Prolog: Probleme

- Responsabil: Vlad Neculae [mailto:neculae.vlad@gmail.com]

- Data publicării: 11.05.2019
- Data ultimei modificări: 13.05.2019

Obiective

Scopul acestui laborator îl reprezintă învățarea unor concepte avansate de programare în Prolog, legate în special de controlul execuției, precum și de obținerea tuturor soluțiilor ce satisfac un scop.

Controlul execuției: operatorul cut (!), negația (\+) și fail

Negația ca eșec (\+)

`\+` este operatorul folosit pentru negație în Prolog (meta-predicatul `not` nu mai este recomandat). Așa cum ați observat nu se pot adăuga în baza de date fapte în forma negată și nici nu se pot scrie reguli pentru acestea. Dacă nu se poate demonstra ¬p, atunci ce semnificație are în Prolog `not(Goal)` sau `\+ Goal`?

Prolog utilizează *presupunerea lumii închise*: ceea ce nu poate fi demonstrat, este fals. De aceea, în Prolog `\+ p` trebuie citit ca „scopul `p` nu poate fi satisfăcut“ sau „`p` nu poate fi demonstrat“. Faptul că Prolog utilizează negația ca eșec (eng. *negation as failure*) are implicații asupra execuției programelor.

În logica de ordinul întâi, următoarele două expresii sunt echivalente: `¬a(X) & b(X)` și `b(X) & ¬a(X)`. În Prolog, următoarele 2 clauze (`p1` și `p2`) vor produce rezultate diferite:

```
student(andrei).
student(marius).
lazy(marius).

p1(X):- student(X), \+ lazy(X).
p2(X):- \+ lazy(X), student(X).
```

Predicatul fail

`fail` este un predicat care eșuează întotdeauna. De ce am vrea să scriem o regulă care să eșueze? Un posibil răspuns este: datorită efectelor laterale pe care le pot avea scopurile satisfăcute până la întâlnirea predicatului `fail`. Un exemplu este următorul:

```
my_reverse(List, Acc, _):-format('List:~w, Acc:~w~n', [List, Acc]), fail.
my_reverse([], Sol, Sol).
my_reverse([Head|Tail], Acc, Sol):-my_reverse(Tail, [Head|Acc], Sol).
```

În codul de mai sus, am scris o regulă pentru funcția `my_reverse` care are un singur rol: acela de a afișa argumentele `List` și `Acc` în orice moment se dorește satisfacerea unui scop cu predicatul `my_reverse/3`.

```
?- my_reverse([1,2,3,4],[],Rev).
List:[1,2,3,4], Acc:[]
List:[2,3,4], Acc:[1]
List:[3,4], Acc:[2,1]
List:[4], Acc:[3,2,1]
List:[], Acc:[4,3,2,1]
Rev = [4, 3, 2, 1].
```

Operatorul cut

În Prolog, operatorul cut (`!`) are rolul de a elimina toate punctele de întoarcere create în predicatul curent.

Dincolo de utilizarea operatorului cut (`!`) pentru a influența execuția (rezultatele) programelor (revedeți exemplul cu `max`), acesta poate avea rol și în optimizarea programelor. Eliminarea unor puncte de întoarcere acolo unde acestea oricum ar fi inutile, poate salva memorie. Uneori, erorile „Out of global stack“ pot fi rezolvate prin introducerea de operatori cut ce nu modifică rezultatele programelor.

Așa cum veți observa din problemele din laborator, cut, fail și negația devin și mai utile atunci când sunt folosite împreună.

Aflarea tuturor soluțiilor pentru satisfacerea unui scop

Prolog oferă un set special de predicate care pot construi liste din toate soluțiile de satisfacere a unui scop. Acestea sunt extrem de utile deoarece altfel este complicată culegerea informațiilor la revenirea din backtracking (o alternativă este prezentată în secțiunea următoare).

findall(+Template, +Goal, -Bag)

Predicatul `findall` pune în Bag câte un element Template pentru fiecare soluție a expresiei `Goal`. Desigur, predicatul este util atunci când `Goal` și `Template` au variabile comune. De exemplu:

```
even(Numbers, Even):-
    findall(X,(member(X, Numbers), X mod 2 =:= 0), Even).

?- even([1,2,3,4,5,6,7,8,9], Even).
Even = [2, 4, 6, 8].
```

bagof(+Template, +Goal, -Bag)

Predicatul `bagof` seamănă cu `findall`, diferența fiind că `bagof` construiește câte o listă `Bag` pentru fiecare instanțiere diferită a variabilelor libere din `Goal`.

```
digits([1,2,3,4,5,6,7,8,9]).
numbers([4,7,9,14,15,18,24,28,33,35]).

multiples(D,L):-
    digits(Digits),
    numbers(Numbers),
    bagof(N,(member(D, Digits), member(N, Numbers), N mod D =:= 0), L).


?- multiples(D,L).
D = 1,
L = [4, 7, 9, 14, 15, 18, 24, 28, 33|...] ;
D = 2,
L = [4, 14, 18, 24, 28] ;
D = 3,
L = [9, 15, 18, 24, 33] ;
D = 4,
L = [4, 24, 28] ;
D = 5,
L = [15, 35] ;
D = 6,
L = [18, 24] ;
D = 7,
L = [7, 14, 28, 35] ;
D = 8,
L = [24] ;
D = 9,
L = [9, 18].
```

Pentru a evita *gruparea* soluțiilor pentru fiecare valoare separată a variabilelor ce apar în scopul lui `bagof/3` se poate folosi construcția `Var^Goal`

setof(+Template, +Goal, -Bag)

Predicatul `setof/3` are aceeași comportare cu `bagof/3`, dar cu diferența că soluțiile găsite sunt sortate și se elimină duplicatele.

Câteva observații asupra purității

În logica de ordinul întâi clauzele `p(A,B) ^ q(A,B)` și `q(A,B) ^ p(A,B)` sunt echivalente. Ordinea termenilor dintr-o conjuncție (sau disjuncție) nu influențează valoarea de adevăr a clauzei.

În Prolog acest lucru nu este întotdeauna adevărat:

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

?- X == Y, X = Y.
false.
```

Resurse

- probleme