Rappresentazione della conoscenza

Lezione 4

Efficienza nei programmi PROLOG

- Controllo dell'inferenza
- Meta-predicati
- Termini e loro rappresentazione
- Taglio per controllare il backtracking
- Meta-interpreti

Efficienza nei programmi PROLOG

- Specifica dei programmi
- Ordine delle clausole
- Ordine dei congiunti
- Taglio per controllare il backtracking

Specifica dei programmi PROLOG

```
discendente(X,Y):-figlio(X,Y). % 1
discendente(X,Y):-figlio(Z,Y),discendente(X,Z).

discendente(X,Y):-figlio(X,Y). % 2
discendente(X,Y):-figlio(X,Z),discendente(Z,Y).

discendente(X,Y):-figlio(X,Y). % 2
discendente(X,Y):-discendente(X,Z),discendente(Z,Y).
```

Controllo esplicito dell'inferenza

"Qual è il reddito della moglie del presidente degli Stati Uniti?"

```
Reddito(S,I),Sposati(S,P),Lavoro(P,presidenteStatiUniti).
```

♢ **metaragionamento** per decidere quale ordine degli atomi è più vantaggioso

Con un predicato che restituisce il numero di istanze di una relazione, posso ordinare i congiunti in modo da ottimizzare il calcolo della risposta.

Efficienza nelle implementazioni PROLOG

- **backtracking guidato dalle dipendenze** invece del **backtracking cronologico**
- **intelligent backtracking** che ricorda le derivazioni già fatte evitando di ripeterle

Predicati meta-logici

♢ Servono per trattare gli elementi di un programma come strutture di dati.

♢ Accedono alla rappresentazione interna dei programmi.

♢ Consentono la meta-programmazione.

♢ Meta-programmazione consente il controllo **esplicito** dell'esecuzione

Predicati meta-logici di tipo

- `var(Term)` ha successo se `Term` è una variabile
- `nonvar(Term)` ha successo se `Term` non è una variabile

```
nonno(X,Z):- nonvar(X), genitore(X,Y), genitore(Y,Z).
nonno(X,Z):- nonvar(Z), genitore(Y,Z), genitore(X,Y).
```

Termini ground

```
ground(Term) :- nonvar(Term), constant(Term).
ground(Term) :-  nonvar(Term), compound(Term),
                functor(Term,F,N), ground(N,Term).

ground(N,Term) :- N > 0, arg(N,Term,Arg),
                  ground(Arg), N1 is N-1,
                 ground(N1,Term).
ground(0,Term).

%  verifica se un termine e' ground
```

Uguaglianza e Unificazione

- `X == Y` ha successo se `X` ed `Y` sono la stessa costante, la stessa variabile, o strutture con identico funtore e ricorsivamente == (negato \ ==)
- `X = Y` ha successo se `X` ed `Y` unificano

Unificazione

```
unify1(X,Y) :- var(X), var(Y), X=Y.
unify1(X,Y) :- var(X), nonvar(Y),
               not_occurs_in(X,Y), X=Y.
unify1(X,Y) :- var(Y), nonvar(X),
               not_occurs_in(Y,X), Y=X.
unify1(X,Y) :- nonvar(X), nonvar(Y),
               constant(X), constant(Y), X=Y.
unify1(X,Y) :- nonvar(X), nonvar(Y), compound(X),
               compound(Y), term_unify1(X,Y).

term_unify1(X,Y) :- functor(X,F,N), functor(Y,F,N),
                    unify1_args(N,X,Y).

%     Unificazione con occurs check
```

Unificazione degli argomenti

```
unify1_args(N,X,Y) :- N > 0, unify1_arg(N,X,Y),
                      N1 is N-1, unify1_args(N1,X,Y).
unify1_args(0,X,Y).

unify1_arg(N,X,Y) :- arg(N,X,ArgX), arg(N,Y,ArgY),
                     unify1(ArgX,ArgY).
```

Occurs check

```
not_occurs_in(X,Y) :- var(Y), X \== Y.
not_occurs_in(X,Y) :- nonvar(Y), constant(Y).
not_occurs_in(X,Y) :- nonvar(Y), compound(Y),
                      functor(Y,F,N),
                      not_occurs_in(N,X,Y).

not_occurs_in(N,X,Y) :- N > 0, arg(N,Y,Arg),
                        not_occurs_in(X,Arg),
                        N1 is N-1,
                        not_occurs_in(N1,X,Y).
not_occurs_in(0,X,Y).
```

Collezionare soluzioni

Spesso è utile utilizzare tutte le le soluzioni di un programma.

`findall(Term,Goal,List).`

`List` è la lista ottenuta, valutando `Goal` e prendendo i legami associati a `Term`.

`findall(X,nonno(giovanni,X),Nipoti).`

Taglio

Una generica clausola con taglio ha la seguente forma:

$$A \leftarrow B_1, \ldots, B_k, !, B_{k+2}, B_n.$$

Se l'obiettivo corrente G unifica con A e $B_1, \ldots, B_k$ vengono raggiunti con successo,

- ogni altra dimostrazione che si potrebbe ottenere unificando G con un'altra clausola viene eliminata dall'albero di ricerca;
- ogni ulteriore alternativa di dimostrazione di $B_1, \ldots, B_k$, viene viene eliminata dall'albero di ricerca.

Taglio: un esempio

```
merge1([X|Xs],[Y|Ys],[X|Zs]):-
        X<Y,!,
        merge1(Xs,[Y|Ys],Zs).
merge1([X|Xs],[Y|Ys],[X,Y|Zs]):-
    X=:=Y,!,
    merge1(Xs,Ys,Zs).
merge1([X|Xs],[Y|Ys],[Y|Zs]):-
    X>Y,!,
    merge1([X|Xs],Ys,Zs).
merge1([],X,X):- !.
merge1(Y,[],Y):- !.
```

Tagli verdi e rossi: eliminare soluzioni valide

Il taglio si dice **verde**, quando non altera le soluzioni del programma, **rosso** altrimenti.

```
antenato(X,Y):- genitore(X,Y).
antenato(X,Y):- antenato(X,Z),!,genitore(Z,Y).
```

Ferma la ricerca al primo antenato di X: Z. Poiché non è detto che Y sia genitore di Z, la ricerca fallisce.

Omissione delle condizioni esplicite

$$A \leftarrow B_1, \ldots, B_k, !, B_{k+2}, B_n$$

Taglio verde:

```
if (B1,B2,..Bk) return f(y);      minimum(X,Y,X):-X=<Y,!.
if (B1',B2',..,Bj')return f'(y); minimum(X,Y,Y):-X>Y,!.
```

Taglio rosso:

```
if (B1,B2,..Bk) return f(y);     minimum(X,Y,X):-X=<Y,!.
return f'(y);                    minimum(X,Y,Y).
                                 minimum(2,5,5)/YES????????
```

Regola: Scrivere sempre il programma prolog in versione senza tagli, ed aggiungere solo tagli verdi.

Tagli verdi e rossi

Taglio verde:

```
sort2(Xs,Ys) :- append1(As,[X,Y|Bs],Xs), X > Y, !,
                        append1(As,[Y,X|Bs],Xs1), sort2(Xs1,Ys).
sort2(Xs,Xs) :- ordered(Xs), !.
```

Taglio rosso:

```
sort3(Xs,Ys) :- append1(As,[X,Y|Bs],Xs), X > Y, !,
                append1(As,[Y,X|Bs],Xs1),
                sort3(Xs1,Ys).
sort3(Xs,Xs).
/*
 * verificare che sort3([1,3,2],[1,3,2]). \‘e si
 * mentre sort2([1,3,2],[1,3,2]). \‘e no
 */
```

Controllo del backtracking

```
cousin(massimo, daniele),!,american(daniele)

member(a,C), !, q(a).

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

Negazione come fallimento

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

`fail` è un predicato che fallisce sempre.

Variabili come programmi

- `call(X)` esegue il goal `X` che deve essere istanziato, si può semplicemente scrivere la variabile nel corpo di una clausola.

Es: Disgiunzione

```
X ; Y :- X.
X ; Y :- Y.
```

Meta interprete di base

Un **meta-interprete** Prolog è un interprete per il Prolog scritto in Prolog.

```
solve1(true).
solve1((A,B)) :- solve1(A), solve1(B).
solve1(A) :- clause(A,B), solve1(B).
```

`clause/2` restituisce la testa ed il corpo di una clausola definite nel programma (almeno un arg istanziato).

Meta interprete con l'unificazione

```
solve2(true).
solve2((A,B)) :- solve2(A), solve2(B).
solve2(A) :- clause(X,B), unify1(X,A), solve2(B).
```

Meta interprete che traccia la prova

```
solve_trace(Goal) :- solve_trace(Goal,0).

solve_trace(true,Depth) :- !.
solve_trace((A,B),Depth) :-  !, solve_trace(A,Depth),
                             solve_trace(B,Depth).
solve_trace(A,Depth) :- builtin(A), !, A,
                        display(A,Depth), nl.
solve_trace(A,Depth) :- clause(A,B), display(A,Depth),
                        nl, Depth1 is Depth + 1,
                        solve_trace(B,Depth1).

display(A,Depth) :-  Spacing is 3*Depth,
                     put_spaces(Spacing), write(A).
```

```
put_spaces(N) :- between(1,N,I), put_char(' '), fail.
put_spaces(N).

between(I,J,I) :- I =< J.
between(I,J,K) :- I < J, I1 is I + 1, between(I1,J,K).

% Program 17.7 A tracer for Prolog
```

Meta interprete che costruisce l'albero di prova

```
solve(true,true) :- !.
solve((A,B),(ProofA,ProofB)) :- !, solve(A,ProofA),
                                 solve(B,ProofB).
solve(A,(A:-builtin)) :- builtin(A), !, A.
solve(A,(A:-Proof)) :- clause(A,B), solve(B,Proof).

% Program 17.8
% A meta-interpreter for building a proof tree
```

Meta interprete con fattori di certezza

```
solve(true,1) :- !.
solve((A,B),C) :- !, solve(A,C1),
                  solve(B,C2),
                  minimum(C1,C2,C).
solve(A,1) :- builtin(A), !, A.
solve(A,C) :-clause_cf(A,B,C1),
              solve(B,C2), C is C1 * C2.

minimum(X,Y,X) :- X =< Y, !.
minimum(X,Y,Y) :- X > Y, !.

%  Program 17.9
% A meta-interpreter for reasoning with uncertainty
```