Logică și structuri discrete

Logica predicatelor. Programare logică

Casandra Holotescu

casandra@cs.upt.ro

https://tinyurl.com/lecturesLSD

Programare logică

specificăm *ce* vrem să obținem

caută automat *soluția* (decide *cum*)

bazată pe rezoluție

Programare logică: Prolog

Programatorul specifică *declarativ* ce se știe despre problemă.
Interpretorul *găsește soluțiile* căutând demonstrații.

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

fapte (predicate adevărate) `fiu(ion, petre).`
reguli: `:-` e implicație $\leftarrow$ (stânga *dacă* dreapta)
virgula e conjuncție $\wedge$

X e descendentul lui Y dacă X e fiul lui Y.
X e descendentul lui Z dacă X e fiul lui Y și Y e descendentul lui Z

Ce înseamnă clauzele

(R1) `desc(X, Y) :- fiu(X, Y).`
(R2) `desc(X, Z) :- fiu(X, Y), desc(Y, Z).`

Variabilele din *stânga* sunt cuantificate *universal* (în ambele părți)
Variabilele care apar doar în *dreapta* sunt cuantificate *existențial*.

Ce înseamnă clauzele

(R1) `desc(X, Y) :- fiu(X, Y).`
(R2) `desc(X, Z) :- fiu(X, Y), desc(Y, Z).`

Variabilele din *stânga* sunt cuantificate *universal* (în ambele părți)
Variabilele care apar doar în *dreapta* sunt cuantificate *existențial*.

R1: $\forall X \forall Y . desc(X, Y) \leftarrow fiu(X, Y)$

R2: X e descendent al lui Z dacă *există* Y
astfel încât X e fiul lui Y și Y e descendent al lui Z

R2: $\forall X \forall Z . desc(X, Z) \leftarrow \exists Y . fiu(X, Y) \wedge desc(Y, Z)$
$\forall X \forall Z . desc(X, Z) \vee \neg \exists Y . fiu(X, Y) \wedge desc(Y, Z)$
$\forall X \forall Z . desc(X, Z) \vee \forall Y \neg (fiu(X, Y) \wedge desc(Y, Z))$

Ce înseamnă clauzele

(R1) `desc(X, Y) :- fiu(X, Y).`
(R2) `desc(X, Z) :- fiu(X, Y), desc(Y, Z).`

Variabilele din *stânga* sunt cuantificate *universal* (în ambele părți)
Variabilele care apar doar în *dreapta* sunt cuantificate *existențial*.

R1: $\forall X \forall Y . desc(X, Y) \leftarrow fiu(X, Y)$

R2: X e descendent al lui Z dacă *există* Y
astfel încât X e fiul lui Y și Y e descendent al lui Z

R2: $\forall X \forall Z . desc(X, Z) \leftarrow \exists Y . fiu(X, Y) \wedge desc(Y, Z)$
$\forall X \forall Z . desc(X, Z) \vee \neg \exists Y . fiu(X, Y) \wedge desc(Y, Z)$
$\forall X \forall Z . desc(X, Z) \vee \forall Y \neg (fiu(X, Y) \wedge desc(Y, Z))$

Eliminând cuantificatorii universali rezultă:
R1: $desc(X, Y) \vee \neg fiu(X, Y)$
R2: $desc(X, Z) \vee \neg fiu(X, Y) \vee \neg desc(Y, Z)$

Rezoluția în Prolog

Fie ca întrebare/scop/țintă (engl. *goal*) `desc(X, vasile).`

soluție = o valoare `xs` pentru `X` care face predicatul adevărat.

Rezoluția în Prolog

Fie ca întrebare/scop/țintă (engl. *goal*) `desc(X, vasile).`

soluție = o valoare `xs` pentru `X` care face predicatul adevărat.
$\Rightarrow$ negația $\neg$ `desc(xs, vasile)` va da o contradicție.

Folosim *rezoluția* încercând să găsim o contradicție.

Rezoluția în Prolog

Fie ca întrebare/scop/țintă (engl. *goal*) `desc(X, vasile).`

soluție = o valoare `xs` pentru `X` care face predicatul adevărat.
$\Rightarrow$ negația $\neg$ `desc(xs, vasile)` va da o contradicție.

Folosim *rezoluția* încercând să găsim o contradicție.

Unificăm cu regula R1 (redenumind-o cu variabile noi):

	$\neg$ `desc(X, vasile)`	
(R1) `desc(X1, Y1)` $\vee \neg$ `fiu(X1, Y1)`		
	$\neg$ `fiu(X, vasile)`	`X1=X, Y1=vasile`
faptul (4)	`fiu(petre, vasile)`	
	$\emptyset$ (clauza vidă)	`X=petre`

Rezoluția în Prolog

Fie ca întrebare/scop/țintă (engl. *goal*) `desc(X, vasile).`

soluție = o valoare `xs` pentru `X` care face predicatul adevărat.
$\Rightarrow$ negația $\neg$ `desc(xs, vasile)` va da o contradicție.

Folosim *rezoluția* încercând să găsim o contradicție.

Unificăm cu regula R1 (redenumind-o cu variabile noi):

	$\neg$ `desc(X, vasile)`	
(R1) `desc(X1, Y1)`	$\vee \neg$ `fiu(X1, Y1)`	
	$\neg$ `fiu(X, vasile)`	`X1=X, Y1=vasile`
faptul (4)	`fiu(petre, vasile)`	
	$\emptyset$ (clauza vidă)	`X=petre`

Pentru `X=petre` am obținut o contradicție.
`desc(petre, vasile)` e *adevărat*. `X=petre` e o *soluție*.

Continuând, putem găsi și alte soluții.

Soluții multiple în Prolog

Pornim tot cu negația întrebării: ¬`desc(X, vasile)`.

Unificăm cu regula 2 (redenumind din nou variabilele):

¬`desc(X,vasile)`		
`desc(X2,Z2)` ∨ ¬`fiu(X2,Y2)` ∨ ¬`desc(Y2,Z2)`		
	¬`fiu(X, Y2)` ∨ ¬`desc(Y2,vasile)`	`X2=X,Z2=vasile`
faptul (1)	`fiu(ion,petre)`	
	¬`desc(petre,vasile)`	`X=ion,Y2=petre`
obținut anterior	`desc(petre,vasile)`	
	∅ (clauza vidă)	

⇒ `X=ion` e încă o soluție pentru întrebarea inițială.

Dacă întrebarea are variabile, interpretorul Prolog generează *toate soluțiile posibile* (toate substituțiile pentru variabile).

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

Exemplu Prolog: inversarea listelor (1)

Noțiunile recursive se scriu similar în logică și programare funcțională.

Pentru liste folosim constanta `nil` (lista vidă) 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.

```
let rec rev2 acc = function
  | [] -> acc
  | h :: t -> rev2 (h::acc) t
```

Exemplu Prolog: inversarea listelor (2)

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

Cu întrebarea $rev(c(1, c(2, c(3, nil))), X)$ obținem derivarea:

$rev(c(1, c(2, c(3, nil))), X)$

L1=c(1,c(2,c(3,nil))), R1=X

$\leftarrow rev3(c(1, c(2, c(3, nil))), nil, X)$

H1=1, T1=c(2,c(3,nil)), Ac1=nil

$\leftarrow rev3(c(2, c(3, nil)), c(1, nil), X)$

H2=2, T2=c(3,nil), Ac2=c(1,nil)

$\leftarrow rev3(c(3, nil), c(2, c(1, nil)), X)$

H3=3, T3=nil, Ac3=c(2,c(1,nil))

$\leftarrow rev3(nil, c(3, c(2, c(1, nil))), X)$

X=c(3,c(2,c(1,nil)))

Rezoluție și programare logică

Prolog folosește un caz particular de rezoluție: *clauzele Horn*
= clauze cu *cel mult un literal pozitiv*

Rezoluție și programare logică

Prolog folosește un caz particular de rezoluție: *clauzele Horn*
= clauze cu *cel mult un literal pozitiv*

clauze definite (reguli): $p_i \leftarrow h_1 \wedge ... \wedge h_k$ (implicație)
se transformă în $p_i \vee \neg h_1 \vee ... \vee \neg h_k$ doar p_i e pozitiv

fapte: p_j (se afirmă, ipoteze) p_j pozitiv

Rezoluție și programare logică

Prolog folosește un caz particular de rezoluție: *clauzele Horn*
= clauze cu *cel mult un literal pozitiv*

clauze definite (reguli): $p_i \leftarrow h_1 \wedge ... \wedge h_k$ (implicație)
se transformă în $p_i \vee \neg h_1 \vee ... \vee \neg h_k$ doar p_i e pozitiv

fapte: p_j (se afirmă, ipoteze) p_j pozitiv

întrebare/scop/țintă (*goal*): $false \leftarrow g_1 \wedge ... \wedge g_m$
= arată prin contradicție că $g_1 \wedge ... \wedge g_m$ e adevărată
se transformă în $\neg g_1 \vee ... \vee \neg g_m$

Rezoluție și programare logică

Pentru clauze Horn, rezoluția se aplică mai simplu:

- se pornește de la *țintă* ($false \leftarrow g_1 \wedge ... \wedge g_m$)
- se caută pe rând fiecare scop parțial g_j în *concluziile* regulilor și se înlocuiește cu *conjuncția premiselor* din regulă (clauză) ($p_i \leftarrow h_1 \wedge ... \wedge h_k$)
- până când în toate cazurile se ajunge la *fapte*

Rezoluție și programare logică

Pentru clauze Horn, rezoluția se aplică mai simplu:

se pornește de la *țintă* ($false \leftarrow g_1 \wedge ... \wedge g_m$)

se caută pe rând fiecare scop parțial g_j în *concluziile* regulilor
și se înlocuiește cu *conjuncția premiselor* din regulă (clauză)
($p_i \leftarrow h_1 \wedge ... \wedge h_k$)

până când în toate cazurile se ajunge la *fapte*

Privit ca rezoluție:

ținta $\neg g_1 \vee ... \vee \neg g_m$
regulile $p_i \vee \neg h_1 \vee ... \vee \neg h_k$

se unifică un *scop* $\neg g_j$ cu o *concluzie* p_i,
rezultă înlocuirea cu $\neg h_1 \vee ... \vee \neg h_k$

Aplicații ale programării logice

Gestiune de dependențe/configurații/compilare în sisteme software
reguli de tip Makefile

```
main:  main.c file.c lib.c
    gcc -o main main.c file.c lib.c
```

Sisteme expert: reguli cu cunoștințe dintr-un domeniu (medical,...)

Datalog: un subset de Prolog
- interogări în baze de date
- specificări de politici de securitate

Programare logică cu constrângeri (*constraint logic programming*)
- planificare (orare, zboruri, personal)
- gestiune de portofolii financiare
- optimizări de circuite