$, iar $R \in \mathbf{R}_2$.

Exercițiu

Fie alfabetul definit prin $\mathbf{R} = \{<\}$, $\mathbf{F} = \{s, +\}$, $\mathbf{C} = \{0\}$ și $ari(s) = 1$, $ari(+) = ari(<) = 2$.

Dați exemple de 3 termeni, 3 formule atomice și 3 formule.

Exercițiu

Fie alfabetul definit prin $\mathbf{R} = \{<\}$, $\mathbf{F} = \{s, +\}$, $\mathbf{C} = \{0\}$ și $ari(s) = 1$, $ari(+) = ari(<) = 2$.

Dați exemple de 3 termeni, 3 formule atomice și 3 formule.

Exemple de termeni:

$$0,\ x,\ s(0),\ s(s(0)),\ s(x),\ s(s(x)),\ \ldots,$$

$$+(0,0),\ +(s(s(0)),+(0,s(0))),\ +(x,s(0)),\ +(x,s(x)),\ \ldots,$$

Exemple de formule atomice:

$$<(0,0),\ <(x,0),\ <(s(s(x)),s(0)),\ \ldots$$

Exemple de formule:

$$\forall x\, \forall y\ <(x,+(x,y)),\ \forall x\ <(x,s(x))$$

Literali

Un literal este o formulă atomică sau negația unei formule atomice.

O formulă este în formă normală conjunctivă (FNC) dacă este o conjuncție de disjuncții de literali.

Exemplu:

$(P(f(x_1, c)) \vee R(c, x_2)) \ \wedge \ \neg R(x_1, x_2) \ \wedge \ (R(x_1, x_1) \vee \neg P(c))$

unde $c \in \mathbf{C}$, $x_1, x_2 \in V$, $f \in \mathbf{F}_2$, $P \in \mathbf{R}_1$, iar $R \in \mathbf{R}_2$.

Clauze

- O clauză este o disjuncție de literali.
- Dacă $L_1, \ldots, L_n$ sunt literali atunci vom reprezenta clauza $L_1 \vee \ldots \vee L_n$ ca mulțimea $\{L_1, \ldots, L_n\}$

  clauză = mulțime de literali

- O clauză C este trivială dacă conține un literal și complementul lui.
- Când $n = 0$ obținem clauza vidă, care se notează $\square$.

Clauze

Putem reprezenta o clauză prin mulțimea

$$\{\neg Q_1, \ldots, \neg Q_n, P_1, \ldots, P_k\}$$

unde $n, k \geq 0$ și $Q_1, \ldots, Q_n, P_1, \ldots, P_k$ sunt formule atomice.

Formula corespunzătoare este

$$\forall x_1 \ldots \forall x_m (\neg Q_1 \vee \ldots \vee \neg Q_n \vee P_1 \vee \ldots \vee P_k)$$

unde $x_1, \ldots, x_m$ sunt toate variabilele care apar în clauză

Echivalent, putem scrie

$$\forall x_1 \ldots \forall x_m (Q_1 \wedge \ldots \wedge Q_n \rightarrow P_1 \vee \ldots \vee P_k)$$

Presupunem cuantificarea universală a clauzelor implicită:

$$Q_1 \wedge \ldots \wedge Q_n \rightarrow P_1 \vee \ldots \vee P_k$$

Clauze program definite

- Clauză $Q_1 \wedge \ldots \wedge Q_n \rightarrow P_1 \vee \ldots \vee P_k$
  unde $n, k \geq 0$ și $Q_1, \ldots, Q_n, P_1, \ldots, P_k$ sunt formule atomice.

- Dacă $k = 1$, atunci avem o clauză program definită:
  - cazul $n > 0$: $Q_1 \wedge \ldots \wedge Q_n \rightarrow P$
  - cazul $n = 0$: $\top \rightarrow P$ (clauză unitate, fapt)
    $\top$ este simbol pentru o formula mereu adevărată

- Program logic definit = mulțime finită de clauze definite

Clauze Horn

- Clauză $Q_1 \wedge \ldots \wedge Q_n \rightarrow P_1 \vee \ldots \vee P_k$
  unde $n, k \geq 0$ și $Q_1, \ldots, Q_n, P_1, \ldots, P_k$ sunt formule atomice.

- Dacă $k = 0$, atunci avem o clauză scop definită (țintă, întrebare):

$$Q_1 \wedge \ldots \wedge Q_n \rightarrow \bot$$

$\bot$ este simbol pentru o formula mereu falsă

  Vom scrie o clauza scop definită ca $Q_1, \ldots, Q_n$.

- În plus, dacă $n = k = 0$, atunci avem clauza vidă $\square$

Clauză Horn = clauză program definită sau clauză scop ($k \leq 1$)

Limbajul PROLOG are la bază logica clauzelor Horn.

Clauze Horn

Clauza Horn = clauză program definită sau clauză scop ($k \leq 1$)

- Clauză $Q_1 \wedge \ldots \wedge Q_n \rightarrow P_1 \vee \ldots \vee P_k$
  unde $n, k \geq 0$ și $Q_1, \ldots, Q_n, P_1, \ldots, P_k$ sunt formule atomice.

- Fie $x_1, \ldots, x_m$ toate variabilele care apar într-o clauza scop $Q_1, \ldots, Q_n$. Atunci avem echivalența

$$\forall x_1 \ldots \forall x_m (\neg Q_1 \vee \ldots \vee \neg Q_n) \equiv \neg \exists x_1 \ldots \exists x_m (Q_1 \wedge \ldots \wedge Q_n)$$

Negația unei "întrebări" în PROLOG este clauză scop.

Programare logica

- Logica clauzelor Horn: un fragment al logicii de ordinul I în care singurele formule admise sunt clauze Horn
  - formule atomice: $P(t_1, \ldots, t_n)$
  - $Q_1 \wedge \ldots \wedge Q_n \rightarrow P$
    unde toate Q_i, P sunt formule atomice, $\top$ sau $\bot$

- Problema programării logice: reprezentăm cunoștințele ca o mulțime de clauze definite KB și suntem interesați să aflăm răspunsul la o întrebare de forma $Q_1 \wedge \ldots \wedge Q_n$, unde toate Q_i sunt formule atomice

$$KB \models Q_1 \wedge \ldots \wedge Q_n$$

  - Variabilele din KB sunt cuantificate universal.
  - Variabilele din $Q_1, \ldots, Q_n$ sunt cuantificate existențial.

Un exemplu

Fie următoarele clauze definite:

$father(jon, ken)$.

$father(ken, liz)$.

$father(X, Y) \rightarrow ancestor(X, Y)$

$daugther(X, Y) \rightarrow ancestor(Y, X)$

$ancestor(X, Y) \wedge ancestor(Y, Z) \rightarrow ancestor(X, Z)$

Putem pune întrebările:

- $ancestor(jon, liz)$?
- $ancestor(ken, Z)$?
  (există Z astfel încât $ancestor(ken, Z)$)

Sistem de deducție pentru logica clauzelor Horn

Pentru un program logic definit KB avem

- Axiome: orice clauză din KB
- Regula de deducție *backchain*:

$$\frac{\theta(Q_1) \quad \theta(Q_2) \quad \ldots \quad \theta(Q_n) \quad (Q_1 \wedge Q_2 \wedge \ldots \wedge Q_n \rightarrow P)}{\theta(Q)}$$

unde $Q_1 \wedge Q_2 \wedge \ldots \wedge Q_n \rightarrow P \in KB$, iar θ cmgu pentru Q și P.

Regula *backchain* conduce la un sistem de deducție complet:

Pentru o mulțime de clauze KB și o țintă Q,
dacă $KB \models Q$,
atunci există o derivare a lui Q folosind regula *backchain*.

Cum răspundem la întrebări

Pentru o țintă Q, trebuie să găsim o clauză din KB

$$Q_1 \wedge \ldots \wedge Q_n \rightarrow P,$$

și un unificator θ pentru Q și P.

În continuare vom verifica $\theta(Q_1), \ldots, \theta(Q_n)$.

Exemplu. Pentru ținta

$$ancestor(ken, Z),$$

putem folosi clauză

$$father(X, Y) \rightarrow ancestor(X, Y)$$

cu unificatorul

$$\{X \mapsto ken, Y \mapsto Z\}$$

pentru a obține o nouă țintă

$$father(ken, Z).$$

Sistem de deducție

Rergula backchain

$$\frac{\theta(Q_1) \quad \theta(Q_2) \quad \ldots \quad \theta(Q_n) \quad (Q_1 \wedge Q_2 \wedge \ldots \wedge Q_n \rightarrow P)}{\theta(Q)}$$

unde $Q_1 \wedge Q_2 \wedge \ldots \wedge Q_n \rightarrow P \in KB$, iar θ este cgu pentru Q și P.

Exemplu. Presupunem că în KB avem:

$father(ken, liz)$.

$father(X, Y) \rightarrow ancestor(X, Y)$

$$\frac{\dfrac{father(ken, liz)}{father(ken, Z)} \quad father(X, Y) \rightarrow ancestor(X, Y)}{ancestor(ken, Z)}$$

Puncte de decizie în programarea logica

Având doar această regulă, care sunt punctele de decizie în căutare?

- Ce clauză să alegem.
  - Pot fi mai multe clauze a căror parte dreaptă se potrivește cu o țintă.
  - Aceasta este o alegere de tip **SAU**: este suficient ca oricare din variante să reușească.
- Ordinea în care rezolvăm noile ținte.
  - Aceasta este o alegere de tip **ȘI:** trebuie arătate toate țintele noi.
  - Ordinea în care le rezolvăm poate afecta găsirea unei derivări, depinzând de strategia de căutare folosită.

Strategia de căutare din Prolog

Strategia de căutare din Prolog este de tip *depth-first*

- de sus în jos
  - pentru alegerile de tip **SAU**
  - alege clauzele în ordinea în care apar în program
- de la stânga la dreapta
  - pentru alegerile de tip **ȘI**
  - alege noile ținte în ordinea în care apar în clauza aleasă

Regula backchain și rezoluția SLD

- Regula *backchain* este implementată în programarea logică prin rezoluția SLD (Selected, Linear, Definite).
- Prolog are la bază rezoluția SLD.

Rezoluția SLD

Fie KB o mulțime de clauze definite.

$$\text{SLD}\ \boxed{\frac{\neg Q_1 \vee \cdots \vee \neg Q_i \vee \cdots \vee \neg Q_n}{\theta(\neg Q_1 \vee \cdots \vee \neg P_1 \vee \cdots \vee \neg P_m \vee \cdots \vee \neg Q_n)}}$$

unde

- $Q \vee \neg P_1 \vee \cdots \vee \neg P_m$ este o clauză definită din KB
- în care toate variabilele au fost redenumite cu variabile noi
- θ este cmgu pentru Q_i și Q

Rezoluția SLD - exemplu

```
father(eddard,sansa).
father(eddard,jonSnow).

stark(eddard).                                  ?- stark(jonSnow)
stark(catelyn).

stark(X) :- father(Y,X), stark(Y).
```

$$\text{SLD}\ \boxed{\frac{\neg Q_1 \vee \cdots \vee \neg Q_i \vee \cdots \vee \neg Q_n}{\theta(\neg Q_1 \vee \cdots \vee \neg P_1 \vee \cdots \vee \neg P_m \vee \cdots \vee \neg Q_n)}}$$

- $Q \vee \neg P_1 \vee \cdots \vee \neg P_m$ este o clauză definită din *KB*
- variabilele din $Q \vee \neg P_1 \vee \cdots \vee \neg P_m$ și Q_i se redenumesc
- θ este cmgu pentru Q_i și Q.

Rezoluția SLD - exemplu

$father(eddard, sansa)$
$father(eddard, jonSnow)$

$stark(eddard)$
$stark(catelyn)$

$stark(X) \vee \neg father(Y, X) \vee \neg stark(Y)$

$$\neg stark(jonSnow)$$

$$\theta(X) = jonSnow$$

$$\text{SLD}\ \boxed{\frac{\neg Q_1 \vee \cdots \vee \neg Q_i \vee \cdots \vee \neg Q_n}{\theta(\neg Q_1 \vee \cdots \vee \neg P_1 \vee \cdots \vee \neg P_m \vee \cdots \vee \neg Q_n)}}$$

- $Q \vee \neg P_1 \vee \cdots \vee \neg P_m$ este o clauză definită din KB
- variabilele din $Q \vee \neg P_1 \vee \cdots \vee \neg P_m$ și Q_i se redenumesc
- θ este cmgu pentru Q_i și Q.

Rezoluția SLD - exemplu

$father(eddard, sansa)$
$father(eddard, jonSnow)$

$stark(eddard)$
$stark(catelyn)$

$$\frac{\neg stark(jonSnow)}{\neg father(Y, jonSnow) \vee \neg stark(Y)}$$

$\theta(X) = jonSnow$

$stark(X) \vee \neg father(Y, X) \vee \neg stark(Y)$

$$\text{SLD} \quad \boxed{\frac{\neg Q_1 \vee \cdots \vee \neg Q_i \vee \cdots \vee \neg Q_n}{\theta(\neg Q_1 \vee \cdots \vee \neg P_1 \vee \cdots \vee \neg P_m \vee \cdots \vee \neg Q_n)}}$$

- $Q \vee \neg P_1 \vee \cdots \vee \neg P_m$ este o clauză definită din KB
- variabilele din $Q \vee \neg P_1 \vee \cdots \vee \neg P_m$ și Q_i se redenumesc
- θ este cmgu pentru Q_i și Q.

Rezoluția SLD - exemplu

$father(eddard, sansa)$
$father(eddard, jonSnow)$

$stark(eddard)$
$stark(catelyn)$

$stark(X) \vee \neg father(Y, X) \vee \neg stark(Y)$

$$\frac{\neg stark(jonSnow)}{\neg father(Y, jonSnow) \vee \neg stark(Y)}$$

$$\frac{\neg father(Y, jonSnow) \vee \neg stark(Y)}{\neg stark(eddard)}$$

$$\frac{\neg stark(eddard)}{\square}$$

$$\text{SLD}\ \boxed{\frac{\neg Q_1 \vee \cdots \vee \neg Q_i \vee \cdots \vee \neg Q_n}{\theta(\neg Q_1 \vee \cdots \vee \neg P_1 \vee \cdots \vee \neg P_m \vee \cdots \vee \neg Q_n)}}$$

- $Q \vee \neg P_1 \vee \cdots \vee \neg P_m$ este o clauză definită din KB
- variabilele din $Q \vee \neg P_1 \vee \cdots \vee \neg P_m$ și Q_i se redenumesc
- θ este cmgu pentru Q_i și Q.

Rezoluția SLD

Fie KB o mulțime de clauze definite și $Q_1 \wedge \ldots \wedge Q_m$ o întrebare, unde Q_i sunt formule atomice.

- O derivare din KB prin rezoluție SLD este o secvență

$$G_0 := \neg Q_1 \vee \ldots \vee \neg Q_m, \quad G_1, \quad \ldots, \quad G_k, \ldots$$

  în care G_{i+1} se obține din G_i prin regula SLD.
- Dacă există un k cu $G_k = \square$ (clauza vidă), atunci derivarea se numește SLD-respingere.

Rezoluția SLD

Fie KB o mulțime de clauze definite și $Q_1 \wedge \ldots \wedge Q_m$ o întrebare, unde Q_i sunt formule atomice.

- O derivare din KB prin rezoluție SLD este o secvență

$$G_0 := \neg Q_1 \vee \ldots \vee \neg Q_m, \quad G_1, \quad \ldots, \quad G_k, \ldots$$

  în care G_{i+1} se obține din G_i prin regula SLD.
- Dacă există un k cu $G_k = \square$ (clauza vidă), atunci derivarea se numește SLD-respingere.

Teoremă. Sunt echivalente:

1. există o SLD-respingere a lui $Q_1 \wedge \ldots \wedge Q_m$ din KB,
2. $KB \models Q_1 \wedge \cdots \wedge Q_m$.

Rezoluția SLD - arbori de căutare

Arbori SLD

- Presupunem că avem o mulțime de clauze definite KB și o țintă $G_0 = \neg Q_1 \vee \ldots \vee \neg Q_m$
- Construim un arbore de căutare (arbore SLD) astfel:
  - Fiecare nod al arborelui este o țintă (posibil vidă)
  - Rădăcina este G_0
  - Dacă arborele are un nod G_i, iar G_{i+1} se obține din G_i folosind regula SLD folosind o clauză $C_i \in KB$, atunci nodul G_i are copilul G_{i+1}.
    Muchia dintre G_i și G_{i+1} este etichetată cu C_i.
- Dacă un arbore SLD cu rădăcina G_0 are o frunză $\square$ (clauza vidă), atunci există o SLD-respingere a lui G_0 din KB.

Rezoluția SLD - arbori de căutare

Exemplu.

Fie *KB* următoarea mulțime de clauze definite:

1. $grandfather(X, Z) :- father(X, Y), parent(Y, Z)$
2. $parent(X, Y) :- father(X, Y)$
3. $parent(X, Y) :- mother(X, Y)$
4. $father(ken, diana)$
5. $mother(diana, brian)$

Găsiți o respingere din *KB* pentru

$$? - grandfather(ken, Y)$$

Rezoluția SLD - arbori de căutare

Exemplu.

Fie *KB* următoarea mulțime de clauze definite:

1. $grandfather(X, Z) \vee \neg father(X, Y) \vee \neg parent(Y, Z)$
2. $parent(X, Y) \vee \neg father(X, Y)$
3. $parent(X, Y) \vee \neg mother(X, Y)$
4. $father(ken, diana)$
5. $mother(diana, brian)$

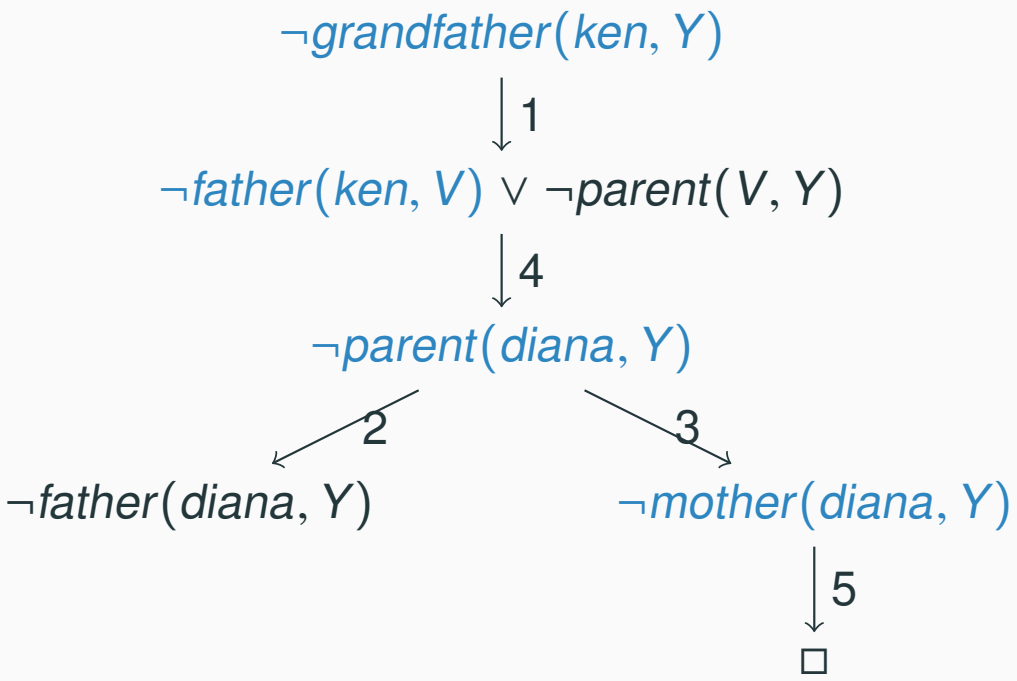
Găsiți o respingere din *KB* pentru

$$\neg grandfather(ken, Y)$$

Rezoluția SLD - arbori de căutare

Exemplu.

1. $grandfather(X, Z) \vee \neg father(X, Y) \vee \neg parent(Y, Z)$
2. $parent(X, Y) \vee \neg father(X, Y)$
3. $parent(X, Y) \vee \neg mother(X, Y)$
4. $father(ken, diana)$
5. $mother(diana, brian)$



Limbajul Prolog

- Am arătat că sistemul de inferență din spatele Prolog-ului este complet.
  - Dacă o întrebare este consecință logică a unei mulțimi de clauze, atunci există o derivare a întrebării.
- Totuși, strategia de căutate din Prolog este incompletă!
  - Chiar dacă o întrebare este consecință logică a unei mulțimi de clauze, Prolog nu găsește mereu o derivare a întrebării.

Limbajul Prolog - exemplu

```
warmerClimate :- albedoDecrease.
warmerClimate :- carbonIncrease.
iceMelts :- warmerClimate.
albedoDecrease :- iceMelts.
carbonIncrease.

?- iceMelts.
! Out of local stack
```

Limbajul Prolog - exemplu

```
warmerClimate :- albedoDecrease.
warmerClimate :- carbonIncrease.
iceMelts :- warmerClimate.
albedoDecrease :- iceMelts.
carbonIncrease.

?- iceMelts.
! Out of local stack
```

Limbajul Prolog - exemplu

Totuși, există o derivare a lui *iceMelts* în sistemul de deducție din clauzele:

$$
\begin{aligned}
\mathit{albedoDecrease} &\rightarrow \mathit{warmerClimate} \\
\mathit{carbonIncrease} &\rightarrow \mathit{warmerClimate} \\
\mathit{warmerClimate} &\rightarrow \mathit{iceMelts} \\
\mathit{iceMelts} &\rightarrow \mathit{albedoDecrease} \\
\top &\rightarrow \mathit{carbonIncrease}
\end{aligned}
$$

$$
\dfrac{\dfrac{\mathit{carbonInc}. \qquad \mathit{carbonInc}. \rightarrow \mathit{warmerClim}.}{\mathit{warmerClim}.} \qquad \mathit{warmerClim}. \rightarrow \mathit{iceMelts}}{\mathit{iceMelts}}
$$

Rezoluția SLD - arbori de căutare

Exercițiu Desenați arborele SLD pentru programul Prolog de mai jos și ținta

```
?- p(X,X).
```

```
1.  p(X,Y) :- q(X,Z), r(Z,Y).
2.  p(X,X) :- s(X).
3.  q(X,b).
4.  q(b,a).
5.  q(X,a) :- r(a,X).
6.  r(b,a).
```

```
7.  s(X) :- t(X,a).
8.  s(X) :- t(X,b).
9.  s(X) :- t(X,X).
10.  t(a,b).
11.  t(b,a).
```

Rezoluția SLD - arbori de căutare

```
1.  p(X,Y) :- q(X,Z), r(Z,Y).
2.  p(X,X) :- s(X).
3.  q(X,b).
```

$p(X,Y) \vee \neg q(X,Z) \vee \neg r(Z,Y)$
$p(X,X) \vee \neg s(X)$
$q(X,b)$

```
4.  q(b,a).
5.  q(X,a) :- r(a,X).
6.  r(b,a).
```

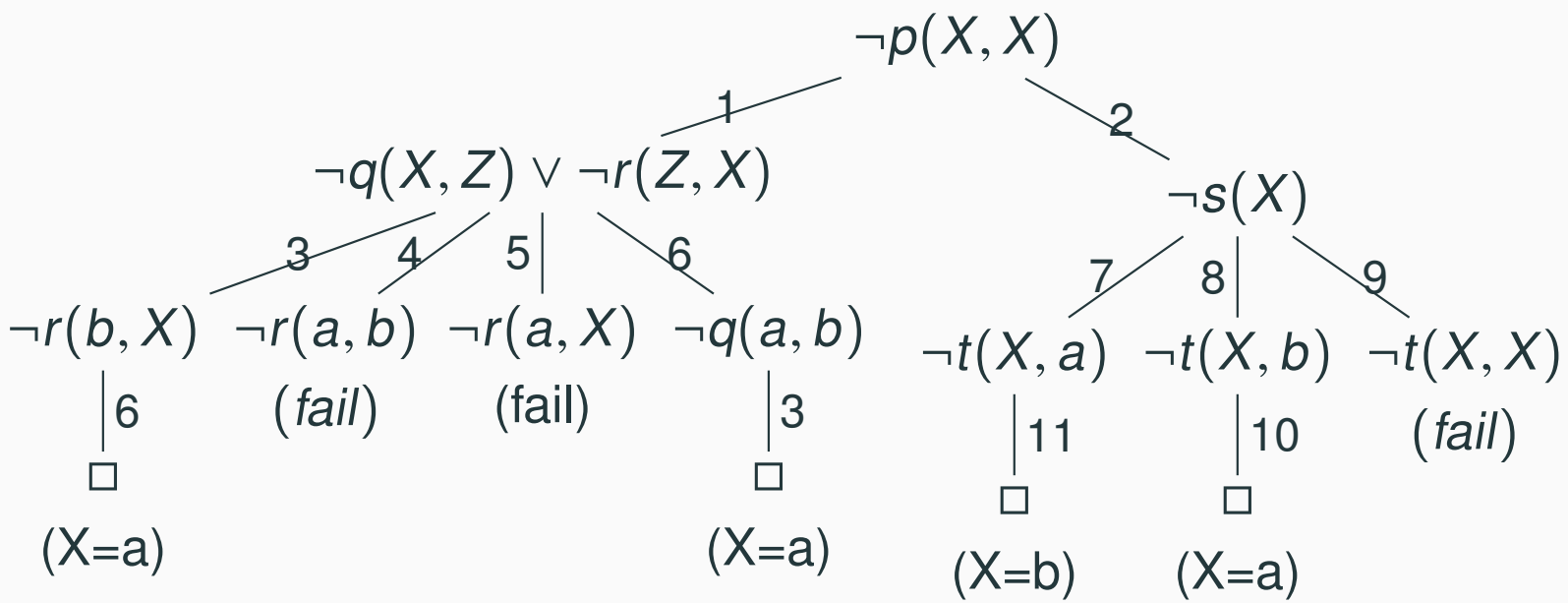
$q(b,a)$
$q(X,a) \vee \neg r(a,X)$
$r(b,a)$

```
7.  s(X) :- t(X,a).
8.  s(X) :- t(X,b).
9.  s(X) :- t(X,X).
```

$s(X) \vee \neg t(X,a)$
$s(X) \vee \neg t(X,b)$
$s(X) \vee \neg t(X,X)$

```
10.  t(a,b).
11.  t(b,a).
```

$t(a,b)$
$t(b,a)$



Pe data viitoare!

Fundamentele limbajelor de programare

C11

Denisa Diaconescu
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Semantica limbajelor de programare

Principalele paradigme de programare

- Imperativă (cum calculăm)
  - Procedurală
  - Orientată pe obiecte
- Declarativă (ce calculăm)
  - Logică
  - Functională

Fundamentele paradigmelor de programare

Imperativă Execuția unei Mașini Turing

Logică Rezoluția în logica clauzelor Horn

Funcțională Beta-reducție în Lambda Calcul

Ce înseamnă semantică formală?

Ce definește un limbaj de programare?

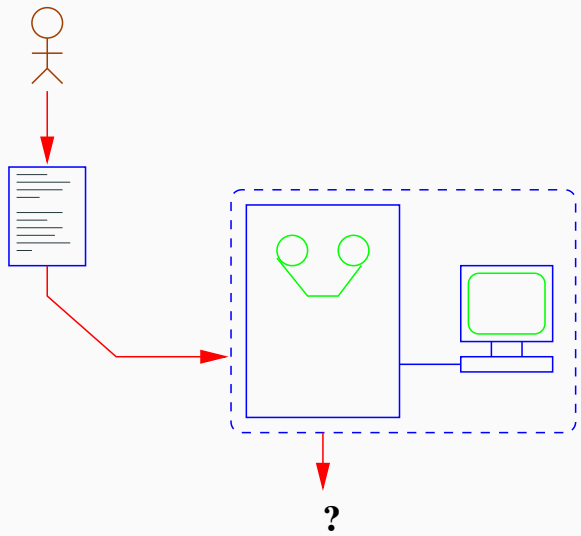
- Sintaxa – Simboluri de operație, cuvinte cheie, descriere (formală) a programelor/expresiilor bine formate
- Practic – Un limbaj e definit de modul cum poate fi folosit
  - Manual de utilizare și exemple de bune practici
  - Implementare (compilator/interpretor)
  - Instrumente ajutătoare (analizor de sintaxă, depanator)
- Semantica – Ce înseamnă/care e comportamentul unei instrucțiuni?
  - De cele mai multe ori se dă din umeri și se spune că Practica e suficientă
  - Limbajele mai utilizate sunt standardizate

La ce folosește semantica?

- Să înțelegem un limbaj în profunzime
  - Ca programator: pe ce mă pot baza când programez
  - Ca implementator al limbajului: ce garanții trebuie să ofer
- Ca instrument în proiectarea unui nou limbaj/a unei extensii
  - Înțelegerea componentelor și a relațiilor dintre ele
  - Exprimarea (și motivarea) deciziilor de proiectare
  - Demonstrarea unor proprietăți generice ale limbajului
- Ca bază pentru demonstrarea corectitudinii programelor

Problema corectitudinii programelor

- Pentru anumite metode de programare (e.g., imperativă, orientată pe obiecte), nu este ușor să stabilim dacă un program este corect sau să înțelegem ce înseamnă că este corect (e.g, în raport cu ce?!).
- Corectitudinea programelor devine o problemă din ce în ce mai importantă, nu doar pentru aplicații "safety-critical".
- Avem nevoie de metode ce asigură "calitate", capabile să ofere "garanții".



C

```
#include <iostream>
using namespace std;
int main()
{
  int square;
  for(int i = 1; i <= 5; ++i)
  {
    square = i * i;
    cout <<  square << endl;
  }
}
```

C

```
#include <iostream>
using namespace std;
int main()
{
  int square;
  for(int i = 1; i <= 5; ++i)
  {
    square = i * i;
    cout <<  square << endl;
  }
}
```

- Este corect?

C

```
#include <iostream>
using namespace std;
int main()
{
  int square;
  for(int i = 1; i <= 5; ++i)
  {
    square = i * i;
    cout <<  square << endl;
  }
}
```

- Este corect? În raport cu ce?

C

```
#include <iostream>
using namespace std;
int main()
{
  int square;
  for(int i = 1; i <= 5; ++i)
  {
    square = i * i;
    cout <<  square << endl;
  }
}
```

- Este corect? În raport cu ce?
- Un formalism adecvat trebuie:
  - să permită descrierea problemelor (specificații), și
  - să raționeze despre implementarea lor (corectitudinea programelor).

Care este comportamentul corect?

```
int main(void) {
  int x = 0;
  return (x = 1) + (x = 2);
}
```

Care este comportamentul corect?

```
int main(void) {
  int x = 0;
  return (x = 1) + (x = 2);
}
```

- GCC4, MSVC: valoarea întoarsă e 4
- GCC3, ICC, Clang: valoarea întoarsă e 3

Conform standardului limbajului C (ISO/IEC 9899:2018)
Comportamentul programului este nedefinit.

Tipuri de semantică

- Limbaj natural – descriere textuală a efectelor
- Statică – un sistem de tipuri care exclude programe eronate
- Operațională – asocierea unei demonstrații pentru execuție
  - $\langle cod, \sigma \rangle \to \langle cod', \sigma' \rangle$
  - modelează execuția unui program pe o mașină abstractă
  - utilă pentru implementarea de compilatoare și interpretoare
- Axiomatică – aproximarea logică a efectelor unei instrucțiuni
  - $\vdash \{\varphi\} cod \{\psi\}$
  - modelează comportamentul un program prin formulele logice pe care le satisface
  - utilă pentru demonstrarea corectitudinii
- Denotațională – asocierea unui obiect matematic (denotație)
  - $[\![cod]\!]$
  - modelează un program ca obiecte matematice
  - utilă pentru fundamente matematice

Limbajul IMP

Vom folosi ca exemplu un mic limbaj imperativ IMP care conține:

- Expresii
  - Aritmetice: `x + 3`
  - Booleene: `x >= 7`
- Instrucțiuni
  - De atribuire: `x = 5`
  - Condiționale: `if(x >= 7, x = 5, x = 0)`
  - De ciclare: `while(x >= 7, x = x - 1)`
- Compunerea instrucțiunilor: `x=7; while(x>=0, x=x-1)`
- Blocuri de instrucțiuni: `{x=7; while(x>=0, x=x-1)}`

Limbajul IMP

Un exemplu de program în limbajul IMP

```
{ x = 10 ; sum = 0;
  while(0 =< x,
     {sum = sum + x; x = x-1}
  )},
  sum
```

Semantica: după executia programului, se evaluează `sum`

Sintaxa BNF a limbajului IMP

$$
\begin{aligned}
E &::= n \mid x \\
&\quad \mid E + E \mid E - E \mid E * E \\
\\
B &::= \texttt{true} \mid \texttt{false} \\
&\quad \mid E =< E \mid E >= E \mid E == E \\
&\quad \mid \texttt{not}(B) \mid \texttt{and}(B\ ,\ B) \mid \texttt{or}(B\ ,\ B) \\
\\
C &::= \texttt{skip} \\
&\quad \mid x = E \\
&\quad \mid \texttt{if(}\ B\ \texttt{,}\ C\ \texttt{,}\ C\ \texttt{)} \\
&\quad \mid \texttt{while(}\ B\ \texttt{,}\ C\ \texttt{)} \\
&\quad \mid \texttt{\{}\ C\ \texttt{\}} \mid C\ \texttt{;}\ C \\
\\
P &::= \texttt{\{}\ C\ \texttt{\},}\ E
\end{aligned}
$$

Semantica operațională small-step

Imagine de ansamblu

- Semantica operațională descrie cum se execută un program pe o masină abstractă (ideală).
- Semantica operațională **small-step**
  - semantica structurală, a pașilor mici
  - descrie cum o execuție a programului avansează în functie de reduceri succesive.

$$\langle cod, \sigma \rangle \rightarrow \langle cod', \sigma' \rangle$$

- Semantica operațională **big-step**
  - semantică naturală, într-un pas mare

Starea execuției

- Starea execuției unui program IMP la un moment dat este dată de valorile deținute în acel moment de variabilele declarate în program.
- Formal, starea execuției unui program IMP la un moment dat este o funcție parțială (cu domeniu finit):

$$\sigma : \mathit{Var} \rightharpoonup \mathit{Int}$$

- Notatii:
  - Descrierea funcției prin enumerare: $\sigma = n \mapsto 10, \mathit{sum} \mapsto 0$
  - Funcția vidă $\perp$, nedefinită pentru nicio variabilă
  - Obținerea valorii unei variabile: $\sigma(x)$
  - Suprascrierea valorii unei variabile:

$$\sigma_{x \leftarrow v}(y) = \begin{cases} \sigma(y), \text{ dacă } y \neq x \\ v, \text{ dacă } y = x \end{cases}$$

Semantica small-step

- Introdusă de Gordon Plotkin (1981)
- Denumiri alternative:
  - Semantică Operațională Structurală
  - semantică prin tranziții
  - semantică prin reducere
- Definește cel mai mic pas de execuție ca o relație „de tranziție” între configurații:

$$\langle cod\ ,\ \sigma \rangle \rightarrow \langle cod'\ ,\ \sigma' \rangle$$

- Execuția se obține ca o succesiune de astfel de tranziții:

$$\begin{aligned} \langle x = 0\ ;\ x = x + 1\ ,\ \bot \rangle &\rightarrow \langle x = x + 1\ ,\ x \mapsto 0 \rangle \\ &\rightarrow \langle x = 0 + 1\ ,\ x \mapsto 0 \rangle \\ &\rightarrow \langle x = 1\ ,\ x \mapsto 0 \rangle \\ &\rightarrow \langle \{\}\ ,\ x \mapsto 1 \rangle \end{aligned}$$

- Cum definim această relație?

  Prin inductie după elementele din sintaxă.

Redex. Reguli structurale. Axiome

- Expresie reductibilă (redex)
  - Fragmentul de sintaxă care va fi procesat la pasul următor

$$\texttt{if}\ (0 \texttt{<=}\ 5 + 7 \texttt{*}\ x\ \texttt{,}\ r = 1\ \texttt{,}\ r = 0)$$

Redex. Reguli structurale. Axiome

- Expresie reductibilă (redex)
  - Fragmentul de sintaxă care va fi procesat la pasul următor

$$\texttt{if}\ (0 \texttt{ <= } 5 \texttt{ + } 7 \texttt{ * } x \texttt{ , } r = 1 \texttt{ , } r = 0)$$

Redex. Reguli structurale. Axiome

- Expresie reductibilă (redex)
  - Fragmentul de sintaxă care va fi procesat la pasul următor

$$\texttt{if}\ (0 \texttt{ <= } 5 \texttt{ + } 7 \texttt{ * } x \texttt{ , } r = 1 \texttt{ , } r = 0)$$

- Reguli structurale
  - Folosesc la identificarea următorului redex
  - Definite recursiv pe structura termenilor

Redex. Reguli structurale. Axiome

- Expresie reductibilă (redex)
  - Fragmentul de sintaxă care va fi procesat la pasul următor

$$\texttt{if}\ (0 \texttt{ <= } 5 \texttt{ + } 7 \texttt{ * } x\ \texttt{ , } r = 1\ \texttt{ , } r = 0)$$

- Reguli structurale
  - Folosesc la identificarea următorului redex
  - Definite recursiv pe structura termenilor

$$\frac{\langle b\ ,\ \sigma\rangle \to \langle b'\ ,\ \sigma\rangle}{\langle \texttt{if}\ (b, bl_1, bl_2)\ ,\ \sigma\rangle \to \langle \texttt{if}\ (b', bl_1, bl_2)\ ,\ \sigma\rangle}$$

Redex. Reguli structurale. Axiome

- Expresie reductibilă (redex)
  - Fragmentul de sintaxă care va fi procesat la pasul următor

$$\texttt{if}\ (0 \texttt{ <= } 5 \texttt{ + } 7 \texttt{ * } x\ \texttt{ , } r = 1\ \texttt{ , } r = 0)$$

- Reguli structurale
  - Folosesc la identificarea următorului redex
  - Definite recursiv pe structura termenilor

$$\frac{\langle b\ ,\ \sigma\rangle \to \langle b'\ ,\ \sigma\rangle}{\langle \texttt{if}\ (b, bl_1, bl_2)\ ,\ \sigma\rangle \to \langle \texttt{if}\ (b', bl_1, bl_2)\ ,\ \sigma\rangle}$$

- Axiome
  - Realizează pasul computațional

Redex. Reguli structurale. Axiome

- Expresie reductibilă (redex)
  - Fragmentul de sintaxă care va fi procesat la pasul următor

$$\texttt{if}\ (0 \texttt{ <= } 5 \texttt{ + } 7 \texttt{ * } x\ \texttt{,}\ r = 1\ \texttt{,}\ r = 0)$$

- Reguli structurale
  - Folosesc la identificarea următorului redex
  - Definite recursiv pe structura termenilor

$$\frac{\langle b\ ,\ \sigma\rangle \to \langle b'\ ,\ \sigma\rangle}{\langle \texttt{if}\ (b, bl_1, bl_2)\ ,\ \sigma\rangle \to \langle \texttt{if}\ (b', bl_1, bl_2)\ ,\ \sigma\rangle}$$

- Axiome
  - Realizează pasul computațional

$$\langle \texttt{if}\ (\texttt{true}, bl_1, bl_2)\ ,\ \sigma\rangle \to \langle bl_1\ ,\ \sigma\rangle$$

Semantica expresiilor aritmetice

- Semantica unui întreg este o valoare
  - nu poate fi redex, deci nu avem regulă
- Semantica unei variabile

  (Id) $\langle x\ ,\ \sigma\rangle \to \langle i\ ,\ \sigma\rangle \quad \textit{dacă}\ \sigma(x) = i$
- Semantica adunării a două expresii aritmetice

  (Add) $\langle i_1 + i_2\ ,\ \sigma\rangle \to \langle i\ ,\ \sigma\rangle \quad \textit{dacă}\ i_1 + i_2 = i$

$$\frac{\langle a_1\ ,\ \sigma\rangle \to \langle a_1'\ ,\ \sigma\rangle}{\langle a_1 + a_2\ ,\ \sigma\rangle \to \langle a_1' + a_2\ ,\ \sigma\rangle}$$

$$\frac{\langle a_2\ ,\ \sigma\rangle \to \langle a_2'\ ,\ \sigma\rangle}{\langle a_1 + a_2\ ,\ \sigma\rangle \to \langle a_1 + a_2'\ ,\ \sigma\rangle}$$

  Observatie: ordinea de evaluare a argumentelor este nespecificată.

Semantica expresiilor booleene

- Semantica operatorului de comparație

(LEQ-FALSE) $\langle i_1 =< i_2 \ , \ \sigma \rangle \to \langle \mathbf{false} \ , \ \sigma \rangle \quad$ *dacă* $i_1 > i_2$

(LEQ-TRUE) $\langle i_1 =< i_2 \ , \ \sigma \rangle \to \langle \mathbf{true} \ , \ \sigma \rangle \quad$ *dacă* $i_1 \leq i_2$

$$\frac{\langle a_1 \ , \ \sigma \rangle \to \langle a_1' \ , \ \sigma \rangle}{\langle a_1 =< a_2 \ , \ \sigma \rangle \to \langle a_1' =< a_2 \ , \ \sigma \rangle} \qquad \frac{\langle a_2 \ , \ \sigma \rangle \to \langle a_2' \ , \ \sigma \rangle}{\langle a_1 =< a_2 \ , \ \sigma \rangle \to \langle a_1 =< a_2' \ , \ \sigma \rangle}$$

- Semantica negației

(!-FALSE) $\langle \mathbf{not(true)} \ , \ \sigma \rangle \to \langle \mathbf{false} \ , \ \sigma \rangle$

(!-TRUE) $\langle \mathbf{not(false)} \ , \ \sigma \rangle \to \langle \mathbf{true} \ , \ \sigma \rangle$

$$\frac{\langle a \ , \ \sigma \rangle \to \langle a' \ , \ \sigma \rangle}{\langle \mathbf{not} \ (a) \ , \ \sigma \rangle \to \langle \mathbf{not} \ (a') \ , \ \sigma \rangle}$$

Semantica expresiilor booleene

- Semantica și-ului

(AND-FALSE) $\langle \mathbf{and}\ (\mathbf{false},\ b_2)\ ,\ \sigma\rangle \to \langle \mathbf{false}\ ,\ \sigma\rangle$

(AND-TRUE) $\langle \mathbf{and}\ (\mathbf{true},\ b_2)\ ,\ \sigma\rangle \to \langle b_2\ ,\ \sigma\rangle$

$$\frac{\langle b_1\ ,\ \sigma\rangle \to \langle b_1'\ ,\ \sigma\rangle}{\langle \mathbf{and}\ (b_1\ ,\ b_2)\ ,\ \sigma\rangle \to \langle \mathbf{and}\ (b_1'\ ,\ b_2)\ ,\ \sigma\rangle}$$

Semantica compunerii și a blocurilor

- Semantica blocurilor

  (Block) $\langle \texttt{\{ } s \texttt{ \}} , \sigma \rangle \to \langle s , \sigma \rangle$

- Semantica compunerii secvențiale

  (Next-stmt) $\langle \texttt{skip; } s_2 , \sigma \rangle \to \langle s_2 , \sigma \rangle$

  $$\frac{\langle s_1 , \sigma \rangle \to \langle s_1' , \sigma' \rangle}{\langle s_1 \texttt{ ; } s_2 , \sigma \rangle \to \langle s_1' \texttt{ ; } s_2 , \sigma' \rangle}$$

- Semantica atribuirii

  (Asgn) $\langle x = i , \sigma \rangle \to \langle \texttt{skip} , \sigma' \rangle \quad \textit{dacă } \sigma' = \sigma_{x \leftarrow i}$

  $$\frac{\langle a , \sigma \rangle \to \langle a' , \sigma \rangle}{\langle x = a , \sigma \rangle \to \langle x = a' , \sigma \rangle}$$

Semantica lui `if`

- Semantica lui `if`

(IF-TRUE) $\langle \mathbf{if}\ (\mathbf{true}, bl_1, bl_2)\ ,\ \sigma\rangle \to \langle bl_1\ ,\ \sigma\rangle$

(IF-FALSE) $\langle \mathbf{if}\ (\mathbf{false}, bl_1, bl_2)\ ,\ \sigma\rangle \to \langle bl_2\ ,\ \sigma\rangle$

$$\frac{\langle b\ ,\ \sigma\rangle \to \langle b'\ ,\ \sigma\rangle}{\langle \mathbf{if}\ (b, bl_1, bl_2)\ ,\ \sigma\rangle \to \langle \mathbf{if}\ (b', bl_1, bl_2)\ ,\ \sigma\rangle}$$

- Semantica lui `while`

(WHILE) $\langle \mathbf{while}\ (b, bl)\ ,\ \sigma\rangle \to \langle \mathbf{if}\ (b, bl\ \mathbf{;while}\ (b, bl), \mathbf{skip})\ ,\ \sigma\rangle$

- Semantica programelor

(PGM) $$\frac{\langle a_1\ ,\ \sigma_1\rangle \to \langle a_2\ ,\ \sigma_2\rangle}{\langle (\mathbf{skip}, a_1)\ ,\ \sigma_1\rangle \to \langle (\mathbf{skip}, a_2)\ ,\ \sigma_2\rangle}$$

$$\frac{\langle s_1\ ,\ \sigma_1\rangle \to \langle s_2\ ,\ \sigma_2\rangle}{\langle (s_1, a)\ ,\ \sigma_1\rangle \to \langle (s_2, a)\ ,\ \sigma_2\rangle}$$

Semantica small-step a lui IMP

Execuție pas cu pas

$$\langle i = 3 \texttt{ ; while } (0 \texttt{ <= } i \texttt{ ,\{ } i = i + -4 \texttt{ \}}) \texttt{ , } \bot \rangle \xrightarrow{\textsc{Asgn}}$$

Semantica small-step a lui IMP

Execuție pas cu pas

$$\langle i = 3 \texttt{ ; while } (0 \texttt{ <= } i \texttt{ ,\{ } i = i + -4 \texttt{ \}}) \texttt{ , } \bot \rangle \xrightarrow{\textsc{Asgn}}$$

$$\langle \texttt{skip; while } (0 \texttt{ <= } i \texttt{ ,\{ } i = i + -4 \texttt{ \}}) \texttt{ , } i \mapsto 3 \rangle \xrightarrow{\textsc{Next-stmt}}$$

Semantica small-step a lui IMP

Execuție pas cu pas

$$\langle i = 3 \texttt{ ; while } (0 \texttt{ <= } i \texttt{ ,\{ } i = i + -4 \texttt{ \}}) \texttt{ , } \bot \rangle \xrightarrow{\textsc{Asgn}}$$

$$\langle \texttt{skip; while } (0 \texttt{ <= } i \texttt{ ,\{ } i = i + -4 \texttt{ \}}) \texttt{ , } i \mapsto 3 \rangle \xrightarrow{\textsc{Next-stmt}}$$

$$\langle \texttt{while } (0 \texttt{ <= } i \texttt{ ,\{ } i = i + -4 \texttt{ \}}) \texttt{ , } i \mapsto 3 \rangle \xrightarrow{\textsc{While}}$$

Semantica small-step a lui IMP

Execuție pas cu pas

$$\langle i = 3 \texttt{ ; while } (0 \texttt{ <= } i \texttt{ ,\{ } i = i + -4 \texttt{ \}}) \,,\ \bot \rangle \xrightarrow{\textsc{Asgn}}$$

$$\langle \texttt{skip; while } (0 \texttt{ <= } i \texttt{ ,\{ } i = i + -4 \texttt{ \}}) \,,\ i \mapsto 3 \rangle \xrightarrow{\textsc{Next-stmt}}$$

$$\langle \texttt{while } (0 \texttt{ <= } i \texttt{ ,\{ } i = i + -4 \texttt{ \}}) \,,\ i \mapsto 3 \rangle \xrightarrow{\textsc{While}}$$

$$\langle \texttt{if } (0 \texttt{ <= } i \texttt{ , } i = i + -4 \texttt{ ;while } (0 \texttt{ <= } i, \texttt{\{ } i = i + -4 \texttt{ \}}) \texttt{ ,skip}) \,,\ i \mapsto 3 \rangle \xrightarrow{\textsc{Id}}$$

Semantica small-step a lui IMP

Execuție pas cu pas

$$\langle i = 3 \texttt{ ; while } (0 \texttt{ <= } i \texttt{ ,\{ } i = i + -4 \texttt{ \}}) \texttt{ , } \bot \rangle \xrightarrow{\textsc{Asgn}}$$

$$\langle \texttt{skip; while } (0 \texttt{ <= } i \texttt{ ,\{ } i = i + -4 \texttt{ \}}) \texttt{ , } i \mapsto 3 \rangle \xrightarrow{\textsc{Next-stmt}}$$

$$\langle \texttt{while } (0 \texttt{ <= } i \texttt{ ,\{ } i = i + -4 \texttt{ \}}) \texttt{ , } i \mapsto 3 \rangle \xrightarrow{\textsc{While}}$$

$$\langle \texttt{if } (0 \texttt{ <= } i \texttt{ , } i = i + -4 \texttt{ ;while } (0 \texttt{ <= } i, \texttt{\{ } i = i + -4 \texttt{ \}}) \texttt{ ,skip}) \texttt{ , } i \mapsto 3 \rangle \xrightarrow{\textsc{Id}}$$

$$\langle \texttt{if } (0 \texttt{ <= } 3 \texttt{ , } i = i + -4 \texttt{ ;while } (0 \texttt{ <= } i, \texttt{\{ } i = i + -4 \texttt{ \}}) \texttt{ ,skip}) \texttt{ , } i \mapsto 3 \rangle \xrightarrow{\textsc{Leq-tru}}$$

Semantica small-step a lui IMP

Execuție pas cu pas

$$\langle i = 3 \texttt{ ; while } (0 \texttt{ <= } i \texttt{ ,\{ } i = i + -4 \texttt{ \}}) \texttt{ , } \bot\rangle \xrightarrow{\textsc{Asgn}}$$

$$\langle \texttt{skip; while } (0 \texttt{ <= } i \texttt{ ,\{ } i = i + -4 \texttt{ \}}) \texttt{ , } i \mapsto 3\rangle \xrightarrow{\textsc{Next-stmt}}$$

$$\langle \texttt{while } (0 \texttt{ <= } i \texttt{ ,\{ } i = i + -4 \texttt{ \}}) \texttt{ , } i \mapsto 3\rangle \xrightarrow{\textsc{While}}$$

$$\langle \texttt{if } (0 \texttt{ <= } i \texttt{ , } i = i + -4 \texttt{ ;while } (0 \texttt{ <= } i, \texttt{\{ } i = i + -4 \texttt{ \}}) \texttt{ ,skip}) \texttt{ , } i \mapsto 3\rangle \xrightarrow{\textsc{Id}}$$

$$\langle \texttt{if } (0 \texttt{ <= } 3 \texttt{ , } i = i + -4 \texttt{ ;while } (0 \texttt{ <= } i, \texttt{\{ } i = i + -4 \texttt{ \}}) \texttt{ ,skip}) \texttt{ , } i \mapsto 3\rangle \xrightarrow{\textsc{Leq-tru}}$$

$$\langle \texttt{if } (\texttt{true, } i = i + -4 \texttt{ ;while } (0 \texttt{ <= } i, \texttt{\{ } i = i + -4 \texttt{ \}}) \texttt{ ,skip}) \texttt{ , } i \mapsto 3\rangle \xrightarrow{\textsc{If-true}}$$

Semantica small-step a lui IMP

Execuție pas cu pas

$$\langle i = 3 \texttt{ ; while } (0 \texttt{ <= } i \texttt{ ,\{ } i = i + -4 \texttt{ \}}) \texttt{ , } \bot\rangle \xrightarrow{\textsc{Asgn}}$$

$$\langle \texttt{skip; while } (0 \texttt{ <= } i \texttt{ ,\{ } i = i + -4 \texttt{ \}}) \texttt{ , } i \mapsto 3\rangle \xrightarrow{\textsc{Next-stmt}}$$

$$\langle \texttt{while } (0 \texttt{ <= } i \texttt{ ,\{ } i = i + -4 \texttt{ \}}) \texttt{ , } i \mapsto 3\rangle \xrightarrow{\textsc{While}}$$

$$\langle \texttt{if } (0 \texttt{ <= } i \texttt{ , } i = i + -4 \texttt{ ;while } (0 \texttt{ <= } i, \texttt{\{ } i = i + -4 \texttt{ \}}) \texttt{ ,skip}) \texttt{ , } i \mapsto 3\rangle \xrightarrow{\textsc{Id}}$$

$$\langle \texttt{if } (0 \texttt{ <= } 3 \texttt{ , } i = i + -4 \texttt{ ;while } (0 \texttt{ <= } i, \texttt{\{ } i = i + -4 \texttt{ \}}) \texttt{ ,skip}) \texttt{ , } i \mapsto 3\rangle \xrightarrow{\textsc{Leq-tru}}$$

$$\langle \texttt{if } (\texttt{true, } i = i + -4 \texttt{ ;while } (0 \texttt{ <= } i, \texttt{\{ } i = i + -4 \texttt{ \}}) \texttt{ ,skip}) \texttt{ , } i \mapsto 3\rangle \xrightarrow{\textsc{If-true}}$$

$$\langle i = i + -4 \texttt{ ;while } (0 \texttt{ <= } i, \texttt{\{ } i = i + -4 \texttt{ \}}) \texttt{ , } i \mapsto 3\rangle \xrightarrow{\textsc{Id}}$$

$\ldots$

Semantica axiomatică

Semantica Axiomatică

- Dezvoltată de Tony Hoare în 1969
  (inspirată de rezultatele lui Robert Floyd).
- Definește triplete (triplete Hoare) de forma

$$\{P\}\ \mathbb{C}\ \{Q\}$$

  unde:
  - $\mathbb{C}$ este o instrucțiune
  - P (precondiție), Q (postcondiție) sunt aserțiuni logice asupra stării sistemului înaintea, respectiv după execuția lui $\mathbb{C}$
  - Limbajul aserțiunilor este un limbaj de ordinul I.

Semantica Axiomatică

Interpretarea unui triplet Hoare $\{P\}\ \mathbb{C}\ \{Q\}$

- dacă programul se execută dintr-o stare inițială care satisface P
- și execuția se termină
- atunci se ajunge într-o stare finală care satisface Q.

Semantica Axiomatică

Interpretarea unui triplet Hoare $\{P\}\ \mathbb{C}\ \{Q\}$

- dacă programul se execută dintr-o stare inițială care satisface P
- și execuția se termină
- atunci se ajunge într-o stare finală care satisface Q.

Exemple:

- $\{\mathtt{x} = 1\}$ `x = x+1` $\{\mathtt{x} = 2\}$ este corect
- $\{\mathtt{x} = 1\}$ `x = x+1` $\{\mathtt{x} = 3\}$ **nu** este corect
- $\{\top\}$ `if (x <= y) z = x; else z = y;` $\{\mathtt{z} = \mathit{min}(\mathtt{x}, \mathtt{y})\}$ este corect

Logica Hoare ne ajută să verificăm corectitudinea programelor

- Se asociază fiecărei construcții sintactice o regulă de deducție care definește recursiv tripletele corecte pentru un limbaj.
- Se exprimă o aserțiune de corectitudine a programului ca un tripet Hoare
- Se verifică dacă tripletul dat e corect folosind definiția recursivă

Sistem de reguli pentru logica Floyd-Hoare

Reguli generale pentru logică propozițională

$$(\rightarrow) \quad \frac{P1 \rightarrow P2 \quad \{P2\}\ \mathbb{C}\ \{Q2\} \quad Q2 \rightarrow Q1}{\{P1\}\ \mathbb{C}\ \{Q1\}}$$

$$(\vee) \quad \frac{\{P1\}\ \mathbb{C}\ \{Q\} \quad \{P2\}\ \mathbb{C}\ \{Q\}}{\{P1 \vee P2\}\ \mathbb{C}\ \{Q\}}$$

$$(\wedge) \quad \frac{\{P\}\ \mathbb{C}\ \{Q1\} \quad \{P\}\ \mathbb{C}\ \{Q2\}}{\{P\}\ \mathbb{C}\ \{Q1 \wedge Q2\}}$$

Logica Floyd-Hoare pentru IMP1

$$(\textsc{Skip}) \quad \frac{}{\{P\}\ \{\}\ \{P\}}$$

$$(\textsc{Seq}) \quad \frac{\{P\}\ \mathbb{C}_1\ \{Q\} \quad \{Q\}\ \mathbb{C}_2\ \{R\}}{\{P\}\ \mathbb{C}_1 ; \mathbb{C}_2\ \{R\}}$$

$$(\textsc{Asign}) \quad \frac{}{\{P[x/e]\}\ x = e\ \{P\}}$$

$$(\textsc{If}) \quad \frac{\{\mathbb{B} \wedge P\}\ \mathbb{C}_1\ \{Q\} \quad \{\neg\mathbb{B} \wedge P\}\ \mathbb{C}_2\ \{Q\}}{\{P\}\ \texttt{if}\ (\mathbb{B})\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2\ \{Q\}}$$

$$(\textsc{While}) \quad \frac{\{\mathbb{B} \wedge P\}\ \mathbb{C}\ \{P\}}{\{P\}\ \texttt{while}\ (\mathbb{B})\ \mathbb{C}\ \{\neg\mathbb{B} \wedge P\}}$$

Regula pentru atribuire

$$(\textsc{Asign}) \quad \frac{}{\{P[x/e]\}\ x = e\ \{P\}}$$

Exemplu:

$$\{\mathtt{x} + \mathtt{y} = \mathtt{y} + 10\}\ \mathtt{x}\ =\ \mathtt{x}\ +\ \mathtt{y}\ \{\mathtt{x} = \mathtt{y} + 10\}$$

Regula pentru condiții

$$(\textsc{If}) \quad \frac{\{\mathbb{B} \wedge P\}\ \mathbb{C}_1\ \{Q\} \quad \{\neg\mathbb{B} \wedge P\}\ \mathbb{C}_2\ \{Q\}}{\{P\}\ \texttt{if}\ (\mathbb{B})\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2\ \{Q\}}$$

Exemplu:

Pentru a demonstra

$\{\top\}\ \texttt{if (x <= y) z = x; else z = y;}\ \{\texttt{z} = \mathit{min}(\texttt{x}, \texttt{y})\}$

este suficient să demonstrăm

- $\{\texttt{x} \leq \texttt{y}\}\ \texttt{z = x}\ \{\texttt{z} = \mathit{min}(\texttt{x}, \texttt{y})\}$
- $\{\neg(\texttt{x} \leq \texttt{y})\}\ \texttt{z = y}\ \{\texttt{z} = \mathit{min}(\texttt{x}, \texttt{y})\}$

Invarianți pentru `while`

Cum demonstrăm $\{P\}$ `while(`$\mathbb{B}$`)` $\mathbb{C}$ $\{Q\}$?

Se determină un invariant I și se folosește următoarea regulă:

$$(\textsc{Inv}) \quad \frac{P \to I \quad \{\mathbb{B} \wedge I\}\,\mathbb{C}\,\{I\} \quad (I \wedge \neg\mathbb{B}) \to Q}{\{P\}\ \mathbf{while}\ (\mathbb{B})\,\mathbb{C}\,\{Q\}}$$

Invariantul trebuie să satisfacă următoarele proprietăți:

- să fie adevărat inițial
- să rămână adevărat după execuția unui ciclu
- să implice postcondiția la ieșirea din buclă

Invarianți pentru `while`

$\{\texttt{x} = 0 \ \wedge \ 0 \leq n \ \wedge \texttt{y} = 1\}$

```
while (x < n) { x = x + 1; y = y * x}
```

$\{\texttt{y} = n!\}$

Invarianți pentru `while`

$\{\texttt{x} = 0 \ \wedge \ 0 \leq n \ \wedge \texttt{y} = 1\}$

```
while (x < n) { x = x + 1; y = y * x}
```

$\{\texttt{y} = n!\}$

- Invariantul I este $\texttt{y} = x!$

Invarianți pentru `while`

$\{\mathtt{x} = 0 \ \wedge \ 0 \leq n \ \wedge \mathtt{y} = 1\}$

```
while (x < n) { x = x + 1; y = y * x}
```

$\{\mathtt{y} = n!\}$

- Invariantul I este $\mathtt{y} = x!$
- $(\mathtt{x} = 0 \ \wedge \ 0 \leq n \ \wedge \mathtt{y} = 1) \rightarrow I$
- $\{I \wedge (x < n)\}$ `x = x + 1; y = y * x` $\{I\}$
- $I \wedge \neg(\mathtt{x} < n) \rightarrow (\mathtt{y} = n!)$

Pe data viitoare!

Fundamentele limbajelor de programare

C12

Denisa Diaconescu
Traian Șerbănuță

Departamentul de Informatică, FMI, UB

Imagine de ansamblu

Ce am parcurs la acest curs

- Lambda calcul fără tipuri
  - Terminologie
  - β-reducții
  - Strategii de evaluare
  - Expresivitate (codări)
  - Puncte fixe (combinatori de punct fix)

Ce am parcurs la acest curs

- Lambda calcul cu tipuri simple
  - Terminologie
  - Church-typing vs. Curry-typing
  - Sistem de deducție ($\lambda\!\rightarrow$) pentru calculul Church
  - Alte tipuri (unit, void, produs, suma)
  - Corespondența Curry-Howard
  - Type checking/ Type inference
  - Sistem de deducție ($\lambda\!\rightarrow$) cu constrângeri pentru calculul Church

Ce am parcurs la acest curs

- Unificare
  - Terminologie
  - Algoritmul de unificare

- Programare logică
  - Terminologie
  - Prolog
  - Logica clauzelor Horn
  - Sistem de deducție pentru logica clauzelor Horn (regula backchain)
  - Strategia de căutare din Prolog
  - Rezoluția SLD (arbori de căutare)

Ce am parcurs la acest curs

- Semantica limbajelor de programare
  - Terminologie
  - Semantica operațională small-step
  - Semantica axiomatică

Examen

Examen final

- valorează 4 puncte din nota finală
- durata 1 oră
- în sesiune, fizic
- acoperă toată materia
- exerciții asemănătoare cu exemplele de la curs (nu grile)
- ~~materiale ajutătoare: suporturile de curs și de laborator~~
  un material pus la dispoziție de noi și **o foaie** cu ce notițe vreți voi
- **aparatele electronice nu sunt permise la examen**

Model examen

Mulțumim că ați participat la acest curs!

Baftă la examen și mai departe!