NEGAȚIA ÎN INTERPRETAREA ORIENTATĂ SPRE BAZA DE DATE A PROGRAMELOR LOGICE

Ing. Sabin Chiricescu

Institutul de Cercetări în Informatică

Rezumat

Se prezinta o etapa în proiectarea unei baze de date deductive și anume introducerea negației. În prima parte, facem o scurta prezentare a negației în PROLOG, urmată de prezentarea negației în DATALOG. Aici, unde negația apare în corpul regulilor, se prezintă o clasă de programe, programele stratificate, pentru care există un model unic. În a doua parte se prezintă o discutie despre influența negației din capul regulilor: efectele laterale de actualizare a bazei de date și împarțirea regulilor în clase de cunoștințe ce își moștenesc, anuleaza proprietăți - o îmbinare a programelor logice cu programarea orientata obiect.

Cuvinte cheie: PROLOG, DATALOG, negație, model.

Introducere

Se va prezenta o etapă în proiectarea unei baze de date deductive: introducerea negației. Discuția va fi asupra influenței negației în semantica programelor logice. Astfel:
Prin introducerea negației în limbajele logice de tip DATALOG se va mari puterea de expresie a limbajului, astfel încît programele, vor putea să exprime, diferența din calculul rațional, să utilizeze reguli implicite sau excepții de la acestea, să facă modificari în baza de date.
Datorita aparitiei negației în corpul regulilor, este influențată completitudinea limbajelor logice de tip PROLOG și unicitatea modelelor în programele DATALOG. Mai mult chiar, datorita aparitiei negației în capul regulilor, se modifică înseși programele DATALOG și, implicit, modelul lor.
În continuare discuția va avea două parți:

1. negația în corpul regulilor

Aici se prezintă cîteva aspecte din semantica negației în PROLOG, apoi vom introduce programele DATALOG stratificate. Evaluarea programelor logice stratificate, va fi completa din punct de vedere al modelului - rezultatului (evaluarea se va face într-o manieră bottom-up) și se va putea face prin translatarea către algebra relațională (operațiile acesteia putînd fi executate de către bazele de date clasice).

2. negația, atît în corpul regulilor, cît și în capul regulilor.

Se vor examina două direcții de extindere a programării logice:

- adăugarea de facilitați de actualizare (efecte laterale asupra bazei de date)
- îmbinarea programelor logice cu programarea orientată pe obiecte.

Și într-un caz și în altul, negația din capul regulilor este caracterizata de o acțiune de ștergere (a datelor sau a proprietaților anterioare).

1. Negația în corpul regulilor

Înainte de a discuta despre semantica negației în limbajele de tip DATALOG, vom încerca să evidențiem cîteva aspecte ale semanticii negației folosite în cel mai popular limbaj logic, PROLOG.

1.1. Negația în PROLOG

Majoritatea implementărilor PROLOG suferă de două importante imperfecțiuni: nu dispun de o implementare completă a negației și nu suportă cuantificări existențiali decît în set-predicate (septof, bagof...). În implementările clasice de PROLOG, negația este definită astfel:

```
not(P):- P,!,fail.
not(_).
```

Astfel, dacă scopul P nu este satisfăcut, negația succede, altfel eșuează. Semantica acestei negații este cunoscută ca "negația prin eșec" (negation as failure) [CLAR78].
Fie întrebarea:

```
? - not P
```

care se rescrie astfel

```
(p,!,fail);true
```

scopuri în a căror evaluare se poate întîlni:
a) p care succede (există o substituție de răspuns pentru p) și atunci not p eșuează;
b) p eșuează și atunci este implicit succesul lui not p.
Singurele răspunsuri la întrebare sînt yes(succes) sau no(eșec) fără ca în cazul succesului să fie enumerate instanțele variabilelor din p. Negația, dacă succede, lasă variabilele neinstanțiate.
Exemplu:
Fie sistemul de reguli:

```
bird(coco).
penguin(coco).
bird(fifi).
bird(lulu).
fly(X):-bird(X), not penguin(X).
```

La întrebarea

```
? fly(X).
```

răspunsul va fi

```
X=fifi;
X=lulu;
no
```

La întrebarea

```
? - not fly(X).
```

răspunsul va fi

```
no
```

cu toate că la întrebarea

```
? - not fly(coco)
```

răspunsul va fi

```
yes
```

deci, X=coco ar fi un răspuns corect la întrebarea anterioară.

Evaluarea negației în PROLOG

Din cele arătate mai sus, vom reține trei concluzii:

- implementarea negației nu ar fi posibilă fără existența predicatului de control al evaluării 'cut' (!), care va interzice revenirea și căutarea de alternative (eșecul lui not fly(X) din exemplul anterior este datorat unificării lui X cu 'fifi', deși și alternativa X=lulu conduce la eșec).
- implementarea negației este incompletă (X=coco este o consecință logică a programului și a scopului propus not fly(X)).
- dubla negație not not p nu este echivalentă cu p, în sensul că, dacă p succede, not not p succede și reciproc, dar variabilele care apar în p rămîn neinstanțiate după evaluarea lui not not p.

Exemplu.
În cazul reformulării regulii fly(X) astfel:

fly(X):-not penguin(X)

întrebarea ?-not fly(X) echivalează cu dubla negație not not penguin(X) care va lăsa variabila X neinstanțiată față de penguin(X), unde X instantiat la 'coco'. Totuși, această proprietate poate fi folosită în anumite programe pentru a testa unele condiții, fără ca variabilele ce apar în aceste condiții să fie instanțiate.
O altă concluzie importantă este aceea că, ordinea de evaluare este importantă. Acest lucru implică faptul că în acest caz proprietatea Church_Roser nu este satisfăcută (vezi exemplul următor).

Alte abordări ale mecanismului de evaluare a negației în PROLOG

După cum am observat, discuția s-a axat pe negarea formulelor ce conțin variabile, deoarece apare o tendință de generalizare a negației în momentul în care variabilele nu sînt instanțiate(ground). Apare, astfel, naturală încercarea de a amîna evaluarea negației pînă în momentul în care variabilele vor fi cunoscute.
În PROLOG și MU-PROLOG există posibilitatea 'înghețării' unei formule-scop pînă cînd variabilele din aceasta vor fi instanțiate, dar momentul 'dezghețării' depinde de program, el trebuind să instanțieze variabilele 'înghețate'.
Exemplu:
Fie programul

```
bird(coco).
bird(fifi).
bird(lulu).
penguin(coco).
fly(X): - not penguin(X),bird(X).
```

În implementările clasice, la întrebarea ? - fly(X) răspunsul va fi no datorită lui penguin(coco), care va anula celelalte alternative (a fost schimbată ordinea de evaluare, astfel încît să apară în negație o variabilă neinstanțată).

Dacă se amîna evaluarea negației pînă cînd variabilele vor fi instanțiate, răspunsul la întreabarea ? - fly(X) este identic cu cel de exemplul clasic X=fifi și X=lulu, dar autorul succesului lui not fly(X) (X=coco) va rămîne în continuare necunoscut. Și această implementare a negației este incompletă.

1.2. Programe stratificate

După cum am văzut, în PROLOG formula negată poate conține variabile ce sînt neinstanțiate în momentul evaluării; ele, fiind nelegate, pot să se unifice cu orice obiect, deși scopurile ulterioare vor selecta doar anumite obiecte.
Apare naturală restricția ca un obiect să fie cunoscut înainte de a participa într-o formulă negată(evaluarea formulei negate să fie sigură). Această restricție poate fi respectată, fie introducînd o constrîngere sintactică asupra programelor, fie modificînd mecanismul de evaluare, astfel încît să poată să înghețe anumite scopuri pînă în momentul instanțierii variabilelor nelegate la momentul înghețării scopului.
Restricția sintactică de care ne vom ocupa în continuare, impune o relație de ordine între regulile din program, astfel încît programul să fie descompus în straturi ce se pot evalua succesiv, fără ca evaluarea curentă să modifice ceva din evaluările anterioare. Într-un capitol ulterior vom vedea că ideea stratificării se poate folosi și în definirea semanticii altor operații specifice bazelor de date.
În cele ce urmează, ne vom restrînge la prezentarea unui limbaj simplificat (fără simboluri funcționale și predicate predefinite) ce admite negația, concentrîndu-ne atenția pe aspectele declarative și operaționale ale acesteia.

Sintaxa și semantica programelor stratificate

Principalele notații și concepte folosite în discuția noastră:

- variabilele sînt asociate (încep) cu literele mari: ex. X, Y, Z;
- constantele sînt asociate (încep) cu literele mici: ex. a, b, c;
- termenii sînt, fie constante, fie variabile;
- atomii (predicatele) sînt formulele de forma p(t1,....,tn), unde p este un simbol predicativ și ti sînt termeni.

Simbolurile predicative sînt constante.
Literalii sînt: atomi pozitivi p(t1,....,tn) sau
atomi negativi not p(t1,....,tn);
Regulile (clauzele) sînt formule de forma

cap ← corp

unde cap(head) este un atom pozitiv și corp(body) este o conjuncție de literali separați prin virgulă.
O regulă fără corp este un fapt (fact).
Un program este o colecție de reguli.
Pentru simplificarea discuției, introducem următoarele

clase de programe:

- programe pozitive în care literalii din capul și corpul regulilor sînt atomi pozitivi;
- programe negative în care corpul regulilor pot să conțină orice fel de literal.

Unui P, un program definit ca mai sus, îi vom asocia următoarele concepte:

- universul Herbrand al lui P, notat cu Hp, este mulțimea tuturor constantelor ce apar în program;
- baza Herbrand a lui P, notată cu Bp, este mulțimea tuturor formulelor atomice de bază (ground atomic formula) de forma p(t1...tn), unde p face parte din mulțimea simbolurilor predicative ale lui P și ti aparține lui Hp;
- o substituție σ pentru o regulă r din P este o funcție din mulțimea de variabile în Hp, care în regula r înlocuiește fiecare variabilă X cu o(X). Astfel, oricare atom A din r este înlocuit cu atomul oA ∈ Bp;
- o interpretare I pentru P este orice submulțime din Bp în care:
- orice formulă atomica A este adevărată, dacă există o substituție o pentru care A aparține lui I;
- orice formulă atomică not A este adevărată, dacă A este falsă; (vom reveni cu explicații asupra contextului de evaluare)
- orice regulă A ← B_1,...,B_n este adevărată în I, dacă există o substituție o, astfel încît A să fie adevărat sau i=1..n a.i. oBi este fals; altfel, regula este falsă (dacă există o substituție S a.i. oricare i=1..n oBi este adevărat și oA este fals).
- orice interpretare I a lui P pentru care toate regulile din P sînt adevărate se numește model Herbrand al lui P. Un model al lui P este chiar Bp;
- un model minimal este acel model a lui P în care nici o submulțime a sa nu mai este model P.

Să încercăm să definim răspunsul pe care îl așteptăm de la program.

Fie MM(P)={M_1,...,M_n} mulțimea tuturor modelelor pentru programul P. Dacă LM = intersecție (după i) din Mi este un model, atunci acesta este înțelesul natural (răspunsul) al programului P. Dacă LM este numit cel mai mic model al programului. Această proprietate a programelor este cunoscută drept proprietatea de intersecție a modelelor (model intersection property).

Atomii celui mai mic model (least model) LM sînt o consecință sigura a formulelor din P.

Exemplu:

Programul P={p(X),←q(X); q(X) ← r(X); r(1); p(2); p(3)←t(X)} admite modelele

M1={r(1), p(2), q(1), p(1); q(2)} și

M2={r(1), p(2), q(1), p(1) } care este și modelul minimal.

M3={r(1), p(2), q(1), p(1), p(3), q(2)}

Oare exista întotdeauna cel mai mic model? În continuare vom prezenta semantica programelor pozitive, care admit cel mai mic model, și vom discuta despre cel mai mic model în programele negative.

Semantica programelor pozitive

Emden și Kowalski [1] și alții au arătat că programele pozitive bine definite au proprietatea de intersecție a modelelor. În consecință, orice program pozitiv bine definit are cel mai mic model.

Un program pozitiv bine definit este o mulțime închisă de formule a.i.; pentru fiecare atom din corpul regulilor există o regulă care să-l definească.

Orice întrebare Q dintr-un program pozitiv P este adevărată, dacă și numai dacă Q aparține lui LM (cel mai mic model a lui P).

În continuare, vom descrie metoda procedurală de construire a celui mai mic model.

Fie P un program. Considerăm secvența de evaluare definită astfel:

T_o este mulțimea faptelor

T_{i+1}={TA astfel încît, dacă există o regulă A ← B_1... B_p din P și o substituție T astfel încît

$TB_i \in T_i \; \forall \; i \in 1.....p\}$ U T_i

Intuitiv T_{i+1} conține toate faptele ce sînt consecințe logice ale regulilor din P .Această secvență mărginită de B_p este monotonă, iar limita (punctul fix) există și coincide cu modelul cel mai mic a lui P. Practic, LM este obținut în timpul evaluării, în momentul în care $T_n=T_{n+1}$ (LM=T_n)

Exemplu.

Fie programul P

p(X)←q(X)
q(X)←(X)
și faptele {r(1); p(2);}

Secvența de evaluare a lui P este:

To={r(1), p(2)}
T1={r(1), p(2), q(1)}
T2={r(1), p(2), q(1), p(1)} și $T_3=T_2$

cel mai mic model LM este T2.

Fie programul N

r1. p(1)←not q(1);
r2. q(X)←r (X);
r3. r(1);

Secvența de evaluare este

T_0={r(1)}
T_1= {r(1); p(1); q(1)}
T_2= {r(1); q(1)} cel mai mic model

În evaluarea programelor negative secvența de evaluare poate fi nemonotonă, nefiind garantată limita sa (punctul fix). În programul N, $T_2 \subset T_1$, deoarece regula ri este adevărată cînd not q(1) este fals, iar q(1) este adevărat deoarece aparține secvenței de interpretare T1.

O posibilitate de a păstra o evaluare monotonă este de a introduce o restricție asupra programelor (numai anumite programe vor fi acceptate pentru evaluare), astfel încît, în momentul în care apare un atom negat not A(...), să fie cunoscută deja mulțimea de formule

atomice de forma A(...)∈ Bp, deduse din regulile programului. Practic, această mulțime nu mai poate fi modificată de evaluările ulterioare.

Semantica programelor negative

Ipoteza lumii închise CWA (the Closed World Assumption) [7] extinde puterea limbajelor logice, creind o bază pentru definirea semanticii negației. Semantica negației prin eșec este inclusă în semantica CWA.

Definiția sintactică a ipotezei lumii închise:
Fie P un program pozitiv și fie LM cel mai mic model; dacă Q este o întrebare, atunci presupunem că not Q este adevărat, dacă și numai dacă, Q nu aparține lui LM. Dacă not (P⊢ Q), atunci este adevărat not Q (relație ce nu trebuie ținută în baza de date, ea putînd fi demonstrată).
Ipoteza lumii închise CWA nu permite definirea semanticii complete pentru negație (full negation):
"Orice element din Hb este fals, atît timp cît nu a fost dovedit adevărul despre el (nu există o regulă a cărei consecință logică să fie)."
Naqvi a demonstrat că există o clasă de programe (programele stratificate) bazate pe logica de ordinul întîi (sau incluse în logica de ordinul întîi), unde, atît ipoteza lumii închise [7], cît și proprietatea intersecției modelelor (Model Intersection Property) [1], rămîn adevărate. Aceste programe au un cel mai mic model.
Dacă pentru a implementa "negația prin eșec" în Prolog este nevoie de un predicat extra-logic (predicatul de control a execuției ! 'cut'), în descrierea noastră "negația completă" într-o lume închisă are o semantică pur logică.

Programe admisibile și stratificarea acestora

Fie P un program definit conform sintaxei prezentate mai sus. Vom defini relațiile de derivare → și -*→ între atomul pozitiv p și literalul p' astfel:

p → p' (p derivă p' sau p' derivă din p) dacă există o regulă p← ..., p',..., în P.

p *→ p' închiderea tranzitivă a relației de derivare p → p'.

Spunem că există o dependență mutuală între p și p' dacă există relațiile p – *> p' și p' – *> p.
Definim relațiile de ordine > și ≥ între simbolurile predicative p și p' astfel:

p ≥ p' dacă există o regulă p← ..., p',... în P

p > p' dacă există o regulă p ← ...; not p' ,... în P

Observație.
Se poate construi un graf orientat, asociat relațiilor de ordine definite anterior. Astfel:

- pentru o relație p≥p' se construiește un arc p p'
- pentru o relație p>p' se construiește un arc marcat p p'.

Aceste relații au următoarele semnificații:

≥ coincide cu "putem deduce p în același timp cu q"

> coincide cu "deducem p după ce am dedus q"

Definiție:
Un program este admisibil [6] dacă nu există o derivare p *→ not p.
Cu alte cuvinte, un program este admisibil dacă nu conține un ciclu care traversează un arc marcat.
Exemplu:
Fie programul

P1={ p ←not q; q←p,q }

Relațiile de derivare sînt:

p – *> not q și q – *>p, de unde deducem q – *not q

Relațiile de ordine sînt:

p>q și q≥p

Se observă ca P1 nu este un program admisibil q ≥ p>q.
Orice program admisibil poate fi descompus într-o succesiune de părți disjuncte Lo ... L_m, în care toate simbolurile predicative p și q din P, aflate într-o relație de ordine, se regăsesc astfel:

a) dacă p ≥q, atunci p aparține lui Li și q aparține lui Lj cu i ≥ j;
b) dacă p > q, atunci p aparține lui Li și q aparține lui Lj cu i > j.

În partiția Lo intră, în general, fapte aflate în baza de date și regulile pozitive, (al căror corp nu conțin atomi negativi).
Exemplu:
Programul P2={r(a); s(b); q(X)← r(X);
p(X)← not q(X),s(X);}
are relațiile de ordine:

p > q ≥ r și p ≥ s

și partițiile (fiind stratificabil)

L_0={r,q,s} și L_1 = {p}

Evaluarea programelor stratificate

Fără a intra în detalii se observă că modelul minimal se obține evaluînd întîi partiția L_0 (avînd ca model M), apoi partiția L1(M1) și așa mai departe, ajungîndu-se la ultima partiție Ln (Mn). Evaluarea este monotonă, conducînd la cel mai mic model. Datorită stratificării, în momentul evaluării lui not Q cunoaștem deja relațiile, pentru q neexistînd posibilitatea de revenire și modificare a acestora.
Exemplu:
Fie programul P alcătuit din:
relațiile în baza de date (fapte):

1. child (a,b)
2. child (b,c)
3. child (c,d)
4. child (a,z)
5. child (a,f)
6. child (f,c)

și regulile:

7. descendent (X,Y) ← child (X,Y)
8. descendent (X,Y) ← child (X,Y), descendent (X,Y).
9. sp-desc (X,Y) ← descendent (X,Y), not descendent (b,Y).

Stratificarea lui P este compusă din partițiile L_0={1...8} și L_1={9}.

Evaluarea lui P, echivalentă în operațiile specifice bazelor de date se face în modul următor:

- pentru partiția L_0, relația 'descendent' este închiderea tranzitivă a relației 'child';
- pentru partiția L1 vom construi relația 'sp-desc' astfel: pentru not descendent vom selecta întîi toate tupelele care au în prima coloană b; vom face o proiecție pe a doua coloană, obținînd toate elementele Y ce sînt în descendent (b,Y); apoi, vom elimina din descendent toate tuplele care au în a doua coloană elemente Y, selectate anterior. Vom face un join între relația descendent (X,Y) și relația obținută anterior.

Dacă notăm relația descendent cu D(X,Y) vom obține relația sp_desc astfel:

$$D(x,y)* \left(\overline{D}(x,y) - T_{y \in \pi_y \left(T_{x=b}(D(x,y))\right)}^{(D(x,y))}\right)$$

2. Efectul negației în capul regulilor

Există două direcții ale programării logice în care regulile cu cap negativ au un rol important:

- extinderea programării logice cu facilități de actualizare a bazei de date: astfel, fiecare regulă r ce redefinește un fapt A, este dublată de o acțiune de inserare (cînd H(r)=A) sau ștergere (cînd H(r)=not A);
- îmbinarea programării logice cu programarea pe obiecte: obiectele reprezintă clase de cunoștințe, între ele apărînd o relație de rafinare ce va conduce la o ierarhie de obiecte. Regulile se vor moșteni de la clasa mai generală către clasa inferioară (mai rafinată din punctul de vedere al cunoștințelor), regulile locale putînd să ascundă reguli globale (să fie excepții de la acestea).

Pentru prima direcție vom prezenta problemele care apar, iar pentru cea de a doua, vom arăta modificarea semanticii programelor logice, ordonate după clase de cunoștințe.

Sintaxa programelor se modifică în definirea regulilor astfel:

- regulile sînt formule de tipul cap ←corp, unde 'cap' este un literal (ce poate fi negativ sau pozitiv) și 'corp' este o conjuncție de literali separați prin virgulă.

Fie forma generală a regulilor A←B1 ... Bn.

În scopul simplificării prezentării vom introduce următoarele clase de programe:

- programe pozitive, în care A, Bi sînt atomi pozitivi;
- programe seminegative, în care A este atom pozitiv și Bi sînt literali;
- programe negative în care A, Bi sînt literali.

2.1. Extinderea programării logice cu facilități de actualizare

O interpretare a programelor este posibilă făcînd o asemănare cu regulile de producție. Astfel, fiecare regulă ce redefinește un fapt este asemănătoare unei reguli de producție de tip acțiune←condiție.

Fie p un fapt, atunci:

- oricare regulă de forma
  p(...) ← C

este dublată de regula de producție

  if C then insert(p(...))

- oricare regulă
  not p(...) ← C, este dublată de producția
  if C then delete(p(...)).

În cele ce urmează vom studia programele seminegative, care admit cel mai mic model (au o interpretare unică).

Exemplu:

Fie P1 un program cu regulile:

r1. p(X)← q(X);
r2. q(X) ←r(X); și faptele
r(1), r(2), p(2)

Dacă T_0={(r(1), r(2), p(2)} este mulțimea de fapte inițiale (tuple în baza de date) și LM={(r(1),r(2),p(2),q(1),q(2),p(1)}, este naturală interpretarea următoare:

dacă LM conține noile fapte ce au fost deduse, aceste fapte ({p(1)}) trebuie adăugate la baza de date.

Apare tendința de a folosi r1 în două scopuri:

- 'creștere' a relației p prin adăugarea de noi fapte deduse în momentul evaluării;
- adăugarea la baza de date de noi tuple, acelea care sînt consecințele logice ale programului. Apare un efect lateral de pseudo-modificare a programului prin actualizarea bazei de fapte, dar care nu ne deranjează deoarece este prinsă în model.

Exemplu:

Fie P2 programul negativ

r1. not p(X)← q(X);
r2. q(X)←r(X);

și faptele

r(2), p(2)

Atunci secvența de evaluare a lui P este:

T_0={r(2), p(2)} T1={r(2), p(2), q(2)}
1. din r1 rezultă $T_2=T_1$ - {p(2)}
2. datorită faptului p(2), rezultă
$T_3=T_2 \cup \{p(2)\}$;
3. se reia 1.

Înțelesul dorit al lui P este eliminarea din baza de date a lui p(2). Ne trebuie însă o ordine de evaluare a regulilor, astfel încît, o dată cu ștergerea tuplelor din baza de date, ele să fie eliminate și din model. Modelul lui P2 ar fi T_2={r(2),q(2)}.

Observații:

- secvența de evaluare nu mai este monotonă;
- ordinea de evaluare este importantă, deoarece apar contradicții între regulile cu cap pozitiv și cele negative;

Ordonarea evaluării în programele negative

Contradicțiile dintre regulile cu cap negativ și cele cu cap pozitiv pot fi eliminate folosind următoarea metaregulă asemănătoare stratificării:

"o regulă de ștergere este evaluată după ce au fost

evaluate toate regulile ce pot produce tuple."

Observație.

Dacă pentru programe seminegative, ce admit cel mai mic model, actualizarea este cuprins în construirea celui mai mic model, pentru programe negative vom accepta ca model cel care este obținut, atît după ștergere în program, cît și în model.
Făcînd analogia cu programele stratificate în care construcția relației 'p' nu se putea face simultan cu folosirea lui not p, putem să impunem o restricție programelor ce vor fi evaluate într-o ordine implicită, asemănătoare stratificării. Astfel:

- relațiile de ordine între simbolurile predicative din capul regulilor sînt:
  p ≥ p', dacă exista o regulă p ←..., p',... în P sau not p ← ..., p'...,
  p > p', dacă există o regulă p ← ..., not p',... în P sau not p ..., not p',...;
- relațiile de derivare se păstrează, numai ca acum, atît p, cît q sînt literali (atomi pozitivi sau negativi);
- un program este admisibil dacă nici p –*> not p și nici not p*→p cu p orice simbol predicativ din P.

Vom construi partițiile astfel:
a) dacă p > =q, atunci p în Li, q în Lj și not q în Lk cu i>=max(j,k);
b) dacă p > q, atunci p în Li, q în Lj și not q în Lk cu i<max(j,k);
c) not p în Li și p în Lj cu i>j.

Exemplu:
Fie P3 un program care conține faptele { p(2),r(2)} și regulile

r1. not p(X) ← q(X)
r2. q(X)← r(X)
r3. t(X) ← p(X)

Efectul lateral al lui r1 este de ștergere a lui p(2) din BD.

Efectul negației în regulile cu mai multe capete

Vom încerca prin cîteva exemple să arătăm că nu este suficientă această metaregulă, de aceea trebuie lăsată programatorului posibilitatea de specificare a ordinii de evaluare a regulilor (metaregula de stratificare a programelor).
Următoarele exemple, pe care le considerăm sugestive, nu vor respecta sintaxa acceptată în discuție, dar ele pun în evidență unele situații limită.
Fie programul P4, alcătuit din regula:
r. ancestor (X,Z), not ancestor (Z,X) ←
parent (X,Y), ancestor (Y,Z)
P4 este constituit dintr-o regulă și de relațiile 'ancestor' și 'parent' aflate în BD. Regula dorește să mențină consistentă relația ancestor din BD.

Fie BD avînd următoarele tuple
{(1,2),a(2,1),p(1,2),p(2,3),p(3,4),a(4,3)}
Să remarcăm că:

- din punctul de vedere a lui r, BD este inconsistentă (a(1,2)/\ a(2,1)← ... este fals).
- dacă facem o evaluare (fără a elimina inconsistența):
  a(4,3) este eliminată, iar a(1,1), (implicat datorită lui a(1,2), este inserat în BD.

Metaregula de ordonare în evaluare ar fi corectă dacă am fi siguri că BD este consistentă sau dacă toate faptele de tip ancestor pot fi redemonstrate. Altfel, trebuie să se înceapă cu verificarea consistenței.
Exemplu:
Fie P5 compus dintr-o relație ce descrie un circuit electric și regula de transformare paralelă:

r. not circuit(X,Y,R1), not circuit(X,Y,R2),
circuit(X,Y,R12) ← circuit (X,Y,R1),
circuit(X,Y,R2), R12 = R1*R2/(R1+R2).

Pentru a putea să obținem un rezultat corect este necesar ca, după fiecare transformare, să eliminăm din baza de date vechile tuple și să le adăugăm e cele noi. Ordinea implicită de evaluare va duce la echivalări infinite între R1 și R2 sau R1 și R12 sau R12 și R1.
În consecință, trebuie lăsată utilizatorului posibilitatea de a alege ordinea de evaluare.

2.2. Negația și programarea logică ordonată

Cea de a doua tendință în interpretarea programelor negative este apropierea de paradigma programării pe obiecte.
Un program va fi alcătuit dintr-o mulțime de module (obiecte), fiecare alcătuit dintr-o mulțime de reguli (posibil cu cap negativ) și dintr-o mulțime de relații între module.
Fiecare obiect poate reprezenta o clasă de cunoștințe.
Între obiecte pot să existe următoarele relații:

- o relație de rafinare notată cu <. Astfel, între obiectele O1 și O2, aflate în relația O1 < O2, obiectul O1 este o rafinare a conceptelor din O2. Obiectul O1 moștenește regulile din O2, dar el poate să-și impună propriile reguli locale prin ascunderea (suprapunerea) peste regulile globale introduse de O2;
- o relație de indiferență notată cu <> (implicită între obiecte); nu avem nici O1 < O2 nici O2 > O1. În timpul evaluării, regulile din modulele O1 și O2, aflate în relația O1<>O2 sînt pe același nivel, ele putîndu-se completa sau anula (datorită contrazicerii).

În timpul evaluării, anumite reguli pot să fie blocate datorită faptelor deduse pînă acum (regula să fie adevărată din punct de vedere logic). Dacă regula r este de forma A ← ... B .... și not B este adevărat, atunci regula r este blocată (formula A V... not B este adevărată).
Exemplu:

Fie următoarele componente
bird={bird(coco); bird(fifi); bird(lulu);
fly(X) ← bird(X); }
penguin = { penguin(coco);
not fly(X) ← penguin(X); }
query={ground_animal(X) ← bird(X), not fly(X); }
a)
Fie P1 = <{bird, penguin}, {penguin <bird }> .
Datorită regulilor fly(X) și not fly(X) apare o contradicție între fly(coco) și not fly(coco). Contradicția se rezolvă prin surapunere: acceptarea consecinței not fly(coco) din modulul 'penguin' va ascunde regula fly(coco) dedusă în modulul 'bird'. Această interpretare este naturală considerînd conceptul 'penguin' ca pe o rafinare a conceptului 'bird', not fly(coco) fiind o excepție a regulii fly(X)← bird(X).
Modelul lui P1 este:

{bird(coco), bird(fifi), bird(lulu), penguin(coco), fly(fifi), fly(lulu), not fly(coco)}

b)
Fie P2 = <{bird, penguin}, {penguin >bird}>.
Aceeași contradicție va fi rezolvată prin eliminarea consecințelor ambelor reguli fly(coco) și not fly(coco), ea trebuind să fie interpretată astfel: desre 'coco' nu deducem, nici că are proprietate 'fly', nici că nu o are ('not fly' adevărat). Modelul lui P2 este:

{bird(coco), bird(fifi), bird(lulu), penguin(coco), fly(fifi), fly(lulu)}

c)
Fie P3 = {(bird, penguin, query}, {query < penguin <bird}.
Regula ground_animal (X) este blocată pentru X=fifi; datorită lui fly(fifi), nu vom putea deduce ground_animal(fifi). Modelul lui P3 este:

{bird(coco), bird(fifi), bird(lulu), penguin(coco), fly(fifi), fly(lulu), not fly(coco), ground_animal(coco)}

Sintaxa și semantica programelor ordonate

Sintactic s-au modificat regulile ce pot avea acum cap negativ și programele pentru care trebuie să specificăm mulțimea de componente și de relații între acestea (implicit modulele sînt indiferente - același nivel de evaluare). Componentele sînt mulțimi de reguli (programele anterioare).
Vom relua prezentarea semanticii. Unui program P îi vom asocia următoarele concepte:

- universul Herbrand - Hp este mulțimea constantelor din P;
- baza Herbrand Bp este mulțimea formulelor de bază p(t1,...,tn) cu p aparținînd mulțimii de simboluri predicative a lui P și ti aparținînd lui Hp;
- o substituție de bază S pentru o regulă r înlocuiește fiecare apariție de variabilă din r cu o constantă din Hp;

Fie C o componentă, notăm cu C* mulțimea {r : r aparține lui Cj cu C < Cj sau C <> Cj};

- ground (C)este mulțimea instantierii tuturor regulilor din C; ground (C*) este mulțimea instantierilor din C*;
- o interpretare I pentru P este orice submulțime din Bp U not Bp; atunci ~I reprezintă mulțimea de predicate {A : A din Bp pentru care nici A și nici not A nu apațin lui I};
- fiind data o interpretare I pentru P o regulă r: A ← B1...Bn din ground (C), unde C este componenta în care este definită r, poate fi:
- aplicabilă, dacă Bi aparține lui I;
- aplicată, dacă A,Bi aparține lui I;
- blocată, dacă există i a.i not Bi apaține lui I;
- suprapunere, dacă există o regulă neblocată r' aparținînd modulului Cj cu C <...<C_j și H(r')=not H(r) (r' de forma not A← ...); regula r' este suprapusă de r;
- anulată, dacă există o regulă neblocată r' aparținînd modulului Cj cu C <> Cj și H(r') = not H(r); și regula r' este anulată;

un model M a lui P este orice interpretare a lui P, care:

- pentru fiecare fapt A din M, oricare regulă r din P cu H(r)=not A este blocată sau suprapusă. Prin blocarea regulii r nu se permite atribuirea nici unei valori unui element nedefinit (regula este adevărată datorită echivalenței logice a implicației). Regula r fiind suprapusă înseamnă că există o regulă ce o rafinează;
- pentru fiecare A din ~M (mulțimea de predicate despre care nu știm, nici că sînt adevarate, nici că sînt false) oricare regula r din P cu H(r)=A sau H(r)=not A este exclusă.

Observație
Baza Herbrand nu mai este un model ca în cazul programelor seminegative.

Exemplu:
Fie P un program alcătuit dintr-o singură componentă C. C are regulile:

r1. a ← b
r2. not a ← b

admite ca modele:

{}
{b} regula r1 anulează pe r2 și reciproc
{not b} atît r1 cît și r2 sînt anulate
{a, not b} și {not a,not b}

în timp ce baza Herbrand {a,b} nu este model (regula r2 se contrazice cu r1).

Definiție:

Fie P un program și fie I familia interpretărilor lui P. Operatorul de transformare imediată pentru P este funcția Vp(I) definită astfel:

Vp(I)={A | a.i. există o regulă r: A←B1...Bn în ground(P) cu Bi în I și care nu este, nici

suprapusă, nici anulată}.

Laenens [4] a demonstrat că operatorul VP este monoton și admite un cel mai mic punct fix. Limita VP (0) la infinit este cel mai mic model a lui P.

Secvența de evaluare este:

$T_0 = \{\}$

T_n = {A : a.i există o regulă r:A←B1,..Bn în ground(P) cu Bi în Tn-1 și care nu este nici suprapusă nici anulată} cînd Tn=Tn+1 cel mai mic model va fi LM=Tn.

Exemplu:

Fie P1 alcătuit dintr-o componentă not b; b; a← b.

Secvența de evaluare este: T0={} T1={} deoarece nu se pot introduce, nici b, nici not b în T1, deoarece se anulează reciproc.

Fie P2 alcătuit din componentele

C0 < C1

C1: not b

C0: b; a← b.

Secvența de evaluare este T0={}. Dacă se introduce not b în T1, este anulat de b din C1. Altfel, dacă se introduce b în T1, T1={}, deoarece b suprapune not b (este declarată pe un nivel inferior), T2={b,a} care este și cel mai mic model.

Fie P3 alcătuit din componentele

C0<C1

C1: not a, not b;

C0: a ← b; b ← a;

Secvența de evaluare este T0={} și T1={}deoarece not a și not b sînt suprapuse.

Fie P4 alcătuit dintr-o singură componentă C care conține

faptele F={t(a),t(b),not s(a)}

regulile

r1. not p(X) ← t(X)

r.2 q(X) ← t(X)

r3. p(X) ← not q(X), s(X), not q(a)

r4. not q(X) ← p(X)

Se compune ușor T0=F. Apoi se deduce {not p(a), not p(b), q(a), q(b)} din r1 și r2, dar singurul fapt care se poate reține este not p(a): (regula r3 p(b) exclude not p(b), dar aceeași regulă pentru p(a) este blocată de not s(a); q(a) și q(b) sînt excluse de r4), deci T1={not p(a) U F. La pasul următor se deduce q(a), deoarece r4 este blocată de not p(a), deci T2={not p(a), q(a). Abia la pasul următor deducem p(b), care nu poate fi exclus de r3 blocată de q(a) și q(b), r4 fiind blocată de not p(b).

Modelul este:

LM=T3={not p(a), q(a), not p(b), q(b)} U F.

Concluzii

Am arătat că există o clasă de programe, programele stratificate pentru care, atît corpul, cît și capul lor pot să conțină atomi negativi. Dar recursivitatea mutuală nu poate traversa negația. Datorită simbolurilor funcționale și mulțimilor, această clasă de programe nu este strict determinabilă sintatic (în timpul analizei sintactice-semantice), deoarece există programe care admit cel mai mic model fără ca programul să fie stratificabil. Deci, noi nu putem oferi utilizatorului decît informația ca programul este stratificabil sau nu, lăsînd pe seama evaluatorului găsirea unei eventuale soluții.

Soluția programării logice ordonate este elegantă, dar este relativ dificilă de implementat deoarece trebuie menținute informații despre toate scopurile blocate sau suprapuse, care se pot reactiva (se aseamănă cu înghețarea din PROLOG).

BIBLIOGRAFIE

1. van EMDEM, M.H., KOWALSKI, R.: The Semantics of Predicate Logic as a Programming Language. În: Journal of ACM, vol.23, no.4, 1976, pp.733-742.
2. LLOYD, J.W.: Foundations of Logic Programming, Springer Verlag, Berlin, 1987, pp.65-131.
3. LAENENS, E., VERMEIER, D.: A Fixpoint Semantics for Ordered Logic. Technical Report 89-27, University of Antwerp UIA, iunie, 1989.
4. LAENENS, E., SACCA, D., VERMEIER, D.: Extending Logic Programming. În: Proceedings of ACM SIGMOD, 1990, pp.178-210.
5. MINKER, J. (ed.) Foundations of Deductive Databases and Logic Programming, Morgan Kaufman, Los Altos, 1987, pp.53-141.
6. NAQVI, S.A.: A Logic for Negation in Database Systems. În: Foundations of Deductive Databases and Logic Programming, Morgan Kaufman, Los Altos, 1987, pp.101-126.
7. REITER, R.: On Closed World Databases. În: Logic and Databases, Plenum Press, Net York, 1978, pp.55-76.