Logică și structuri discrete

Metoda rezoluției. Aplicații ale logicii predicatelor

Marius Minea
marius@cs.upt.ro

http://www.cs.upt.ro/~marius/curs/lsd/

18 noiembrie 2013

Metoda rezoluției (în calculul propozițional)

O formulă e o tautologie $\leftrightarrow$ negația ei nu e realizabilă

O formulă poate avea multe propoziții / predicate
dorim să *reducem numărul lor*, simplificăm formula,
dar *păstrând realizabilitatea* ei

Reprezentăm ca de obicei clauzele ca *mulțimi* de *literali*.
Fiind dat un literal l și două clauze C_1, C_2 cu $l \in C_1, (\neg l) \in C_2$,
rezolventul lui C_1 și C_2 în raport cu literalul l e clauza

$$\text{rez}_l(C_1, C_2) = (C_1 \setminus \{l\}) \cup (C_2 \setminus \{\neg l\})$$

Exemplu: $\text{rez}_p(\{p, q, r\}, \{\neg p, s\}) = \{q, r, s\}$.

Propoziție: $C_1, C_2 \models \text{rez}_l(C_1, C_2)$.

Corolar: dacă $C_1 \wedge C_2$ e realizabilă, atunci $\text{rez}_l(C_1, C_2)$ e realizabilă.
se vede ușor pe cazuri, după valoarea literalului eliminat

Cum folosim rezoluția (în calculul propozițional)

Modus ponens poate fi privit ca un *caz particular de rezoluție*:

$$\frac{p \vee \mathit{false} \qquad \neg p \vee q}{q \vee \mathit{false}}$$

Determinăm dacă o formulă e o tautologie construind negația ei (în CNF) și determinând dacă e realizabilă.
Adăugăm rezolvenți, încercând să *obținem clauza vidă:*

alegem un literal l, adăugăm toți rezolvenții
  dacă avem m clauze cu l și n clauze cu $\neg l$, creem $m \cdot n$ rezolvenți
  putem șterge cele $m + n$ clauze inițiale

dacă vreun rezolvent e clauza vidă, formula e nerealizabilă
altfel alegem alt literal
dacă nu mai putem crea rezolvenți, formula e realizabilă

În logica predicatelor, trebuie o noțiune *mai generală* de rezolvent

De ce substituția și unificarea de termeni ?

Avem două formule în care un predicat apare pozitiv și negativ:
$\forall x.\forall y.P(x, g(y))$ și $\forall z.\neg P(z, a)$.
sau
$\forall x.\forall y.P(x, g(y))$ și $\forall z.\neg P(a, z)$
Se contrazic ?

Putem *substitui* o variabilă cuantificată universal cu *orice* termen
$\Rightarrow$ în al doilea caz, putem substitui $x \mapsto a$, $z \mapsto g(y)$
$\Rightarrow$ obținem $P(a, g(y))$ și $\neg P(a, g(y))$, *contradicție*

În primul caz, nu putem face la fel. Putem defini precis acest lucru?

Substituții de termeni

O *substituție* e o funcție care asociază unor variabile niște termeni:

$$\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$$

De exemplu $f(x, g(y, z), a, t)\{x \mapsto g(y), y \mapsto f(b), t \mapsto u\}$
$= f(g(y), g(f(b), z), a, u)$

Obs: uneori se folosește notația x_i/t_i, alteori invers, t_i/x_i

Substituția σ pe termenul T se notează uzual postfix: $T\sigma$

Compunerea a două substituții e o substituție

Unificare de termeni

Doi termeni t_1 și t_2 se pot *unifica* dacă există o substituție σ care îi face egali: $t_1\sigma = t_2\sigma$.
O astfel de substituție se numește *unificator*.

Exemplu: $f(x, g(y))\{x \mapsto a\} = f(a, g(y)) = f(a, z)\{z \mapsto g(y)\}$
deci substituția $\{x \mapsto a, z \mapsto g(y)\}$ e un *unificator* .

Mai general: avem o mulțime de ecuații (perechi de termeni)
$\{l_1 = r_1, l_2 = r_2, \ldots, l_n = r_n\}$ (numită și *problemă de unificare*).
Un unificator (o soluție a problemei) e o substituție σ astfel încât $l_i\sigma = r_i\sigma$ pentru orice i.

Cel mai general unificator (most general unifier) este acela pentru care orice alt unificator poate fi obținut aplicând încă o substituție.

Relevant pentru rezoluție: având $P(l_1, l_2, \ldots l_n)$ și $\neg P(r_1, r_2, \ldots r_n)$ dacă găsim un *unificator*, avem o *contradicție*.

Reguli de unificare

O variabilă x poate fi unificată cu orice termen t
dacă x *nu apare* în t (nu: x cu $f(g(y), h(x, z))$
(pentru că altfel, substituția ar duce la un termen infinit)
Două constante pot fi unificate doar dacă sunt identice

Doi termeni funcționali pot fi unificați doar dacă au funcții identice, și termenii argument corespunzători pot fi unificați

$\Rightarrow$ cu aceste reguli, se poate scrie un algoritm recursiv de unificare care determină *cel mai general unificator*
(orice alt unificator se poate obține din el printr-o altă substituție)

Union-Find

Structură de date / algoritm pentru mulțimi cu clase de echivalență.

Operații:
find(element): găsește reprezentantul clasei de echivalență
union(elem1, elem2): declară elementele ca fiind echivalente
(și rămân așa mai departe)

Implementare: pădure de arbori cu legături de la fiu la părinte
find: returnează rădăcina (chiar nodul, dacă e singur)
union: leagă rădăcina unuia de rădăcina celuilalt

Rezoluția în calculul predicatelor

Considerăm două clauze A și B, și un predicat P.

Redenumim variabilele din B ca să nu fie comune cu A
(fiecare clauză are spațiul ei de variabile)

Alegem literalii $P_1, \ldots, P_k$ din A și $\neg P_{k+1}, \ldots \neg P_{k+l}$ din B
(toți cu predicatul P)

Unificăm termenii $\{P_1, \ldots, P_k, P_{k+1}, \ldots P_{k+l}\}$
Fie σ unificatorul cel mai general rezultat.

Formăm clauza $(A \cup B \setminus \{P_1, \ldots P_k, P_{k+1}, \ldots P_{k+l}\})$, îi aplicăm substituția σ, și o adăugăm la mulțimea clauzelor.

Dacă repetând obținem clauza vidă, formula inițială nu e realizabilă.
Dacă nu mai putem crea rezolvenți noi, formula inițială e realizabilă.

Rezoluție și programare logică

Metoda rezoluției e *completă* relativ la refutație

pentru orice formulă nerealizabilă, va ajunge la clauza vidă

dar nu poate *determina* realizabilitatea *oricărei formule*

(există formule pentru care rulează la infinit)

Caz particular: *clauzele Horn*: clauze cu *cel mult un literal pozitiv*

clauză definită: $p \leftarrow h_1 \wedge ... \wedge h_k$ (implicație)

fapt: p (se afirmă, ipoteză)

scop (goal): $false \leftarrow h_1 \wedge ... \wedge h_k$

(arată prin contradicție că $h_1 \wedge ... \wedge h_k$ e adevărată)

Pentru astfel de clauze, există o metodă de rezoluție mai simplă:

se pornește de la țintă

se înlocuiește (cu unificare) pe rând fiecare sub-scop cu conjuncția premiselor care îl fac adevărat

Clauzele Horn și acest caz particular de rezoluție stau la baza *programării logice* (limbajul Prolog)

Exemplu de program Prolog

```
desc(X, Y) :- fiu(X, Y).
desc(X, Z) :- fiu(X, Y), desc(Y, Z).
fiu(ion, petre).
fiu(george, ion).
fiu(radu, ion).
fiu(petre, vasile).
```

Specificând ca scop (goal) `desc(X, vasile)`, se aplică rezoluția pornind de la negația clauzei: $\neg$ `desc(X, vasile)`.
Se găsește un rezolvent cu $\neg$ `fiu(X, Y)` $\wedge$ `desc(X, Y)`, rezultă $\neg$ `fiu(X, vasile)` care are rezolvent cu `fiu(petre, vasile)`
Deci `X = vasile` e soluție pentru scopul inițial.

Alte soluții se obțin prin rezoluție cu a doua clauză `desc`, etc.

Logica matematică în verificarea programelor

La sfârșitul anilor '60 - începutul anilor 70: definirea riguroasă a semanticii programelor de către Floyd, Hoare și Dijkstra

Cum se poate defini efectul unei instrucțiuni ?

Dacă înainte de a executa o instrucțiune e adevărată o condiție (un predicat peste variabilele programului), ce relație e adevărată după execuția instrucțiunii ?

ce se va *întâmpla în program* ?

Dacă într-un punct de program e adevărată o relație, ce condiție trebuie să fi fost adevărată înainte de execuția instrucțiunii ?

de ce s-a întâmplat ceva anume ? (*depanare*)

Axiomele (regulile) lui Hoare

Sunt definite pentru fiecare tip de instrucțiune în parte;
prin combinația lor, se poate raționa despre programe întregi

Notație: *corectitudine parțială* $\{P\}\ S\ \{Q\}$
Dacă S e executat într-o stare care satisface P, și execuția lui S se termină, starea rezultantă satisface Q

Atribuire:
$$\frac{}{\{Q[x/E]\}\ x := E\ \{Q\}}$$
unde $Q[x/E]$ e substituția lui x cu E în Q

Exemplu: $\{x = y - 2\}\ x := x + 2\ \{x = y\}$

în rezultat, $x = y$, substituim x cu expresia atribuită, $x + 2$ și obținem $x + 2 = y$, deci $x = y - 2$

Obs: e mai simplu să scriem regula "înapoi" (P în funcție de Q)

Secvențiere:
$$\frac{\{P\}\ S_1\ \{Q\} \qquad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}$$

Decizie:
$$\frac{\{P \wedge E\}\ S_1\ \{Q\} \qquad \{P \wedge \neg E\}\ S_2\ \{Q\}}{\{P\}\ \text{if } E \text{ then } S_1 \text{ else } S_2\ \{Q\}}$$

Regulile lui Hoare (continuare)

Ciclul cu test (inițial): cheia în raționamentul despre programe

Trebuie găsit un *invariant* I = o proprietate menținută adevărată de fiecare execuție a ciclului (exprimată în punctul dintre iterații)

Dacă intrăm în ciclu (E), invariantul e menținut după o iterație S

Dacă nu mai intrăm ($\neg E$), invariantul implică concluzia Q

$$\frac{\{I \wedge E\}\ S\ \{I\} \qquad I \wedge \neg E \Rightarrow Q}{\{I\}\ \text{while}\ E\ \text{do}\ S\ \{Q\}}$$

```
int s = 0;
for (int i = 0; i < n; ++i)
  s += a[i];
```

Invariant: $s = \sum_{k=0}^{i-1} a[k]$. Inițial, $\sum_{k=0}^{0-1}$: 0 termeni în sumă.
La fiecare iterație, i crește, și se adaugă termenul $a[i]$.
La ieșirea din ciclu: $i = n$, si $s = \sum_{k=0}^{n-1} a[k]$

Operatorul *weakest precondition* al lui Dijkstra

Pentru o instrucțiune S și postcondiție dată Q pot exista mai multe precondiții P astfel încât $\{P\}\ S\ \{Q\}$

Dijkstra definește o precondiție *necesară și suficientă* $wp(S, Q)$ pentru terminarea cu succes a lui S cu postcondiția Q.

Atribuire: $wp(x := E, Q) = Q[x/E]$ (v. regula lui Hoare)

Secvențiere: $wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q))$

Condițional:
$wp(\text{if } E \text{ then } S_1 \text{ else } S_2, Q) =$
$(E \Rightarrow wp(S_1, Q)) \wedge (\neg E \Rightarrow wp(S_2, Q))$

Pentru iterație e nevoie de un calcul recursiv.

Principiul inducției matematice

Logica de ordinul I nu e legată de un anumit univers.
însă frecvent, folosim universul numerelor (întregi sau reali)

Principiul *inducției* matematice e (în ciuda numelui) o *regulă de deducție* în *teoria aritmetică* a numerelor naturale

S-ar putea formula:

$$\forall P[P(0) \wedge \forall k \in \mathbb{N}.P(k) \rightarrow P(k+1)] \rightarrow \forall n \in \mathbb{N}\ P(n)$$

dar formula e în logică de *ordinul 2* (cuantificare peste predicate)

În aritmetica lui Peano, e definit ca o *schemă de axiome* (o axiomă pentru fiecare predicat) $\Rightarrow$ nu necesită cuantificare peste predicate

$$\forall \bar{x}[P(0,\bar{x}) \wedge \forall n(P(n,\bar{x}) \rightarrow P(S(n),\bar{x})) \rightarrow \forall n P(n,\bar{x})]$$

Principiul inducției matem.: echivalent cu *principiul bunei ordonări*:
Orice mulțime nevidă de nr. naturale are un cel mai mic element

Mai general: *inducția structurală*: demonstrăm proprietăți despre obiecte tot mai complexe (pt. obiecte definite inductiv/recursiv)