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 2014

Rezoluția: de la propoziții la predicate

În logica propozițională:
Rezoluția e o *regulă de inferență* care produce o nouă clauză
din două clauze cu literale complementare (p și $\neg p$).

$$\frac{p \vee \alpha \qquad \neg p \vee \beta}{\alpha \vee \beta}$$ *rezoluție*

Rezoluția e o regulă de inferență *validă*:
- dacă premisele sunt adevărate, și concluzia e adevărată
- dacă concluzia e contradicție, premisele formează o contradicție

Folosim rezoluția pentru a arăta că o formulă e o *contradicție*.
- e o metodă de *refutație* (respingere a unei afirmații)

Putem *demonstra* o teoremă (tautologie) prin *reducere la absurd*
arătând prin rezoluție că negația ei e o contradicție (nerealizabilă).

În logica predicatelor, un *literal* nu e o propoziție, ci un *predicat*
- nu mai avem p și $\neg p$, ci $P(\ldots)$ și $\neg P(\ldots)$
- $\Rightarrow$ trebuie să analizăm și argumentele predicatului

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 substitui y și obține a din $g(y)$
interpretare: nu putem presupune că funcția arbitrară g
ia obligatoriu și valoarea constantă a.

Definim precis acest lucru: noțiunile de *substituție* și *unificare*

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: se întâlnesc și notațiile x_i/t_i, sau 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 din care orice alt unificator poate fi obținut aplicând încă o substituție.

În *rezoluție*: având clauzele $P(l_1, l_2, \ldots l_n)$ 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 dar 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)

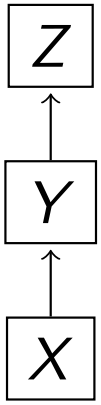
Union-Find

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

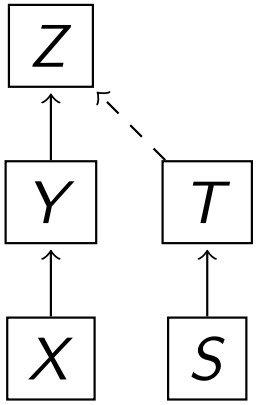
Operații:
find(element): găsește reprezentantul clasei de echivalență
union(elem1, elem2): declară elementele ca fiind echivalente
(și rămân așa mai departe)

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

Exemplu pentru Union-Find



$find(X) = find(Y)$
$= find(Z) = Z$



$union(Y, S)$
leagă $find(Y)$ și $find(S)$

Rezoluția în calculul predicatelor

Fie 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 literale $P_1, ..., P_k \in A$ și $\neg P_{k+1}, ...\neg P_{k+l} \in B$ cu pred. 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.

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)

Rezoluție și programare logică

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

clauze definite: $p_i \leftarrow h_1 \wedge ... \wedge h_k$ (implicație)

se transformă în $p_i \vee \neg h_1 \vee ... \vee \neg h_k$

fapte: p (se afirmă, ipoteze)

scop/țintă (*goal*): $false \leftarrow g_1 \wedge ... \wedge g_m$

se transformă în $\neg g_1 \vee ... \vee \neg g_m$

(arată prin contradicție că $g_1 \wedge ... \wedge g_m$ 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 scop parțial g_j cu conjuncția premiselor care îl fac adevărat

(privit ca rezoluție: se unifică un scop g_j cu o concluzie p_i, rezultă înlocuirea cu $\neg h_1 \vee ... \vee \neg h_k$)

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

Programare logică: 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).
```

Avem: *fapte* (ipoteze, predicate adevărate) ex. fiu(..., ...)
reguli: `:-` înseamnă *dacă*, virgula e conjuncție $\wedge$

Presupunem ca țintă/scop (goal) `desc(X, vasile)`.
Aplicăm rezoluția pornind de la negația ei: $\neg$ `desc(X, vasile)`.
Redenumind, prima regulă dă `desc(X1, Y1)` $\vee \neg$ `fiu(X1, Y1)`
Ca rezolvent, obținem $\neg$ `fiu(X, vasile)`. `Y1=vasile`
Acesta admite rezolvent cu `fiu(petre, vasile)`: clauza vidă.
Deci `X = petre` e soluție pentru ținta inițială.

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

Exemplu Prolog (cont.)

Unificăm ținta $\neg$`desc(X, vasile)` cu concluzia regulii 2:
`desc(X1, Z1)` $\vee \neg$ `fiu(X1, Y1)` $\vee \neg$ `desc(Y1, Z1)`
Obținem $\neg$ `fiu(X, Y1)` $\vee \neg$ `desc(Y1, vasile)` `Z1=vasile`

Unificăm primul predicat cu `fiu(ion, petre)` `X=ion, Y1=petre`
rămâne rezolventul $\neg$ `desc(petre, vasile)`.

Am văzut deja că pentru acesta putem ajunge la clauza vidă
$\Rightarrow$ `X=ion` e încă o soluție pentru ținta inițială

Dacă ținta conține variabile, interpretorul Prolog va genera toate soluțiile posibile (toate unificările/substituțiile pentru variabile).

Altfel, determină dacă predicatul dat (fără variabile) e adevărat.

Exemplu Prolog: inversarea listelor

Noțiunile recursive se scriu similar în logică și programare funcțională.
Pentru liste, folosim constanta `nil` și constructorul `cons`.
Inversarea devine predicat cu 3 argumente: lista, acumulator, rezultat.

```
rev3(nil, R, R).
rev3(cons(H, T), Ac, R) :- rev3(T, cons(H, Ac), R).
rev(L, R) :- rev3(L, nil, R)
```

Când lista e vidă, rezultatul e acumulatorul.
Altfel, adăugând capul listei la acumulator, obținem același rezultat.
Inversarea se obține luând acumulatorul (auxiliar) lista vidă.

Dând ca țintă $rev(c(1, c(2, c(3, nil))), X)$ obținem
$X = c(3, c(2, c(1, nil)))$, derivarea (raționamentul) fiind:

$rev3(nil,\ c(3, c(2, c(1, nil))),\ c(3, c(2, c(1, nil))))$
$\quad \rightarrow rev3(c(3, nil),\ c(2, c(1, nil)),\ c(3, c(2, c(1, nil))))$
$\quad \rightarrow rev3(c(2, c(3, nil)),\ c(1, nil),\ c(3, c(2, c(1, nil))))$
$\quad \rightarrow rev3(c(1, c(2, c(3, nil))),\ nil,\ c(3, c(2, c(1, nil))))$
$\quad \rightarrow rev(c(1, c(2, c(3, nil))),\ c(3, c(2, c(1, nil))))$

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)