Controlul procedural al rationamentului

Metodele de demonstrare automată (Automated theorem proving ATP) sunt modelităţi de raţionament generale, independente de domeniul aplicaţiei. Se încearcă toate variantele logice permise pentru a căuta un răspuns la o întrebare. În unele situaţii acest lucru nu este fezabil.

De multe ori avem o idee despre cum să folosim cunoştinţele şi putem să ghidăm o procedură ATP pe baza proprietăţilor domeniului aplicaţiei.

Vom vedea cum putem exprima cunoştinţele pentru a controla procedura de întrunire înapoi.

Fapte şi reguli

Clauzele dintr-o bază de cunoştinţe se pot împărţi în două categorii:

- fapte - reprezentate prin atomi fără variabile.
- reguli - exprimă relaţii noi şi sunt de obicei condiţionale cuantificate universal

Mother(jane, john)
Father(john, billy)

Parent(x, y) ⇐ Mother(x, y)
Parent(x, y) ⇐ Father(x, y)

Regulile implică înlănţuirea, deci punctul cheie în problema controlului este cum să folosim cât mai eficient regulile dintr-o bază de cunoştinţe.

Construirea regulilor şi strategii de căutare

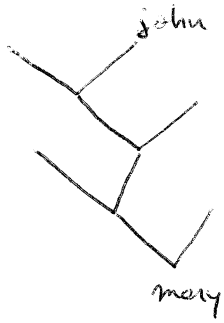
Pentru a reprezenta relaţia Ancestor putem scrie trei variante echivalente logic:

1. Ancestor(x, y) ⇐ Parent(x, y)
   Ancestor(x, y) ⇐ Parent(x, z) ∧ Ancestor(z, y)

2. Ancestor(x, y) ⇐ Parent(x, y)
   Ancestor(x, y) ⇐ Parent(z, y) ∧ Ancestor(x, z)

3. $Ancestor(x,y) \Leftarrow Parent(x,y)$

$Ancestor(x,y) \Leftarrow Ancestor(x,z) \wedge Ancestor(z,y)$



1. se coboară în arborele de familie
2. se urcă în arbore
3. se caută în ambele direcții în același timp

Dacă în medie oamenii ar avea un copil, varianta (1) ar fi de ordin d iar varianta (2) de ordin 2^d, unde d este adâncimea căutării. Dacă oamenii ar avea mai mult de doi copii, varianta (2) ar fi de preferat

Designul algoritmilor

Șirul Fibonacci $\begin{cases} x_0 = 0 \\ x_1 = 1 \\ x_{n+2} = x_{n+1} + x_n \quad , n \geq 0 \end{cases}$

în KB avem $\begin{cases} Fib(0,1) \\ Fib(1,1) \\ Fib(s(s(n)), v) \Leftarrow Fib(n,y) \wedge Fib(s(n),z) \wedge Plus(y,z,v) \\ Plus(0,z,z) \\ Plus(s(x),y,s(z)) \Leftarrow Plus(x,y,z) \end{cases}$

Multe calcule sunt redundante

$Fib(10,\_)$ invocă $Fib(9,\_)$ și $Fib(8,\_)$

$Fib(11,\_)$ invocă $Fib(10,\_)$ și $Fib(9,\_)$

Se generează un număr exponențial de subscopuri Plus deoarece fiecare apelare Fib invocă Fib de două ori

O alternativă este $\begin{cases} Fib(n,v) \Leftarrow F(n,1,0,v) \\ F(0,y,z,y) \\ F(s(n),y,z,v) \Leftarrow Plus(y,z,s) \wedge F(n,s,y,v) \end{cases}$

$F(n, \_, \_, v)$ — rezultatul dorit când n ajunge 0

| 2 numere Fib consecutive

se pleacă de la n spre 0

Ordinea scopurilor

Dpv logic, toate ordonările scopurilor sunt permise dar diferențele dpv computațional pot fi semnificative.

De exemplu, pentru

$$AmericanCousin(x,y) \Leftarrow American(x) \wedge Cousin(x,y)$$

avem două opțiuni:

- să găsim un American și apoi verificăm dacă este vărul cuiva
- găsim un văr și apoi verificăm dacă este american.

În acest caz particular, rezolvarea întâi a $Cousin(x,y)$ și apoi $American(x)$ este mai bună decât inordinea inversă.

Limbajul PROLOG ia în seamă constrângerile de ordine.

Predicatul ! ("cut") în PROLOG – controlul backtracking-ului și negația ca eșec

! este întotdeauna adevărat; previne backtrackingul în punctul unde apare în program.

Dacă ! nu modifică înțelesul declarativ al programului, se numește cut verde; altfel se numește cut roșu.

Funcția $f(x) = \begin{cases} 0, & x \leq 3 \\ 2, & x \in (3,6] \\ 4, & x > 6 \end{cases}$ poate fi implementată astfel:

KB
```
f(X, 0) :- X =< 3.      % regula 1
f(X, 2) :- 3 < X, X =< 6.   % reg 2
f(X, 4) :- 6 < X.       % reg 3
```

```
?- f(1,Y), 2 < Y.
```

X = 1, Y = 0 1 =< 3, 2 < 0 fals

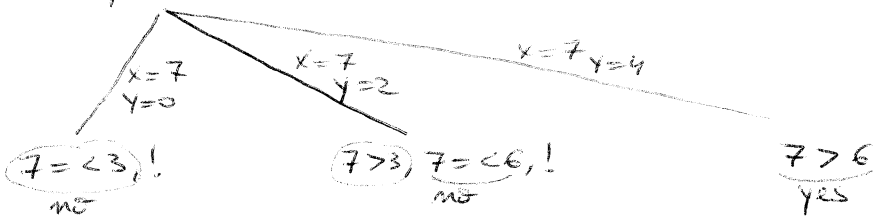
X = 1 Y = 2 fals

X = 1 Y = 4 fals

Programul ar fi trebuit să se oprească după prima verificare.

```
f(X, 0) :- X =< 3, !.
f(X, 2) :- 3 < X, X =< 6, !.      ! verde
f(X, 4) :- 6 < X.
```

?- f(7, Y).



teste redundante

```
f(x,0) :- x =< 3, !.
f(x,2) :- x =< 6, !.
f(x,4).
```

! roșu - dacă eliminăm cut-ul și întrebăm

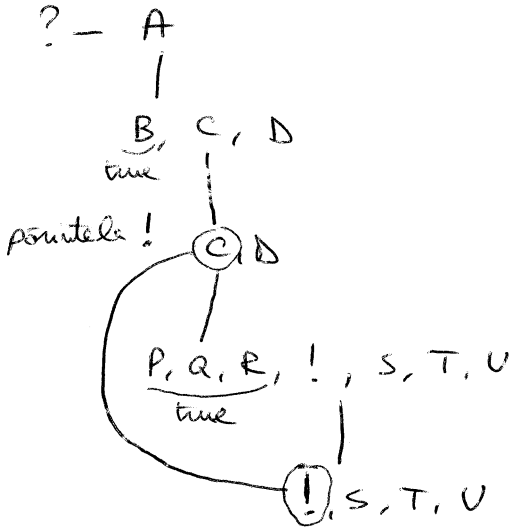
?- f(1, Y).
Y=0;
Y=2;
Y=4;
no

Se numește părinte al cut-ului acel scop PROLOG care unifică cu capul regulii ce conține cut-ul respectiv.

```
C :- P, Q, R, !, S, T, U
C :- V.
A :- B, C, D.
```

?- A.

Backtracking-ul va fi posibil pentru P, Q, R, dar de îndată ce ! este atins, toate soluțiile alternative pentru P, Q, R sunt suprimate. De asemenea, alternativa C :- V va fi suprimată.



- în arborele de scop de la ! până la părintele lui nu se mai aplică backtracking
- ! afectează doar execuția lui C

```
max(X,Y,X) :- X >= Y, !.
max(X,Y,Y).
```

! roșu

```
member(X, [X|L]) :- !.
member(X, [Y|L]) :- member(X, L).
```

Fie KB
```
p(1).
p(2) :- !.
p(3).
```

Ce răspunde PROLOG la următoarele întrebări:

?- p(X).
?- p(X), p(Y).
?- p(X), !, p(Y).

Negație ca eșec - predicatul fail este evaluat întotdeauna fals.

Lui John îi plac toate animalele, cu excepția șerpilor.

```
likes(john, X) :- snake(X), !, fail.
likes(john, X) :- animal(X).
```

Definim predicatul <u>not</u> unar astfel: not(G) execută dacă G este demonstrat; altfel not(G) este demonstrat.

```
not(G) :- G, !, fail.
not(G).
```

```
likes(john, X) :- animal(X), not(snake(X)).
```

Procedural, distingem două tipuri de situații "negative" în raport cu G:
- se poate rezolva scopul ¬G
- nu se poate rezolva scopul G - se poate întâmple când epuizăm toate opțiunile din KB încercând să arătăm că G advine

not în PROLOG nu corespunde exact negației matematice. Acest lucru se întâmplă deoarece atunci când procesează un scop not, PROLOG-ul nu încearcă să demonstreze direct scopul, ci încearcă să demonstreze opusul. Dacă opusul nu poate fi demonstrat, atunci PROLOG-ul presupune că scopul not este demonstrat.
Un astfel de raționament se bazează pe presupunerea "lumii închise". Adică, dacă ceva nu este în KB sau nu poate fi deivat din KB atunci nu este adevărat și în consecință negația este adevărată.

De exemplu, dacă introducem

?- not(human(mary)).

răspunsul va fi "yes" (dacă în KB nu avem human(mary)). Dar acest răspuns nu ar trebui înțeles ca "Mary nu este ființă umană" ci mai degrabă "Nu este suficientă informație în program pentru a dovedi că Mary este ființă umană".

În mod normal, noi nu presupunem "lumea închisă" - dacă nu spunem explicit human(mary), noi nu deducem implicit că Mary nu este ființă umană.

Alte exemple:

1) [composite(N) :- N > 1, not(primeNumber(N)).

   ușor de a dovedi că un număr este prim este suficient pentru a deduce că numărul este compus.

2) [good(renault).
   good(audi).
   expensive(audi).
   reasonable(Car) :- not(expensive(Car)).

   ?- good(X), reasonable(X).

   ?- reasonable(X), good(X).

Cut-ul este folosit și de multe ori necesar, însă trebuie utilizat cu atenție specială.