Curs 6

λ-calcul

- În 1929-1932 Church a propus λ-calculul ca sistem formal pentru logica matematică. În 1935 a argumentat că orice funcție calculabilă peste numere naturale poate fi calculată in λ-calcul.
- În 1935, independent de Church, Turing a dezvoltat mecanismul de calcul numit astăzi Mașina Turing. În 1936 și el a argumentat câ orice funcție calculabilă peste numere naturale poate fi calculată de o mașină Turing. De asemenea, a arătat echivalența celor două modele de calcul. Această echivalență a constituit o indicație puternică asupra "universalității" celor două modele, conducând la ceea ce numim astăzi "Teza Church-Turing".

Referințe

- Benjamin C. Pierce, Types and Programming Languages, The MIT Press 2002
- J.R. Hindley, J.P. Seldin, Lambda-Calculus and Combinators, an Introduction, Cambridge University Press, 2008
- R. Nederpelt, H. Geuvers, Type Theory and Formal Proof, an Introduction, Cambridge University Press 2014

λ-calcul: sintaxa

Lambda Calcul - sintaxă

$$
\begin{array}{rll}
t = & x & \text{(variabilă)} \\
 & \mid \lambda x.\, t & \text{(abstractizare)} \\
 & \mid t\ \ t & \text{(aplicare)}
\end{array}
$$

λ-calcul: sintaxa

Lambda Calcul - sintaxă

$$
\begin{array}{rll}
t = & x & \text{(variabilă)} \\
 & \mid \lambda x.\, t & \text{(abstractizare)} \\
 & \mid t\ t & \text{(aplicare)}
\end{array}
$$

λ-termeni

Fie $Var = \{x, y, z, \ldots\}$ o mulțime infinită de variabile.
Mulțimea λ-termenilor ΛT este definită inductiv astfel:

[Variabilă] $Var \subseteq \Lambda T$

[Aplicare] dacă $t_1, t_2 \in \Lambda T$ atunci $(t_1 t_2) \in \Lambda T$

[Abstractizare] dacă $x \in Var$ și $t \in \Lambda T$ atunci $(\lambda x.t) \in \Lambda T$

Lambda termeni

λ-termeni: exemple

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

Lambda termeni

λ-termeni: exemple

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

Convenții:

- se elimină parantezele exterioare
- aplicarea este asociativă la stînga: $t_1 t_2 t_3$ este $(t_1 t_2)t_3$
- corpul abstractizării este extins la dreapta: $\lambda x.t_1 t_2$ este $\lambda x.(t_1 t_2)$ (nu $(\lambda x.t_1)t_2$)
- scriem $\lambda xyz.t$ în loc de $\lambda x.\lambda y.\lambda z.t$

Lambda termeni / Funcții anonime

λ-termeni: exemple

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

λ-termeni/ funcții anonime în Haskell

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`

Variabile libere și legate

Apariții libere și legate

Pentru un termen $\lambda x.t$ spunem că:

- aparițiile variabilei x în t sunt legate (*bound*)
- λx este legătura (*binder*), iar t este domeniul (*scope*) legării
- o apariție a unei variabile este liberă (*free*) dacă apare într-o poziție în care nu e legată.

Un termen fără variable libere se numește închis (*closed*).

Exemplu:

- $\lambda x.x$ este un termen închis
- $\lambda x.xy$ nu este termen închis, x este legată, y este liberă
- în termenul $x(\lambda x.xy)$ prima apariție a lui x este liberă, a doua este legată.

Variabile libere

Mulțimea variabilelor libere $FV(t)$

Pentru un λ-termen t mulțimea variabilelor libere este definită astfel:

[Variabilă] $FV(x) = x$

[Aplicare] $FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$

[Abstractizare] $FV(\lambda x.t) = FV(t) \setminus \{x\}$

Exemplu:

$$\begin{aligned} FV(\lambda x.xy) = \quad & FV(xy) \setminus \{x\} \\ & (FV(x) \cup FV(y)) \setminus \{x\} \\ & (\{x\} \cup \{y\}) \setminus \{x\} \\ & \{y\} \end{aligned}$$

Variabile libere

Mulțimea variabilelor libere $FV(t)$

Pentru un λ-termen t mulțimea variabilelor libere este definită astfel:

[Variabilă] $FV(x) = x$

[Aplicare] $FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$

[Abstractizare] $FV(\lambda x.t) = FV(t) \setminus \{x\}$

Exemplu:

$$
\begin{aligned}
FV(\lambda x.xy) = \quad & FV(xy) \setminus \{x\} \\
& (FV(x) \cup FV(y)) \setminus \{x\} \\
& (\{x\} \cup \{y\}) \setminus \{x\} \\
& \{y\}
\end{aligned}
$$

$FV(x\lambda x.xy) =$

Variabile libere

Mulțimea variabilelor libere $FV(t)$

Pentru un λ-termen t mulțimea variabilelor libere este definită astfel:

[Variabilă] $FV(x) = x$

[Aplicare] $FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$

[Abstractizare] $FV(\lambda x.t) = FV(t) \setminus \{x\}$

Exemplu:

$$
\begin{aligned}
FV(\lambda x.xy) = \quad & FV(xy) \setminus \{x\} \\
& (FV(x) \cup FV(y)) \setminus \{x\} \\
& (\{x\} \cup \{y\}) \setminus \{x\} \\
& \{y\} \\
FV(x\lambda x.xy) = \quad & \{x, y\}
\end{aligned}
$$

Substituții

Fie t un λ-termen $x \in Var$.

Definiție intuitivă

Pentru un λ-termen u vom nota prin $[u/x]t$ rezultatul înlocuirii tuturor aparițiilor libere ale lui x cu u în t.

Exemple: Dacă x, y, z sunt variabile distincte atunci

- $[y/x]\lambda z.x = \lambda z.y$
- $[(\lambda z.zw)/x](\lambda y.x) = \lambda y.\lambda z.zw$

Substituții

Definirea substituției

Rezultatul substituirii lui x cu u în t este definit astfel:

[Variabilă] $[u/x]x = u$

[Variabilă] $[u/x]y = y$ dacă $x \neq y$

[Aplicare] $[u/x](t_1 t_2) = [u/x]t_1 [u/x]t_2$

[Abstractizare] $[u/x]\lambda y.t = \lambda y.[u/x]t$ unde
$y \neq x$ și $y \notin FV(u)$

Substituții

Exemple: Dacă x, y, z sunt variabile distincte atunci

- $[y/x]\lambda z.x = \lambda z.y$
- Cine este $[y/x]\lambda y.x$?

Substituții

Exemple: Dacă x, y, z sunt variabile distincte atunci

- $[y/x]\lambda z.x = \lambda z.y$

- Cine este $[y/x]\lambda y.x$?
  Dacă folosim definiția intuitivă obținem
  $[y/x]\lambda y.x = \lambda y.y$ ceea ce este greșit!

Substituții

Exemple: Dacă x, y, z sunt variabile distincte atunci

- $[y/x]\lambda z.x = \lambda z.y$

- Cine este $[y/x]\lambda y.x$?
  Dacă folosim definiția intuitivă obținem
  $[y/x]\lambda y.x = \lambda y.y$ ceea ce este greșit!

  Cum procedăm pentru a repara greșeala? Observăm că $\lambda y.x$ desemneaza o funcție constantă, aceeași funcție putând fi reprezentată prin $\lambda z.x$. Aplicarea corectă a substituției este:
  $[y/x]\lambda y.x = [y/x]\lambda z.x = \lambda z.y$

Substituții

Exemple: Dacă x, y, z sunt variabile distincte atunci

- $[y/x]\lambda z.x = \lambda z.y$
- Cine este $[y/x]\lambda y.x$?
  Dacă folosim definiția intuitivă obținem
  $[y/x]\lambda y.x = \lambda y.y$ ceea ce este greșit!

  Cum procedăm pentru a repara greșeala? Observăm că $\lambda y.x$ desemneaza o funcție constantă, aceeași funcție putând fi reprezentată prin $\lambda z.x$. Aplicarea corectă a substituției este:
  $[y/x]\lambda y.x = [y/x]\lambda z.x = \lambda z.y$

  Avem libertatea de a redenumi variabilele legate!

α-conversie (α-echivalență)

α-conversia $=_\alpha$

[Reflexivitate] $t =_\alpha t$

[Simetrie] $t_1 =_\alpha t_2$ implică $t_2 =_\alpha t_1$

[Tranzitivitate] $t_1 =_\alpha t_2$ și $t_2 =_\alpha t_3$ implică $t_1 =_\alpha t_3$

[Redenumire] $\lambda x.t =_\alpha \lambda y.[y/x]t$ dacă $y \notin FV(t)$

[Compatibilitate] $t_1 =_\alpha t_2$ implică
$tt_1 =_\alpha tt_2$, $t_1 t =_\alpha t_2 t$ și $\lambda x.t_1 =_\alpha \lambda x.t_2$

α-conversie (α-echivalență)

α-conversia $=_\alpha$

[Reflexivitate] $t =_\alpha t$

[Simetrie] $t_1 =_\alpha t_2$ implică $t_2 =_\alpha t_1$

[Tranzitivitate] $t_1 =_\alpha t_2$ și $t_2 =_\alpha t_3$ implică $t_1 =_\alpha t_3$

[Redenumire] $\lambda x.t =_\alpha \lambda y.[y/x]t$ dacă $y \notin FV(t)$

[Compatibilitate] $t_1 =_\alpha t_2$ implică

$$tt_1 =_\alpha tt_2,\ t_1 t =_\alpha t_2 t \text{ și } \lambda x.t_1 =_\alpha \lambda x.t_2$$

$$t_1 =_\alpha t_2 \text{ și } u_1 =_\alpha u_2 \text{ implică } [u_1/x]t_1 =_\alpha [u_2/x]t_2$$

Exemplu:

$[xy/x](\lambda y.yx) =_\alpha [xy/x](\lambda z.zx) =_\alpha \lambda z.z(xy)$

α-conversie (α-echivalență)

α-conversia $=_\alpha$

[Reflexivitate] $t =_\alpha t$

[Simetrie] $t_1 =_\alpha t_2$ implică $t_2 =_\alpha t_1$

[Tranzitivitate] $t_1 =_\alpha t_2$ și $t_2 =_\alpha t_3$ implică $t_1 =_\alpha t_3$

[Redenumire] $\lambda x.t =_\alpha \lambda y.[y/x]t$ dacă $y \notin FV(t)$

[Compatibilitate] $t_1 =_\alpha t_2$ implică

$tt_1 =_\alpha tt_2$, $t_1 t =_\alpha t_2 t$ și $\lambda x.t_1 =_\alpha \lambda x.t_2$

$t_1 =_\alpha t_2$ și $u_1 =_\alpha u_2$ implică $[u_1/x]t_1 =_\alpha [u_2/x]t_2$

Exemplu:
$[xy/x](\lambda y.yx) =_\alpha [xy/x](\lambda z.zx) =_\alpha \lambda z.z(xy)$

Vom lucra modulo α-conversie, doi termeni α-echivalenți vor fi considerați "egali".

Substituție

Exemplu:

- $[xy/x]\lambda x.yx =_\alpha [xy/x]\lambda z.yz = \lambda z.[xy/x](yz) = \lambda z.yz$

  Observăm că $\lambda z.yz =_\alpha \lambda x.yx$

Substituție

Exemplu:

- $[xy/x]\lambda x.yx =_\alpha [xy/x]\lambda z.yz = \lambda z.[xy/x](yz) = \lambda z.yz$

  Observăm că $\lambda z.yz =_\alpha \lambda x.yx$

Substituție

Exemplu:

- $[xy/x]\lambda x.yx =_\alpha [xy/x]\lambda z.yz = \lambda z.[xy/x](yz) = \lambda z.yz$

  Observăm că $\lambda z.yz =_\alpha \lambda x.yx$

- $[y/z]\lambda xy.zzx = \lambda x.[y/z]\lambda y.zzx =_\alpha \lambda x.[y/z]\lambda v.zzx = \lambda x.\lambda v.[y/z](zzx) = \lambda xv.yyx$

β-reducție

β-reducția este o relație pe mulțimea α-termenilor.

β-reducția $\rightarrow_\beta$, $\stackrel{*}{\rightarrow}_\beta$

- un singur pas $\rightarrow_\beta \subseteq \Lambda T \times \Lambda T$

  [Aplicarea] $(\lambda x.t)u \rightarrow_\beta [u/x]t$

  [Compatibilitatea] $t_1 \rightarrow_\beta t_2$ implică

  $tt_1 \rightarrow_\beta tt_2$, $t_1 t \rightarrow_\beta t_2 t$ și $\lambda x.t_1 \rightarrow_\beta \lambda x.t_2$

- zero sau mai mulți pași $\stackrel{*}{\rightarrow}_\beta \subseteq \Lambda T \times \Lambda T$

  $t_1 \stackrel{*}{\rightarrow}_\beta t_2$ dacă există $n \geq 0$ și $u_0, \ldots, u_n$ astfel încât

  $t_1 =_\alpha u_0 \rightarrow_\beta u_1 \rightarrow_\beta \cdots \rightarrow_\beta u_n =_\alpha t_2$

β-reducție

Să considerăm termenul $(\lambda x.(\lambda y.yx)z)v$

- $(\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda y.yv)z \to_\beta zv$

β-reducție

Să considerăm termenul $(\lambda x.(\lambda y.yx)z)v$

- $(\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda y.yv)z \to_\beta zv$
- $(\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda x.zx)v \to_\beta zv$

β-reducție

Să considerăm termenul $(\lambda x.(\lambda y.yx)z)v$

- $(\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda y.yv)z \to_\beta zv$
- $(\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda x.zx)v \to_\beta zv$

Observăm că un termen poate fi β-redus în mai multe moduri.

Proprietatea de confluență ne asigură că vom ajunge întotdeauna la același rezultat.

Confluența β-reducției

Teorema Church-Rosser

Dacă $t \xrightarrow{*}_{\beta} t_1$ și $t \xrightarrow{*}_{\beta} t_2$

$$\begin{tikzcd} & t \arrow[dl, "*"'] \arrow[dr, "*"] & \\ t_1 & & t_2 \end{tikzcd}$$

Confluența β-reducției

Teorema Church-Rosser

Dacă $t \xrightarrow{*}_\beta t_1$ și $t \xrightarrow{*}_\beta t_2$

$$\begin{tikzcd}
& t \arrow[dl, "*"'] \arrow[dr, "*"] & \\
t_1 & & t_2
\end{tikzcd}$$

atunci există u astfel încât $t_1 \xrightarrow{*}_\beta u$ și $t_2 \xrightarrow{*}_\beta u$.

$$\begin{tikzcd}
& t \arrow[dl, "*"'] \arrow[dr, "*"] & \\
t_1 \arrow[dr, "*"'] & & t_2 \arrow[dl, "*"] \\
& u &
\end{tikzcd}$$

β-forma normală

Intuitiv, o formă normală este un termen care nu mai poate fi redus (sau punctul final al unui calcul).

Formă normal ă

- un λ-termen căruia nu i se mai poate aplica reducerea într-un pas $\rightarrow_\beta$ se numește *β-formă normală*
- dacă $t \stackrel{*}{\rightarrow}_\beta u_1$, $t \stackrel{*}{\rightarrow}_\beta u_2$ și u_1, u_2 sunt η-forme normale atunci, datorită confluenței, $u_1 =_\alpha u_2$
- un λ-termen poate avea cel mult o β-formă normală (modulo α-echivalență)

β-forma normală

Formă normal ă

- un λ-termen căruia nu i se mai poate aplica reducerea într-un pas $\to_\beta$ se numește *β-formă normală*
- dacă $t \overset{*}{\to}_\beta u_1$, $t \overset{*}{\to}_\beta u_2$ și u_1, u_2 sunt η-forme normale atunci, datorită confluenței, $u_1 =_\alpha u_2$
- un λ-termen poate avea cel mult o β-formă normală (modulo α-echivalență)

Exemplu:

- zv este β-formă normală pentru $(\lambda x.(\lambda y.yx)z)v$
  $(\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda y.yv)z \to_\beta zv$
- există termeni care **nu** pot fi reduși la o β-formă normală, de exemplu $(\lambda x.xx)(\lambda x.xx)$

β-conversia

Intuitiv, β-conversia extinde β-reducția în ambele direcții.

- $(\lambda y.yv)z \rightarrow_\beta zv \leftarrow_\beta (\lambda x.zx)v$

β-conversia

Intuitiv, β-conversia extinde β-reducția în ambele direcții.

- $(\lambda y.yv)z \to_\beta zv \leftarrow_\beta (\lambda x.zx)v$
- $(\lambda y.yv)z \leftarrow_\beta (\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda x.zx)v$

β-conversia

Intuitiv, β-conversia extinde β-reducția în ambele direcții.

- $(\lambda y.yv)z \to_\beta zv \leftarrow_\beta (\lambda x.zx)v$
- $(\lambda y.yv)z \leftarrow_\beta (\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda x.zx)v$

β-conversia $=_\beta$

- $=_\beta \subseteq \Lambda T \times \Lambda T$
  $t_1 =_\beta t_2$ dacă există $n \geq 0$ și $u_0, \ldots, u_n$ astfel încât
  $t_1 =_\alpha u_0$, $u_n =_\alpha t_2$ și, pentru orice i, $u_i \to_\beta u_{i+1}$ sau $u_{i+1} \to_\beta u_i$

β-conversia

Intuitiv, β-conversia extinde β-reducția în ambele direcții.

- $(\lambda y.yv)z \rightarrow_\beta zv \leftarrow_\beta (\lambda x.zx)v$
- $(\lambda y.yv)z \leftarrow_\beta (\lambda x.(\lambda y.yx)z)v \rightarrow_\beta (\lambda x.zx)v$

β-conversia $=_\beta$

- $=_\beta \subseteq \Lambda T \times \Lambda T$
  $t_1 =_\beta t_2$ dacă există $n \geq 0$ și $u_0, \ldots, u_n$ astfel încât
  $t_1 =_\alpha u_0$, $u_n =_\alpha t_2$ și, pentru orice i, $u_i \rightarrow_\beta u_{i+1}$ sau $u_{i+1} \rightarrow_\beta u_i$

Exemplu: $(\lambda y.yv)z =_\beta (\lambda x.zx)v$

β-conversia

β-conversia $=_\beta$

- $=_\beta \subseteq \Lambda T \times \Lambda T$
  $t_1 =_\beta t_2$ dacă există $n \geq 0$ și $u_0, \ldots, u_n$ astfel încât
  $t_1 =_\alpha u_0$, $u_n =_\alpha t_2$ și, pentru orice i, $u_i \to_\beta u_{i+1}$ sau $u_{i+1} \to_\beta u_i$

β-conversia

β-conversia $=_\beta$

- $=_\beta \subseteq \Lambda T \times \Lambda T$
  $t_1 =_\beta t_2$ dacă există $n \geq 0$ și $u_0, \ldots, u_n$ astfel încât
  $t_1 =_\alpha u_0$, $u_n =_\alpha t_2$ și, pentru orice i, $u_i \rightarrow_\beta u_{i+1}$ sau $u_{i+1} \rightarrow_\beta u_i$

Observații

- $=_\beta$ este o relație de echivalență

β-conversia

β-conversia $=_\beta$

- $=_\beta \subseteq \Lambda T \times \Lambda T$
  $t_1 =_\beta t_2$ dacă există $n \geq 0$ și $u_0, \ldots, u_n$ astfel încât
  $t_1 =_\alpha u_0$, $u_n =_\alpha t_2$ și, pentru orice i, $u_i \rightarrow_\beta u_{i+1}$ sau $u_{i+1} \rightarrow_\beta u_i$

Observații

- $=_\beta$ este o relație de echivalență
- pentru t_1, t_2 λ-termeni și u_1, u_2 β-forme normale
  dacă $t_1 \stackrel{*}{\rightarrow}_\beta u_1$, $t_2 \stackrel{*}{\rightarrow}_\beta u_2$ și $u_1 =_\alpha u_2$ atunci $t_1 =_\beta t_2$

β-conversia

β-conversia $=_\beta$

- $=_\beta \subseteq \Lambda T \times \Lambda T$
  $t_1 =_\beta t_2$ dacă există $n \geq 0$ și $u_0, \ldots, u_n$ astfel încât
  $t_1 =_\alpha u_0$, $u_n =_\alpha t_2$ și, pentru orice i, $u_i \to_\beta u_{i+1}$ sau $u_{i+1} \to_\beta u_i$
- $=_\beta$ este o relație de echivalență

β-conversia

β-conversia $=_\beta$

- $=_\beta \subseteq \Lambda T \times \Lambda T$
  $t_1 =_\beta t_2$ dacă există $n \geq 0$ și $u_0, \ldots, u_n$ astfel încât
  $t_1 =_\alpha u_0$, $u_n =_\alpha t_2$ și, pentru orice i, $u_i \rightarrow_\beta u_{i+1}$ sau $u_{i+1} \rightarrow_\beta u_i$
- $=_\beta$ este o relație de echivalență
- pentru t_1, t_2 λ-termeni și u_1, u_2 β-forme normale
  dacă $t_1 \stackrel{*}{\rightarrow}_\beta u_1$, $t_2 \stackrel{*}{\rightarrow}_\beta u_2$ și $u_1 =_\alpha u_2$ atunci $t_1 =_\beta t_2$

β-conversia reprezintă "egalitatea prin calcul", iar β-reducția (modulo α-conversie) oferă o procedură de decizie pentru aceasta.

Pe săptămâna viitoare!

Curs 7

λ-calcul și calculabilitate

- În 1929-1932 Church a propus λ-calculul ca sistem formal pentru logica matematică. În 1935 a argumentat că orice funcție calculabilă peste numere naturale poate fi calculată in λ-calcul.
- În 1935, independent de Church, Turing a dezvoltat mecanismul de calcul numit astăzi Mașina Turing. În 1936 și el a argumentat câ orice funcție calculabilă peste numere naturale poate fi calculată de o mașină Turing. De asemenea, a arătat echivalența celor două modele de calcul. Această echivalență a constituit o indicație puternică asupra "universalității" celor două modele, conducând la ceea ce numim astăzi "Teza Church-Turing".

λ-calcul: sintaxa

$$
\begin{array}{rlll}
t = & x & & \text{(variabilă)} \\
 & \mid (\lambda x.\, t) & & \text{(abstractizare)} \\
 & \mid (t\ t) & & \text{(aplicare)}
\end{array}
$$

Convenții:

- se elimină parantezele exterioare
- aplicarea este asociativă la stînga: $t_1 t_2 t_3$ este $(t_1 t_2) t_3$
- corpul abstractizării este extins la dreapta: $\lambda x. t_1 t_2$ este $\lambda x.(t_1 t_2)$ (nu $(\lambda x. t_1) t_2$)
- scriem $\lambda xyz.t$ în loc de $\lambda x.\lambda y.\lambda z.t$

λ-calcul: sintaxa

$$\begin{array}{lll} t = & x & \text{(variabilă)} \\ & \mid (\lambda x.\, t) & \text{(abstractizare)} \\ & \mid (t\ t) & \text{(aplicare)} \end{array}$$

Convenții:

- se elimină parantezele exterioare
- aplicarea este asociativă la stînga: $t_1 t_2 t_3$ este $(t_1 t_2) t_3$
- corpul abstractizării este extins la dreapta: $\lambda x.t_1 t_2$ este $\lambda x.(t_1 t_2)$ (nu $(\lambda x.t_1) t_2$)
- scriem $\lambda xyz.t$ în loc de $\lambda x.\lambda y.\lambda z.t$

Întrebare

Ce putem exprima / calcula folosind **doar** λ-calcul?

Rezumat

- Reprezentarea valorilor de adevăr și a expresiilor condiționale
- Reprezentarea perechilor (tuplurilor) și a funcțiilor proiecție
- Reprezentarea numerelor și a operatiilor aritmetice de bază
- Recursie

Ideea generală

Intuiție

Tipurile de date sunt codificate de capabilități

Boole	capabilitatea de a alege între două alternative
Perechi	capabilitatea de a calcula ceva bazat pe două valori
Numere naturale	capabilitatea de a itera de un număr dat de ori

Valori de adevăr

Intuiție: Capabilitatea de a alege între două alternative.

Codificare: Un Boolean este o funcție cu 2 argumente reprezentând ramurile unei alegeri.

true ::= $\lambda t\ f.t$ — din cele două alternative o alege pe prima

false ::= $\lambda t\ f.f$ — din cele două alternative o alege pe a doua

Operații Booleene

true ::= $\lambda t\ f.t$ — din cele două alternative o alege pe prima

false ::= $\lambda t\ f.f$ — din cele două alternative o alege pe a doua

`if` ::= $\lambda c\ then\ else.c\ then\ else$ — pur și simplu folosim valoarea de adevăr pentru a alege între alternative

`if` *false* $(\lambda x.x\ x)\ (\lambda x.x)$

Operații Booleene

true ::= $\lambda t\ f.t$ — din cele două alternative o alege pe prima

false ::= $\lambda t\ f.f$ — din cele două alternative o alege pe a doua

if ::= $\lambda c\ then\ else.c\ then\ else$ — pur și simplu folosim valoarea de adevăr pentru a alege între alternative
if $false\ (\lambda x.x\ x)\ (\lambda x.x) \rightarrow^3_\beta$
$false\ (\lambda x.x\ x)\ (\lambda x.x) \rightarrow^2_\beta \lambda x.x$

and ::= $\lambda b1\ b2.$ **if** $b1\ b2\ false$ sau $\lambda b1\ b2.b1\ b2\ b1$
and $true\ false$

Operații Booleene

true ::= $\lambda t\ f.t$ — din cele două alternative o alege pe prima

false ::= $\lambda t\ f.f$ — din cele două alternative o alege pe a doua

if ::= $\lambda c\ then\ else.c\ then\ else$ — pur și simplu folosim valoarea de adevăr pentru a alege între alternative
$\mathbf{if}\ false\ (\lambda x.x\ x)\ (\lambda x.x) \rightarrow_\beta^3$
$false\ (\lambda x.x\ x)\ (\lambda x.x) \rightarrow_\beta^2 \lambda x.x$

and ::= $\lambda b1\ b2.\ \mathbf{if}\ b1\ b2\ false$ sau $\lambda b1\ b2.b1\ b2\ b1$
$\mathbf{and}\ true\ false \rightarrow_\beta^2 true\ false\ true \rightarrow_\beta^2 false$

or ::= $\lambda b1\ b2.\ \mathbf{if}\ b1\ true\ b2$ sau $\lambda b1\ b2.b1\ b1\ b2$
$\mathbf{or}\ true\ false$

Operații Booleene

true ::= $\lambda t\ f.t$ — din cele două alternative o alege pe prima

false ::= $\lambda t\ f.f$ — din cele două alternative o alege pe a doua

if ::= $\lambda c\ then\ else.c\ then\ else$ — pur și simplu folosim valoarea de adevăr pentru a alege între alternative
if $false\ (\lambda x.x\ x)\ (\lambda x.x) \rightarrow^3_\beta$
$false\ (\lambda x.x\ x)\ (\lambda x.x) \rightarrow^2_\beta \lambda x.x$

and ::= $\lambda b1\ b2.$ **if** $b1\ b2\ false$ sau $\lambda b1\ b2.b1\ b2\ b1$
and $true\ false \rightarrow^2_\beta true\ false\ true \rightarrow^2_\beta false$

or ::= $\lambda b1\ b2.$ **if** $b1\ true\ b2$ sau $\lambda b1\ b2.b1\ b1\ b2$
or $true\ false \rightarrow^2_\beta true\ true\ false \rightarrow^2_\beta true$

not ::= $\lambda b.$ **if** $b\ false\ true$ sau $\lambda b\ t\ f.b\ f\ t$
not $true$

Operații Booleene

true ::= $\lambda t\ f.t$ — din cele două alternative o alege pe prima

false ::= $\lambda t\ f.f$ — din cele două alternative o alege pe a doua

if ::= $\lambda c\ then\ else.c\ then\ else$ — pur și simplu folosim valoarea de adevăr pentru a alege între alternative
$\mathbf{if}\ false\ (\lambda x.x\ x)\ (\lambda x.x) \to_\beta^3$
$false\ (\lambda x.x\ x)\ (\lambda x.x) \to_\beta^2 \lambda x.x$

and ::= $\lambda b1\ b2.\ \mathbf{if}\ b1\ b2\ false$ sau $\lambda b1\ b2.b1\ b2\ b1$
$\mathbf{and}\ true\ false \to_\beta^2 true\ false\ true \to_\beta^2 false$

or ::= $\lambda b1\ b2.\ \mathbf{if}\ b1\ true\ b2$ sau $\lambda b1\ b2.b1\ b1\ b2$
$\mathbf{or}\ true\ false \to_\beta^2 true\ true\ false \to_\beta^2 true$

not ::= $\lambda b.\ \mathbf{if}\ b\ false\ true$ sau $\lambda b\ t\ f.b\ f\ t$
$\mathbf{not}\ true \to_\beta \lambda t\ f.true\ f\ t \to_\beta \lambda t\ f.f$

Perechi

Intuiție: Capabilitatea de a aplica o funcție componentelor perechii

Codificare: O funcție cu 3 argumente reprezentând componentele perechii și funcția ce vrem să o aplicăm lor.

`pair` ::= $\lambda x\ y.\lambda f.f\ x\ y$
Constructorul de perechi

Exemplu: `pair` $3\ 5 \rightarrow_{\beta}^{2} \lambda f.f\ 3\ 5$

perechea $(3, 5)$ reprezintă capabilitatea de a aplica o funcție de două argumente lui 3 si apoi lui 5.

Operații pe perechi

$$\mathbf{pair} ::= \lambda x\, y.\lambda f.f\ x\ y$$

$$\mathbf{pair}\ xy \equiv_\beta f\ x\ y$$

$\mathbf{fst} ::= \lambda p.p\ \mathit{true}$ — *true* alege prima componentă

$\mathbf{fst}\ (\mathbf{pair}\ 3\ 5) \rightarrow_\beta \mathbf{pair}\ 3\ 5\ \mathit{true} \rightarrow^3_\beta \mathit{true}\ 3\ 5 \rightarrow^2_\beta 3$

$\mathbf{snd} ::= \lambda p.p\ \mathit{false}$ — *false* alege a doua componentă

$\mathbf{snd}\ (\mathbf{pair}\ 3\ 5) \rightarrow_\beta \mathbf{pair}\ 3\ 5\ \mathit{false} \rightarrow^3_\beta \mathit{false}\ 3\ 5 \rightarrow^2_\beta 5$

Numere naturale

Intuiție: Capabilitatea de a itera o funcție de un număr de ori peste o valoare inițială

Codificare: Un număr natural este o funcție cu 2 argumente

- s funcția care se iterează
- z valoarea inițială

0 ::= $\lambda s\ z.z$ — s se iterează de 0 ori, deci valoarea inițială

Numere naturale

Intuiție: Capabilitatea de a itera o funcție de un număr de ori peste o valoare inițială

Codificare: Un număr natural este o funcție cu 2 argumente

- s funcția care se iterează
- z valoarea inițială

0 ::= $\lambda s\ z.z$ — s se iterează de 0 ori, deci valoarea inițială

1 ::= $\lambda s\ z.s\ z$ — funcția iterată o dată aplicată valorii inițiale

Numere naturale

Intuiție: Capabilitatea de a itera o funcție de un număr de ori peste o valoare inițială

Codificare: Un număr natural este o funcție cu 2 argumente

- s funcția care se iterează
- z valoarea inițială

0 ::= $\lambda s\ z.z$ — s se iterează de 0 ori, deci valoarea inițială

1 ::= $\lambda s\ z.s\ z$ — funcția iterată o dată aplicată valorii inițiale

2 ::= $\lambda s\ z.s(s\ z)$ — s iterată de 2 ori, aplicată valorii inițiale

Numere naturale

Intuiție: Capabilitatea de a itera o funcție de un număr de ori peste o valoare inițială

Codificare: Un număr natural este o funcție cu 2 argumente

- s funcția care se iterează
- z valoarea inițială

0 ::= $\lambda s\ z.z$ — s se iterează de 0 ori, deci valoarea inițială

1 ::= $\lambda s\ z.s\ z$ — funcția iterată o dată aplicată valorii inițiale

2 ::= $\lambda s\ z.s(s\ z)$ — s iterată de 2 ori, aplicată valorii inițiale

...

8 ::= $\lambda s\ z.s(s(s(s(s(s(s(s\ z)))))))$

...

Numere naturale

Intuiție: Capabilitatea de a itera o funcție de un număr de ori peste o valoare inițială

Codificare: Un număr natural este o funcție cu 2 argumente

- s funcția care se iterează
- z valoarea inițială

0 ::= $\lambda s\ z.z$ — s se iterează de 0 ori, deci valoarea inițială

1 ::= $\lambda s\ z.s\ z$ — funcția iterată o dată aplicată valorii inițiale

2 ::= $\lambda s\ z.s(s\ z)$ — s iterată de 2 ori, aplicată valorii inițiale

...

8 ::= $\lambda s\ z.s(s(s(s(s(s(s(s\ z)))))))$

...

Observație: $0 = \mathit{false}$

Operații aritmetice de bază

$0 ::= \lambda s\ z.z$ — s se iterează de 0 ori, deci valoarea inițială

$8 ::= \lambda s\ z.s(s(s(s(s(s(s(s\ z)))))))$

$\mathbf{S} ::= \lambda n\ s\ z.s\ (n\ s\ z)$ sau $\lambda n\ s\ z.n\ s\ (sz)$

$\mathbf{S}\ 0$

Operații aritmetice de bază

0 ::= $\lambda s\ z.z$ — s se iterează de 0 ori, deci valoarea inițială

8 ::= $\lambda s\ z.s(s(s(s(s(s(s(s\ z)))))))$

S ::= $\lambda n\ s\ z.s\ (n\ s\ z)$ sau $\lambda n\ s\ z.n\ s\ (sz)$
$\mathbf{S}\ 0 \rightarrow_\beta \lambda s\ z.0s(sz) \rightarrow_\beta^2 \lambda s\ z.sz = 1$

+ ::= $\lambda m\ n.m\ \ \mathbf{S}\ \ n$ sau $\lambda m\ n.\lambda s\ z.m\ s\ (n\ s\ z)$
+ 3 2

Operații aritmetice de bază

$0 ::= \lambda s\ z.z$ — s se iterează de 0 ori, deci valoarea inițială

$8 ::= \lambda s\ z.s(s(s(s(s(s(s(s\ z)))))))$

$\mathbf{S} ::= \lambda n\ s\ z.s\ (n\ s\ z)$ sau $\lambda n\ s\ z.n\ s\ (sz)$

$\mathbf{S}\ 0 \rightarrow_\beta \lambda s\ z.0s(sz) \rightarrow_\beta^2 \lambda s\ z.sz = 1$

$+ ::= \lambda m\ n.m\ \ \mathbf{S}\ \ n$ sau $\lambda m\ n.\lambda s\ z.m\ s\ (n\ s\ z)$

$+\ 3\ 2 \rightarrow_\beta^2 \lambda s\ z.3\ s\ (2\ s\ z) \rightarrow_\beta^2$

$\lambda s\ z.s(s(s(2\ s\ z))) \rightarrow_\beta^2 \lambda s\ z.s(s(s(s(s\ z))) = 5$

$* ::= \lambda m\ n.m\ (+\ n)\ 0$ sau $\lambda m\ n.\lambda s.m\ (n\ s)$

$*\ 3\ 2$

Operații aritmetice de bază

$0 ::= \lambda s\ z.z$ — s se iterează de 0 ori, deci valoarea inițială

$8 ::= \lambda s\ z.s(s(s(s(s(s(s(s\ z)))))))$

$\mathbf{S} ::= \lambda n\ s\ z.s\ (n\ s\ z)$ sau $\lambda n\ s\ z.n\ s\ (sz)$

$\mathbf{S}\ 0 \rightarrow_\beta \lambda s\ z.0s(sz) \rightarrow_\beta^2 \lambda s\ z.sz = 1$

$+ ::= \lambda m\ n.m\ \ \mathbf{S}\ \ n$ sau $\lambda m\ n.\lambda s\ z.m\ s\ (n\ s\ z)$

$+\ 3\ 2 \rightarrow_\beta^2 \lambda s\ z.3\ s\ (2\ s\ z) \rightarrow_\beta^2$

$\lambda s\ z.s(s(s(2\ s\ z))) \rightarrow_\beta^2 \lambda s\ z.s(s(s(s(s\ z))) = 5$

$* ::= \lambda m\ n.m\ (+\ n)\ 0$ sau $\lambda m\ n.\lambda s.m\ (n\ s)$

$*\ 3\ 2 \rightarrow_\beta^2 3\ (+\ 2)0 \rightarrow_\beta^2 +\ 2(+\ 2(+\ 2\ 0)) \rightarrow_\beta^4$

$+\ 2(+\ 2\ 2) \rightarrow_\beta^4 +\ 2\ 4 \rightarrow_\beta^4 6$

$\mathbf{exp} ::= \lambda m\ n.n\ (*\ m)\ 1$ sau $\lambda m\ n.n\ m$

$\mathbf{exp}\ 3\ 2$

Operații aritmetice de bază

$\mathbf{0} ::= \lambda s\ z.z$ — s se iterează de 0 ori, deci valoarea inițială

$\mathbf{8} ::= \lambda s\ z.s(s(s(s(s(s(s(s\ z)))))))$

$\mathbf{S} ::= \lambda n\ s\ z.s\ (n\ s\ z)$ sau $\lambda n\ s\ z.n\ s\ (sz)$

$\mathbf{S}\ 0 \rightarrow_\beta \lambda s\ z.0s(sz) \rightarrow_\beta^2 \lambda s\ z.sz = 1$

$+ ::= \lambda m\ n.m\ \ \mathbf{S}\ \ n$ sau $\lambda m\ n.\lambda s\ z.m\ s\ (n\ s\ z)$

$+\ 3\ 2 \rightarrow_\beta^2 \lambda s\ z.3\ s\ (2\ s\ z) \rightarrow_\beta^2$

$\lambda s\ z.s(s(s(2\ s\ z))) \rightarrow_\beta^2 \lambda s\ z.s(s(s(s(s\ z)))) = 5$

$* ::= \lambda m\ n.m\ (+\ n)\ 0$ sau $\lambda m\ n.\lambda s.m\ (n\ s)$

$*\ 3\ 2 \rightarrow_\beta^2 3\ (+\ 2)0 \rightarrow_\beta^2 +\ 2(+\ 2(+\ 2\ 0)) \rightarrow_\beta^4$

$+\ 2(+\ 2\ 2) \rightarrow_\beta^4 +\ 2\ 4 \rightarrow_\beta^4 6$

$\mathbf{exp} ::= \lambda m\ n.n\ (*\ m)\ 1$ sau $\lambda m\ n.n\ m$

$\mathbf{exp}\ 3\ 2 \rightarrow_\beta^2 2\ 3 \rightarrow_\beta^2 \lambda z.3(3\ z) \rightarrow_\beta$

$\lambda z.\lambda z'.3\ z(3\ z(3\ z\ z')) \equiv_\alpha \lambda s\ z.3\ s(3\ s(3\ s\ z)) \rightarrow_\beta^6$

$\lambda s\ z.s(s(s(s(s(s(s(s(s\ z)))))))) = 9$

Scăderi și comparații

Presupunem că avem o funcție predecesor $\mathbf{P}\ x = \begin{cases} 0 & x = 0 \\ x - 1 & x \neq 0 \end{cases}$

Scăderi și comparații

Presupunem că avem o funcție predecesor $\mathbf{P}\ x = \begin{cases} 0 & x = 0 \\ x - 1 & x \neq 0 \end{cases}$

- $-$ ::= $\lambda m\ n.n\ \mathbf{P}\ m$ — dă 0 dacă $m \leq n$

Scăderi și comparații

Presupunem că avem o funcție predecesor $\mathbf{P}\ x = \begin{cases} 0 & x = 0 \\ x - 1 & x \neq 0 \end{cases}$

- ::= $\lambda m\ n.n\ \mathbf{P}\ m$ — dă 0 dacă $m \leq n$

?0= ::= $\lambda n.n(\lambda x.\mathit{false})\mathit{true}$ — testează dacă n e 0

Scăderi și comparații

Presupunem că avem o funcție predecesor $\mathbf{P}\ x = \begin{cases} 0 & x = 0 \\ x - 1 & x \neq 0 \end{cases}$

$\texttt{-} ::= \lambda m\ n.n\ \mathbf{P}\ m$ — dă 0 dacă $m \leq n$

$\texttt{?0=} ::= \lambda n.n(\lambda x.false)true$ — testează dacă n e 0

$\texttt{<=} ::= \lambda m\ n.\ \texttt{?0=}\ (\texttt{-}\ m\ n)$

Scăderi și comparații

Presupunem că avem o funcție predecesor $\mathbf{P}\ x = \begin{cases} 0 & x = 0 \\ x - 1 & x \neq 0 \end{cases}$

$$\texttt{-} ::= \lambda m\ n.n\ \mathbf{P}\ m \text{ — dă 0 dacă } m \leq n$$

$$\texttt{?0=} ::= \lambda n.n(\lambda x.\mathit{false})\mathit{true} \text{ — testează dacă } n \text{ e } 0$$

$$\texttt{<=} ::= \lambda m\ n.\ \texttt{?0=}\ (\texttt{-}\ m\ n)$$

$$\texttt{>} ::= \lambda m\ n.\ \texttt{not}\ (\texttt{<=}\ m\ n)$$

$$\texttt{>=} ::= \lambda m\ n.\ \texttt{<=}\ n\ m$$

$$\texttt{<} ::= \lambda m\ n.\ \texttt{>}\ n\ m$$

Scăderi și comparații

Presupunem că avem o funcție predecesor $\mathbf{P}\ x = \begin{cases} 0 & x = 0 \\ x - 1 & x \neq 0 \end{cases}$

$$
\begin{aligned}
\texttt{-} &::= \lambda m\ n.n\ \mathbf{P}\ m \text{ — dă 0 dacă } m \leq n \\
\texttt{?0=} &::= \lambda n.n(\lambda x.\mathit{false})\mathit{true} \text{ — testează dacă } n \text{ e } 0 \\
\texttt{<=} &::= \lambda m\ n.\ \texttt{?0=}\ (\texttt{-}\ m\ n) \\
\texttt{>} &::= \lambda m\ n.\ \texttt{not}\ (\texttt{<=}\ m\ n) \\
\texttt{>=} &::= \lambda m\ n.\ \texttt{<=}\ n\ m \\
\texttt{<} &::= \lambda m\ n.\ \texttt{>}\ n\ m \\
\texttt{=} &::= \lambda m\ n.\ \texttt{and}\ (\texttt{<=}\ m\ n)\ (\texttt{>=}\ m\ n) \\
\texttt{<>} &::= \lambda m\ n.\ \texttt{not}\ (\texttt{=}\ m\ n)
\end{aligned}
$$

Problemă

Cum definim funcția $\mathbf{P}$?

Definirea funcției **P** folosind perechi

$$\mathbf{P}\ x = \begin{cases} 0 & x = 0 \\ x - 1 & x \neq 0 \end{cases}$$

Definirea funcției $\mathbf{P}$ folosind perechi

$$\mathbf{P}\ x = \begin{cases} 0 & x = 0 \\ x - 1 & x \neq 0 \end{cases}$$

$$\mathbf{P}'' = \lambda n.n\ \mathbf{S}''\ (\mathbf{pair}\ 0\ 0)$$

Definirea funcției $\mathbf{P}$ folosind perechi

$$\mathbf{P}\ x = \begin{cases} 0 & x = 0 \\ x - 1 & x \neq 0 \end{cases}$$

$$\mathbf{P}'' = \lambda n.n\ \mathbf{S}''\ (\mathbf{pair}\ 0\ 0)$$

$$\mathbf{S}'' = \lambda p.(\lambda x.\ \mathbf{pair}\ x(\mathbf{S}\ x))\ (\mathbf{snd}\ p)$$

Definirea funcției $\mathtt{P}$ folosind perechi

$$\mathtt{P}\ x = \begin{cases} 0 & x = 0 \\ x - 1 & x \neq 0 \end{cases}$$

$$\mathtt{P''} = \lambda n.n\ \mathtt{S''}\ (\mathtt{pair}\ 0\ 0)$$

$$\mathtt{S''} = \lambda p.(\lambda x.\ \mathtt{pair}\ x(\mathtt{S}\ x))\ (\mathtt{snd}\ p)$$

$$\mathtt{P} = \lambda n.\ \mathtt{fst}\ (\mathtt{P''}\ n)$$

Definirea funcției $\mathbf{P}$ folosind perechi

$$\mathbf{P}\ x = \begin{cases} 0 & x = 0 \\ x - 1 & x \neq 0 \end{cases}$$

$$\mathbf{P}'' = \lambda n.n\ \mathbf{S}''\ (\mathbf{pair}\ 0\ 0)$$

$$\mathbf{S}'' = \lambda p.(\lambda x.\ \mathbf{pair}\ x(\mathbf{S}\ x))\ (\mathbf{snd}\ p)$$

$$\mathbf{P} = \lambda n.\ \mathbf{fst}\ (\mathbf{P}''\ n)$$

$\mathbf{P}\ 2 \to_\beta \mathbf{fst}\ (\mathbf{P}''\ 2) \to_\beta \mathbf{fst}\ (2\ \mathbf{S}''\ (\mathbf{pair}\ 0\ 0)) \to_\beta^2$
$\mathbf{fst}\ (\mathbf{S}''\ (\mathbf{S}''\ (\mathbf{pair}\ 0\ 0))) \to_\beta \mathbf{fst}\ (\mathbf{S}''\ (\mathbf{S}''\ (\mathbf{pair}\ 0\ 0))) \to_\beta$
$\mathbf{fst}\ (\mathbf{S}''\ ((\lambda x.\ \mathbf{pair}\ x(\mathbf{S}\ x))\ (\mathbf{snd}\ (\mathbf{pair}\ 0\ 0)))) \to_\beta^6$
$\mathbf{fst}\ (\mathbf{S}''\ ((\lambda x.\ \mathbf{pair}\ x(\mathbf{S}\ x))\ 0)) \to_\beta \mathbf{fst}\ (\mathbf{S}''\ (\mathbf{pair}\ 0(\mathbf{S}\ 0))) \to_\beta^8$
$\mathbf{fst}\ (\mathbf{pair}\ (\mathbf{S}\ 0)(\mathbf{S}\ (\mathbf{S}\ 0))) \to_\beta^6 \mathbf{S}\ 0 \to_\beta^3 1$

Funcția predecesor — definiție alternativă directă

Idee 1: $\mathbf{P} ::= \lambda n.\ \mathbf{?0=}\ n\ 0\ (\mathbf{P'}\ n)$ — am tratat primul caz. acum vrem o funcție $\mathbf{P'}$ care calculeaza $n - 1$ dacă $n \neq 0$

Idee 2: folosim iterația $\mathbf{P'} ::= \lambda n.n\ \mathbf{S'}\ Z'$, unde

- $\mathbf{S'}$ e un fel de succesor
- Z' e un fel de -1, i.e. $\mathbf{S'}\ Z' = 0$

$\mathbf{S'} ::= \lambda n.n\ \mathbf{S}\ 1$

$Z' ::= \lambda s\ z.0$ — Z' nu e codificarea unui număr
Dar se verifică că $\mathbf{S'}\ Z' \to_\beta Z'\ \mathbf{S}\ 1 \to_\beta^2 0$

Totul e OK deoarece $\mathbf{P'}$ e aplicată **doar** pe numere diferite de 0.

$\mathbf{P} ::= \lambda n.\ \mathbf{?0=}\ n\ 0\ (n\ (\lambda n.n\ \ \mathbf{S}\ 1)\ (\lambda s\ z.0))$

Liste

Intuiție: Capabilitatea de a agrega o listă

Codificare: O funcție cu 2 argumente
funcția de agregare și valoarea inițială

null ::= $\lambda a\ i.i$ — lista vidă

cons ::= $\lambda x\ l.\lambda a\ i.a\ x\ (l\ a\ i)$
Constructorul de liste

Exemplu: $\mathbf{cons}\ 3\ (\mathbf{cons}\ 5\ \mathbf{null}) \rightarrow_{\beta}^{2} \lambda a\ i.a\ 3\ (\mathbf{cons}\ 5\ \mathbf{null}\ a\ i) \rightarrow_{\beta}^{4} \lambda a\ i.a\ 3\ (a\ 5\ (\mathbf{null}\ a\ i)) \rightarrow_{\beta}^{2} \lambda a\ i.a\ 3\ (a\ 5\ i)$

Lista $[3, 5]$ reprezintă capabilitatea de a agrega elementele 3 si apoi 5 dată fiind o funcție de agregare a și o valoare implicită i.
Pentru aceste liste, operația de bază este `foldr`.

Operații pe liste

$\texttt{null} ::= \lambda a\ i.i$ — lista vidă

$\texttt{cons} ::= \lambda x\ l.\lambda a\ i.a\ x\ (l\ a\ i)$

Operații pe liste

null ::= $\lambda a\ i.i$ — lista vidă

cons ::= $\lambda x\ l.\lambda a\ i.a\ x\ (l\ a\ i)$

?null= ::= $\lambda l.l\ (\lambda x\ v.false)\ true$

Operații pe liste

`null` ::= $\lambda a\ i.i$ — lista vidă

`cons` ::= $\lambda x\ l.\lambda a\ i.a\ x\ (l\ a\ i)$

`?null=` ::= $\lambda l.l\ (\lambda x\ v.false)\ true$

`head` ::= $\lambda d\ l.l\ (\lambda x\ v.x)\ d$

primul element al listei, sau d dacă lista e vidă

Operații pe liste

$\textbf{null} ::= \lambda a\ i.i$ — lista vidă

$\textbf{cons} ::= \lambda x\ l.\lambda a\ i.a\ x\ (l\ a\ i)$

$\textbf{?null=} ::= \lambda l.l\ (\lambda x\ v.\mathit{false})\ \mathit{true}$

$\textbf{head} ::= \lambda d\ l.l\ (\lambda x\ v.x)\ d$
primul element al listei, sau d dacă lista e vidă

$\textbf{tail} ::= \lambda l.\ \textbf{fst}\ (l\ (\lambda x\ p.\ \textbf{pair}\ (\textbf{snd}\ p)\ (\textbf{cons}\ x\ (\textbf{snd}\ p)))\ (\textbf{pair}\ \textbf{null}\ \textbf{null}))$
coada listei, sau lista vidă dacă lista e vidă

$\textbf{foldr} ::= \lambda f\ i\ l.l\ f\ i$

$\textbf{map} ::= \lambda f\ l.l\ (\lambda x\ t.\ \textbf{cons}\ (f\ x)\ t)\ \ \textbf{null}$

$\textbf{filter} ::= \lambda p\ l.l\ (\lambda x\ t.p\ x\ (\textbf{cons}\ x\ t)\ t)\ \ \textbf{null}$

Factorial, Fibonacci, împărțire folosind perechi

fact ::= $\lambda n.$ **snd** (n ($\lambda p.$ **pair** (**S** (**fst** p))(***** (**fst** p)(**snd** p))) (**pair** 1 1))

Factorial, Fibonacci, împărțire folosind perechi

$\mathbf{fact} ::= \lambda n.\, \mathbf{snd}\ (n\ (\lambda p.\, \mathbf{pair}\ (\mathbf{S}\ (\mathbf{fst}\ p))(\mathbf{*}\ (\mathbf{fst}\ p)(\mathbf{snd}\ p)))\ (\mathbf{pair}\ 1\ 1))$

$\mathbf{fib} ::= \lambda n.\, \mathbf{fst}\ (n\ (\lambda p.\, \mathbf{pair}\ (\mathbf{snd}\ p)(+\ (\mathbf{fst}\ p)\ (\mathbf{snd}\ p)))\ (\mathbf{pair}\ 0\ 1))$

Factorial, Fibonacci, împărțire folosind perechi

fact ::= $\lambda n.$ **snd** $(n\ (\lambda p.$ **pair** (**S** (**fst** p))(***** (**fst** p)(**snd** p))) (**pair** 1 1))

fib ::= $\lambda n.$ **fst** $(n\ (\lambda p.$ **pair** (**snd** p)(**+** (**fst** p) (**snd** p))) (**pair** 0 1))

divMod ::= $\lambda m\ n.m\ (\lambda p. > n$ (**snd** p) p (**pair** (**S** (**fst** p)) (**-** (**snd** p) n))) (**pair** 0 m)

Factorial, Fibonacci, împărțire folosind perechi

$$\texttt{fact} ::= \lambda n.\, \mathbf{snd}\ (n\ (\lambda p.\, \mathbf{pair}\ (\mathbf{S}\ (\mathbf{fst}\ p))(*\ (\mathbf{fst}\ p)(\mathbf{snd}\ p)))\ (\mathbf{pair}\ 1\ 1))$$

$$\texttt{fib} ::= \lambda n.\, \mathbf{fst}\ (n\ (\lambda p.\, \mathbf{pair}\ (\mathbf{snd}\ p)(+\ (\mathbf{fst}\ p)\ (\mathbf{snd}\ p)))\ (\mathbf{pair}\ 0\ 1))$$

$$\texttt{divMod} ::= \lambda m\ n.m\ (\lambda p.\, >\ n\ (\mathbf{snd}\ p)\ p\ (\mathbf{pair}\ (\mathbf{S}\ (\mathbf{fst}\ p))\ (-\ (\mathbf{snd}\ p)\ n)))\ (\mathbf{pair}\ 0\ m)$$

Observații

- Nu toate funcțiile pot fi definite prin iterare fixată
- Pentru **divMod** obținem răspunsul (de obicei) din mult mai puțin de *m* iterații

Recursie

- există termeni care **nu** pot fi reduși la o β-formă normală, de exemplu
  $(\lambda x.x\ x)(\lambda x.x\ x) \to_\beta (\lambda x.x\ x)(\lambda x.x\ x)$
  În λ-calcul putem defini calcule infinite!

Recursie

- există termeni care **nu** pot fi reduși la o β-formă normală, de exemplu
  $(\lambda x.x\ x)(\lambda x.x\ x) \to_\beta (\lambda x.x\ x)(\lambda x.x\ x)$
  În λ-calcul putem defini calcule infinite!
- Dacă notăm $Af ::= \lambda x.f\ (x\ x)$ atunci
  $Af\ Af =_\beta (\lambda x.f\ (x\ x))\ Af =_\beta f\ (Af\ Af)$
- Dacă notăm $Yf ::= Af\ Af$ atunci $Yf =_\beta f\ Yf$.

Recursie

- există termeni care **nu** pot fi reduși la o β-formă normală, de exemplu
  $(\lambda x.x\ x)(\lambda x.x\ x) \rightarrow_\beta (\lambda x.x\ x)(\lambda x.x\ x)$
  În λ-calcul putem defini calcule infinite!
- Dacă notăm $Af ::= \lambda x.f\ (x\ x)$ atunci
  $Af\ Af =_\beta (\lambda x.f\ (x\ x))\ Af =_\beta f\ (Af\ Af)$
- Dacă notăm $Yf ::= Af\ Af$ atunci $Yf =_\beta f\ Yf$.
- Fie $Y ::= \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$
  Atunci $Y\ f =_\beta f\ (Y\ f)$

Puncte fixe

- pentru o funcție $f : X \to X$ un **punct fix** este un element $x_0 \in X$ cu $f(x_0) = x_0$.
  - $f : \mathbb{N} \to \mathbb{N}, f(x) = x + 1$ nu are puncte fixe
  - $f : \mathbb{N} \to \mathbb{N}, f(x) = 2x$ are punctul fix $x = 0$

Puncte fixe

- pentru o funcție $f : X \to X$ un **punct fix** este un element $x_0 \in X$ cu $f(x_0) = x_0$.
  - $f : \mathbb{N} \to \mathbb{N}, f(x) = x + 1$ nu are puncte fixe
  - $f : \mathbb{N} \to \mathbb{N}, f(x) = 2x$ are punctul fix $x = 0$
- În λ-calcul

$$\mathbf{Y} ::= \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$$

are proprietatea că $Y\ f =_\beta f\ (Y\ f)$, deci $Y\ f$ este un **punct fix** pentru f.

Puncte fixe

- pentru o funcție $f : X \to X$ un **punct fix** este un element $x_0 \in X$ cu $f(x_0) = x_0$.
  - $f : \mathbb{N} \to \mathbb{N}, f(x) = x + 1$ nu are puncte fixe
  - $f : \mathbb{N} \to \mathbb{N}, f(x) = 2x$ are punctul fix $x = 0$
- În λ-calcul

  $$\mathbf{Y} ::= \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$$

  are proprietatea că $Y\ f =_\beta f\ (Y\ f)$, deci $Y\ f$ este un **punct fix** pentru f.

  Y se numește **combinator de punct fix**.

Puncte fixe

- pentru o funcție $f : X \to X$ un **punct fix** este un element $x_0 \in X$ cu $f(x_0) = x_0$.
  - $f : \mathbb{N} \to \mathbb{N}, f(x) = x + 1$ nu are puncte fixe
  - $f : \mathbb{N} \to \mathbb{N}, f(x) = 2x$ are punctul fix $x = 0$
- În λ-calcul

  $$\mathbf{Y} ::= \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$$

  are proprietatea că $Y\ f =_\beta f\ (Y\ f)$, deci $Y\ f$ este un **punct fix** pentru f.

  Y se numește **combinator de punct fix.**

  Avem $Y\ f =_\beta f\ (Y\ f) =_\beta f\ (f\ (Y\ f)) =_\beta \ldots$

  Putem folosi Y pentru a obține apeluri recursive!

Puncte fixe - funcția factorial

$\texttt{fact} ::= \lambda n.\, \texttt{if}\, (\texttt{?0=}\ n)\ 1\ (\texttt{*}\ n\ (\mathit{fact}\ (\texttt{P}\ \ n)))$

Puncte fixe - funcția factorial

$$\texttt{fact} ::= \lambda n.\, \texttt{if}\ (\texttt{?0=}\ n)\ 1\ (\texttt{*}\ n\ (\mathit{fact}\ (\texttt{P}\ \ n)))$$

Această definiție nu este corectă conform regulilor noastre, cum procedăm?

Puncte fixe - funcția factorial

$$\texttt{fact} ::= \lambda n.\,\mathtt{if}\,(\mathtt{?0=}\ n)\ 1\ (\mathtt{*}\ n\ (\mathit{fact}\ (\mathtt{P}\ \ n)))$$

Această definiție nu este corectă conform regulilor noastre, cum procedăm?

- Pasul 1: abstractizăm, astfel încât construcția să fie corectă

$$\texttt{factA} ::= \lambda f.\lambda n.\,\mathtt{if}\,(\mathtt{?0=}\ n)\ 1\ (\mathtt{*}\ n\ (f\ (\mathtt{P}\ \ n)))$$

Puncte fixe - funcția factorial

$$\texttt{fact} ::= \lambda n.\ \texttt{if}\ (\texttt{?0=}\ n)\ 1\ (\texttt{*}\ n\ (\mathit{fact}\ (\texttt{P}\ \ n)))$$

Această definiție nu este corectă conform regulilor noastre, cum procedăm?

- Pasul 1: abstractizăm, astfel încât construcția să fie corectă

$$\texttt{factA} ::= \lambda f.\lambda n.\ \texttt{if}\ (\texttt{?0=}\ n)\ 1\ (\texttt{*}\ n\ (f\ (\texttt{P}\ \ n)))$$

- Pasul 2: aplicăm combinatorul de punct fix

$$\texttt{fact} ::= Y\ \texttt{factA}$$

Puncte fixe - funcția factorial

$$\texttt{fact} ::= \lambda n.\, \texttt{if}\, (\texttt{?0=}\, n)\, 1\, (\texttt{*}\, n\, (\mathit{fact}\, (\mathbf{P}\ \ n)))$$

Această definiție nu este corectă conform regulilor noastre, cum procedăm?

- Pasul 1: abstractizăm, astfel încât construcția să fie corectă

  $$\texttt{factA} ::= \lambda f.\lambda n.\, \texttt{if}\, (\texttt{?0=}\, n)\, 1\, (\texttt{*}\, n\, (f\, (\mathbf{P}\ \ n)))$$

- Pasul 2: aplicăm combinatorul de punct fix

  $$\texttt{fact} ::= Y\ \texttt{factA}$$

  Deoarece $Y\ \texttt{factA} =_\beta \texttt{factA}\, (Y\ \texttt{factA})$ obținem

$$\texttt{fact} =_\beta \lambda n.\, \texttt{if}\, (\texttt{?0=}\, n)\, 1\, (\texttt{*}\, n\, (\ \mathit{fact}\, (\mathbf{P}\ \ n))$$

divMod — definiție recursivă

Definiția primitiv recursivă (fără Y)

$$\texttt{divMod} ::= \lambda m\, n.m\ (\lambda p.\ >\ n\ (\mathbf{snd}\ p)\ p\ (\mathbf{pair}\ (\mathbf{S}\ (\mathbf{fst}\ p))\ (\mathbf{-}\ (\mathbf{snd}\ p)\ n)))\ (\mathbf{pair}\ 0\ m)$$

Definiția recursivă (incorectă)

$$\texttt{divMod} ::= \lambda m\, n.\ \mathbf{?0=}\ n\ (\mathbf{pair}\ 0\ m)\ (\mathbf{divMod'}\ \ 0\ m)\ \text{where}$$

$$\texttt{divMod'} ::= \lambda q\, r.\ >\ n\ r\ (\mathbf{pair}\ q\ r)\ (\mathbf{divMod'}\ \ (\mathbf{S}\ q)(\mathbf{-}\ m\ n))$$

Definiția recursivă (corectă, folosind Y)

$$\texttt{divMod} ::= \lambda m\, n.\ \mathbf{?0=}\ n\ \ (\mathbf{pair}\ 0\ m)\ (\mathbf{Y}\ (\lambda f.\lambda q\, r.\ >\ n\ r\ (\mathbf{pair}\ q\ r)\ (f\ (\mathbf{S}\ q)\ (\mathbf{-}\ m\ n)))\ 0\ m)$$

Apel prin valoare (Call by Value)

Pentru a reduce $e_1 \; e_2$:

- Reducem e_1 până la o funcție $\lambda x.e$
- Apoi reducem e_2 până la o valoare v
- Apoi reducem $(\lambda x.e) \; v$ la $[v/x]e$

Nu simplificăm sub λ

Apel prin valoare (Call by Value)

Pentru a reduce $e_1 \;\; e_2$:

- Reducem e_1 până la o funcție $\lambda x.e$
- Apoi reducem e_2 până la o valoare v
- Apoi reducem $(\lambda x.e) \;\; v$ la $[v/x]e$

Nu simplificăm sub λ

$$\texttt{?0=}\ 0\ (+\ 2\ 1)\ (+\ 3\ 4)$$
$$\rightarrow_{\beta(V)} 0\ (\lambda x.\mathit{false})\ \mathit{true}\ (+\ 2\ 1)\ (+\ 3\ 4)$$
$$\rightarrow^{2}_{\beta(V)} \mathit{true}\ (+\ 2\ 1)\ (+\ 3\ 4)$$
$$\rightarrow^{2}_{\beta(V)} \mathit{true}\ (\lambda s\ z.2\ s\ (1\ s\ z))\ (+\ 3\ 4)$$
$$\rightarrow_{\beta(V)} (\lambda f.\lambda s\ z.2\ s\ (1\ s\ z))\ (+\ 3\ 4)$$
$$\rightarrow^{2}_{\beta(V)} (\lambda f.\lambda s\ z.2\ s\ (1\ s\ z))\ (\lambda s\ z.3\ s\ (4\ s\ z))$$
$$\rightarrow_{\beta(V)} \lambda s\ z.2\ s\ (1\ s\ z)$$
$$\not\rightarrow_{\beta(V)}$$

$$\texttt{?0=} ::= (\lambda n.n\ (\lambda x.\mathit{false})\ \mathit{true})$$
$$0 ::= (\lambda s\ z.z)$$
$$+ ::= (\lambda m\ n\ s\ z.m\ s\ (n\ s\ z))$$
$$\mathit{true} ::= \lambda t\ f.t$$

Apel prin nume (Limbaje pur funcționale)

Pentru a reduce $e_1 \ \ e_2$:

- Reducem e_1 până la o funcție $\lambda x.e$
- Apoi reducem $(\lambda x.e) \ \ e_2$ la $[e_2/x]e$

Nu simplificăm sub λ nici în dreapta aplicației

Apel prin nume (Limbaje pur funcționale)

Pentru a reduce $e_1 \ \ e_2$:

- Reducem e_1 până la o funcție $\lambda x.e$
- Apoi reducem $(\lambda x.e) \ \ e_2$ la $[e_2/x]e$

Nu simplificăm sub λ nici în dreapta aplicației

$$
\begin{aligned}
&\mathtt{?0}{=}\ 0\ (+\ 2\ 1)\ (+\ 3\ 4)\\
&\rightarrow_{\beta(N)}\ 0\ (\lambda x.\mathit{false})\ \mathit{true}\ (+\ 2\ 1)\ (+\ 3\ 4)\\
&\rightarrow^2_{\beta(N)}\ \mathit{true}\ (+\ 2\ 1)\ (+\ 3\ 4)\\
&\rightarrow^2_{\beta(N)}\ (+\ 2\ 1)\\
&\rightarrow^2_{\beta(N)}\ (\lambda s\ z.2\ s\ (1\ s\ z))\\
&\not\rightarrow_{\beta(N)}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{?0}{=} &::= (\lambda n.n\ (\lambda x.\mathit{false})\ \mathit{true})\\
0 &::= (\lambda s\ z.z)\\
\mathit{true} &::= \lambda t\ f.t\\
+ &::= (\lambda m\ n\ s\ z.m\ s\ (n\ s\ z))
\end{aligned}
$$

Evaluare leneșă

Pentru a reduce $e_1 \; e_2$:

- Reduc e_1 până la o funcție $\lambda x.e$
- Apoi reduc corpul funcției e până la un e' care are nevoie de x
- Apoi reduc e_2 până la o valoare v
- Apoi reduc $(\lambda x.e') \; v$ la $[v/x]e'$

Evaluare nerestricționată

Pentru a reduce $e_1 \ \ e_2$

- Reducem fie e_1 fie e_2
- Putem reduce corpurile funcțiilor
- Oricând avem $(\lambda x.e')e''$, o putem (sau nu) reduce la $[e''/x]e'$
- Reduce până la o formă normală

Nu garantează găsirea unei forme normale

Evaluare „normală“

Pentru a reduce $e_1 \;\; e_2$

- Reducem mereu cel mai din stânga redex din cele de mai sus
- Reducem $(\lambda x.e')e''$ la $[e''/x]e'$
- Reducem e_1 (putem reduce și corpurile funcțiilor)
- Dacă $e_1 \not\rightarrow$, reducem e_2
- Reduce până la o formă normală
- Garantează găsirea unei forme normale

Apel prin valoare

Pentru a reduce $e_1 \ \ e_2$:

- Reducem e_1 până la o funcție $\lambda x.e$
- Apoi reducem e_2 până la o valoare v
- Apoi reducem $(\lambda x.e) \ \ v$ la $[v/x]e$

Reguli

$$\text{(V@S)} \quad \frac{e_1 \to_\beta e_1'}{e_1 \ \ e_2 \to_\beta e_1' \ \ e_2}$$

$$\text{(V@D)} \quad \frac{e_2 \to_\beta e_2'}{(\lambda x.e_1) \ \ e_2 \to_\beta (\lambda x.e_1) \ \ e_2'}$$

$$\text{(V@)} \quad (\lambda x.e_1) \ \ v_2 \to_\beta e \quad \textit{dacă } e = [v_2/x]e_1$$

Apel prin nume

Pentru a reduce $e_1 \;\; e_2$:

- Reducem e_1 până la o funcție $\lambda x.e$
- Apoi reducem $(\lambda x.e) \;\; e_2$ la $[e_2/x]e$

Reguli

$$(\text{N@S}) \quad \frac{e_1 \to_\beta e_1'}{e_1 \;\; e_2 \to_\beta e_1' \;\; e_2}$$

$$(\text{N@}) \quad (\lambda x.e_1) \;\; e_2 \to_\beta e \quad \textit{dacă } e = [e_2/x]e_1$$

Evaluare nerestricționată

Pentru a reduce $e_1 \;\; e_2$

- Reducem fie e_1 fie e_2
- Putem reduce corpurile funcțiilor
- Oricând avem $(\lambda x.e')e''$, o putem (sau nu) reduce la $[e''/x]e'$

Reguli

$$(\text{NR@S}) \quad \frac{e_1 \rightarrow e_1'}{e_1 \;\; e_2 \rightarrow e_1' \;\; e_2}$$

$$(\text{NR@D}) \quad \frac{e_2 \rightarrow e_2'}{e_1 \;\; e_2 \rightarrow e_1 \;\; e_2'}$$

$$(\text{NR}\textsc{Fun}\text{D}) \quad \frac{e \rightarrow e'}{\lambda x.e \rightarrow \lambda x.e'}$$

$$(\text{NR@}) \quad (\lambda x.e_1) \;\; e_2 \rightarrow e \quad \textit{dacă } e = [e_2/x]e_1$$

Evaluare „normală“

Pentru a reduce $e_1 \; e_2$

- Reducem mereu cel mai din stânga redex din cele de mai sus
- Reducem $(\lambda x.e')e''$ la $[e''/x]e'$
- Reducem e_1 (putem reduce și corpurile funcțiilor)
- Dacă $e_1 \nrightarrow$, reducem e_2

Reguli

$$(\text{NOR@}) \quad (\lambda x.e_1) \; e_2 \to e \quad \textit{dacă } e = [e_2/x]e_1$$

$$(\text{NOR@S}) \quad \frac{e_1 \to e_1'}{e_1 \; e_2 \to e_1' \; e_2} \quad \textit{dacă } e_1 \text{ nu e încă funcție}$$

$$(\text{NORFUND}) \quad \frac{e \to e'}{\lambda x.e \to \lambda x.e'}$$

$$(\text{NOR@D}) \quad \frac{e_1 \nrightarrow \qquad e_2 \to e_2'}{e_1 \; e_2 \to e_1 \; e_2'}$$

Evaluare leneșă

Pentru a reduce $e_1 \;\; e_2$:

- Reduc e_1 până la o funcție $\mathbf{fun}\ (x : T) \rightarrow e$
- Apoi reduc corpul funcției e până la un e' care are nevoie de x
- Apoi reduc e_2 până la o valoare v
- Apo reduc $(\lambda x.e') \;\; v$ la $[v/x]e'$

Reguli?

E mai complicat decât pare, deoarece trebuie să ne dăm seama că e' are nevoie de x.

Contexte de evaluare

- Găsirea redex-ului prin analiză sintactică
- Putem înlocui regulile structurale cu reguli gramaticale

Sintaxă: $e ::= x \mid \lambda x.e \mid e\ e$

Reguli structurale

$$\frac{e_1 \longrightarrow_\beta e_1'}{e_1\ e_2 \longrightarrow_\beta e_1'\ e_2}$$

$$\frac{e_2 \longrightarrow_\beta e_2'}{\langle(\lambda x.e_1)\ e_2 \longrightarrow_\beta (\lambda x.e_1)\ e_2'}$$

Contexte de evaluare

$$\begin{array}{lll} c & ::= & \blacksquare \\ & \mid & c\ e \\ & \mid & (\lambda x.e)\ c \end{array}$$

Instanțierea unui context c cu expresia e

$$c[e] = [e/\blacksquare]c$$

Contexte de evaluare

Sintaxă: $e ::= x \mid \lambda x.e \mid e\ e$

Contexte: $c ::= \blacksquare \mid c\ e \mid (\lambda x.e)\ c$

Exemple de contexte

Corecte	Greșite
$\blacksquare$	5
$3\ \blacksquare$	$true\ x\ \blacksquare$

Exemple de contexte instanțiate

- $\blacksquare[x\ 1] = x\ 1$
- $(9\ (\blacksquare\ 7))[x] = 9\ (x\ 7)$
- $(9\ (\blacksquare\ 7))[5] = 9\ (5\ 7)$

Semantica Operațională Contextuală

Un pas de execuție folosind contexte de evaluare

- Descompune expresia în contextul c și redex-ul r
- Aplică o regulă operațională asupra lui r obținând e
- Pune e în contextul inițial, obținând $c[e]$

$$\frac{r \longrightarrow_\beta e}{c[r] \longrightarrow_\beta c[e]}$$

Evaluare leneșă folosind Semantica Contextuală

Contexte de evaluare pentru aplicație

$$\begin{array}{lcl} c & ::= & \blacksquare \\ & | & c\ e \\ & | & (\lambda x.c)\ e \\ & | & (\lambda x.c[x])\ c \end{array}$$

Regulă de evaluare pentru aplicație

$$(\lambda x.c[x])\ v \rightarrow (\lambda x.c[v])\ v$$

Curs 8

Cuprins

Programare logică & Prolog

Programare logică

- Programarea logică este o paradigmă de programare bazată pe logică formală.

Programare logică

- Programarea logică este o paradigmă de programare bazată pe logică formală.
- Unul din sloganurile programării logice:

  Program = Logică + Control (R. Kowalski)

Programare logică

- Programarea logică este o paradigmă de programare bazată pe logică formală.

- Unul din sloganurile programării logice:

  Program = Logică + Control (R. Kowalski)

- Programarea logică poate fi privită ca o deducție controlată.

Programare logică

- Programarea logică este o paradigmă de programare bazată pe logică formală.
- 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

- Programarea logică este o paradigmă de programare bazată pe logică formală.
- 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.
- Exemple de limbaje de programare logică:
  - Prolog
  - Answer set programming (ASP)
  - Datalog

Ce veți vedea la laborator

Prolog

- bazat pe logica clauzelor Horn
- semantica operațională este bazată pe rezoluție
- este Turing complet
- vom folosi implementarea SWI-Prolog

Ce veți vedea la laborator

Prolog

- bazat pe logica clauzelor Horn
- semantica operațională este bazată pe rezoluție
- este Turing complet
- vom folosi implementarea SWI-Prolog

Limbajul Prolog este folosit pentru programarea sistemului IBM Watson!

Puteți citi mai multe detalii aici.

Learn Prolog Now!`http://www.let.rug.nl/bos/lpn/`

Programare logică - în mod idealist

- Un "program logic" este o colecție de proprietăți presupuse (sub formă de formule logice) despre lume (sau mai degrabă 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

$$\begin{array}{rcl}
\mathtt{oslo} & \rightarrow & \mathtt{windy} \\
\mathtt{oslo} & \rightarrow & \mathtt{norway} \\
\mathtt{norway} & \rightarrow & \mathtt{cold} \\
\mathtt{cold} \land \mathtt{windy} & \rightarrow & \mathtt{winterIsComing} \\
 & & \mathtt{oslo}
\end{array}$$

Exemplu de program logic

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

Exemplu de întrebare

Este adevărat `winterIsComing`?

Putem să testăm în SWI-Prolog

Program:

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

Intrebare:

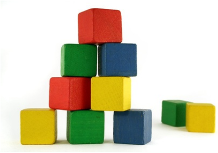
```
?- winterIsComing.
true
```

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

Sintaxă: constante, variabile, termeni compuși

- Atomi: `sansa, 'Jon Snow', jon_snow`
- Numere: `23, 23.03,-1`

  Atomii și numerele sunt constante.
- Variabile: `X, Stark, _house`
- Termeni compuși: `father(eddard, jon_snow),`
  `and(son(bran,eddard), daughter(arya,eddard))`
  - forma generală: atom(termen,. . ., termen)
  - atom-ul care denumește termenul se numește functor
  - numărul de argumente se numește aritate



Un mic exercițiu sintactic

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'

Un mic exercițiu sintactic

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

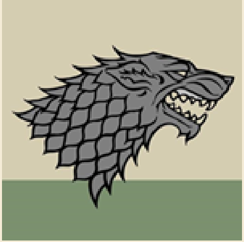
Un program în Prolog:

```
father(eddard,sansa).
father(eddard,jon_snow).

mother(catelyn,sansa).
mother(wylla,jon_snow).

stark(eddard).
stark(catelyn).

stark(X) :- father(Y,X),  stark(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(eddard,sansa).
father(eddard,jon_snow).

mother(catelyn,sansa).
mother(wylla,jon_snow).

stark(eddard).
stark(catelyn).

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

Predicate:
father/2
mother/2
stark/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.

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.

Exemplu

- Exemplu de regulă: `stark(X) :- father(Y,X), stark(Y).`
- Exemplu de fapt: `father(eddard, jon_snow).`

Interpretarea din punctul de vedere al logicii

- operatorul **:-** este implicația logică $\leftarrow$

Exemplu

`winterfell(X) :- stark(X)`

dacă `stark(X)` este adevărat, atunci `winterfell(X)` este adevărat.

Interpretarea din punctul de vedere al logicii

- operatorul :- este implicația logică $\leftarrow$

Exemplu

`winterfell(X) :- stark(X)`

dacă `stark(X)` este adevărat, atunci `winterfell(X)` este adevărat.

- virgula , este conjuncția $\wedge$

Exemplu

`stark(X) :- father(Y,X), stark(Y)`

dacă `father(Y,X)` și `stark(Y)` sunt adevărate,
atunci `stark(X)` este adevřat.

Interpretarea din punctul de vedere al logicii

- mai multe reguli cu același `Head` definesc același predicat, între defiții fiind un sau logic.

Exemplu

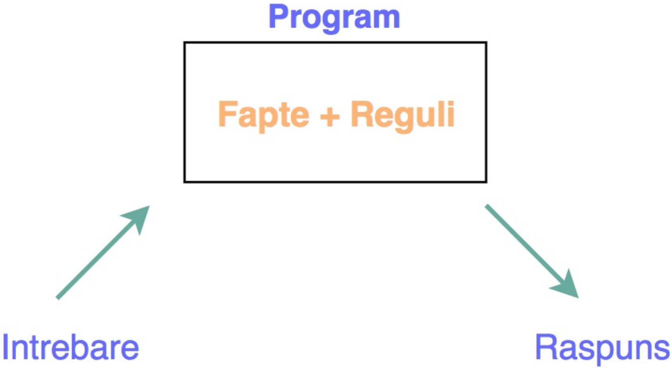
```
got_house(X) :- stark(X).
got_house(X) :- lannister(X).
got_house(X) :- targaryen(X).
got_house(X) :- baratheon(X).
```

dacă
`stark(X)` este adevărat sau `lannister(X)` este adevărat sau `targaryen(X)` este adevărat sau `baratheon(X)` este adevărat,
atunci
`got_house(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 se răspunde la o întrebare se numește țintă (goal).

Întrebări în Prolog

Prolog poate da 2 tipuri de răspunsuri:

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

Întrebări în Prolog

Prolog poate da 2 tipuri de răspunsuri:

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

Exemplu

```
?- stark(jon_snow)
true
?- stark(wylla)
false
```

```
?- stark(X)
X = eddard ;
X = catelyn ;
X = sansa ;
X = jon_snow ;
false
```

Cum găsește Prolog răspunsul

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

Exemplu

Să presupunem că avem programul:

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

și că punem următoarea întrebare:

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

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

Răspunsul la întrebare este găsit prin unificare!

Cum găsește Prolog răspunsul

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

Exemplu

Să presupunem că avem programul:

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

și că punem următoarea întrebare:

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

?- foo(d).
false
```

Dacaă nu se poate face potrivirea, răspunsul este false.

Cum găsește Prolog răspunsul

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

Exemplu

Să presupunem că avem programul:

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

și că punem următoarea întrebare:

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

Cum găsește Prolog răspunsul

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

Exemplu

Să presupunem că avem programul:

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

și că punem următoarea întrebare:

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

Dacă dorim mai multe răspunsuri, tastăm ;

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

Cum găsește Prolog răspunsul

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

Exemplu

Să presupunem că avem programul:

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

și că punem următoarea întrebare:
`?- 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.

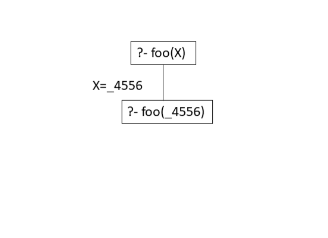
Exemplu

Să presupunem că avem programul:

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

și că punem următoarea întrebare:

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



Cum găsește Prolog răspunsul

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

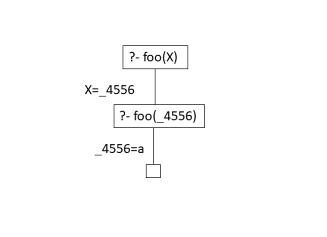
Exemplu

Să presupunem că avem programul:

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

și că punem următoarea întrebare:

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



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

Cum găsește Prolog răspunsul

Pentru a găsi un raspuns, Prolog încearcă clauzele în ordinea apariției lor.

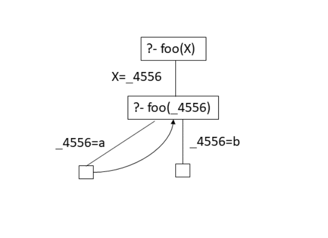
Exemplu

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

Pentru a găsi un raspuns, Prolog încearcă clauzele în ordinea apariției lor.

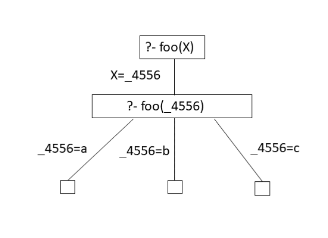
Exemplu

Să presupunem că avem programul:

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

și că punem următoarea întrebare:

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



arborele de căutare

Cum găsește Prolog răspunsul

Exemplu

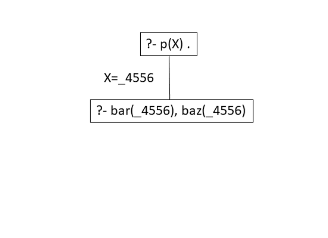
Să presupunem că avem programul:

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

baz(c).
```

și că punem următoarea întrebare:

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



Cum găsește Prolog răspunsul

Prolog se întoarce la ultima alegere dacă o sub-țintă eșuează.

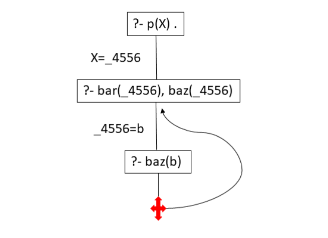
Exemplu

Să presupunem că avem programul:

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

și că punem următoarea întrebare:

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



Cum găsește Prolog răspunsul

Prolog se întoarce la ultima alegere dacă o sub-țintă eșuează.

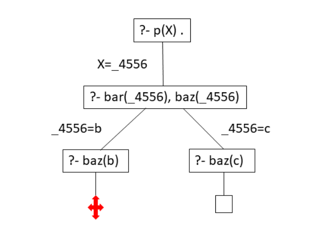
Exemplu

Să presupunem că avem programul:

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

și că punem următoarea întrebare:

`?- 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 următoarea întrebare:

```
?- 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 următoarea întrebare:

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

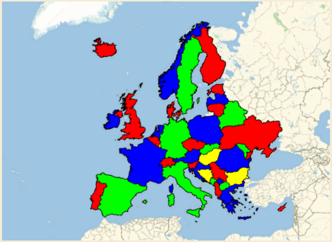
Vă explicați ce s-a întâmplat? Desenați arborele de căutare!

Un program mai complicat

Problema colorării hărților

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

Exemplu



Sursa imaginii

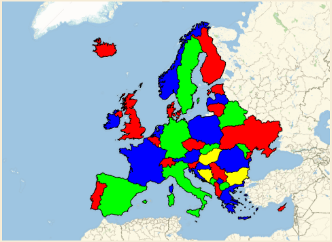
Un program mai complicat

Problema colorării hărților

<u>Să se coloreze o hartă dată cu un număr minim de culori astfel încât oricare două țări vecine să fie colorate diferit.</u>

Cum modelăm această problemă în Prolog?

Exemplu



Sursa imaginii

Un program mai complicat

Problema colorării hărților

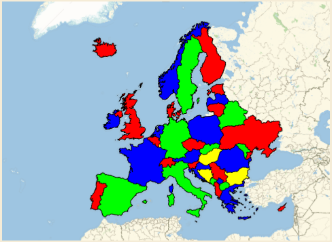
Să se coloreze o hartă dată cu un număr minim de culori astfel încât oricare două țări vecine să fie colorate diferit.

Cum modelăm această problemă în Prolog?

Exemplu

Trebuie să definim:

- culorile
- harta
- constrângerile



Sursa imaginii

Problema colorării hărților

Definim culorile

Exemplu

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

Problema colorării hărților

Definim culorile, harta

Exemplu

```
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).
```

Problema colorării hărților

Definim culorile, harta și constrângerile.

Exemplu

```
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?

Exemplu

```
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?

Exemplu

```
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?

Exemplu

```
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

Exemplu

```
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).
RO = albastru,
SE = UA, UA = rosu,
MD = BG, BG = HU, HU = verde
```

Compararea termenilor: =,\=, ==,\==

`T = U`	reușește dacă există o potrivire (termenii se unifică)
`T \= U`	reușește dacă nu există o potrivire
`T == U`	reușește dacă termenii sunt identici
`T \== U`	reușește dacă termenii sunt diferiți

Compararea termenilor: =,\=, ==,\==

`T = U`	reușește dacă există o potrivire (termenii se unifică)
`T \= U`	reușește dacă nu există o potrivire
`T == U`	reușește dacă termenii sunt identici
`T \== U`	reușește dacă termenii sunt diferiți

Exemplu

```
?- X = Y.
X = Y.

?- p(X,q(Z)) = p(Y,X).
X = Y, Y = q(Z).

?- 2 = 1 + 1
false
```

```
?- X == Y .
false

?- p(X,Y) == p(X,Y).
true

?- 2 == 1 + 1
false
```

- În exemplul de mai sus, `1+1` este privită ca o expresie, nu este evaluată. Există și predicate care forțează evaluarea.

Negarea unui predicat: `\+ pred(X)`

Exemplu

```
animal(dog). animal(elephant). animal(sheep).

?- animal(cat).
false
?- \+ animal(cat).
true
```

Negarea unui predicat: `\+ pred(X)`

Exemplu

```
animal(dog). animal(elephant). animal(sheep).

?- animal(cat).
false
?- \+ animal(cat).
true
```

- Clauzele din Prolog dau doar condiții suficiente, dar nu și necesare pentru ca un predicat să fie adevărat.
- Pentru a da un răspuns pozitiv la o țintă, Prolog trebuie să construiască o "demonstrație" pentru a arată că mulțimea de fapte și reguli din program implică acea țintă.
- Astfel, un răspuns `false` nu înseamnă neapărat că ținta nu este adevărată, ci doar că Prolog nu a reușit să găsească o demonstrație.

Operatorul \+

- Negarea unei ținte se poate defini astfel:

```
neg(Goal) :- Goal, !, fail.
neg(Goal)
```

  unde `fail/0` este un predicat care eșuează întotdeauna.

Operatorul \+

- Negarea unei ținte se poate defini astfel:

```
neg(Goal) :- Goal, !, fail.
neg(Goal)
```

  unde `fail/0` este un predicat care eșuează întotdeauna.
- În PROLOG acest predicat este predefinit sub numele `\+`.
- Operatorul `\+` se foloseste pentru a nega un predicat.
- `!`(cut) este un predicat predefinit (de aritate 0) care restricționează mecanismul de backtracking: execuția subțintei `!` se termină cu succes, deci alegerile (instanțierile) făcute înaite de a se ajunge la `!` nu mai pot fi schimbate.
- O țintă `\+ Goal` reușește dacă Prolog nu găsește o demonstrație pentru `Goal`. Negația din Prolog este definită ca incapacitatea de a găsi o demonstrație.
- Semantica operatorului `\+` se numește negation as failure.

Negația ca eșec ("negation as failure")

Exemplu

Să presupunem că avem o listă de fapte cu perechi de oameni căsătoriți între ei:

```
married(peter, lucy).
married(paul, mary).
married(bob, juliet).
married(harry, geraldine).
```

Negația ca eșec

Exemplu (cont.)

Putem să definim un predicat `single/1` care reușește dacă argumentul său nu este nici primul nici al doilea argument în faptele pentru `married`.

```
single(Person) :-
       \+ married(Person, _),
       \+ married(_, Person).
```

```
?- single(mary).     ?- single(anne).     ?- single(X).
false                true                 false
```

Răspunsul la întrebarea `?- single(anne).` trebuie gândit astfel:

Presupunem că Anne este single,
deoarece nu am putut demonstra că este maritată.

Predicatul -> /2 (`if-then-else`)

- `if-then`

```
If -> Then :- If, !, Then.
```

Predicatul -> /2 (if-then-else)

- if-then

```
If -> Then :- If, !, Then.
```

- if-then-else

```
If -> Then; _Else :- If, !, Then.
If -> Then; Else :- !, Else.
```

Se încearcă demonstrarea predicatului `If`. Dacă întoarce true atunci se încearcă demonstrarea predicatului `Then`, iar dacă întoarce false se încearcă demonstrarea predicatului `Else`.

```
max(X,Y,Z) :- (X =< Y) -> Z = Y ;  Z = X
```

```
?- max(2,3,Z).
Z = 3.
```

Predicatul -> /2 (`if-then-else`)

- `if-then`

```
If -> Then :- If, !, Then.
```

- `if-then-else`

```
If -> Then; _Else :- If, !, Then.
If -> Then; Else :- !, Else.
```

  Se încearcă demonstrarea predicatului `If`. Dacă întoarce true atunci se încearcă demonstrarea predicatului `Then`, iar dacă întoarce false se încearcă demonstrarea predicatului `Else`.

```
max(X,Y,Z) :- (X =< Y) -> Z = Y ;  Z = X
```

```
?- max(2,3,Z).
Z = 3.
```

Observăm că `If -> Then` este echivalent cu `If -> Then ; fail`.

Tipuri de date compuse

Termeni compuși `f(t1,..., tn)`

- Termenii sunt unitățile de bază prin care Prolog reprezintă datele.
- Sunt de 3 tipuri:
  - Constante: `23, sansa, 'Jon Snow'`
  - Variabile: `X, Stark, _house`
  - Termeni compuși:
    - predicate
    - termeni prin care reprezentăm datele

Exemplu

- `born(john, date(20,3,1977))`
  - `born/2` și `date/3` sunt functori
  - `born/2` este un predicat
  - `date/3` definește date compuse

Tipuri de date definite recursiv

- Am văzut că listele sunt definite recursiv astfel:
  - `[]` este listă
  - `[X|L]` este listă, unde `X` este element și `L` este listă

Tipuri de date definite recursiv

- Am văzut că listele sunt definite recursiv astfel:
  - `[]` este listă
  - `[X|L]` este listă, unde `X` este element și `L` este listă

- Cum definim arborii binari în Prolog?

Tipuri de date definite recursiv

- Am văzut că listele sunt definite recursiv astfel:
  - `[]` este listă
  - `[X|L]` este listă, unde `X` este element și `L` este listă
- Cum definim arborii binari în Prolog? Soluție posibilă:

Tipuri de date definite recursiv

- Am văzut că listele sunt definite recursiv astfel:
  - `[]` este listă
  - `[X|L]` este listă, unde `X` este element și `L` este listă
- Cum definim arborii binari în Prolog? Soluție posibilă:
  - `void` este arbore

Tipuri de date definite recursiv

- Am văzut că listele sunt definite recursiv astfel:
  - `[]` este listă
  - `[X|L]` este listă, unde `X` este element și `L` este listă

- Cum definim arborii binari în Prolog? Soluție posibilă:
  - `void` este arbore
  - `tree(X,A1,A2)` este arbore, unde `X` este un element, iar `A1` și `A2` sunt arbori

Tipuri de date definite recursiv

- Am văzut că listele sunt definite recursiv astfel:
  - `[]` este listă
  - `[X|L]` este listă, unde `X` este element și `L` este listă
- Cum definim arborii binari în Prolog? Soluție posibilă:
  - `void` este arbore
  - `tree(X,A1,A2)` este arbore, unde `X` este un element, iar `A1` și `A2` sunt arbori

`tree(X,A1,A2)` este un termen compus, dar nu este un predicat!

Arbori binari în Prolog

- Cum arată un arbore?

Arbori binari în Prolog

- Cum arată un arbore?

  tree(a, tree(b, tree(d, void, void), void), tree(c, void, tree(e, void, void)))

Arbori binari în Prolog

- Cum arată un arbore?

  tree(a, tree(b, tree(d, void, void), void), tree(c, void, tree(e, void, void)))

- Cum dăm un "nume" arborelui de mai sus?

Arbori binari în Prolog

- Cum arată un arbore?

  tree(a, tree(b, tree(d, void, void), void), tree(c, void, tree(e, void, void)))

- Cum dăm un "nume" arborelui de mai sus?   Definim un predicat:

```
def(arb, tree(a, tree(b,
                          tree(d,void,void),
                          void),
                 tree(c, void,
                          tree(e,void,void)))).
```

Arbori binari în Prolog

- Cum arată un arbore?

  tree(a, tree(b, tree(d, void, void), void), tree(c, void, tree(e, void, void)))

- Cum dăm un "nume" arborelui de mai sus? Definim un predicat:

```
def(arb, tree(a, tree(b,
                          tree(d,void,void),
                          void),
                 tree(c, void,
                          tree(e,void,void)))).
```

Deoarece în Prolog nu avem declarații explicite de date, pentru a defini arborii vom scrie un predicat care este adevărat atunci când argumentul său este un arbore.

Arbori binari în Prolog

- Scrieți un predicat care verifică că un termen este arbore binar.

Arbori binari în Prolog

- Scrieți un predicat care verifică că un termen este arbore binar.

```
binary_tree(void).
binary_tree(tree(Element,Left,Right)) :- binary_tree(Left),
                                         binary_tree(Right).
```

Arbori binari în Prolog

- Scrieți un predicat care verifică că un termen este arbore binar.

```
binary_tree(void).
binary_tree(tree(Element,Left,Right)) :- binary_tree(Left),
                                          binary_tree(Right).
```

Eventual putem defini și un predicat pentru elemente:

```
binary_tree(void).
binary_tree(tree(Element,Left,Right)) :- binary_tree(Left),
                                      binary_tree(Right),
                                      element_binary_tree(Element)

element_binary_tree(X):- integer(X). /* de exemplu */
```

Arbori binari în Prolog

- Scrieți un predicat care verifică că un termen este arbore binar.

```
binary_tree(void).
binary_tree(tree(Element,Left,Right)) :- binary_tree(Left),

                                          binary_tree(Right).
```

Eventual putem defini și un predicat pentru elemente:

```
binary_tree(void).
binary_tree(tree(Element,Left,Right)) :- binary_tree(Left),

                                     binary_tree(Right),
                                     element_binary_tree(Element)

element_binary_tree(X):- integer(X). /* de exemplu */


test:- def(arb,T), binary_tree(T).
```

Arbori binari în Prolog

Exercițiu

Scrieți un predicat care verifică că un element aparține unui arbore.

```
tree_member(X,tree(X,Left,Right)).

tree_member(X,tree(Y,Left,Right)) :- tree_member(X,Left).

tree_member(X,tree(Y,Left,Right)) :- tree_member(X,Right).
```

Arbori binari în Prolog

Exercițiu

Scrieți un predicat care determină parcurgerea în preordine a unui arbore binar.

```
preorder(tree(X,L,R),Xs) :- preorder(L,Ls),
                            preorder(R,Rs),
                            append([X|Ls],Rs,Xs).
```

Arbori binari în Prolog

Exercițiu

Scrieți un predicat care determină parcurgerea în preordine a unui arbore binar.

```
preorder(tree(X,L,R),Xs) :- preorder(L,Ls),
                            preorder(R,Rs),
                            append([X|Ls],Rs,Xs).
preorder(void,[]).
```

Arbori binari în Prolog

Exercițiu

Scrieți un predicat care determină parcurgerea în preordine a unui arbore binar.

```
preorder(tree(X,L,R),Xs) :- preorder(L,Ls),
                            preorder(R,Rs),
                            append([X|Ls],Rs,Xs).
preorder(void,[]).

test(Tree,Pre):- def(arb, Tree), preorder(Tree,Pre).
```

Arbori binari în Prolog

Exercițiu

Scrieți un predicat care determină parcurgerea în preordine a unui arbore binar.

```
preorder(tree(X,L,R),Xs) :- preorder(L,Ls),
                            preorder(R,Rs),
                            append([X|Ls],Rs,Xs).
preorder(void,[]).

test(Tree,Pre):- def(arb, Tree), preorder(Tree,Pre).
```

```
?- test(T,P).
T = tree(a, tree(b, tree(d, void, void), void), tree(c,
void, tree(e, void, void))),
P = [a, b, d, c, e]
```

Liste și recursie

Listă [t1,...,tn]

- 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 `|` delimitează 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.
```

Listă [t1,...,tn]==[t1 |[t2,...,tn]

- Simbolul | delimitează coada listei:

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

- Variabila anonimă _ este unificată cu orice termen Prolog:

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

- Deoarece Prologul face unificare poate identifica șabloane mai complicate:

```
?- [5,1,1,3,2]=[_|[X|[X|_]]].
X = 1.
?- [5,1,4,3,2]=[_|[X|[X|_]]].
false.
```

Liste

Exercițiu

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

```
is_list([]).
is_list([_|_]).
```

Liste

Exercițiu

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

```
is_list([]).
is_list([_|_]).
```

- 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).
```

Exercițiu

- Definiți un predicat care verifică dacă un termen aparține unei liste.

```
member(H, [H|_]).
member(H, [_|T]) :- member(H,T).
```

Liste

Exercițiu

- Definiți un predicat care verifică dacă un termen aparține unei liste.

```
member(H, [H|_]).
member(H, [_|T]) :- member(H,T).
```

- Definiți un predicat `append/3` care verifică dacă o listă se obține prin concatenarea altor două liste.

```
append([],L,L).
append([X|T],L, [X|R]) :- append(T,L,R).
```

Liste

Exercițiu

- Definiți un predicat care verifică dacă un termen aparține unei liste.

```
member(H, [H|_]).
member(H, [_|T]) :- member(H,T).
```

- Definiți un predicat `append/3` care verifică dacă o listă se obține prin concatenarea altor două liste.

```
append([],L,L).
append([X|T],L, [X|R]) :- append(T,L,R).
```

Există predicatele predefinite `member/2` și `append/3`.

Liste append/3

- Funcția append/3:

```
?- listing(append/3).
append([],L,L).
append([X|T],L, [X|R]) :- append(T,L,R).

?- append(X,Y,[a,b,c]).
X = [],
Y = [a, b, c] ;
X = [a],
Y = [b, c] ;
X = [a, b],
Y = [c] ;
X = [a, b, c],
Y = [] ;
false
```

- Funcția astfel definită poate fi folosită atât pentru verificare, cât și pentru generare.

Liste

Exercițiu

- Definiți un predicat `elim/3` care verifică dacă o listă se obține din alta prin eliminarea unui element.

Liste

Exercițiu

- Definiți un predicat `elim/3` care verifică dacă o listă se obține din alta prin eliminarea unui element.

```
elim(X, [X|T], T).
elim(X, [H|T], [H|L]) :- elim(X,T,L).
```

Liste

Exercițiu

- Definiți un predicat `elim/3` care verifică dacă o listă se obține din alta prin eliminarea unui element.

```
elim(X, [X|T], T).
elim(X, [H|T], [H|L]) :- elim(X,T,L).
```

- Definiți un predicat care `perm/2` care verifică dacă două liste sunt permutări.

Liste

Exercițiu

- Definiți un predicat `elim/3` care verifică dacă o listă se obține din alta prin eliminarea unui element.

```
elim(X, [X|T], T).
elim(X, [H|T], [H|L]) :- elim(X,T,L).
```

- Definiți un predicat care `perm/2` care verifică dacă două liste sunt permutări.

```
perm([],[]). perm([X|T],L) :- elim(X,L,R), perm(R,T).
```

Liste

Exercițiu

- Definiți un predicat `elim/3` care verifică dacă o listă se obține din alta prin eliminarea unui element.

```
elim(X, [X|T], T).
elim(X, [H|T], [H|L]) :- elim(X,T,L).
```

- Definiți un predicat care `perm/2` care verifică dacă două liste sunt permutări.

```
perm([],[]). perm([X|T],L) :- elim(X,L,R), perm(R,T).
```

Predicatele predefinite `select/3` și `permutation/2` au aceeași funcționalitate.

Generează și testează

```
solution(X) :- generate(X), check(X).
```

Exercițiu

Determinați toate cuvintele dintr-o bază de cunoștințe dată, care sunt anagrame ale unui cuvânt dat.

KB: `word(relay).` `word(early).` `word(layer).`

Predicat util:

```
?- name(relay,L). % conversie între atomi și liste
L = [114, 101, 108, 97, 121]
```

Generează și testează

```
solution(X) :- generate(X), check(X).
```

Exercițiu

Determinați toate cuvintele dintr-o bază de cunoștințe dată, care sunt anagrame ale unui cuvânt dat.

KB: `word(relay).` `word(early).` `word(layer).`

Predicat util:

```
?- name(relay,L). % conversie între atomi și liste
L = [114, 101, 108, 97, 121]
```

Două abordări posibile:

- se generează o posibilă, soluție apoi se testează dacă este în KB.
- se parcurge KB și pentru fiecare termen se testează dacă e soluție.

Generează și testează

```
solution(X) :- generate(X), check(X).
```

Exercițiu

Determinați toate cuvintele dintr-o bază de cunoștințe dată, care sunt anagrame ale unui cuvânt dat.

KB: `word(relay).` `word(early).` `word(layer).`

Generează și testează

```
solution(X) :- generate(X), check(X).
```

Exercițiu

Determinați toate cuvintele dintr-o bază de cunoștințe dată, care sunt anagrame ale unui cuvânt dat.

KB: `word(relay).` `word(early).` `word(layer).`

```
anagram1(A,B) :- name(A,L), permutation(L,W),
                 name(B,W), word(B).
```

Generează și testează

```
solution(X) :- generate(X), check(X).
```

Exercițiu

Determinați toate cuvintele dintr-o bază de cunoștințe dată, care sunt anagrame ale unui cuvânt dat.

KB: `word(relay).` `word(early).` `word(layer).`

```
anagram1(A,B) :- name(A,L), permutation(L,W),
                 name(B,W), word(B).
anagram2(A,B) :- name(A,L), word(B),
                 name(B,W), permutation(L,W).
```

Generează și testează

```
solution(X) :- generate(X), check(X).
```

Exercițiu

Determinați toate cuvintele dintr-o bază de cunoștințe dată, care sunt anagrame ale unui cuvânt dat.

KB: `word(relay).  word(early).  word(layer).`

```
anagram1(A,B) :- name(A,L), permutation(L,W),
                 name(B,W), word(B).
anagram2(A,B) :- name(A,L), word(B),
                 name(B,W), permutation(L,W).
```

```
?- anagram1(layre,X).
X = layer ;
X = relay ;
X = early ;
false.
```

```
?- anagram2(layre,X).
X = relay ;
X = early ;
X = layer ;
false.
```

Recursie

Exercițiu

- Definiți un predicat `rev/2` care verifică dacă o listă este inversa altei liste.

Recursie

Exercițiu

- Definiți un predicat `rev/2` care verifică dacă o listă este inversa altei liste.

```
rev([],[]).
rev([X|T],L) :- rev(T,R),append(R,[X],L).
```

Soluția de mai sus este corectă, dar foarte costisitoare computațional, datorită stilului de programare declarativ.

Cum putem defini o variantă mai rapidă?

O metodă care prin care recursia devine mai rapidă este folosirea acumulatorilor, în care se păstrează rezultatele parțiale.

Recursie cu acumulatori

- Varianta inițială:

```
rev([],[]).
rev([X|T],L) :- rev(T,R),append(R,[X],L).
```

- Varianta cu acumulator

```
rev(L,R) :- revac(L,[],R).
% la momentul inițial nu am acumulat nimic.
```

Recursie cu acumulatori

- Varianta inițială:

```
rev([],[]).
rev([X|T],L) :- rev(T,R),append(R,[X],L).
```

- Varianta cu acumulator

```
rev(L,R) :- revac(L,[],R).
% la momentul inițial nu am acumulat nimic.

revac([], R, R).
% cand lista inițială a fost consumată,
% am acumulat rezultatul final.
```

Recursie cu acumulatori

- Varianta inițială:

```
rev([],[]).
rev([X|T],L) :- rev(T,R),append(R,[X],L).
```

- Varianta cu acumulator

```
rev(L,R) :- revac(L,[],R).
% la momentul inițial nu am acumulat nimic.

revac([], R, R).
% cand lista inițială a fost consumată,
% am acumulat rezultatul final.

revac([X|T], Acc, R) :- revac(T,[X|Acc],R).
% Acc conține inversa listei care a fost deja parcursă.
```

- Complexitatea a fost redusă de la $O(n^2)$ la $O(n)$, unde n este lungimea listei.

Recursie

- Multe implementări ale limbajului Prolog aplică "last call optimization" atunci când un apel recursiv este ultimul predicat din corpul unei clauze (tail recursion).
- Atunci când este posibil, se recomandă utilizare recursiei la coadă (tail recursion).
- Vom defini un predicat care generează liste lungi în două moduri și vom analiza performanța folosind predicatul time/1.

Recursie

- Multe implementări ale limbajului Prolog aplică "last call optimization" atunci când un apel recursiv este ultimul predicat din corpul unei clauze (tail recursion).
- Atunci când este posibil, se recomandă utilizare recursiei la coadă (tail recursion).
- Vom defini un predicat care generează liste lungi în două moduri și vom analiza performanța folosind predicatul time/1.

```
biglist(0,[]).
biglist(N,[N|T]) :- N >= 1, M is N-1,biglist(M,T),M=M.

biglist_tr(0,[]).
biglist_tr(N,[N|T]) :- N >= 1, M is N-1,biglist_tr(M,T).
```

Recursie la coadă

- Predicat fără recursie la coadă:

```
biglist(0,[]).
biglist(N,[N|T]) :- N >= 1, M is N-1,biglist(M,T),M=M.
```

Apelul recursiv întoarce valoarea găsită în predicatul apelant, acestă valoare urmând a fi prelucrată.

Recursie la coadă

- Predicat fără recursie la coadă:

```
biglist(0,[]).
biglist(N,[N|T]) :- N >= 1, M is N-1,biglist(M,T),M=M.
```

Apelul recursiv întoarce valoarea găsită în predicatul apelant, acestă valoare urmând a fi prelucrată.

```
?- time(biglist(50000,X)).
100,000 inferences, 0.016 CPU in 0.038 seconds
(41% CPU, 6400000 Lips)
X = [50000, 49999, 49998|...] .
```

Recursie la coadă

- Predicat fără recursie la coadă:

```
biglist(0,[]).
biglist(N,[N|T]) :- N >= 1, M is N-1,biglist(M,T),M=M.
```

  Apelul recursiv întoarce valoarea găsită în predicatul apelant, acestă valoare urmând a fi prelucrată.

```
?- time(biglist(50000,X)).
100,000 inferences, 0.016 CPU in 0.038 seconds
(41% CPU, 6400000 Lips)
X = [50000, 49999, 49998|...] .
```

- Predicatul cu recursie la coadă:

```
biglist_tr(0,[]).
biglist_tr(N,[N|T]) :- N >= 1, M is N-1,biglist_tr(M,T).
```

Recursie la coadă

- Predicat fără recursie la coadă:

```
biglist(0,[]).
biglist(N,[N|T]) :- N >= 1, M is N-1,biglist(M,T),M=M.
```

  Apelul recursiv întoarce valoarea găsită în predicatul apelant, acestă valoare urmând a fi prelucrată.

```
?- time(biglist(50000,X)).
100,000 inferences, 0.016 CPU in 0.038 seconds
(41% CPU, 6400000 Lips)
X = [50000, 49999, 49998|...] .
```

- Predicatul cu recursie la coadă:

```
biglist_tr(0,[]).
biglist_tr(N,[N|T]) :- N >= 1, M is N-1,biglist_tr(M,T).

?- time(biglist_tr(50000,X)).
100,000 inferences, 0.000 CPU in 0.007 seconds
(0% CPU, Infinite Lips)
X = [50000, 49999, 49998|...]
```

Exemplu: reprezentarea unei GIC

Structura frazelor

- Aristotel, On Interpretation, `http://classics.mit.edu/Aristotle/interpretation.1.1.html`:

  "Every affirmation, then, and every denial, will consist of a noun and a verb, either definite or indefinite."

Structura frazelor

- Aristotel, On Interpretation, `http://classics.mit.edu/Aristotle/interpretation.1.1.html`:

  "Every affirmation, then, and every denial, will consist of a noun and a verb, either definite or indefinite."

- N. Chomsy, Syntactic structure, Mouton Publishers, First printing 1957 - Fourteenth printing 1985 [Chapter 4 (Phrase Structure)]

  (i) *Sentence* $\rightarrow$ *NP* $+$ *VP*
  (ii) *NP* $\rightarrow$ *T* $+$ *N*
  (iii) *VP* $\rightarrow$ *Verb* $+$ *NP*
  (iv) *T* $\rightarrow$ *the*
  (q) *N* $\rightarrow$ *fman*, *ball*, etc.
  (vi) *V* $\rightarrow$ *hit*, *took*, etc.

Gramatică independentă de context

- Definim structura propozițiilor folosind o gramatică independentă de context:

S	$\rightarrow$	NP VP
NP	$\rightarrow$	Det N
VP	$\rightarrow$	V
VP	$\rightarrow$	V NP
Det	$\rightarrow$	*the*
Det	$\rightarrow$	*a*
N	$\rightarrow$	*boy*
N	$\rightarrow$	*girl*
V	$\rightarrow$	*loves*
V	$\rightarrow$	*hates*

- Neterminalele definesc categorii gramaticale:

  S (propozițiile),
  NP (expresiile substantivale),
  VP (expresiile verbale),
  V (verbele),
  N (substantivele),
  Det (articolele).

- Terminalele definesc cuvintele.

Gramatică independentă de context

GIC

			Det	$\rightarrow$	*the*
S	$\rightarrow$	NP VP	Det	$\rightarrow$	*a*
NP	$\rightarrow$	Det N	N	$\rightarrow$	*boy*
VP	$\rightarrow$	V	N	$\rightarrow$	*girl*
VP	$\rightarrow$	V NP	V	$\rightarrow$	*loves*
			V	$\rightarrow$	*hates*

Ce vrem să facem?

- Vrem să scriem un program în Prolog care să recunoască propozițiile generate de această gramatică.
- Reprezentăm propozițiile prin liste.

```
?- atomic_list_concat(SL,' ' , 'a boy loves a girl').
SL = [a, boy, loves, a, girl]
```

Definirea unei gramatici în Prolog

- Reprezentăm propozițiile prin liste.

  ```
  SL = [a, boy, loves, a, girl]
  ```

Definirea unei gramatici în Prolog

- Reprezentăm propozițiile prin liste.

  `SL = [a, boy, loves, a, girl]`

- Fiecărui neterminal îi asociem un predicat care definește listele corespunzătoare categoriei gramaticale respective. `n([boy]).`

  `n([girl]).  det([the]).  v([loves]).`

Definirea unei gramatici în Prolog

- Reprezentăm propozițiile prin liste.

  `SL = [a, boy, loves, a, girl]`

- Fiecărui neterminal îi asociem un predicat care definește listele corespunzătoare categoriei gramaticale respective. `n([boy]).`

  `n([girl]).  det([the]).  v([loves]).`

- Lista asociată unei propoziții se obține prin concatenarea listelor asociate elementelor componente.

Definirea unei gramatici în Prolog

- Reprezentăm propozițiile prin liste.

  `SL = [a, boy, loves, a, girl]`

- Fiecărui neterminal îi asociem un predicat care definește listele corespunzătoare categoriei gramaticale respective. `n([boy]).`

  `n([girl]).  det([the]).  v([loves]).`

- Lista asociată unei propoziții se obține prin concatenarea listelor asociate elementelor componente.

  De exemplu, interpretăm regula S $\rightarrow$ NP VP astfel:

  o propozi tie este o listă `L` care se obține prin concatenarea a două liste, `X` și `Y`, unde `X` reprezintă o expresie substantivală și `Y` reprezintă o expresie verbală.

```
s(L) :- np(X), vp(Y), append(X,Y,L).
```

Definirea unei gramatici în Prolog

Gramatică independentă de context

S	$\rightarrow$	NP VP
NP	$\rightarrow$	Det N
VP	$\rightarrow$	V
VP	$\rightarrow$	V NP

Det	$\rightarrow$	*the*
Det	$\rightarrow$	*a*
N	$\rightarrow$	*boy*
N	$\rightarrow$	*girl*
V	$\rightarrow$	*loves*
V	$\rightarrow$	*hates*

Prolog

```
s(L) :- np(X), vp(Y),
        append(X,Y,L).
np(L) :- det(X), n(Y),
         append(X,Y,L) .
vp(L) :- v(L).
vp(L):- v(X), np(Y),
        append(X,Y,L) .
```

```
det([the]).
det([a]).
n([boy]).
n([girl]).
v([loves]).
v([hates]).
```

Definirea unei gramatici în Prolog

```
s(L) :- np(X), vp(Y),
        append(X,Y,L).

np(L) :- det(X), n(Y),
         append(X,Y,L) .

vp(L) :- v(L).
vp(L):- v(X), np(Y),
        append(X,Y,L) .

det([the]).
det([a]).
n([boy]).
n([girl]).
v([loves]).
v([hates]).
```

```
?- s([a,boy,loves, a,
girl]).
true .

?- s[a, girl|T].
T = [loves] ;
T = [hates] ;
T = [loves, the, boy] ;
⋮

?- s(S).
S = [the, boy, loves] ;
S = [the, boy, hates] ;
⋮
```

Definirea unei gramatici în Prolog

- Reprezentăm propozițiile prin liste.

  ```
  SL = [a, boy, loves, a, girl]
  ```

Definirea unei gramatici în Prolog

- Reprezentăm propozițiile prin liste.

  `SL = [a, boy, loves, a, girl]`

- Fiecărui neterminal îi asociem un predicat care definește listele corespunzătoare categoriei gramaticale respective. `n([boy]).`

  `n([girl]).  det([the]).  v([loves]).`

Definirea unei gramatici în Prolog

- Reprezentăm propozițiile prin liste.

  `SL = [a, boy, loves, a, girl]`

- Fiecărui neterminal îi asociem un predicat care definește listele corespunzătoare categoriei gramaticale respective. `n([boy]).`

  `n([girl]).  det([the]).  v([loves]).`

- Lista asociată unei propoziții se obține prin concatenarea listelor asociate elementelor componente.

Definirea unei gramatici în Prolog

- Reprezentăm propozițiile prin liste.

  `SL = [a, boy, loves, a, girl]`

- Fiecărui neterminal îi asociem un predicat care definește listele corespunzătoare categoriei gramaticale respective. `n([boy]).`

  `n([girl]).  det([the]).  v([loves]).`

- Lista asociată unei propoziții se obține prin concatenarea listelor asociate elementelor componente.

- Deși corectă, reprezentarea anterioară este ineficientă, arborele de căutare este foarte mare. Pentru a optimiza, folosim <u>reprezentarea listelor ca diferențe</u>.

Bibliografie

- M. Ben-Ari, **Mathematical Logic for Computer Science**, Springer, 2012.
- P. Blackburn, J. Bos, K. Striegnitz, **Learn Prolog now**, College Publications, 2006.
- J.W. Lloyd, Foundations of Logic Programming, Springer, 1987.
- L.S. Sterling and E.Y. Shapiro, The Art of Prolog `https://mitpress.mit.edu/books/art-prolog-second-edition`
- Logic Programming, The University of Edinburgh, https://www.inf.ed.ac.uk/teaching/courses/lp/

Pe săptămâna viitoare!

Curs 9

2020-2021

Fundamentele limbajelor de programare

Cuprins

Limbajul IMP

Limbajul IMP

Vom implementa un limbaj 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 instruțiunilor `x=7;while(x>=0,x=x-1)`
- Blocuri de instrucțiuni `{x=7;while(x>=0,x=x-1)}`

Limbajul IMP

Exemplu

Un program în limbajul IMP

```
{x = 10 ; sum = 0;
while(0 =< x,
     {sum = sum + x; x = x-1}
     )},sum
```

- Semantica
  după execuția programului, se evaluează `sum`

Sintaxa BNF a limbajului IMP

$$
\begin{aligned}
E ::= &\ n \mid x \\
&\mid E \texttt{ + } E \mid E \texttt{ - } E \mid E \texttt{ * } E
\end{aligned}
$$

$$
\begin{aligned}
B ::= &\ \texttt{true} \mid \texttt{false} \\
&\mid E \texttt{ =< } E \mid E \texttt{ >= } E \mid E \texttt{ == } E \\
&\mid \texttt{not}(B) \mid \texttt{and}(B \texttt{ , } B) \mid \texttt{or}(B \texttt{ , } B)
\end{aligned}
$$

$$
\begin{aligned}
C ::= &\ \texttt{skip} \\
&\mid x \texttt{ = } E \\
&\mid \texttt{if(} B \texttt{ , } C \texttt{ , } C \texttt{)} \\
&\mid \texttt{while(} B \texttt{ , } C \texttt{)} \\
&\mid \texttt{\{ } C \texttt{ \}} \mid C \texttt{ ; } C
\end{aligned}
$$

$$
P ::= \texttt{\{ } C \texttt{ \}, } E
$$

O implementare a limbajului IMP în Prolog

Decizii de implementare

- {} și ; sunt operatori

```
:- op(100, xf, {}).
:- op(1100, yf, ;).
```

- definim un predicat pentru fiecare categorie sintactică

```
stmt(while(BE,St)) :- bexp(BE),  stmt(St).
```

- `while`, `if`, `and`, etc sunt functori în Prolog
  `while(true,skip)` este un termen compus

- , are semnificația obișnuită

- pentru valori numerice folosim întregii din Prolog

```
aexp(I) :- integer(I).
```

- pentru identificatori folosim atomii din Prolog

```
aexp(X) :- atom(X).
```

Expresiile aritmetice

$$
\begin{aligned}
E ::= {} & n \mid x \\
& \mid E + E \mid E - E \mid E * E
\end{aligned}
$$

Prolog

```
aexp(I) :- integer(I).
aexp(X) :- atom(X).
aexp(A1 + A2) :- aexp(A1), aexp(A2).
aexp(A1 - A2) :- aexp(A1), aexp(A2).
aexp(A1 * A2) :- aexp(A1), aexp(A2).
```

Expresiile aritmetice

Exemplu

```
?- aexp(1000).
true.
?- aexp(id).
true.
?- aexp(id + 1000).
true.
?- aexp(2 + 1000).
true.
?- aexp(x * y).
true.
?- aexp(- x).
false.
```

Expresiile booleene

$B ::=$ **true** | **false**
 | E =< E | E >= E | E == E
 | not(B) | and(B , B) | or(B , B)

Prolog

```
bexp(true). bexp(false).
bexp(and(BE1,BE2)) :- bexp(BE1), bexp(BE2).
bexp(or(BE1,BE2)) :-  bexp(BE1), bexp(BE2).
bexp(not(BE)) :-  bexp(BE).

bexp(A1 =< A2) :- aexp(A1), aexp(A2).
bexp(A1 >= A2) :- aexp(A1), aexp(A2).
bexp(A1 == A2) :- aexp(A1), aexp(A2).
```

Expresiile booleene

Exemplu

```
?- bexp(true).
true.
?- bexp(id).
false.
?- bexp(not(1 =< 2)).
true.
?- bexp(or(1 =< 2,true)).
true.
?- bexp(or(a =< b,true)).
true.
?- bexp(not(a)).
false.
?- bexp(!(a)).
false.
```

Instrucțiunile

$$
\begin{aligned}
C ::= &\ \mathbf{skip} \\
\mid &\ x \,\mathbf{=}\, E \,\mathbf{;} \\
\mid &\ \mathbf{if(}\ B\ \mathbf{)}\ C\ \mathbf{else}\ C \\
\mid &\ \mathbf{while(}\ B\ \mathbf{)}\ C \\
\mid &\ \mathbf{\{}\ C\ \mathbf{\}} \mid C\ \mathbf{;}\ C
\end{aligned}
$$

Prolog

```
stmt(skip).
stmt(X = AE) :- atom(X), aexp(AE).
stmt(St1;St2) :- stmt(St1), stmt(St2).
stmt((St1;St2)) :- stmt(St1), stmt(St2).
stmt({St}) :- stmt(St).
stmt(if(BE,St1,St2)) :-  bexp(BE), stmt(St1), stmt(St2).
stmt(while(BE,St)) :- bexp(BE), stmt(St).
```

Instrucțiunile

Exemplu

```
?- stmt(id = 5).
true.
?- stmt(id = a).
true.
?- stmt(3 = 6).
false.
?- stmt(if(true, x=2;y=3, x=1;y=0)).
true.
?- stmt(while(x =< 0,skip)).
true.
?- stmt(while(x =< 0,)).
false.
?- stmt(while(x =< 0,skip)).
true .
```

Programele

$P ::= \{\ C\ \}, E$

Prolog

```
program(St,AE) :- stmt(St), aexp(AE).
```

Exemplu

```
test0 :- program( {x = 10 ; sum = 0;
                   while(0 =< x,
                         {sum = sum + x; x = x-1}
                        )}
                   , sum).
```

?- test0.
true.

O implementare a semanticii small-step

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:

$$\langle \mathit{cod} \;,\; \sigma \rangle \rightarrow \langle \mathit{cod} \;,\; \sigma' \rangle$$

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:

$$\langle cod\ ,\ \sigma\rangle \to \langle cod\ ,\ \sigma'\rangle$$

- Execuția se obține ca o succesiune de astfel de tranziții:

$$\langle \texttt{int}\ x = 0\ \texttt{;}\ x = x + 1\ \texttt{;}\ ,\ \perp\rangle \quad \longrightarrow \quad \langle x = x + 1\ \texttt{;}\ ,\ x \mapsto 0\rangle$$

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:

$$\langle cod\ ,\ \sigma\rangle \to \langle cod\ ,\ \sigma'\rangle$$

- Execuția se obține ca o succesiune de astfel de tranziții:

$$\begin{array}{rcl} \langle \texttt{int}\ x = 0\ \texttt{;}\ x = x + 1\ \texttt{;}\ ,\ \bot\rangle & \longrightarrow & \langle x = x + 1\ \texttt{;}\ ,\ x \mapsto 0\rangle \\ & \longrightarrow & \langle x = 0 + 1\ \texttt{;}\ ,\ x \mapsto 0\rangle \end{array}$$

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:

$$\langle cod \ , \ \sigma \rangle \to \langle cod \ , \ \sigma' \rangle$$

- Execuția se obține ca o succesiune de astfel de tranziții:

$$\begin{array}{lll}
\langle \mathtt{int}\ x = 0\ \mathtt{;}\ x = x + 1\ \mathtt{;}\ ,\ \perp \rangle & \longrightarrow & \langle x = x + 1\ \mathtt{;}\ ,\ x \mapsto 0 \rangle \\
 & \longrightarrow & \langle x = 0 + 1\ \mathtt{;}\ ,\ x \mapsto 0 \rangle \\
 & \longrightarrow & \langle x = 1\ \mathtt{;}\ ,\ x \mapsto 0 \rangle
\end{array}$$

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:

$$\langle cod\ ,\ \sigma \rangle \to \langle cod\ ,\ \sigma' \rangle$$

- Execuția se obține ca o succesiune de astfel de tranziții:

$$\begin{array}{lll}
\langle \mathbf{int}\ x = 0\ \mathbf{;}\ x = x + 1\ \mathbf{;}\ ,\ \bot \rangle & \longrightarrow & \langle x = x + 1\ \mathbf{;}\ ,\ x \mapsto 0 \rangle \\
& \longrightarrow & \langle x = 0 + 1\ \mathbf{;}\ ,\ x \mapsto 0 \rangle \\
& \longrightarrow & \langle x = 1\ \mathbf{;}\ ,\ x \mapsto 0 \rangle \\
& \longrightarrow & \langle \mathbf{\{\}}\ ,\ x \mapsto 1 \rangle
\end{array}$$

Semantica small-step

- Definește cel mai mic pas de execuție ca o relație de tranziție între configurații:

  $\langle \mathit{cod} , \sigma\rangle \rightarrow \langle \mathit{cod}' , \sigma'\rangle$ `step(Cod,S1,Cod',S2)`

- Execuția se obține ca o succesiune de astfel de tranziții.
- Starea executiei unui program IMP la un moment dat este o funcție parțială: $\sigma = n \mapsto 10, \mathit{sum} \mapsto 0$, etc.

Reprezentarea stărilor în Prolog

```
get(S,X,I) :- member(vi(X,I),S).
get(_,_,0).
set(S,X,I,[vi(X,I)|S1]) :- del(S,X,S1).

del(S,X,S1) :- select(vi(X,_), S, S1), !.
del(S, _, S).
```

Semantica expresiilor aritmetice

- Semantica unei variabile

  $\langle x\ ,\ \sigma\rangle \rightarrow \langle i\ ,\ \sigma\rangle \quad \textit{dacă } i = \sigma(x)$

Prolog

```
step(X,S,I,S) :-
  atom(X),
  get(S,X,I).
```

Semantica expresiilor aritmetice

- Semantica adunării a două expresii aritmetice

$\langle i_1 + i_2 \, , \, \sigma \rangle \to \langle i \, , \, \sigma \rangle$ *dacă* $i = i_1 + i_2$

$$\frac{\langle a_1 \, , \, \sigma \rangle \to \langle a_1' \, , \, \sigma' \rangle}{\langle a_1 + a_2 \, , \, \sigma \rangle \to \langle a_1' + a_2 \, , \, \sigma' \rangle} \qquad \frac{\langle a_2 \, , \, \sigma \rangle \to \langle a_2' \, , \, \sigma' \rangle}{\langle a_1 + a_2 \, , \, \sigma \rangle \to \langle a_1 + a_2' \, , \, \sigma' \rangle}$$

Prolog

```
step(I1 + I2,S,I,S):- integer(I1),integer(I2),
                            I is I1 + I2.

step(AE + AE1,S1,AE + AE2,S1):- step(AE1,S1,AE2,S2).

step(AE1 + AE,S1,AE2 + AE,S2):- step(AE1,S1,AE2,S2).
```

Semantica expresiilor aritmetice

Exemplu

```
?- step(a + b, [vi(a,1),vi(b,2)],AE, S).
AE = 1+b,
S = [vi(a, 1), vi(b, 2)] .

?- step(1 + b, [vi(a,1),vi(b,2)],AE, S).
AE = 1+2,
S = [vi(a, 1), vi(b, 2)] .

?- step(1 + 2, [vi(a,1),vi(b,2)],AE, S).
AE = 3,
S = [vi(a, 1), vi(b, 2)]
```

Semantica expresiilor aritmetice

Exemplu

```
?- step(a + b, [vi(a,1),vi(b,2)],AE, S).
AE = 1+b,
S = [vi(a, 1), vi(b, 2)] .

?- step(1 + b, [vi(a,1),vi(b,2)],AE, S).
AE = 1+2,
S = [vi(a, 1), vi(b, 2)] .

?- step(1 + 2, [vi(a,1),vi(b,2)],AE, S).
AE = 3,
S = [vi(a, 1), vi(b, 2)]
```

- Semantica * și – se definesc similar.

Semantica expresiilor booleene

- Semantica operatorului de comparație

$\langle i_1 \texttt{ =< } i_2 \,,\, \sigma\rangle \to \langle \mathbf{false} \,,\, \sigma\rangle \quad dacă\ i_1 > i_2$

$\langle i_1 \texttt{ =< } i_2 \,,\, \sigma\rangle \to \langle \mathbf{true} \,,\, \sigma\rangle \quad dacă\ i_1 \leq i_2$

$$\frac{\langle a_1 \,,\, \sigma\rangle \to \langle a_1' \,,\, \sigma'\rangle}{\langle a_1 \texttt{ =< } a_2 \,,\, \sigma\rangle \to \langle a_1' \texttt{ =< } a_2 \,,\, \sigma'\rangle} \qquad \frac{\langle a_2 \,,\, \sigma\rangle \to \langle a_2' \,,\, \sigma'\rangle}{\langle a_1 \texttt{ =< } a_2 \,,\, \sigma\rangle \to \langle a_1 \texttt{ =< } a_2' \,,\, \sigma'\rangle}$$

Prolog

```
step(I1 =< I2,S,true,S):- integer(I1),integer(I2),
                                 (I1 =< I2).
step(I1 =< I2,S,false,S):- integer(I1),integer(I2),
                                  (I1 > I2).
step(AE =< AE1,S1,AE =< AE2,S2):- step(AE1,S1,AE2,S2).
step(AE1 =< AE,S1,AE2 =< AE,S2):- step(AE1,S1,AE2,S2).
```

Semantica expresiilor Booleene

- Semantica negației

$\langle \mathbf{not(true)} , \sigma\rangle \to \langle \mathbf{false} , \sigma\rangle$

$\langle \mathbf{not(false)} , \sigma\rangle \to \langle \mathbf{true} , \sigma\rangle$

$$\frac{\langle a , \sigma\rangle \to \langle a' , \sigma'\rangle}{\langle \mathbf{not}\ (a) , \sigma\rangle \to \langle \mathbf{not}\ (a') , \sigma'\rangle}$$

Prolog

```
step(not(true),S,false,S) .
step(not(false),S,true,S) .

step(not(BE1),S1,not(BE2),S2) :- step(BE1,S1,BE2,S2).
```

Semantica compunerii și a blocurilor

- Semantica blocurilor
  $\langle \texttt{\{}\ s\ \texttt{\}}\ ,\ \sigma\rangle \rightarrow \langle s\ ,\ \sigma\rangle$
- Semantica compunerii secvențiale
  $\langle \texttt{\{\}};s_2\ ,\ \sigma\rangle \rightarrow \langle s_2\ ,\ \sigma\rangle \qquad \dfrac{\langle s_1\ ,\ \sigma\rangle \rightarrow \langle s_1'\ ,\ \sigma'\rangle}{\langle s_1;s_2\ ,\ \sigma\rangle \rightarrow \langle s_1';s_2\ ,\ \sigma'\rangle}$

Prolog

```
step({E},S,E,S).

step((skip;St2),S,St2,S).
step((St1;St),S1,(St2;St),S2) :-
                              step(St1,S1,St2,S2) .
```

Semantica atribuirii

- Semantica atribuirii

$\langle x = i \ , \ \sigma\rangle \rightarrow \langle \{\} \ , \ \sigma'\rangle \quad dacă\ \sigma' = \sigma[i/x]$

$$\frac{\langle a \ , \ \sigma\rangle \rightarrow \langle a' \ , \ \sigma'\rangle}{\langle x = a \ , \ \sigma\rangle \rightarrow \langle x = a' \ ; \ , \ \sigma'\rangle}$$

Prolog

```
step(X = I,S,skip,S1) :- integer(I),set(S,X,I,S1).

step(X = AE1,S1,X = AE2,S2) :-
                                        step(AE1,S1,AE2,S2).
```

Semantica lui `if`

- Semantica lui `if`

$\langle \mathtt{if}\ (\mathtt{true}, bl_1, bl_2)\ ,\ \sigma\rangle \to \langle bl_1\ ,\ \sigma\rangle$

$\langle \mathtt{if}\ (\mathtt{false}, bl_1, bl_2)\ ,\ \sigma\rangle \to \langle bl_2\ ,\ \sigma\rangle$

$$\frac{\langle b\ ,\ \sigma\rangle \to \langle b'\ ,\ \sigma'\rangle}{\langle \mathtt{if}\ (b, bl_1, bl_2)\ ,\ \sigma\rangle \to \langle \mathtt{if}\ (b', bl_1, bl_2)\ ,\ \sigma'\rangle}$$

Prolog

```
step(if(true,St1,_),S,St1,S).
step(if(false,_,St2),S,St2,S).

step(if(BE1,St1,St2),S1,if(BE2,St1,St2),S2) :-
                                  step(BE1,S1,BE2,S2) .
```

Semantica lui `while`

- Semantica lui `while`

  $\langle \mathtt{\textbf{while}}\ (b, bl)\ ,\ \sigma\rangle \to \langle \mathtt{\textbf{if}}\ (b, bl\ \mathtt{\textbf{;while}}\ (b, bl), \mathtt{\textbf{skip}})\ ,\ \sigma\rangle$

Prolog

```
step(while(BE,St),S,if(BE,(St;while(BE,St)),skip),S).
```

Semantica programelor

- Semantica programelor

$$\frac{\langle a_1\ ,\ \sigma_1\rangle \rightarrow \langle a_2\ ,\ \sigma_2\rangle}{\langle (\mathbf{skip}, a_1)\ ,\ \sigma_1\rangle \rightarrow \langle (\mathbf{skip}, a_2)\ ,\ \sigma_2\rangle}$$

$$\frac{\langle s_1\ ,\ \sigma_1\rangle \rightarrow \langle s_2\ ,\ \sigma_2\rangle}{\langle (s_1, a)\ ,\ \sigma_1\rangle \rightarrow \langle (s_2, a)\ ,\ \sigma_2\rangle}$$

Prolog

```
step((skip,AE1),S1,(skip,AE2),S2) :-
                                  step(AE1,S1,AE2,S2) .
step((St1,AE),S1,(St2,AE),S2) :-
                                  step(St1,S1,St2,S2).
```

Execuția programelor

Prolog

```
all_steps(P1, S1, PF, SF) :-
    step(P1, S1, P2, S2)
    -> all_steps(P2, S2, PF, SF)
    ;  PF = P1, SF = S1.

run_program(Name) :- defpg(Name,{P},E),
                     all_steps((P,E), [],(skip, I), _),
                     write(I).
```

Exemplu

```
defpg(pg2, {x = 10 ; sum = 0; while(0 =< x, {
                                    sum = sum + x;
                                    x = x - 1})},sum)
```

?- run_program(pg2).
55
true

Execuția programelor: `trace`

Putem defini o funcție care ne permite să urmărim execuția unui program în implementarea noastră?

Execuția programelor: `trace`

Putem defini o funcție care ne permite să urmărim execuția unui program în implementarea noastră?

Prolog

```
all_steps(P1, S1, PF, SF, [(P1, S1)| Trace]) :-
    step(P1, S1, P2, S2)
    -> all_steps(P2, S2, PF, SF, Trace)
    ;  PF = P1, SF = S1, Trace=[].

trace_program(Name) :- defpg(Name,{P},E),
                       all_steps((P,E), [],_, _, Trace),
                       write(Trace).
```

Execuția programelor: `trace_program`

Exemplu

```
?- trace_program(pg2).
. . .
((if(0=<x,(sum=sum+x;x=x-1;while(0=<x,sum=sum+x;x=x-1)),skip), sum),
[vi(x,-1),vi(sum,55)]),
((if(0=<-1,(sum=sum+x;x=x-1;while(0=<x,sum=sum+x;x=x-1)),skip), sum),
[vi(x,-1),vi(sum,55)]),
((if(false,(sum=sum+x;x=x-1;while(0=<x,sum=sum+x;x=x-1)),skip), sum),
[vi(x,-1),vi(sum,55)]),
((skip, sum), [vi(x,-1),vi(sum,55)]),
((skip, 55), [vi(x,-1),vi(sum,55)]),
```

Sintaxa limbajului LAMBDA

BNF

$$
\begin{array}{rcl}
e & ::= & x \\
& | & \lambda x.e \\
& | & e\ e \\
& | & \textbf{let}\ x = e\ \textbf{in}\ e
\end{array}
$$

Verificarea sintaxei în Prolog

```
exp(Id) :- atom(Id).                               % identifier
exp(Id -> Exp) :- atom(Id), exp(Exp).              % lambda
exp(Exp1 $ Exp2) :- exp(Exp1), exp(Exp2).          % application
exp(let(Id, Exp1, Exp2)) :- atom(Id), exp(Exp1), exp(Exp2).
```

Semantica small-step pentru Lambda

- Definește cel mai mic pas de execuție ca o relație de tranziție între expresii dată fiind o stare cu valori pentru variabilele libere

  $\rho \vdash cod \to cod'$ `step(Env, Cod1,Cod2)`
- Execuția se obține ca o succesiune de astfel de tranziții.

Semantica variabilelor

$$\rho \vdash x \to v \quad \textit{dacă}\ \rho(x) = v$$

Prolog

```
step(Env, X, V) :- atom(X), get(Env, X, V).
```

Semantica λ-abstracției

$$\rho \vdash \lambda x.e \rightarrow closure(x, e, \rho)$$

λ-abstracția se evaluează la o valoare specială numită closure care capturează valorile curente ale variabilelor pentru a se putea executa în acest mediu atunci când va fi aplicată.

Prolog

```
step(Env, X -> E, closure(X, E, Env)).
```

Semantica construcției `let`

$$\rho \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \rightarrow (\lambda x.e2)\ e_1$$

A îi da lui x valoarea lui e_1 în e_2 este același lucru cu a aplica funcția de x cu corpul e_2 expresiei e_1.

Prolog

```
step(_, let(X, E1, E2), (X -> E2) $ E1).
```

Semantica operatorului de aplicare

$$\frac{\rho_e[v/x] \vdash e \rightarrow e'}{\rho \vdash closure(x, e, \rho_e)\ v \rightarrow closure(x, e', \rho_e)\ v} \quad \textit{dacă } v \text{ valoare}$$

$$\rho \vdash closure(x, v, \rho_e)\ e \rightarrow v \quad \textit{dacă } v \text{ valoare}$$

$$\frac{\rho \vdash e_1 \rightarrow e_1'}{\rho \vdash e_1\ e_2 \rightarrow e_1'\ e_2} \qquad \frac{\rho \vdash e_2 \rightarrow e_2'}{\rho \vdash e_1\ e_2 \rightarrow e_1\ e_2'}$$

Prolog

```
step(Env, E $ E1, E $ E2) :- step(Env, E1, E2).
step(Env, E1 $ E, E2 $ E) :- step(Env, E1, E2).
step(Env, closure(X, E, EnvE) $ V, Result) :-
    \+ step(Env, V, _),
    set(EnvE, X, V, EnvEX),
    step(EnvEX, E, E1)
    ->  Result = closure(X, E1, EnvE) $ V
    ;   Result = E.
```

Pe săptămâna viitoare!

Curs 10

2020-2021

Fundamentele limbajelor de programare

Cuprins

Sintaxa limbajului LAMBDA

BNF

$$
\begin{array}{rcl}
e & ::= & x \mid n \mid \mathit{true} \mid \mathit{false} \\
 & \mid & e + e \mid e < e \mid \texttt{not}\,(e) \\
 & \mid & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
 & \mid & \lambda x.e \mid e\ e \\
 & \mid & \texttt{let}\ x = e\ \texttt{in}\ e
\end{array}
$$

Verificarea sintaxei în Prolog

```
exp(Id) :- atom(Id).                              % identifier
exp(Lit) :- Lit = true ; Lit = false ; integer(Lit).
exp(E1 + E2) :- exp(E1), exp(E2).                 % same for any op
exp(if(E1, E2, E3)) :- exp(E1), exp(E2), exp(E3).
exp(Id -> Exp) :- atom(Id), exp(Exp).             % lambda
exp(Exp1 $ Exp2) :- exp(Exp1), exp(Exp2).         % application
exp(let(Id, Exp1, Exp2)) :- atom(Id), exp(Exp1), exp(Exp2).
```

Semantica small-step pentru Lambda

- Definește cel mai mic pas de execuție ca o relație de tranziție între expresii dată fiind o stare cu valori pentru variabilele libere

  $\rho \vdash cod \to cod'$ `step(Env, Cod1,Cod2)`

- Execuția se obține ca o succesiune de astfel de tranziții.

Semantica variabilelor

$$\rho \vdash x \to v \quad \textit{dacă}\ \rho(x) = v$$

Prolog

```
step(Env, X, V) :- atom(X), get(Env, X, V).
```

Semantica expresiilor aritmetice

- Semantica adunării a două expresii aritmetice

$\langle i_1 + i_2 \, , \, \sigma \rangle \to \langle i \, , \, \sigma \rangle \quad \textit{dacă } i = i_1 + i_2$

$$\frac{\langle a_1 \, , \, \sigma \rangle \to \langle a_1' \, , \, \sigma' \rangle}{\langle a_1 + a_2 \, , \, \sigma \rangle \to \langle a_1' + a_2 \, , \, \sigma' \rangle} \qquad \frac{\langle a_2 \, , \, \sigma \rangle \to \langle a_2' \, , \, \sigma' \rangle}{\langle a_1 + a_2 \, , \, \sigma \rangle \to \langle a_1 + a_2' \, , \, \sigma' \rangle}$$

- Pentru alți operatori (artimetici, de comparație, booleeni, condițional)
  - Similar cu regulile din IMP

Prolog

```
step(_, I1 + I2,I):- integer(I1),integer(I2),
                      I is I1 + I2.

step(Env, AE + AE1,AE + AE2):- step(Env, AE1,AE2).

step(Env, AE1 + AE,AE2 + AE):- step(Env, AE1,AE2).
```

Semantica λ-abstracției

$$\rho \vdash \lambda x.e \rightarrow closure(x, e, \rho)$$

λ-abstracția se evaluează la o valoare specială numită closure care capturează valorile curente ale variabilelor pentru a se putea executa în acest mediu atunci când va fi aplicată.

Prolog

```
step(Env, X -> E, closure(X, E, Env)).
```

Semantica construcției `let`

$$\rho \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rightarrow (\lambda x.e2)\ e_1$$

A îi da lui x valoarea lui e_1 în e_2 este același lucru cu a aplica funcția de x cu corpul e_2 expresiei e_1.

Prolog

```
step(_, let(X, E1, E2), (X -> E2) $ E1).
```

Semantica operatorului de aplicare

$$\frac{\rho_e[v/x] \vdash e \rightarrow e'}{\rho \vdash closure(x, e, \rho_e)\ v \rightarrow closure(x, e', \rho_e)\ v} \quad \textit{dacă } v \text{ valoare}$$

$$\rho \vdash closure(x, v, \rho_e)\ e \rightarrow v \quad \textit{dacă } v \text{ valoare}$$

$$\frac{\rho \vdash e_1 \rightarrow e_1'}{\rho \vdash e_1\ e_2 \rightarrow e_1'\ e_2} \qquad \frac{\rho \vdash e_2 \rightarrow e_2'}{\rho \vdash e_1\ e_2 \rightarrow e_1\ e_2'}$$

Prolog

```
step(Env, E $ E1, E $ E2) :- step(Env, E1, E2).
step(Env, E1 $ E, E2 $ E) :- step(Env, E1, E2).
step(Env, closure(X, E, EnvE) $ V, Result) :-
    \+ step(Env, V, _),
    set(EnvE, X, V, EnvEX),
    step(EnvEX, E, E1)
    ->  Result = closure(X, E1, EnvE) $ V
    ;   Result = E.
```

Problemă: Sintaxa este prea permisivă

Problemă: Mulți termeni acceptați de sintaxă nu pot fi evaluați

- $2\ (\lambda x.x)$
- $(\lambda x.x) + 1$
- $(\lambda x.x + 1)\ (\lambda x.x)$

Problemă: Sintaxa este prea permisivă

Problemă: Mulți termeni acceptați de sintaxă nu pot fi evaluați

- $2\ (\lambda x.x)$— expresia din stânga aplicației trebuie să reprezinte o funcție
- $(\lambda x.x) + 1$ — adunăm funcții cu numere
- $(\lambda x.x + 1)\ (\lambda x.x)$ — pot face o reducție, dar tot nu pot evalua

Soluție: Identificarea (precisă) a programelor corecte

- Definim tipuri pentru fragmente de program corecte (e.g., int, bool)
- Definim (recursiv) o relație care să lege fragmente de program de tipurile asociate

$$((\lambda x.x + 1)\ ((\lambda x.x)\ 3)) : \text{int}$$

Relația de asociere de tipuri

Definim (recursiv) o relație de forma $\Gamma \vdash e : \tau$, unde

- τ este un tip

  $\tau ::= int$ [întregi]
  $\quad | \; bool$ [valori de adevăr]
  $\quad | \; \tau \to \tau$ [funcții]
  $\quad | \; a$ [variabile de tip]

- e este un termen (potențial cu variabile libere)
- Γ este mediul de tipuri, o funcție parțială finită care asociază tipuri variabilelor (libere ale lui e)
- Variabilele de tip sunt folosite pentru a indica polimorfismul

Cum citim $\Gamma \vdash e : \tau$?

Dacă variabila x are tipul $\Gamma(x)$ pentru orice $x \in dom(\Gamma)$,
atunci termenul e are tipul τ.

Axiome

$(:\text{VAR}) \quad \Gamma \vdash x : \tau \quad \textit{dacă}\ \Gamma(x) = \tau$

$(:\text{INT}) \quad \Gamma \vdash n : \textit{int} \quad \textit{dacă}\ n$ întreg

$(:\text{BOOL}) \quad \Gamma \vdash b : \textit{bool} \quad \textit{dacă}\ b = \textit{true or}\ b = \textit{false}$

Expresii

$$(:\text{IOP}) \quad \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \ o \ e_2 : int} \quad dacă\ o \in \{+, -, *, /\}$$

$$(:\text{COP}) \quad \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \ o \ e_2 : bool} \quad dacă\ o \in \{\leq, \geq, <, >, =\}$$

$$(:\text{BOP}) \quad \frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : bool}{\Gamma \vdash e_1 \ o \ e_2 : bool} \quad dacă\ o \in \{\mathbf{and}, \mathbf{or}\}$$

$$(:\text{IF}) \quad \frac{\Gamma \vdash e_b : bool \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathtt{\mathbf{if}}\ e_b\ \mathtt{\mathbf{then}}\ e_1\ \mathtt{\mathbf{else}}\ e_2 : \tau}$$

Fragmentul funcțional

$$(\textsc{:fn}) \quad \frac{\Gamma' \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \quad \textit{dacă } \Gamma' = \Gamma[x \mapsto \tau]$$

$$(\textsc{:app}) \quad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

Programe în execuție

Problemă:

- În timpul execuției programul conține valori de tip closure
- Care este tipul lor?

Soluție

- Adăugam regula (:CL) $\dfrac{\Gamma_\rho \vdash \lambda x.e : \tau}{\Gamma \vdash closure(x, e, \rho) : \tau}$ *unde*
- Mediul de tipuri Γ_ρ asociat unui mediu de execuție ρ satisface:
  - $Dom(\Gamma_\rho) = Dom(\rho)$
  - Pentru orice variabilă $x \in Dom(\rho)$, există τ tip și v valoare astfel încât $\Gamma_\rho(x) = \tau$, $\rho(x) = v$ și $\vdash v : \tau$

Proprietăți

Theorem (Proprietatea de a progresa)

Dacă $\Gamma_\rho \vdash e : \tau$ atunci e este valoare sau e poate progresa în ρ: există e' astfel încât $\rho \vdash e \to e'$.

Theorem (Proprietatea de conservare a tipului)

Dacă $\Gamma_\rho \vdash e : \tau$ și $\rho \vdash e \to e'$, atunci $\Gamma'_\rho \vdash e' : \tau$.

Theorem (Siguranță—programele bine formate nu se împotmolesc)

Dacă $\Gamma_\rho \vdash e : \tau$ și $\rho \vdash e \longrightarrow^* e'$, atunci e' este valoare sau există e'', astfel încât $\rho \vdash e' \to e''$.

Probleme computaționale

Verificarea tipului

Date fiind Γ, e și τ, verificați dacă $\Gamma \vdash e : \tau$.

Determinarea (inferarea) tipului

Date fiind Γ și e, găsiți (sau arătați ce nu există) un τ astfel încât $\Gamma \vdash e : \tau$.

- A doua problemă e mai grea în general decât prima
- Algoritmi de inferare a tipurilor
  - Colectează constrângeri asupra tipului
  - Folosesc metode de rezolvare a constrângerilor (programare logică)

Probleme computaționale

Theorem (Determinarea tipului este decidabilă)

Date fiind Γ și e, poate fi găsit (sau demonstrat că nu există) un τ astfel încât $\Gamma \vdash e : \tau$.

Theorem (Verificarea tipului este decidabilă)

Date fiind Γ, e și τ, problema $\Gamma \vdash e : \tau$ este decidabilă.

Theorem (Unicitatea tipului)

Dacă $\Gamma \vdash e : \tau$ și $\Gamma \vdash e : \tau'$, atunci $\tau = \tau'$.

Exemplu

Care este tipul expresiei următoare (dacă are)

$$\lambda x.\lambda y.\lambda z.\ \mathbf{if}\ y = 0\ \mathbf{then}\ z\ \mathbf{else}\ x/y$$

Aplicăm regula

(:FN) $\dfrac{\Gamma' \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}$ *dacă* $\Gamma' = \Gamma[x \mapsto \tau]$

$\vdash \lambda x.\lambda y.\lambda z.\ \mathbf{if}\ y = 0\ \mathbf{then}\ z\ \mathbf{else}\ x/y : t_x \to t$ dacă

$x \mapsto t_x \vdash \lambda y.\lambda z.\ \mathbf{if}\ y = 0\ \mathbf{then}\ z\ \mathbf{else}\ x/y : t$

Exemplu

Care este tipul expresiei următoare (dacă are)

$$\lambda x.\lambda y.\lambda z.\ \mathbf{if}\ y = 0\ \mathbf{then}\ z\ \mathbf{else}\ x/y$$

Aplicăm regula

$$(:\text{FN})\quad \frac{\Gamma' \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \quad \textit{dacă}\ \Gamma' = \Gamma[x \mapsto \tau]$$

$\vdash \lambda x.\lambda y.\lambda z.\ \mathbf{if}\ y = 0\ \mathbf{then}\ z\ \mathbf{else}\ x/y : t_x \to t$ dacă
$x \mapsto t_x \vdash \lambda y.\lambda z.\ \mathbf{if}\ y = 0\ \mathbf{then}\ z\ \mathbf{else}\ x/y : t$
Mai departe: $x \mapsto t_x \vdash \lambda y.\lambda z.\ \mathbf{if}\ y = 0\ \mathbf{then}\ z\ \mathbf{else}\ x/y : t_y \to t_0$ dacă
$x \mapsto t_x, y \mapsto t_y \vdash \lambda z.\ \mathbf{if}\ y = 0\ \mathbf{then}\ z\ \mathbf{else}\ x/y : t_0$ și, de mai sus,
$t = t_y \to t_0$

Exemplu

Care este tipul expresiei următoare (dacă are)

$$\lambda x.\lambda y.\lambda z.\ \mathbf{if}\ y = 0\ \mathbf{then}\ z\ \mathbf{else}\ x/y$$

Aplicăm regula

$$(:\text{FN})\quad \frac{\Gamma' \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \quad \textit{dacă}\ \Gamma' = \Gamma[x \mapsto \tau]$$

$\vdash \lambda x.\lambda y.\lambda z.\ \mathbf{if}\ y = 0\ \mathbf{then}\ z\ \mathbf{else}\ x/y : t_x \to t$ dacă
$x \mapsto t_x \vdash \lambda y.\lambda z.\ \mathbf{if}\ y = 0\ \mathbf{then}\ z\ \mathbf{else}\ x/y : t$
Mai departe: $x \mapsto t_x \vdash \lambda y.\lambda z.\ \mathbf{if}\ y = 0\ \mathbf{then}\ z\ \mathbf{else}\ x/y : t_y \to t_0$ dacă $x \mapsto t_x, y \mapsto t_y \vdash \lambda z.\ \mathbf{if}\ y = 0\ \mathbf{then}\ z\ \mathbf{else}\ x/y : t_0$ și, de mai sus, $t = t_y \to t_0$
Mai departe: $x \mapsto t_x, y \mapsto t_y \vdash \lambda z.\ \mathbf{if}\ y = 0\ \mathbf{then}\ z\ \mathbf{else}\ x/y : t_z \to t_1$ dacă $x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash \mathbf{if}\ y = 0\ \mathbf{then}\ z\ \mathbf{else}\ x/y : t_1$ și, de mai sus, $t_0 = t_z \to t_1$

Exemplu

Unde suntem

$\vdash \lambda x.\lambda y.\lambda z.\ \mathbf{if}\ y = 0\ \mathbf{then}\ z\ \mathbf{else}\ x/y : t_x \to t$ dacă
$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash \mathbf{if}\ y = 0\ \mathbf{then}\ z\ \mathbf{else}\ x/y : t_1$ și $t_0 = t_z \to t_1$,
$t = t_y \to t_0$.

Aplicăm regula (:IF) $\dfrac{\Gamma \vdash e_b : bool \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{if}\ e_b\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau}$

$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash \mathbf{if}\ y = 0\ \mathbf{then}\ z\ \mathbf{else}\ x/y : t_1$ dacă
$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash y = 0 : \mathbf{bool}$ și $x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash z : t_1$ și
$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash x/y : t_1$

Exemplu

Aplicăm regula

$$(\text{:COP})\quad \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1\ o\ e_2 : bool} \quad \textit{dacă } o \in \{\leq, \geq, <, >, =\}$$

$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash y = 0$:**bool** dacă
$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash y$:**int** și $x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash 0$:**int**

Aplicăm regula (:INT) $\Gamma \vdash n : int$ *dacă n întreg*

$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash 0$:**int** este adevărat

Aplicăm regula

$$(\text{:IOP})\quad \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1\ o\ e_2 : int} \quad \textit{dacă } o \in \{+, -, *, /\}$$

$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash x/y$:**int** dacă
$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash x$:**int** și $x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash y$:**int**
și, de mai sus, $t_1 =$**int**

Exemplu

Recapitulăm

$\vdash \lambda x.\lambda y.\lambda z.\,\mathtt{if}\ y = 0\ \mathtt{then}\ z\ \mathtt{else}\ x/y : t_x \to t$ dacă
$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash y : \mathtt{int}$ și $x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash z : t_1$
$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash x : \mathtt{int}$ și $x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash y : \mathtt{int}$
și $t_0 = t_z \to t_1$, $t = t_y \to t_0$, $t_1 = \mathtt{int}$.

Aplicăm regula (:VAR) $\Gamma \vdash x : \tau$ *dacă* $\Gamma(x) = \tau$

$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash y : t_y$ adevărat și, de mai sus $t_y = \mathtt{int}$
$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash z : t_z$ adevărat și, de mai sus, $t_1 = t_z$
$x \mapsto t_x, y \mapsto t_y, z \mapsto t_z \vdash x : t_x$ adevărat și, de mai sus, $t_x = \mathtt{int}$

Exemplu

Finalizăm

$\vdash \lambda x.\lambda y.\lambda z.\ \mathbf{if}\ y = 0\ \mathbf{then}\ z\ \mathbf{else}\ x/y : t_x \to t$ dacă
$t_0 = t_z \to t_1$, $t = t_y \to t_0$, $t_1 = \mathbf{int}$, $t_y = \mathbf{int}$, $t_1 = t_z$ și $t_x = \mathbf{int}$.

Rezolvăm constrângerile și obținem

$\vdash \lambda x.\lambda y.\lambda z.\ \mathbf{if}\ y = 0\ \mathbf{then}\ z\ \mathbf{else}\ x/y : \mathbf{int} \to \mathbf{int} \to \mathbf{int} \to \mathbf{int}$

Relația de asociere de tipuri în Prolog

Definim (recursiv) o relație de forma `type(Gamma, E, T)`, unde

- `Gamma` este o listă de perechi de forma `(X, T)` unde `X` este un identificator și `T` este o expresie de tip cu variabile
- `E` este o λ-expresie scrisă cu sintaxa descrisă mai sus
- `T` este o expresie de tip cu variabile

Sintaxa limbajului LAMBDA

BNF

$$
\begin{array}{rcl}
e & ::= & x \mid n \mid \mathit{true} \mid \mathit{false} \\
 & \mid & e + e \mid e < e \mid \mathbf{not}\,(e) \\
 & \mid & \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \\
 & \mid & \lambda x.e \mid e\ e \\
 & \mid & \mathbf{let}\ x = e\ \mathbf{in}\ e
\end{array}
$$

Verificarea sintaxei în Prolog

```
exp(Id) :- atom(Id).                                   % identifier
exp(Lit) :- Lit = true ; Lit = false ; integer(Lit).
exp(E1 + E2) :- exp(E1), exp(E2).                      % same for any op
exp(if(E1, E2, E3)) :- exp(E1), exp(E2), exp(E3).
exp(Id -> Exp) :- atom(Id), exp(Exp).                  % lambda
exp(Exp1 $ Exp2) :- exp(Exp1), exp(Exp2).              % application
exp(let(Id, Exp1, Exp2)) :- atom(Id), exp(Exp1), exp(Exp2).
```

Sintaxa tipurilor

BNF

τ ::= *int* [întregi]
  | *bool* [valori de adevăr]
  | $\tau \to \tau$ [funcții]
  | a [variabile de tip]

Verificarea sintaxei tipurilor în Prolog

```
is_type(X) :- variable(X).       % variabile
is_type(int).                    % intregi
is_type(bool).                   % valori de adevar
is_type(T1 -> T2) :-             % functii
    is_type(T1), is_type(T2).
```

Axiome

(:VAR) $\Gamma \vdash x : \tau$ *dacă* $\Gamma(x) = \tau$

```
type(Gamma, X, T) :- atom(X), get(Gamma, X, T).
```

(:INT) $\Gamma \vdash n : int$ *dacă* n întreg

```
type(_, I, int) :- integer(I).
```

(:BOOL) $\Gamma \vdash b : bool$ *dacă* $b = true$ *or* $b = false$

```
type(_, true, bool).
type(_, false, bool).
```

Expresii

$$(:\text{IOP})\quad \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1\ o\ e_2 : int} \quad dac\breve{a}\ o \in \{+, -, *, /\}$$

```
type(Gamma, E1 + E2, int) :-
    type(Gamma, E1, int), type(Gamma, E2, int).
```

$$(:\text{COP})\quad \frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1\ o\ e_2 : bool} \quad dac\breve{a}\ o \in \{\leq, \geq, <, >, =\}$$

```
type(Gamma, E1 < E2, bool) :-
    type(Gamma, E1, int), type(Gamma, E2, int).
```

$$(:\text{BOP})\quad \frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : bool}{\Gamma \vdash e_1\ o\ e_2 : bool} \quad dac\breve{a}\ o \in \{\mathbf{and}, \mathbf{or}\}$$

```
type(Gamma, and(E1, E2), bool) :-
    type(Gamma, E1, bool), type(Gamma, E2, bool).
```

Expresia condițională

$$(\textsc{:if}) \quad \frac{\Gamma \vdash e_b : bool \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{if}\ e_b\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau}$$

```
type(Gamma, if(E, E1, E2), T) :-
    type(Gamma, E, bool),
    type(Gamma, E1, T),
    type(Gamma, E2, T).
```

Fragmentul funcțional

$$(:\text{FN})\quad \frac{\Gamma' \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'} \quad \textit{dacă } \Gamma' = \Gamma[x \mapsto \tau]$$

```
type(Gamma, X -> E, TX -> TE) :-
    atom(X), set(Gamma, X, TX, GammaX), type(GammaX, E, TE).
```

$$(:\text{APP})\quad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

```
type(Gamma, E1 $ E2, T) :-
    type(Gamma, E, TE2 -> T), type(Gamma, E2, TE2).
```

Tipurile variabile nu sunt suficiente

Tipurile variabile sunt destul de flexibile

- $\vdash \lambda x.x : t \to t$ pentru orice t
- $\vdash$ **if** $(\lambda x.x)$ *true* **then** $(\lambda x.x)$ 3 **else** 4 :**int**

Tipurile variabile nu sunt suficiente

Tipurile variabile sunt destul de flexibile

- $\vdash \lambda x.x : t \to t$ pentru orice t
- $\vdash \mathtt{\textbf{if}}\ (\lambda x.x)\ \mathit{true}\ \mathtt{\textbf{then}}\ (\lambda x.x)\ 3\ \mathtt{\textbf{else}}\ 4 : \mathtt{\textbf{int}}$

Dar tipul unei expresii este fixat:

$$\nvdash (\lambda id.\mathtt{\textbf{if}}\ id\ \mathit{true}\ \mathtt{\textbf{then}}\ id\ 3\ \mathtt{\textbf{else}}\ 4)(\lambda x.x) : \mathtt{\textbf{int}}$$

Tipurile variabile nu sunt suficiente

Tipurile variabile sunt destul de flexibile

- $\vdash \lambda x.x : t \rightarrow t$ pentru orice t
- $\vdash \mathtt{if}\ (\lambda x.x)\ \mathit{true}\ \mathtt{then}\ (\lambda x.x)\ 3\ \mathtt{else}\ 4 : \mathtt{int}$

Dar tipul unei expresii este fixat:

$$\nvdash (\lambda \mathit{id}.\mathtt{if}\ \mathit{id}\ \mathit{true}\ \mathtt{then}\ \mathit{id}\ 3\ \mathtt{else}\ 4)(\lambda x.x) : \mathtt{int}$$

Soluție

Pentru funcțiile cu nume, am vrea să fie ca și cum am calcula mereu tipul

$$\vdash \mathtt{let}\ \mathit{id} = (\lambda x.x)\ \mathtt{in}\ \mathtt{if}\ \mathit{id}\ \mathit{true}\ \mathtt{then}\ \mathit{id}\ 3\ \mathtt{else}\ 4) : \mathtt{int}$$

Operațional: redenumim variabilele de tip când instanțiem numele funcției

Scheme de tipuri

- Numim schemă de tipuri o expresie de forma $\langle\tau\rangle$, unde τ este e un expresie tip cu variabile
- variabilele dintr-o schemă nu pot fi constrânse e ca și cum ar fi cuantificate universal
- O schemă poate fi concretizată la un tip obișnuit substituindu-i fiecare variabilă cu orice tip (poate fi și variabilă)
  - Pentru orice substituție θ de la variabile de tip la tipuri cu variabile

Reguli pentru scheme

$$(\textsc{:let})\quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma_1 \vdash e_2 : \tau}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e2 : \tau} \quad \textit{dacă}\ \Gamma_1 = \Gamma[\langle\tau_1\rangle/x]$$

Reguli pentru scheme

(:LET) $\dfrac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma_1 \vdash e_2 : \tau}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e2 : \tau}$ *dacă* $\Gamma_1 = \Gamma[\langle\tau_1\rangle/x]$

```
type(Gamma, let(X, E1, E2), T) :-
    type(Gamma, E1, T1),
    copy_term(T1, FreshT1),    % redenumeste variabilele
                               % ca sa nu poata fi constranse
    set(Gamma, X, scheme(FreshT1), GammaX),
    type(GammaX, E2, T).
```

Reguli pentru scheme

(:LET) $\dfrac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma_1 \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e2 : \tau}$ *dacă* $\Gamma_1 = \Gamma[\langle \tau_1 \rangle / x]$

```
type(Gamma, let(X, E1, E2), T) :-
    type(Gamma, E1, T1),
    copy_term(T1, FreshT1),   % redenumeste variabilele
                              % ca sa nu poata fi constranse
    set(Gamma, X, scheme(FreshT1), GammaX),
    type(GammaX, E2, T).
```

(:SCH) $\Gamma \vdash x : \tau'$ *dacă* $\Gamma(x) = \langle \tau \rangle$ și $\tau' = \theta(\tau)$

```
type(Gamma, X, T) :-
    atom(X), get(Gamma, X, T), is_type(T), !.
type(Gamma, X, T) :-
    atom(X), get(Gamma, X, scheme(TX)),
    copy_term(T1, T).         % redenumeste variabilele
                              % ca sa poata fi constranse
```

Pe săptămâna viitoare!

Curs 11

Cuprins

Logica propozițională (recap.)

Limbajul și formulele PL

- Limbajul PL
  - variabile propoziționale: $Var = \{p, q, v, \ldots\}$
  - conectori logici: $\neg$ (unar), $\rightarrow$, $\wedge$, $\vee$, $\leftrightarrow$ (binari)

- Formulele PL

$$
\begin{array}{lcl}
\mathit{var} & ::= & p \mid q \mid v \mid \ldots \\
\mathit{form} & ::= & \mathit{var} \mid (\neg \mathit{form}) \mid \mathit{form} \wedge \mathit{form} \mid \mathit{form} \vee \mathit{form} \\
& & \mid \mathit{form} \rightarrow \mathit{form} \mid \mathit{form} \leftrightarrow \mathit{form}
\end{array}
$$

- Conectorii sunt împărțiți în conectori de bază și conectori derivați (în funcție de formalism).

- Legături între conectori:

$$
\begin{array}{rcl}
\varphi \vee \psi & := & \neg \varphi \rightarrow \psi \\
\varphi \wedge \psi & := & \neg(\varphi \rightarrow \neg \psi) \\
\varphi \leftrightarrow \psi & := & (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)
\end{array}
$$

Sintaxa și semantica

Un sistem logic are două componente:

- Sintaxa
  - noțiuni sintactice: demonstrație, teoremă
  - notăm prin $\vdash \varphi$ faptul că φ este teoremă
  - notăm prin $\Gamma \vdash \varphi$ faptul că formula φ este demonstrabilă din mulțimea de formule Γ
- Semantica
  - noțiuni semantice: adevăr, model, tautologie (formulă universal adevărată)
  - notăm prin $\models \varphi$ faptul că φ este tautologie
  - notăm prin $\Gamma \models \varphi$ faptul că formula φ este adevărată atunci când toate formulele din mulțimea Γ sunt adevărate

Completitudine: Γ-teoremele și Γ-tautologiile coincid

$\Gamma \vdash \varphi$ dacă și numai dacă $\Gamma \models \varphi$

Logica propozițională

Exemplu

Formalizați următorul raționament:

`If winter is coming and Ned is not alive then Robb is lord of Winterfell. Winter is coming. Rob is not lord of Winterfell. Then Ned is alive.`

O posibilă formalizare este următoarea:

p = `winter is coming`
q = `Ned is alive`
r = `Robb is lord of Winterfel`

$\{(p \wedge \neg q) \rightarrow r, p, \neg r\} \vDash q$

Logica de ordinul I (recap.)

Logica de ordinul I - sintaxa

Limbaj de ordinul I $\mathcal{L}$

- unic determinat de $\tau = (\mathbf{R}, \mathbf{F}, \mathbf{C}, ari)$

Termenii lui $\mathcal{L}$, notați $Trm_{\mathcal{L}}$, sunt definiți inductiv astfel:

- orice variabilă este un termen;
- orice simbol de constantă este un termen;
- dacă $f \in \mathbf{F}$, $ar(f) = n$ și $t_1, \ldots, t_n$ sunt termeni, atunci $f(t_1, \ldots, t_n)$ este termen.

Formulele atomice ale lui $\mathcal{L}$ sunt definite astfel:

- dacă $R \in \mathbf{R}$, $ar(R) = n$ și $t_1, \ldots, t_n$ sunt termeni, atunci $R(t_1, \ldots, t_n)$ este formulă atomică.

Formulele lui $\mathcal{L}$ sunt definite 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 $\varphi \vee \psi$, $\varphi \wedge \psi$, $\varphi \rightarrow \psi$ sunt formule
- dacă φ este o formulă și x este o variabilă, atunci $\forall x\, \varphi$, $\exists x\, \varphi$ sunt formule

Logica de ordinul I - semantică

O structură este de forma $\mathcal{A} = (A, \mathbf{F}^{\mathcal{A}}, \mathbf{R}^{\mathcal{A}}, \mathbf{C}^{\mathcal{A}})$, unde

- A este o mulțime nevidă
- $\mathbf{F}^{\mathcal{A}} = \{f^{\mathcal{A}} \mid f \in \mathbf{F}\}$ este o mulțime de operații pe A; dacă f are aritatea n, atunci $f^{\mathcal{A}} : A^n \to A$.
- $\mathbf{R}^{\mathcal{A}} = \{R^{\mathcal{A}} \mid R \in \mathbf{R}\}$ este o mulțime de relații pe A; dacă R are aritatea n, atunci $R^{\mathcal{A}} \subseteq A^n$.
- $\mathbf{C}^{\mathcal{A}} = \{c^{\mathcal{A}} \in A \mid c \in \mathbf{C}\}$.

O interpretare a variabilelor lui $\mathcal{L}$ în $\mathcal{A}$ ($\mathcal{A}$-interpretare) este o funcție $I : V \to A$.

Inductiv, definim interpretarea termenului t în $\mathcal{A}$ sub I notat $t_I^{\mathcal{A}}$.

Inductiv, definim când o formulă este adevărată în $\mathcal{A}$ în interpretarea I notat $\mathcal{A}, I \vDash \varphi$. În acest caz spunem că $(\mathcal{A}, I)$ este model pentru φ.

O formulă φ este adevărată într-o structură $\mathcal{A}$, notat $\mathcal{A} \vDash \varphi$, dacă este adevărată în $\mathcal{A}$ sub orice interpretare. Spunem că $\mathcal{A}$ este model al lui φ.

O formulă φ este adevărată în logica de ordinul I, notat $\vDash \varphi$, dacă este adevărată în orice structură. O formulă φ este validă dacă $\vDash \varphi$.

O formulă φ este satisfiabilă dacă există o structură $\mathcal{A}$ și o $\mathcal{A}$-interpretare I astfel încât $\mathcal{A}, I \vDash \varphi$.

Algoritmul de unificare

Unificare

- O subtituție σ este o funcție (parțială) de la variabile la termeni,
$$\sigma : V \to Trm_{\mathcal{L}}$$
- 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 unificatorul termenilor t_1 și t_2.
- Un unificator ν pentru t_1 și t_2 este un cel mai general unificator (cgu,mgu) dacă pentru orice alt unificator ν' pentru t_1 și t_2, există o substituție μ astfel încât
$$\nu' = \nu; \mu.$$

Algoritmul de unificare

- Pentru o mulțime finită de termeni $\{t_1, \ldots, t_n\}$, $n \geq 2$, algoritmul de unificare stabilește dacă există un cgu.

- 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 sa 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 dă cgu.

Algoritmul este oprit cu concluzia inexistenței unui cgu 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
	$x \doteq t$, $S[x/t]$	$R'[x/t]$
Final	S	$\emptyset$

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

Exemplu

Exemplu

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

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
$y \doteq z, x \doteq g(z),$ $w \doteq h(g(z))$	$\emptyset$	

- $\nu = \{y/z,\ x/g(z),\ w/h(g(z))\}$ este cgu.

Exemplu

Exemplu

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

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 operații diferite!
- Nu există unificator pentru ecuațiile din U.

Exemplu

Exemplu

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

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 ecuațiile din U.

Validitate și satisfiabilitate

Propoziție

Dacă φ este o formulă atunci

φ este validă dacă și numai dacă $\neg\varphi$ nu este satisfiabilă.

Formă clauzală. Rezoluție

Literali. FNC

- În logica propozițională un literal este o variabilă sau negația unei variabile.

$$literal := \; p \mid \neg p \qquad \text{unde } p \text{ este variabilă propozițională}$$

- În logica de ordinul I un literal este o formulă atomică sau negația unei formule atomice.

$$literal := \; P(t_1, \ldots, t_n) \mid \neg P(t_1, \ldots, t_n)$$

unde $P \in \mathbf{R}, ari(P) = n,$ și $t_1, \ldots, t_n$ sunt termeni.

- Pentru un literal L vom nota cu L^c literalul complement.

O formulă este în formă normală conjunctivă (FNC) dacă este o conjuncție de disjuncții de literali.

Forma clauzală în logica propozițională

- Pentru orice formulă α există o FNC α^{fc} astfel încât $\alpha \mathrel{\text{⫤⊨}} \alpha^{fc}$.
- Pentru o formulă din logica propozițională determinăm FNC corespunzătoare prin următoarele transformări:

1. înlocuirea implicațiilor și echivalențelor

$$
\begin{aligned}
\varphi \to \psi \quad & \mathrel{\text{⫤⊨}} \quad \neg\varphi \vee \psi \\
\varphi \leftrightarrow \psi \quad & \mathrel{\text{⫤⊨}} \quad (\neg\varphi \vee \psi) \wedge (\neg\psi \vee \varphi)
\end{aligned}
$$

2. regulile De Morgan

$$
\begin{aligned}
\neg(\varphi \vee \psi) \quad & \mathrel{\text{⫤⊨}} \quad \neg\varphi \wedge \neg\psi \\
\neg(\varphi \wedge \psi) \quad & \mathrel{\text{⫤⊨}} \quad \neg\varphi \vee \neg\psi
\end{aligned}
$$

3. principiului dublei negații

$$
\neg\neg\psi \quad \mathrel{\text{⫤⊨}} \quad \psi
$$

4. distributivitatea

$$
\begin{aligned}
\varphi \vee (\psi \wedge \chi) \quad & \mathrel{\text{⫤⊨}} \quad (\varphi \vee \psi) \wedge (\varphi \vee \chi) \\
(\psi \wedge \chi) \vee \varphi \quad & \mathrel{\text{⫤⊨}} \quad (\psi \vee \varphi) \wedge (\chi \vee \varphi)
\end{aligned}
$$

Forma clauzală în logica de ordinul I

- O formulă este formă normală conjunctivă prenex (FNCP) dacă
  - este în formă prenex $Q_1 x_1 \ldots Q_n x_n \psi$ ($Q_i \in \{\forall, \exists\}$ oricare i)
  - ψ este FNC
- O formulă este formă clauzală dacă este enunț universal și FNCP:

$$\forall x_1 \ldots \forall x_n \psi \text{ unde } \psi \text{ este FNC}$$

- Pentru orice formulă φ din logica de ordinul I există o formă clauzală φ^{fc} astfel încât

$$\varphi \text{ este satisfiabilă dacă și numai dacă } \varphi^{fc} \text{ este satisfiabilă}$$

- Pentru o formulă φ, forma clauzală φ^{fc} se poate calcula astfel:
  1. se determină forma rectificată
  2. se cuantifică universal variabilele libere
  3. se determină forma prenex
  4. se determină forma Skolem

     în acest moment am obținut o formă Skolem $\forall x_1 \ldots \forall x_n \psi$
  5. se determină o FNC ψ' astfel încât $\psi \mathrel{\text{⫤}\!\vDash} \psi'$
  6. φ^{fc} este $\forall x_1 \ldots \forall x_n \psi'$

Clauze

- O clauză este o disjuncție de literali.
- Dacă $L_1, \ldots, L_n$ sunt literali atunci clauza $L_1 \vee \ldots \vee L_n$ o vom scrie ca mulțimea $\{L_1, \ldots, L_n\}$

  clauză = mulțime de literali

- Clauza $C = \{L_1, \ldots, L_n\}$ este satisfiabilă dacă $L_1 \vee \ldots \vee L_n$ este satisfiabilă.
- O clauză C este trivială dacă conține un literal și complementul lui.
- Când $n = 0$ obținem clauza vidă, care se notează $\square$
- Prin definiție, clauza $\square$ nu este satisfiabilă.

Forma clauzală

- Observăm că o FNC este o conjuncție de clauze.
- Dacă $C_1, \ldots, C_k$ sunt clauze atunci $C_1 \wedge \ldots \wedge C_k$ o vom scrie ca mulțimea $\{C_1, \ldots, C_k\}$

$$\text{FNC} = \text{mulțime de clauze}$$

- O mulțime de clauze $\mathcal{C} = \{C_1, \ldots, C_k\}$ este satisfiabilă dacă $C_1 \wedge \ldots \wedge C_k$ este satisfiabilă
- Când $k = 0$ obținem mulțimea de clauze vidă, pe care o notăm $\{\}$
- Prin definiție, mulțimea de clauze vidă $\{\}$ este satisfiabilă.

$$\{\} \text{ este satisfiabilă, dar } \{\square\} \text{ nu este satisfiabilă}$$

Forma clauzală

- Dacă φ este o formulă în calculul propozițional, atunci
  $\varphi^{fc} = \bigwedge_{i=1}^{k} \bigvee_{j=1}^{n_i} L_{ij}$ unde L_{ij} sunt literali

- Dacă φ o formulă în logica de ordinul I, atunci
  $\varphi^{fc} = \forall x_1 \ldots \forall x_n \left(\bigwedge_{i=1}^{k} \bigvee_{j=1}^{n_i} L_{ij} \right)$ unde L_{ij} sunt literali

φ este satisfiabilă dacă și numai dacă

φ^{fc} este satisfiabilă dacă și numai dacă

$\{\{L_{11}, \ldots, L_{1n_1}\}, \ldots, \{L_{k1}, \ldots, L_{kn_k}\}\}$ este satisfiabilă

Rezoluția în logica propozițională

Regula rezoluției

$$Rez \ \frac{C_1 \cup \{p\}, C_2 \cup \{\neg p\}}{C_1 \cup C_2}$$

unde C_1, C_2 clauze, iar p este variabila propozițională astfel încât $\{p, \neg p\} \cap C_1 = \emptyset$ și $\{p, \neg p\} \cap C_2 = \emptyset$.

Fie $\mathcal{C}$ o mulțime de clauze. O derivare prin rezoluție din $\mathcal{C}$ este o secvență finită de clauze astfel încât fiecare clauză este din $\mathcal{C}$ sau rezultă din clauzele anterioare prin rezoluție (este rezolvent).

Derivare prin rezoluție

Fie $\mathcal{C}$ o mulțime de clauze. O derivare prin rezoluție din $\mathcal{C}$ este o secvență finită de clauze astfel încât fiecare clauză este din $\mathcal{C}$ sau rezultă din clauzele anterioare prin rezoluție (este rezolvent).

Exemplu

Fie $\mathcal{C} = \{\{\neg q, \neg p\}, \{q\}, \{p\}\}$ o mulțime de clauze. O derivare prin rezoluție pentru $\square$ din $\mathcal{C}$ este

$$
\begin{array}{lll}
C_1 = & \{\neg q, \neg p\} & \\
C_2 = & \{q\} & \\
C_3 = & \{\neg p\} & (Rez, C_1, C_2) \\
C_4 = & \{p\} & \\
C_5 = & \square & (Rez, C_3, C_4)
\end{array}
$$

Teorema de completitudine

$\models \varphi$ dacă și numai dacă există o derivare prin rezoluție a lui $\square$ din $(\neg\varphi)^{fc}$.

Procedura Davis-Putnam DPP (informal)

Intrare: o mulțime $\mathcal{C}$ de clauze

Se repetă următorii pași:

- se elimină clauzele triviale
- se alege o variabilă p
- se adaugă la mulțimea de clauze toți rezolvenții obținuti prin aplicarea *Rez* pe variabila p
- se șterg toate clauzele care conțin p sau $\neg p$

Ieșire: dacă la un pas s-a obținut $\square$, mulțimea $\mathcal{C}$ nu este satisfiabilă; altfel $\mathcal{C}$ este satisfiabilă.

Logica Horn

Clauze definite. Programe logice. Clauze Horn

- clauză:

  $$\{\neg Q_1, \ldots, \neg Q_n, P_1, \ldots, P_k\} \quad \text{sau} \quad Q_1 \land \ldots \land Q_n \to P_1 \lor \ldots \lor P_k$$

  unde $n, k \geq 0$ și $Q_1, \ldots, Q_n, P_1, \ldots, P_k$ sunt formule atomice.

- clauză program definită: $k = 1$
  - cazul $n > 0$: $Q_1 \land \ldots \land Q_n \to P$
  - cazul $n = 0$: $\top \to P$ (clauză unitate, fapt)

  Program logic definit = mulțime finită de clauze definite

- scop definit (țintă, întrebare): k=0
  - $Q_1 \land \ldots \land Q_n \to \bot$

- clauza vidă $\square$: $n = k = 0$

  Clauza Horn = clauză program definită sau clauză scop ($k \leq 1$)

Programare logica

- Logica clauzelor definite/Logica 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.

Limbajul PROLOG are la bază logica clauzelor Horn.

Sistem de deducție *backchain*

Sistem de deducție pentru clauze Horn

Pentru un program logic definit KB avem

- Axiome: orice clauză din KB
- Regula de deducție: regula *backchain*

$$\frac{\theta(Q_1) \quad \theta(Q_2) \quad \ldots \quad \theta(Q_n) \quad (Q_1 \land Q_2 \land \ldots \land Q_n \to P)}{\theta(Q)}$$

unde $Q_1 \land Q_2 \land \ldots \land Q_n \to P \in KB$, iar θ este cgu pentru Q și P.

Sistem de deducție

$$\frac{\theta(Q_1) \quad \theta(Q_2) \quad \ldots \quad \theta(Q_n) \quad (Q_1 \land Q_2 \land \ldots \land Q_n \to P)}{\theta(Q)}$$

unde $Q_1 \land Q_2 \land \ldots \land Q_n \to P \in KB$, iar θ este cgu pentru Q și P.

Exemplu

KB conține următoarele clauze definite:

$father(jon, ken)$. $father(ken, liz)$.
$father(X, Y) \to ancestor(X, Y)$
$daugther(X, Y) \to ancestor(Y, X)$
$ancestor(X, Y) \land ancestor(Y, Z) \to ancestor(X, Z)$

atunci

$$\frac{\dfrac{father(ken, liz)}{father(ken, Z)} \quad (father(Y, X) \to ancestor(Y, X))}{ancestor(ken, Z)}$$

Rezoluția SLD

Rezoluția SLD

Fie T o mulțime de clauze definite.

$$\text{SLD} \quad \boxed{\dfrac{\neg Q_1 \vee \cdots \vee \neg Q_i \vee \cdots \vee \neg Q_n}{\theta(\neg Q_1 \vee \cdots \vee \neg P_1 \vee \cdots \vee \neg P_m \vee \cdots \vee \neg Q_n)}}$$

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) și
- variabilele din $Q \vee \neg P_1 \vee \cdots \vee \neg P_m$ și Q_i se redenumesc
- θ este c.g.u 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

Exercițiu

Găsiți o SLD-respingere pentru următorul program Prolog și ținta:

```
1.  p(X) :- q(X,f(Y)), r(a).        ?- p(X), q(Y,Z).
2.  p(X) :- r(X).
3.  q(X,Y) :- p(Y).
4.  r(X) :- q(X,Y).
5.  r(f(b)).
```

Soluție

$$
\begin{array}{ll}
G_0 = \neg p(X) \vee \neg q(Y,Z) & \\
G_1 = \neg r(X_1) \vee \neg q(Y,Z) & (2 \text{ cu } \theta(X) = X_1) \\
G_2 = \neg q(Y,Z) & (5 \text{ cu } \theta(X_1) = f(b)) \\
G_3 = \neg p(Z_1) & (3 \text{ cu } \theta(X) = Y_1 \text{ și } \theta(Y) = Z_1) \\
G_4 = \neg r(X) & (2 \text{ cu } \theta(Z_1) = X) \\
G_5 = \square & (5 \text{ cu } \theta(X) = f(b))
\end{array}
$$

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 - arbore de căutare complet

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 - arbore SLD complet

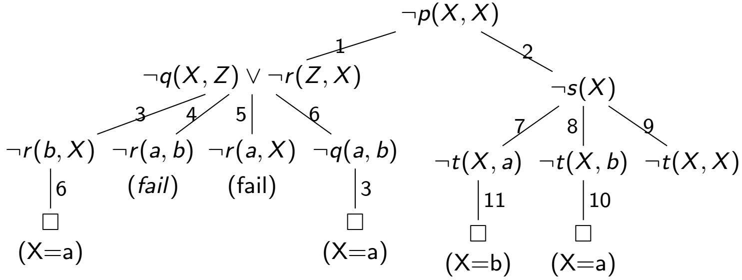
```
1.  p(X,Y) :- q(X,Z), r(Z,Y).
2.  p(X,X) :- s(X).
3.  q(X,b).
4.  q(b,a).
5.  q(X,a) :- r(a,X).
6.  r(b,a).
7.  s(X) :- t(X,a).
8.  s(X) :- t(X,b).
9.  s(X) :- t(X,X).
10.  t(a,b).
11.  t(b,a).
```

$p(X, Y) \vee \neg q(X, Z) \vee \neg r(Z, Y)$
$p(X, X) \vee \neg s(X)$
$q(X, b)$

$q(b, a)$
$q(X, a) \vee \neg r(a, X)$
$r(b, a)$

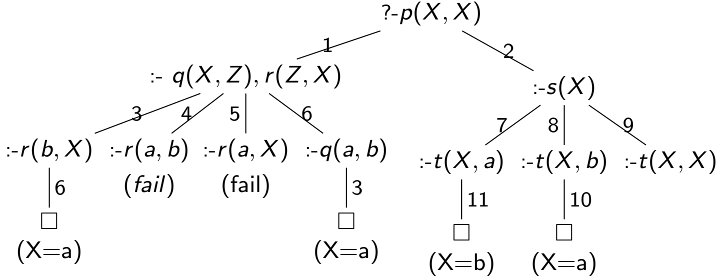
$s(X) \vee \neg t(X, a)$
$s(X) \vee \neg t(X, b)$
$s(X) \vee \neg t(X, X)$

$t(a, b)$
$t(b, a)$



Rezoluția SLD - arbori de execuție

```
1.  p(X,Y) :- q(X,Z), r(Z,Y).       4.  q(b,a).                 7.  s(X) :- t(X,a).      10.  t(a,b).
2.  p(X,X) :- s(X).                 5.  q(X,a) :- r(a,X).       8.  s(X) :- t(X,b).      11.  t(b,a).
3.  q(X,b).                         6.  r(b,a).                 9.  s(X) :- t(X,X).
```



Rezoluția SLD

Exercițiu

Fie KB următoarea bază de cunoștințe definită în Prolog:

1. `r(a, a)`
2. `q(X, a)`
3. `p(X, Y) :- q(X, Z), r(Z, Y)`

(a)Desenați arborele SLD și arborele de execuție pentru întrebarea
`?- p(X, Z)`

(b) Exprimați KB ca o mulțime de formule în logica de ordinul I
demonstrați folosind rezoluția că din KB se deduce `p(X,Z)`, adică
KB $\vdash \exists x \exists z\, p(x, z)$.

Rezoluția SLD

(a) Soluție:

```
1. r(a, a).
2. q(X, a).
3. p(X, Y) :- q(X, Z), r(Z, Y)
```

Arborele SLD:

$$\neg p(X, Z)$$
$$\downarrow 3$$
$$\neg q(X1, Z1) \vee \neg r(Z1, Y1)$$
$$\downarrow 2$$
$$\neg r(a, Y1)$$
$$\downarrow 1$$
$$\square$$

Arborele de execuție:

```
?- p(X,Z)
   ↓ 3
:- q(X1,Z1), r(Z1,Y1)
   ↓ 2
:- r(a,Y1)
   ↓ 1
   □
```

Rezoluția SLD

(cont.)
Fie KB următoarea bază de cunoștințe definită în Prolog:

1. `r(a, a).` 2. `q(X, a).` 3. `p(X, Y) :- q(X, Z), r(Z, Y)`

(b) Soluție:

KB$=\{r(a,a), \forall x\, q(x,a), \forall x \forall y \forall z\, (\neg q(x,y) \vee \neg r(z,y) \vee p(x,y))\}$

KB $\vdash \exists x \exists z\ p(x,z)$ dacă și numai dacă există o derivare prin rezoluție pentru $\square$ din forma clauzală a mulțimii $KB \cup \{\neg(\exists x \exists z\ p(x,z))\}$.

Forma clauzală a mulțimii $KB \cup \{\neg(\exists x \exists z\ p(x,z))\}$ este
$\mathcal{C} = \{\{r(a,a)\}, \{q(x,a)\}, \{\neg q(x,y), \neg r(z,y), p(x,y)\}, \{\neg p(x,z)\}\}$. Se

face derivarea direct sau se construiește arborele SLD.

Pe săptămâna viitoare!