Curs 11

2017-2018 Programare Logică

Cuprins

Rescrierea termenilor

Fie $\mathcal{L}$ un limbaj de ordinul I.

regulă de rescriere	$l \to r$	l, r termeni din $Trm_{\mathcal{L}}$
sistem de rescriere (TRS)	R	mai multe $l \to r$
relația de rescriere	$\to_R$	generată de R
echivalența	$\stackrel{*}{\leftrightarrow}_R$	generată de $\to_R$

$(Trm_{\mathcal{L}}, R)$ este un sistem de rescriere abstract.

Sisteme de rescriere

- Terminarea unui sistem de rescriere este nedecidabilă.
    - echivalentă cu oprirea mașinilor Turing
- Pentru sisteme de rescriere particulare putem decide terminarea.
    - diverse metode
- Pentru sisteme de rescriere care se termină, confluența este decidabilă.
    - algoritmul Knuth-Bendix

Confluență și perechi critice

Fie $\mathcal{L}$ un limbaj de ordinul I și R un sistem de rescriere pentru termeni.

$$\begin{tikzcd} & \theta(l_1) \arrow[dl] \arrow[dr] & \\ \theta(r_1) & & \theta(c)[z \leftarrow \theta(r_2)] \end{tikzcd}$$

$l_1 \to r_1,\ l_2 \to r_2 \in R$,
$Var(l_1) \cap Var(l_2) = \emptyset$,
$l_1 = c[z \to t]$,
$\theta(t) = \theta(l_2)$ c.g.u

Pereche critică: $(\theta(r_1), \theta(c)[z \leftarrow \theta(r_2)])$

Teoremă (Teorema Perechilor Critice)

Dacă R este noetherian, atunci sunt echivalente:

1. *R este confluent,*
2. *$t_1 \downarrow_R t_2$ pentru orice pereche critică (t_1, t_2).*

Terminarea sistemelor de rescriere

Propoziție

Sunt echivalente:

1. Un sistem de rescrire R este noetherian.
2. oricărui termen t îi poate fi asociat un număr natural $\mu(t) \in \mathbb{N}$ astfel încât $t \to_R t'$ implică $\mu(t) > \mu(t')$.

Terminarea sistemelor de rescriere

Propoziție

Sunt echivalente:

1. Un sistem de rescrire R este noetherian.
2. oricărui termen t îi poate fi asociat un număr natural $\mu(t) \in \mathbb{N}$ astfel încât $t \to_R t'$ implică $\mu(t) > \mu(t')$.

Teoremă

Sunt echivalente:

1. Un sistem de rescrire R este noetherian.
2. Există o ordine de reducere $>$ care satisface $l > r$ pentru orice $l \to r \in R$.

Ordine de reducere

Fie $\mathcal{L}$ un limbaj de ordinul I și R un sistem de rescriere pentru termeni.

Definiție

O ordine strictă $>$ pe $Trm_{\mathcal{L}}$ se numește ordine de reducere dacă

- este *well-founded*:
  - nu există lanțuri descrescătoare infinite, i.e. nu există o secvență infinită de termeni $t_0, t_1, t_2, \dots$ astfel încât $t_0 > t_1 > t_2 > \dots$
- este compatibilă cu simbolurile de funcții:
  - dacă $s_1 > s_2$, atunci $f(t_1, \dots, t_{i-1}, s_1, t_{i+1}, \dots, t_n) > f(t_1, \dots, t_{i-1}, s_2, t_{i+1}, \dots, t_n)$, pentru orice simbol de funcție f de aritate n
- este închisă la substituții:
  - dacă $s_1 > s_2$, atunci $\theta(s_1) > \theta(s_2)$ pentru orice substituție θ

Ordine de reducere

Exemplu

Relația de ordine strictă $>$ pe $Trm_{\mathcal{L}}$ definită prin

$$s > t \quad \text{ddacă} \quad |s| > |t| \text{ și } nr_x(s) \geq nr_x(t), \text{ pentru orice } x \in Var$$

este o ordine de reducere.

Ordine de reducere

Exemplu

Relația de ordine strictă $>$ pe $Trm_{\mathcal{L}}$ definită prin

$$s > t \quad \text{ddacă} \quad |s| > |t| \text{ și } nr_x(s) \geq nr_x(t), \text{ pentru orice } x \in Var$$

este o ordine de reducere.

Exemplu

Ordinea lexicografică $>_{lpo}$ indusă pe mulțimea de termeni $Trm_{\mathcal{L}}$ de o relație de ordine strictă $>$ pe limbajul $\mathcal{L}$ este o ordine de reducere.

Algoritmul Knuth-Bendix

Algoritmul Knuth-Bendix

- Procedură pentru a completa un TRS noetherian.

- Intrare: R un sistem de rescriere (TRS) noetherian.
- Ieșire:
  - T un sistem de rescriere (TRS) = completarea lui R.
  - eșec

Algoritmul Knuth-Bendix

- INTRARE: R un sistem de rescriere (TRS) noetherian.

Algoritmul Knuth-Bendix

- INTRARE: R un sistem de rescriere (TRS) noetherian.
- INIȚIALIZARE: T := R și $>$ ordine de reducere pentru T

Algoritmul Knuth-Bendix

- INTRARE: R un sistem de rescriere (TRS) noetherian.
- INIȚIALIZARE: T := R și $>$ ordine de reducere pentru T
- Se execută următorii pași, cât timp este posibil:

Algoritmul Knuth-Bendix

- INTRARE: R un sistem de rescriere (TRS) noetherian.
- INIȚIALIZARE: T := R și $>$ ordine de reducere pentru T
- Se execută următorii pași, cât timp este posibil:
  1. CP := CP(T) $= \{(t_1, t_2) \mid (t_1, t_2)$ pereche critică în $T\}$

Algoritmul Knuth-Bendix

- INTRARE: R un sistem de rescriere (TRS) noetherian.
- INIȚIALIZARE: T := R și > ordine de reducere pentru T
- Se execută următorii pași, cât timp este posibil:
  1. CP := CP(T) $= \{(t_1, t_2) \mid (t_1, t_2)$ pereche critică în $T\}$
  2. Dacă $t_1 \downarrow t_2$, oricare $(t_1, t_2) \in CP$, atunci STOP *(T completarea lui R)*.

Algoritmul Knuth-Bendix

- INTRARE: R un sistem de rescriere (TRS) noetherian.
- INIȚIALIZARE: T := R și $>$ ordine de reducere pentru T
- Se execută următorii pași, cât timp este posibil:
  1. CP := CP(T) $= \{(t_1, t_2) \mid (t_1, t_2)$ pereche critică în $T\}$
  2. Dacă $t_1 \downarrow t_2$, oricare $(t_1, t_2) \in CP$, atunci STOP *(T completetarea lui R)*.
  3. Dacă $(t_1, t_2) \in CP$, $t_1 \not\downarrow t_2$ atunci:
     - dacă $fn(t_1) > fn(t_2)$ atunci T := T $\cup \{fn(t_1) \to fn(t_2)\}$,
     - dacă $fn(t_2) > fn(t_1)$ atunci T := T $\cup \{fn(t_2) \to fn(t_1)\}$,
     - altfel, STOP *(completare eșuată)*.

Algoritmul Knuth-Bendix

- INTRARE: R un sistem de rescriere (TRS) noetherian.
- INIȚIALIZARE: T := R și $>$ ordine de reducere pentru T
- Se execută următorii pași, cât timp este posibil:
  1. CP := CP(T) $= \{(t_1, t_2) \mid (t_1, t_2)$ pereche critică în $T\}$
  2. Dacă $t_1 \downarrow t_2$, oricare $(t_1, t_2) \in CP$, atunci STOP *(T completetarea lui R)*.
  3. Dacă $(t_1, t_2) \in CP$, $t_1 \not\downarrow t_2$ atunci:
     - dacă $fn(t_1) > fn(t_2)$ atunci T := T $\cup \{fn(t_1) \to fn(t_2)\}$,
     - dacă $fn(t_2) > fn(t_1)$ atunci T := T $\cup \{fn(t_2) \to fn(t_1)\}$,
     - altfel, STOP *(completare eșuată)*.
- IEȘIRE: T completarea lui R sau eșec.

Algoritmul Knuth-Bendix

- INTRARE: R un sistem de rescriere (TRS) noetherian.
- INIȚIALIZARE: T := R și $>$ ordine de reducere pentru T
- Se execută următorii pași, cât timp este posibil:
  1. CP := CP(T) = $\{(t_1, t_2) \mid (t_1, t_2)$ pereche critică în $T\}$
  2. Dacă $t_1 \downarrow t_2$, oricare $(t_1, t_2) \in CP$, atunci STOP *(T completarea lui R)*.
  3. Dacă $(t_1, t_2) \in CP$, $t_1 \not\downarrow t_2$ atunci:
     - dacă $fn(t_1) > fn(t_2)$ atunci $T := T \cup \{fn(t_1) \to fn(t_2)\}$,
     - dacă $fn(t_2) > fn(t_1)$ atunci $T := T \cup \{fn(t_2) \to fn(t_1)\}$,
     - altfel, STOP *(completare eșuată)*.
- IEȘIRE: T completarea lui R sau eșec.

Atenție! Succesul completării depinde de ordinea de reducere $>$.

Exemplu

Exemplu

- Fie $\mathcal{L}$ un limbaj de ordinul I cu un simbol de funcție $*$ de aritate 2.
- Fie $R = \{(x * y) * (y * v) \to y\}$.
- Vrem să determinăm completarea sistemului R aplicând algoritmul Knuth-Bendix.
- INIȚIALIZARE:
  - $T = R = \{(x * y) * (y * v) \to y\}$,
  - Ordine de reducere:
    $s > t$ ddacă $|s| > |t|$ și $nr_x(s) \geq nr_x(t)$, pentru orice $x \in X$

Exemplu

Exemplu

- Determinăm perechile critice pentru

  $l_1 := (x * y) * (y * v),\ r_1 := y,\ l_2 := (x' * y') * (y' * v'),\ r_2 := y'$

Exemplu

Exemplu

- Determinăm perechile critice pentru

  $l_1 := (x * y) * (y * v),\ r_1 := y,\ l_2 := (x' * y') * (y' * v'),\ r_2 := y'$

  Subtermenii lui l_1 care nu sunt variabile:

$$(x * y),\ (y * v),\ (x * y) * (y * v)\ .$$

Exemplu

Exemplu

- Determinăm perechile critice pentru

  $l_1 := (x * y) * (y * v),\ r_1 := y,\ l_2 := (x' * y') * (y' * v'),\ r_2 := y'$

  Subtermenii lui l_1 care nu sunt variabile:

  $$(x * y),\ (y * v),\ (x * y) * (y * v)\ .$$

  - $t := x * y,\ c = z * (y * v),\ \theta := \{x \leftarrow x' * y', y \leftarrow y' * v'\}$
    $\theta(r_1) = y' * v',\ \theta(c)[z \leftarrow \theta(r_2)] = y' * ((y' * v') * v)$
    Perechea critică: $(y' * v', y' * ((y' * v') * v))$.

Exemplu

Exemplu

- Determinăm perechile critice pentru

$$l_1 := (x * y) * (y * v),\ r_1 := y,\ l_2 := (x' * y') * (y' * v'),\ r_2 := y'$$

Subtermenii lui l_1 care nu sunt variabile:

$$(x * y),\ (y * v),\ (x * y) * (y * v)\ .$$

  - $t := x * y,\ c = z * (y * v),\ \theta := \{x \leftarrow x' * y', y \leftarrow y' * v'\}$
    $\theta(r_1) = y' * v',\ \theta(c)[z \leftarrow \theta(r_2)] = y' * ((y' * v') * v)$
    Perechea critică: $(y' * v', y' * ((y' * v') * v))$.
  - $t := y * v,\ c = (x * y) * z,\ \theta := \{y \leftarrow x' * y', v \leftarrow y' * v'\}$
    $\theta(r_1) = x' * y',\ \theta(c)[z \leftarrow \theta(r_2)] = (x * (x' * y')) * y'$
    Perechea critică: $(x' * y', (x * (x' * y')) * y')$.

Exemplu

Exemplu

- Determinăm perechile critice pentru

  $l_1 := (x * y) * (y * v),\ r_1 := y,\ l_2 := (x' * y') * (y' * v'),\ r_2 := y'$

  Subtermenii lui l_1 care nu sunt variabile:

  $$(x * y),\ (y * v),\ (x * y) * (y * v)\ .$$

  - $t := x * y,\ c = z * (y * v),\ \theta := \{x \leftarrow x' * y', y \leftarrow y' * v'\}$
    $\theta(r_1) = y' * v',\ \theta(c)[z \leftarrow \theta(r_2)] = y' * ((y' * v') * v)$
    Perechea critică: $(y' * v', y' * ((y' * v') * v))$.

  - $t := y * v,\ c = (x * y) * z,\ \theta := \{y \leftarrow x' * y', v \leftarrow y' * v'\}$
    $\theta(r_1) = x' * y',\ \theta(c)[z \leftarrow \theta(r_2)] = (x * (x' * y')) * y'$
    Perechea critică: $(x' * y', (x * (x' * y')) * y')$.

  - $t := (x * y) * (y * v),\ c = z,\ \theta := \{x \leftarrow x', y \leftarrow y', v \leftarrow v'\}$
    $\theta(r_1) = y',\ \theta(c)[z \leftarrow \theta(r_2)] = y'$
    Perechea critică: (y', y').

Exemplu

Exemplu

- Perechile critice:
  1. $(y' * v', y' * ((y' * v') * v))$,
  2. $(x' * y', (x * (x' * y')) * y')$,
  3. (y', y').

Exemplu

Exemplu

- Perechile critice:
  1. $(y' * v', y' * ((y' * v') * v))$,
  2. $(x' * y', (x * (x' * y')) * y')$,
  3. (y', y').
- Avem
  - $y * ((y * x) * v) > y * x$
  - $(x * (v * y)) * y > v * y$

Exemplu

Exemplu

- Perechile critice:
  1. $(y' * v', y' * ((y' * v') * v))$,
  2. $(x' * y', (x * (x' * y')) * y')$,
  3. (y', y').
- Avem
  - $y * ((y * x) * v) > y * x$
  - $(x * (v * y)) * y > v * y$
- Considerăm
  $T := T \cup \{y * ((y * x) * v) \to y * x,\ (x * (v * y)) * y \to v * y\}$
- T este complet și este completarea lui R_E.

Terminarea programelor în Prolog

Terminarea programelor în Prolog

- Un program logic nu este un sistem de rescriere de termeni (TRS).
- Dar un program logic poate fi transformat într-un TRS astfel încât terminarea TRS-ului să implice terminarea programului logic.
- Vom exemplifica această transformare pentru o clasă de programe logice numite well moded.

Programe logice well moded

- Pentru fiecare poziție dintr-un predicat stabilim dacă este o poziție de intrare sau una de ieșire printr-o funcție *moding* m.
- Pentru fiecare simbol de predicat P de aritate n și fiecare $1 \leq i \leq n$ avem $m(p, i) \in \{\mathbf{in}, \mathbf{out}\}$
- $m(p, i)$ stabilește dacă al i-ulea argument al lui P este de intrare (**in**) sau de ieșire (**out**)

Programe logice well moded

- Funcția *moding* trebuie aleasă astfel încât programul logic să fie *well-moded*.
  - Această proprietate garantează că fiecare atom selectat de strategia Prolog-ului (de la stanga la dreapta) este instanțiat "suficient" în timpul unei derivări pentru o țintă ce conține doar termeni fără variabile.

- Un program este well moded dacă pentru fiecare regulă $\mathtt{H}\ \mathtt{:-}\ \mathtt{B}_1,\ldots,\mathtt{B}_k$ cu $k \geq 0$ avem:

  1. $Var_{out}(\mathtt{H}) \subseteq Var_{in}(\mathtt{H}) \cup Var_{out}(\mathtt{B}_1) \cup \ldots \cup Var_{out}(\mathtt{B}_k)$
  2. $Var_{in}(\mathtt{B}_i) \subseteq Var_{in}(\mathtt{H}) \cup Var_{out}(\mathtt{B}_1) \cup \ldots \cup Var_{out}(\mathtt{B}_{i-1})$ pentru orice $1 \leq i \leq k$

  unde $Var_{in}(\mathtt{B})$ și $Var_{out}(\mathtt{B})$ sunt variabilele din termenii din $\mathtt{B}$ care apar în poziții de intrare și, respectiv, de ieșire.

Programe logice well moded

Exemplu

Să considerăm următorul program logic:

```
p(X,X).
p(f(X),g(Y)) :- p(f(X),f(Z)), p(Z,g(Y)).
```

Fie m funcția $m(p,1) = \textbf{in}$ și $m(p,2) = \textbf{out}$.
Atunci programul este well moded:

- Pentru prima regula este evident.
- Pentru a doua regula, avem:
  1. este adevărată deoarece variabila de ieșire Y din capul regulii este variabilă de ieșire în al doilea atom din corpul regulii.
  2. este adevărată deoarece variabila de intrare X din primul atom din corpul regulii este variabilă de intrare în capul regulii și variabila de intrare Z din al doilea atom din corpul regulii este variabilă de ieșire în primul atom din corpul regulii.

Transformarea clasică

- Pentru transformarea clasică a unui program logic într-un TRS, se introduc două simboluri noi de funcție p_{in} și p_{out} pentru orice simbol de predicat p.
- Vom scrie $p(\vec{s}, \vec{t})$ pentru a indica faptul că $\vec{s}$ și $\vec{t}$ sunt secvențe de termeni în pozițiile de intrare și, respectiv, de ieșire ale lui p.
- Pentru fiecare fapt $p(\vec{s}, \vec{t})$, TRS-ul construit conține regula $p_{in}(\vec{s}) \to p_{out}(\vec{t})$
- Pentru fiecare regulă de forma $p(\vec{s}, \vec{t}) :- p_1(\vec{s_1}, \vec{t_1}), \ldots, p_k(\vec{s_k}, \vec{t_k})$, TRS-ul construit conține regulile:

$$p_{in}(\vec{s}) \to u_{c,1}(p_{1_{in}}(\vec{s_1}), Var(\vec{s}))$$
$$u_{c,1}(p_{1_{out}}(\vec{t_1}), Var(\vec{s})) \to u_{c,2}(p_{2_{in}}(\vec{s_2}), Var(\vec{s}) \cup Var(\vec{t_1}))$$
$$\ldots$$
$$u_{c,k}(p_{k_{in}}(\vec{t_k}), Var(\vec{s}) \cup Var(\vec{t_1}) \cup \ldots \cup Var(\vec{t_{k-1}})) \to p_{out}(\vec{t})$$

Transformarea clasică

Exemplu

Pentru programul logic de mai jos

```
p(X,X).
p(f(X),g(Y)) :- p(f(X),f(Z)), p(Z,g(Y)).
```

cu funcția $m(p,1) = \mathbf{in}$ și $m(p,2) = \mathbf{out}$, TRS-ul obținut este:

$p_{in}(X) \rightarrow p_{out}(X)$

$p_{in}(f(X)) \rightarrow u_1(p_{in}(f(X)), X)$

$u_1(p_{out}(f(Z)), X) \rightarrow u_2(p_{in}(Z), X, Z)$

$u_2(p_{out}(g(Y)), X, Z) \rightarrow p_{out}(g(Y))$

Transformarea clasică

- Pentru un program logic well moded, dacă TRS-ul obținut se termină, atunci și programul logic se termină pentru orice țintă cu termeni fără variabile în pozițiile de intrare.
- Implicația inversă nu este adevărată.
- În literatură există astfel de transformări și pentru programe logice oarecare.
- Pentru mai multe detalii consultați Teza de doctorat a lui Peter Schneider-Kamp:

  Static Termination Analysis for Prolog unig Term Rewriting and SAT Solving, RWTH Aachen, 2008.

Prolog - Backtracking, Cut, Negații

Backtracking, Cut, Negații

- În cursurile trecute, am stabilit ce obiect matematic se obține dintr-un program în Prolog și cum putem raționa cu/despre el.
- În continuare vom introduce un mecanism de control în Prolog (cuts) care ne permite să scriem implementări mai eficiente.

Backtracking

Prolog folosește backtracking pentru a răspunde întrebărilor:

- În momentul în care Prolog încearcă să găsească un răspuns la o întrebare, ține minte toate punctele de decizie.
  - Puncte de decizie = situațiile în care găsește mai multe potriviri.
- De fiecare dată când un drum eșuează sau se termină, Prolog sare la ultima alegere făcută și încearcă următoarea alternativă.

Exemplu (Permutările unei liste)

- Predicatul `permutation/2` găsește toate permutările unei liste.
- Predicatul folosește predicatul predefinit `select/3` care primește o listă ca al doilea argument și găsește o potrivire între primul argument și un element al listei. Variabila din al treilea argument este lista inițială din care se elimină acel element.

Backtracking

Exemplu (Permutările unei liste)

- Predicatul `permutation/2` găsește toate permutările unei liste.
- Predicatul folosește predicatul predefinit `select/3` care primește o listă ca al doilea argument și găsește o potrivire între primul argument și un element al listei. Variabila din al treilea argument este lista inițială din care se elimină acel element.

```
permutation([],[]).

permutation(List, [Element | Permutation]) :-
       select(Element, List, Rest),
       permutation(Rest, Permutation).
```

Backtracking

Exemplu (Permutările unei liste - cont.)

```
?- permutation([1,2,3], X).

X = [1,2,3]
X = [1,3,2]
X = [2,1,3]
X = [2,3,1]
X = [3,1,2]
X = [3,2,1]
```

Exemplu (Permutările unei liste - cont.)

- Cazul cel mai simplu este pentru lista vidă, în care avem o singură permutare (lista vidă).
- Dacă lista nu este vidă, atunci subținta `select(Element, List, Rest)` o să lege variabila `Element` de un element al listei.
- Mai departe, face acel element capul listei de ieșire și apelează recursiv `permutation/2` pentru restul listei.
- Primul răspuns o să fie chiar lista de intrare, deoarece `Element` o să ia valoarea primului element din lista `List`.
- Totuși, când ne întoarcem prin backtracking la subțintele `select`, `Element` este instanțiat cu un alt element din `List`.
- Astfel ajungem să generăm toate modurile posibile în care putem selecta elementele din lista de intrare.

Problemele backtracking-ului

Sunt totuși cazuri în backtracking-ul care nu ne ajută.

Problemele backtracking-ului

Sunt totuși cazuri în backtracking-ul care nu ne ajută.

Exemplu (Eliminarea duplicatelor)

- Predicatul `remove_duplicates/2` șterge duplicatele elementelor dintr-o listă.

```
remove_duplicates([],[]).

remove_duplicates([Head | Tail], Result) :-
      member(Head,Tail),
      remove_duplicates(Tail,Result).

remove_duplicates([Head | Tail], [Head | Result]) :-
      remove_duplicates(Tail,Result).
```

Problemele backtracking-ului

Exemplu (Eliminarea duplicatelor - cont.)

```
?- remove_duplicates([a,b,b,c,a], List).

List = [b,c,a]
List = [b,b,c,a]
List = [a,b,c,a]
List = [a,b,b,c,a]
```

Problemele backtracking-ului

Exemplu (Eliminarea duplicatelor - cont.)

- Deși prima soluție găsită de Prolog este cea bună, restul soluțiilor nu sunt corecte.
- Ultimele două reguli conduc la puncte de decizie.
- Pentru primul drum în arborele de căutare, Prolog o să folosească mereu prima regulă dacă este posibil (de fiecare dată când capul este membru în coadă, este eliminat).
- Totuși, în timpul backtracking-ului, și toate celelalte drumuri din arborele de căutare o să fie considerate.
- Chiar dacă prima regulă se potrivește, câteodata a doua regulă o să fie aleasă și elementul duplicat o să ramână în listă.

Problemele backtracking-ului

Exemplu (Eliminarea duplicatelor - cont.)

- Deși prima soluție găsită de Prolog este cea bună, restul soluțiilor nu sunt corecte.
- Ultimele două reguli conduc la puncte de decizie.
- Pentru primul drum în arborele de căutare, Prolog o să folosească mereu prima regulă dacă este posibil (de fiecare dată când capul este membru în coadă, este eliminat).
- Totuși, în timpul backtracking-ului, și toate celelalte drumuri din arborele de căutare o să fie considerate.
- Chiar dacă prima regulă se potrivește, câteodata a doua regulă o să fie aleasă și elementul duplicat o să ramână în listă.

Ne trebuie o metodă prin care să anunțăm Prolog-ul când,
chiar dacă și alte soluții sunt solicitate,
nu există alternative și că ținta trebuie să eșueze.

Cuts

- În Prolog putem să "tăiem" punctele de decizie din backtracking, ghidând astfel căutarea soluțiilor și eliminând soluții alternative nedorite.
- O "tăietură" (cut) se introduce prin `!`.
- Este un predicat (de aritate 0) predefinit în Prolog care poate fi inserat oriunde în corpul unei reguli.
- Execuția subțintei `!` este mereu cu succes.
- De fiecare dată când `!` este întâlnit în corpul unei reguli, sunt finale toate alegerile făcute începând cu momentul în care capul acelei reguli a fost unificat cu scopul părinte.

Cuts

Exemplu (Eliminarea duplicatelor)

Să modificăm soluția anterioară pentru eliminarea duplicatelor prin introducerea unui cut după prima subțintă din corpul primei reguli.

```
remove_duplicates([],[]).

remove_duplicates([Head | Tail], Result) :-
       member(Head,Tail), !,
       remove_duplicates(Tail,Result).

remove_duplicates([Head | Tail], [Head | Result]) :-
       remove_duplicates(Tail,Result).

?- remove_duplicates([a,b,b,c,a], List).
List = [b,c,a]
```

Cuts

Exemplu (Eliminarea duplicatelor)

- Acum, de fiecare dată când capul este membru în coadă, prima subțintă din prima regulă (`member(Head,Tail)`) o să reușească.
- Apoi, următoarea subțintă `!` o să reușească de asemenea.
- Fără acel `!`, putem continua backtracking-ul: ținta originală este unificată cu capul următoarei reguli. Astfel, obținem soluții alternative.
- Î n momentul în care Prolog trece de `!`, aceste căutări pentru ținta inițiala nu mai au loc.

Probleme cu cut

- Cuts sunt foarte utile pentru a ghida Prolog spre o soluție.
- Totuși, introducând cuts, renunțăm la anumite caracteristici declarative ale Prolog-ului și ne îndreptăm spre un sistem procedural.

Probleme cu cut

- Cuts sunt foarte utile pentru a ghida Prolog spre o soluție.
- Totuși, introducând cuts, renunțăm la anumite caracteristici declarative ale Prolog-ului și ne îndreptăm spre un sistem procedural.

Exemplu

- Predicatul `add/3` inserează un element într-o listă doar dacă acel element nu este deja un membru al listei.
- Elementul pe care dorim să îl inserăm este dat ca prim argument, iar lista ca al doilea argument. Variabila dată ca al treilea argument este rezultatul.

```
?- add(elephant, [dog, donkey, rabbit], List).
List = [elephant, dog, donkey, rabbit]

?- add(donkey, [dog, donkey, rabbit], List).
List = [dog, donkey, rabbit]
```

Probleme cu cut

Exemplu

O soluție posibilă:

```
add(Element, List, List) :-
       member(Element, List), !.

add(Element, List, [Element | List]).
```

- Dacă elementul se află deja în listă, lista soluție este chiar cea inițială. Cum aceasta este unica soluție posibilă, împiedicăm Prolog-ul să caute o altă soluție introducând `!`.
- Altfel, în rezultat adăugam elementul la începutul listei de intrare.

Probleme cu cut

- Acesta este un exemplu în care cut creează probleme.
- Când predicatul `add/3` este folosit cu o variabilă în al treilea argument funcționează corect.
- Totuși, dacă folosim acest predicat cu o listă instanțiată în al treilea argument, răspunsul Prolog-ului nu este neapărat cel așteptat.

Exemplu

```
?- add(a, [a, b, c, d], [a, b, c, d]).
true

?- add(a, [a, b, c, d], [a, a, b, c, d]).
true

?- add(a, [a, b, c, d], [a, b, a, c, d]).
false
```

Probleme cu cut

Exemplu

O soluție alternativă:

```
add(Element, List, Result) :-
       member(Element, List), !,
       Result = List.

add(Element, List, [Element | List]).
```

- Din punct de vedere declarativ, cele două soluții sunt echivalente, dar procedural se comportă diferit.

Atenție cum folosiți cut!

Răspunsurile din Prolog

- 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, răspunsul `true` la o țintă nu înseamnă doar că ținta este adevarată, ci și că este demonstrabilă.
- Astfel, un răspuns `false` nu înseamnă neapărat că ținta nu este adevărată, ci doar că Prolog nu a reușit să găsească o demonstrație.

Răspunsurile din Prolog

Exemplu

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

?- animal(cat).
false
```

Răspunsurile din Prolog

Exemplu

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

?- animal(cat).
false
```

- Clauzele din Prolog dau doar condiții suficiente, dar nu și necesare pentru ca un predicat să fie adevărat.
- Totuși, dacă specificăm complet o problemă (adică specificăm toate cazurile posibile), atunci noțiunile de nedemonstrabil și fals coincid. Atunci un `false` e chiar un fals.

Operatorul \+

- Câteodată poate dorim să negăm o țintă.
- Negarea unei ținte se poate defini astfel:

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

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

Operatorul \+

- Câteodată poate dorim să negăm o țintă.
- 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.
- O țintă `\+ Goal` reușește dacă Prolog nu găsește o demonstrație pentru `Goal`.
- Semantica operatorului \+ se numește negation as failure.
- Negația din Prolog este definită ca incapacitatea de a găsi o demonstrație.

Operatorul \+

- Câteodată poate dorim să negăm o țintă.
- 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.
- O țintă `\+ Goal` reușește dacă Prolog nu găsește o demonstrație pentru `Goal`.
- Semantica operatorului \+ se numește negation as failure.
- Negația din Prolog este definită ca incapacitatea de a găsi o demonstrație.

  "nevinovat până la proba contrarie"

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).
```

Negation as failure

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(claudia).    ?- single(X).
false               true                   false
```

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

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

Pe săptămâna viitoare!