META-INTERPRETI

Esercizio

Si scriva un meta-interprete depth-first per Prolog puro che quando cerca di selezionare una clausola la cui testa unifica con un dato goal G per fare il passo di risoluzione, presenti all'utente la lista di tutte le clausole ancora non tentate che unificano con G e lasci a lui la scelta. In questo modo le clausole non sono più tentate nell'ordine testuale di Prolog.

Soluzione

```
metadf(true):-!.
metadf((A,B)):-!,metadf(A),metadf(B).
```

% Ora viene costruita una lista `L` che contiene liste del ti-
% po `[Head,Body]` di clausole la cui testa unifica con `X`.

```
metadf(X):-
    findall(G,unificano(X,G),L),!,
    choice(X,L).

unificano(X,G):-clause(X,Body), G=[X,Body].
```

% Nel caso di lista vuota l'alternativa fallisce.

```
choice(_,[]):- !,
    write('Questa alternativa fallisce'),
    fail.
```

% Se esiste un'unica clausola è inutile chiedere
% all'utente di effettuare una scelta.

```
choice(X,[[H,Body]]):- !,
    choice1(X,H,[[H,Body]],Body).
```

% Nel caso di più clausole l'utente effettua una scelta.

```
choice(X,L):-
    scelta(L,[H,Body]),
    choice1(X,H,L,Body).
```

% `choice1` la prima volta che viene chiamato provvede all'unificazione della testa della clausola selezionata con il sottogoal da dimostrare poi cerca di dimostrare il body.

```
choice1(X,H,_,Body):-
    H=X,
    metadf(Body).
```

% In caso di fallimento è necessario tentare le strade %ancora aperte permettendo all'utente di scegliere %nuovamente ma eliminando dalle possibili scelte quelle %già effettuate.

```
choice1(X,H,L,Body):-
    delete([H,Body],L,L1),
    choice(X,L1).

scelta(L,[H,Body]):-
    write('scegli una clausola dalla lista'),
    nl,
    write(L),
    read([H,Body]).
```

NB: La lista che viene presentata all'utente contiene le clausole che unificano con la clausola da dimostrare, <u>già unificate</u>.

Esercizio
Si costruisca un meta-interprete per Prolog che chieda all'utente i goal che non è in grado di dimostrare, solo se per essi non compaiono fatti ground nel database con lo stesso funtore e arità.

Questo meta-interprete può essere utile nella realizzazione di sistemi esperti dove, spesso, il sistema chiede all'utente goal che non è in grado di dimostrare.

L'utente può rispondere in diversi modi:

1) può semplicemente asserire true o false.

2) può rispondere true istanziando eventuali variabili unbound.

Nell'esercizio si è scelta la prima strada.

Soluzione

```
meta1(true):-!.
meta1((X,Y)):- !,meta1(X),meta1(Y).
meta1(X):- clause(X,Body),meta1(Body).
```

% Non ho dimostrato `X`: cerco fatti ground nel database.
% Se non li trovo chiedo all'utente con `ask/1` se `X` è true
% o false

```
meta1(X):-
    not(fatti(X)),
    ask(X,Risp),call(Risp).

fatti(X):-
    functor(X,P,A),
    functor(K,P,A),
    clause(K,true),
    K=..[_|Arg],
    gr(Arg).

ask(X,Risp):-
    write(X),
    write("  true or false ? "),
    read(Risp).
```

% `gr` controlla che gli argomenti siano ground.

```
gr([]).
gr([Argh|Argt]):-controllo(Argh),gr(Argt).
```

% Caso di una costante come argomento

```
controllo(B):-atomic(B),!.
```

% Caso di una lista come argomento.

```
controllo([H|T]):- !,controllo(H),
                     controllo(T).
```

% Caso di un termine composto come argomento.

```
controllo(B):- compound(B), B=..[_|L],
               controllo(L).
```

In alcuni Prolog esistono predicati (non standard) predefiniti `ground/1` o `nonground/1`

Esercizio

Si scriva un meta-interprete che sia in grado di sospendere l'esecuzione dopo un certo numero di passi di risoluzione N, dove N è specificato dall'utente. Il meta-interprete applica, inoltre, una strategia di riconoscimento di alcuni casi di (possibile) loop.
Dunque sospende l'esecuzione, riportando fallimento e il numero dei passi di risoluzione eseguiti, quando incontra un sotto-goal che sia uguale, una variante o un'istanza di un sotto-goal incontrato precedentemente durante la dimostrazione.

Più formalmente, un atomo A è una variante o un'istanza di un altro atomo B se esiste una sostituzione s tale che A= {B}s.

Soluzione

% solve riceve in ingresso il numero `N` massimo di passi %dopo i quali si deve sospendere l'esecuzione del Goal. %`Acc` restituisce, nel caso si concluda la risoluzione, il %numero di passi effettuati.

```
solve(Goal,Acc,N):-
    solve(Goal,[],L,0,Acc,N).
```

% la `solve/6` contiene, come parametri la lista dei goal da risolvere, la lista `L1` dei sotto-goal già incontrati durante la risoluzione, la lista `L3` dei sotto-goal incontrati durante la risoluzione attuale, `Acc` l'accumulatore di ingresso per il calcolo dei livelli già esplorati, `Acc2` l'accumulatore di uscita, e `N` il parametro definito dall'utente sul limite dei livelli di profondità.

```
solve(true,L,L,A,A,_):-!.
solve((A,B),L1,L3,Acc,Acc2,N):- !,
    solve(A,L1,L2,Acc,Acc1,N),
    solve(B,L2,L3,Acc1,Acc2,N).
```

% nel caso in cui l'accumulatore di ingresso sia uguale al parametro N definito dall'utente, si verifica un fallimento

```
solve(_,_,_,N,_,N):-
    write('No. di tentativi maggiore di '),
    write(N), nl,!,fail.
```

% nel caso in cui il goal in ingresso sia uguale, una %variante o un'istanza di un sotto-goal incontrato %precedentemente si verifica un fallimento

```
solve(Goal,Lista,_,_,_,_):-
    loop(Goal,Lista),!,
    write('Possibile loop '), nl, fail.
```

% risoluzione del goal Goal

```
solve(Goal,Lista1,Lista3,Acc,Acc2,N):-
    clause(Goal,Body), Acc1 is Acc+1,
    append([Goal],Lista1,Lista2),
    solve(Body,Lista2,Lista3,Acc1,Acc2,N).
```

% loop è un predicato che consente riconoscere un
%sottogoal che è istanza, uguale o variante di un
%sottogoal già incontrato

```
loop(_,[]):-!,fail.
loop(Goal,[Head|_]):-confronta(Goal,Head),!.
loop(Goal,[_|Tail]):-loop(Goal,Tail).
```

% confronta il funtore, l'arità e tutti i parametri di Goal e
%Head per individuare se Goal è una istanza, uguale o
%variante di Head

```
confronta(Goal,Head):-
    functor(Goal,F,A),
    functor(Head,F,A),
    Goal=..[_|Arg1],
    Head=..[_|Arg2],
    confr_arg(Arg1,Arg2).

confr_arg([],[]).
confr_arg([Arg1h|Arg1t],[Arg2h|Arg2t]):-
    var(Arg1h),var(Arg2h),!,
    conf_arg(Arg1t,Arg2t).
confr_arg([Arg1h|Arg1t],[Arg2h|Arg2t]):-
    nonvar(Arg1h), var(Arg2h),!,
    confr_arg(Arg1t,Arg2t).
confr_arg([Arg1h|Arg1t],[Arg2h|Arg2t]):-
    atomic(Arg1h), atomic(Arg2h),
    Arg1h==Arg2h,!,
    confr_arg(Arg1t,Arg2t).
confr_arg([Arg1h|Arg1t],[Arg2h|Arg2t]):-
    compound(Arg1h), compound(Arg2h),
    confronta(Arg1h,Arg2h),
    confr_arg(Arg1t,Arg2t).
```

Esercizio

Si scriva un meta-interprete PROLOG che assegna un limite massimo alla profondità dell'albero AND-OR che esplora. Tale profondità `D` può essere specificata dall'utente all'atto dell'invocazione del meta-interprete. Nel caso in cui la profondità dell'albero superi `D`, il meta-interprete non fallisce, ma produce come uscita il ramo corrente dell'albero che stava esplorando.
Più in dettaglio, il meta-interprete è invocato con goal della forma:

```
:- solve(Goal, D,Overflow).
```

Dove `Goal` è il goal iniziale e `D` una profondità massima.
Il predicato `solve` ha successo:
1) Nel caso in cui il `Goal` ha successo con una profondità massima che non supera `D` .
In questo caso la variabile `Overflow` verrà istanziata a `no_overflow`;
2) Nel caso in cui si sia superata la profondità massima D. In questo caso la variabile `Overflow` verrà istanziata ad una struttura del tipo `overflow([A|B])`, dove la lista `[A|B]` contiene i sottogoal sul corrente ramo di ricerca.

Esempio:
Dato il programma:

```
p(X):-q(X),r(X).
q(1).
r(X):-p(X).
```

otterremo per `:- solve(p(X),5,Overflow),`
il risultato:

```
X=1,
Overflow=overflow([p(1),r(1),p(1),r(1),p(1)]))
```

Soluzione

% Goal ha successo a una profondità minore di `D`

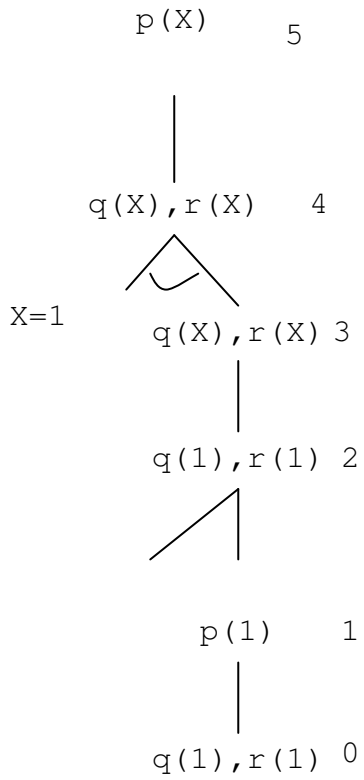
```
solve(true, D1, no_overflow):-
    D1>0.
```

% Raggiunta profondità massima

```
solve(A,0,overflow([])).
```

% Caso dell'and di due sottogoal

```
solve((A,B),D,Overflow):-
    D>0,
    solve(A,D,OverflowA),
    solve_conj(OverflowA,B,
        D,Overflow).
```




```
solve(Goal,D,Overflow):-
    D>0,
    clause(Goal,B),
    D1 is D-1,
    solve(B,D1,OverflowB),
    return_overflow(OverflowB,Goal,Overflow).

solve_conj(overflow(S),B,D,overflow(S)).
solve_conj(no_overflow,B,D,Overflow):-
    solve(B,D,Overflow).
```

% return_overflow torna no_overflow nel caso in cui il %Goal ha successo con una profondità che non supera %D. In caso contrario provvede a costruire la lista di goal e sottogoal richiesta

```
return_overflow(no_overflow,A,no_overflow).
return_overflow(overflow(S),A,overflow([A|S])).
```

Esercizio
Si scriva un metainterprete che riceva in ingresso, oltre al goal da dimostrare, il nome di un predicato e la sua arità e conti, nel corso della computazione (solo i rami di successo) il numero di chiamate al predicato.

Ad esempio, dato il programma:

```
p(X):- q(X), r(X,Y).
q(1).
q(2).
r(1,Y):- s(Y).
r(2,Y):- t(Y).
t(3).
```

Il metainterprete risponde alla chiamata `solve(p(Y), r,2,N)` istanziando `N` a `1`.

Soluzione

```
solve(Goal,Pred,Arity,N):-
    solve(Goal,Pred,Arity,0,N).

solve(true,_,_,N,N).
solve((A,B),Pred,Arity,Acc,Accout):-
    solve(A,Pred,Arity,Acc,AccTemp),
    solve(B,Pred,Arity,AccTemp,Accout).
solve(Goal,Pred,Arity,Accin,Accout):-
    functor(Goal,Pred,Arity),!,
    AccTemp is Accin + 1,
    clause(Goal,Body),

solve(Body,Pred,Arity,AccTemp,Accout).
solve(Goal,Pred,Arity,Accin,Accout):-
    clause(Goal,Body),
    solve(Body,Pred,Arity,Accin,Accout).
```

Esercizio

Si scriva un metainterprete per Prolog che all'interno del body di ogni clausola selezioni sempre prima quei sottogoal con un maggior numero di parametri diversi **prima dell'unificazione**.
Due parametri sono considerati uguali se sono due costanti uguali, oppure due variabili con nome uguale, oppure due termini composti con uguale funtore, uguale arità e argomenti uguali.
Ad esempio, nel body di

```
p(X,Y,Z):- b(Y, Y), a(X, Y), c(Z).
```

selezionerà prima `a(X, Y)`(numero di parametri diversi = 2) poi indifferentemente `b(Y, Y)` (numero di parametri diversi = 1) o `c(Z)` (numero di parametri diversi = 1).
Si supponga di avere a disposizione un predicato `sort(L,L1,L2)` che fornisce in `L2` la lista `L` ordinata (in senso decrescente) secondo i valori contenuti in `L1`.

NOTA: Si supponga inoltre di avere un programma nella forma `clausola(Head, Body)` dove `Body` è una lista di sottogoal.

Soluzione

```
solve([]).
solve([Head|Tail]):-!,
    solve(Head),
    solve(Tail).
solve(Goal):-
    functor(Goal,F,A),
    functor(Temp,F,A),
    clausola(Temp, Body),
    ordina_body(Body, Body_ord),
    Goal = Temp,
    solve(Body_ord).

ordina_body(Body, Body_ord)):-
    numero_par(Body, Parametri),
    sort(Body, Parametri, Body_ord).

numero_par([],[]).
numero_par([Head|Tail],[Nhead |Ntail]):-
    Head=..[_|Arg],
    conta(Arg, Nhead),
    numero_par(Tail, Ntail):

conta(Arg,Nhead):-
    conta(Arg, 0, Nhead).

conta([], N, N):- !.
conta([Arg|Tail], Nin, Nout):-
    member(Arg1, Tail),
    Arg == Arg1,!,
    conta(Tail, Nin, Nout).
conta([Arg|Tail], Nin, Nout):-
    Ntemp is Nin +1,
    conta(Tail, Ntemp, Nout).
```

Esercizio

Si scriva un metainterprete per Prolog che interpreti prima le clausole il cui body contiene un maggior numero di sottogoal diversi. Due sottogoal sono considerati uguali se hanno stesso funtore e stessa arità.

Ad esempio, date le due clausole:

```
p(X,Y,Z):- a(3, 3), a(Y, Z), b(Z, Z).

p(X,Y,Z):-  b(Y, Y), a(X, Y), c(Z).
```

Il metainterprete sceglierà prima la seconda clausola: infatti, il numero di sottogoal diversi della seconda clausola è 3 mentre per la prima è 2 (`a(3,3)` è considerato uguale ad `a(Y,Z)`).
Si supponga di avere a disposizione un predicato `sort(L,L1,L2)` che fornisce in `L2` la lista `L` ordinata (in senso decrescente) secondo i valori contenuti in `L1`.

NOTA: Si supponga di avere un programma nella forma `clausola(Head, Body)` dove `Body` è una lista di sottogoal.

Soluzione

```
solve([]).
solve([Head|Tail]):-
    solve(Head),
    solve(Tail).

solve(Goal):-
    findall((Goal,Body),clausola(Goal,Body),
                                        L),
    ordina(L,Lord),
    member((Goal,FirstBody), Lord),
    solve(FirstBody).

ordina(L,Lord):-
    conta_sottogoal(L,Nsottogoal),
    sort(L,Nsottogoal, Lord).

conta_sottogoal([],[]).
conta_sottogoal([H|T],[NH|NT]):-
    conta_sottogoal1(H,NH),
    conta_sottogoal(T,NT).

conta_sottogoal1((_,Body),Nsottogoal):-
       conta_sottogoal1(Body,0,Nsottogoal).

conta_sottogoal1([],N,N).
conta_sottogoal1([Head|Tail],Nin,Nout):-
    functor(Head, F, A),
    functor(Temp, F, A),
    member(Temp, Tail), !,
    conta_sottogoal1(Tail,Nin,Nout).
conta_sottogoal1([Head|Tail],Nin,Nout):-
    Ntemp is Nin +1,
    conta_sottogoal1(Tail, Ntemp,Nout).
```

Esercizio

Il Prolog adotta una strategia di ricerca depth-first con backtracking cronologico. Tuttavia, si può realizzare un meta-interprete che modifichi la parte di controllo del Prolog.

Si costruisca un meta interprete per Prolog che adotti la strategia breadth-first.

Per risolvere il problema si tenga traccia in una lista (coda) dei nodi sia dei sottogoal da dimostrare sia del *top goal* con le relative istanziazioni.

Si consideri come esempio il programma seguente:

```
a(1) :- b.
a(2): - c.
b:- b,d.
c.
```

Se il goal è `a(X)` l'interprete standard porterebbe a un loop infinito, mentre nel meta-interprete breadth-first l'evoluzione della coda dei sottogoal dovrebbe essere:

```
[([a(X)], a(X))]
[([b],a(1)),([c],a(2))]
[([c],a(2)),([b,d],a(1))]
[([b,d],a(1)),([true],a(2))]
[([true],a(2)),([b,d,d],a(1))]
[([b,d,d],a(1)),([],a(2))]
[([],a(2)),([b,d,d,d],a(1))]
```

Soluzione

%`bf` costruisce una lista (coda) di sottogoal e del relativo top goal in modo da tenere traccia del ramo dell'albero che si sta percorrendo.

```
metabf(Goal):- bf([([Goal],Goal)],Goal).
```

% Se la `bf` trova una soluzione, restituisce il top goal (con le variabili istanziate) che ha portato al successo

```
bf([([true],Istanza)|_],Istanza).
```

%La `bf` precedente ha trovato una soluzione. La `bf` successiva prosegue la ricerca nei restanti rami dell'albero.

```
bf([([true],_)|S],Goal):-!, bf(S,Goal).
```

% La `findall` espande il primo elemento della coda `[A|B]`.

```
bf([([A|B],Goal1)|S],Goal):-
    findall((BB,Goal1),trova(A,B,BB),L),
    append(S,L,Newlist),
    bf(Newlist,Goal).

trova(A,B,BB):-
    clause(A,Body),
    mixed_append(Body,B,BB).
```

% `mixed_append` concatena una struttura (X,Y) a una %lista Z.

```
mixed_append((X,Y),Z,T):- !,
mixed_append(Y,Z,W),T=[X|W].
mixed_append(X,Y,[X|Y]).
```

Esercizio

Si scriva un meta-interprete che intercetti il meccanismo di unificazione del Prolog e che agisca nel modo seguente:

- nel caso in cui i due termini da unificare siano costanti il metainterprete dovrà verificare l'uguaglianza delle costanti stesse

- nel caso in cui siano entrambe variabili, o variabile/costante o ancora variabile/termine composto il metainterprete dovrà fornire un messaggio contenente la sostituzione effettuata senza peraltro effettuarla e senza verificare l'occur-check.

- nel caso in cui siano entrambi termini composti il metainterprete dovrà richiamarsi ricorsivamente sugli argomenti dei termini stessi

In ogni caso non dovrà essere effettuato un legame delle variabili

Si noti che il meccanismo di unificazione del Prolog non dovrà essere mai utilizzato. Inoltre l'unificazione gestita dal meta-interprete dovrà avere carattere locale al termine da unificare.

Esempio

Si supponga di avere un programma nella forma `clausola(Head,Body)` dove `Body` è una lista di sottogoal:

```
clausola(p(X,Y),[q(X),r(Y)]).
clausola(q(3),[]).
clausola(r(2),[]).
```

e di voler chiamare il metainterprete con il goal `p(4,5).`
Un normale programma Prolog fallirebbe. Il metainterprete deve avere successo producendo i messaggi:

```
X/4 Y/5
X/3 Y/2
```

rispettivamente per `p(X,Y)` e per `q(X)` e `r(Y)`.
Le variabili `X` e `Y` al termine della valutazione non dovranno essere istanziate.

Soluzione

```
check([]).
check([A|Tail]):-
    check(A),
    check(Tail).
```

% `check(Goal)` crea un template con lo stesso funtore e arità del Goal da unificare

```
check(Goal):-
    functor(Goal,F,A),
    functor(Templ,F,A), % template
    clausola_unif(Goal,Templ).
```

% `clausola_unif` utilizza il template per l'unificazione
%con la clausola nel database poi controlla se gli
%argomenti unificano

```
clausola_unif(Goal,Templ):-
    clausola(Templ,Body),
    Goal=..[_|Arg],
    Templ=..[_|Arg1],
    unificano(Arg,Arg1),
    check(Body).
```

% `unificano` controlla che due liste di argomenti
%possano unificare

```
unificano([],[]).
```

% <u>Caso 1</u>: due costanti unificano se sono uguali

```
unificano([H|Tail],[H1|Tail1]):-
    atomic(H),atomic(H1),
    H=H1,!,
    unificano(Tail,Tail1).
```

% Caso 2: dal momento che non si effettuano link di
%variabili una variabile unifica con qualunque termine
%(atomico o composto)

```
unificano([H|Tail],[H1|Tail1]):-
    (var(H);var(H1)),!,
    write('sostituzione '),
    write(H/H1), nl,
    unificano(Tail,Tail1).
```

% Caso 3: due termini composti unificano se unificano i
%funtori e gli argomenti per la verifica degli argomenti
%`unificano/2` si richiama ricorsivamente

```
unificano([H|Tail],[H1|Tail1]):-
    compound(H),compound(H1),
    H=..[Head|Arg],
    H1=..[Head|Arg1],
    unificano(Arg,Arg1),!,
    unificano(Tail,Tail1).
```

Esercizio

Si scriva un meta-interprete che intercetti il meccanismo di unificazione del Prolog e che lo utilizzi solo se il termine da unificare (goal o sottogoal) è una generalizzazione del termine che unifica (testa della clausola).

- Due costanti non unificano mai (anche se sono uguali).

- Nel caso in cui i termini da unificare siano entrambi variabili o il termine da unificare è una variabile e il termine che unifica una costante, l'unificazione avviene normalmente. Si noti che se il termine da unificare è una costante e il termine che unifica una variabile l'unificazione fallisce.

- Nel caso in cui i termini da unificare siano entrambi composti, l'unificazione avviene se i funtori sono identici mentre per l'unificazione degli argomenti il metainterprete dovrà richiamarsi ricorsivamente sugli argomenti dei termini stessi

Esempio:

```
p(X,Y):- q(X),r(2).
p(X,Y):- s(Y).
q(3).
r(2).
s(Y):- r(Y).
```

Si supponga di voler chiamare il metainterprete con il goal `p(A,B).`
`A` e `B`, essendo due variabili, unificano con `X` e `Y`. Viene quindi eseguito `q(X)` che è una generalizzazione di `q(3)` e quindi unifica. Per `r(2)`, invece, non esistono termini meno generali. Quindi la computazione fallisce. Rimane quindi choice point per `p(X,Y)` che viene esplorato e porta ad un ramo di successo. Questa volta infatti `r(Y)` è più generale di `r(2)`.
Lo stesso metainterprete, con il goal `p(2,B)`, fallirebbe subito.

Soluzione

```
unif(true):-!.
unif((A,B)):-
    unif(A),
    unif(B).
unif(Goal):-
    functor(Goal,F,A),
    functor(Templ,F,A),
    clausola_unif(Goal,Templ).

clausola_unif(Goal,Templ):-
    clause(Templ,Body),
    Goal=..[_|Arg],
    Templ=..[_|Arg1],
    unificano(Arg,Arg1),
    Goal = Temp1,
    unif(Body).

unificano([],[]).
unificano([H|Tail],[H1|Tail1]):-
    var(H),!,
    unificano(Tail,Tail1).
unificano([H|Tail],[H1|Tail1]):-
    compound(H),compound(H1),
    H=..[A|Arg],H1=..[A|Arg1],
    unificano(Arg,Arg1),!,
    unificano(Tail,Tail1).
```

Esercizio

Si scriva un meta-interprete per Prolog che utilizzi le clausole del programma oggetto non nell'ordine testuale, ma in base al numero di sottogoal che hanno nel body, in particolare dando maggiore priorità a quelle che hanno minor numero di sottogoal (ne segue che i fatti sono sempre i primi ad essere presi in considerazione). Inoltre, a parità di sottogoal si scelga per prima la clausola che, prima dell'unificazione, presenta un maggior numero di parametri ground.

Ad esempio:

Goal: `a(X,L,3)`

```
a(X,Y,K):- b(X,Y,K).
a(3,Y,3):- c(Y).
a(4,5,3).
```

Il metainterprete deve scegliere, come prima clausola, il fatto `a(4,5,3)`, come seconda clausola `a(3,Y,3)` poiché presenta un solo sottogoal nel body come la prima clausola ma ha un numero di parametri istanziati maggiore della `a(X,Y,K)`.

Per la risoluzione dell'esercizio si supponga di avere a disposizione una clausola `sort_cres(L1,L2,L3)` che ordini in modo crescente la lista `L1` in base ai valori contenuti nella lista `L2` e metta il risultato in `L3` e `sort_decr(L1,L2,L3)` che faccia la stessa cosa in senso decrescente.

Esempio:

`sort_cres([El1,El2,El3],[4,7,1],L3)` `L3` risulta istanziato a `[El3,El1,El2]` mentre in `sort_decr([El1,El2,El3],[4,7,1],L3)` `L3` risulta istanziato a `[El2,El1,El3]`.

Soluzione

```
solve(true):-!.
solve((A,B)):- !, solve(A), solve(B).
solve(A):-
  functor(A,Fun,Arity),
  functor(Templ,Fun,Arity),
  findall([Templ,Body],clause(Templ,Body),L)
,
  sottogoal(L,L1),
  parametri_ground(L,L2),
  sort(L,L1,L2,L3),
  member([H,Body],L3),
  H=A,
  solve(Body).
```

% `sottogoal/2` riceve in ingresso una lista di liste del tipo
% `[H,Body]` e fornisce in uscita una lista corrispondente
% alla prima che contiene il numero di sottogoal di
% ciascun elemento della prima lista

```
sottogoal([],[]).
sottogoal([[_,Body]|Tail],[L|Tail1]):-
    length_and(Body,L),
    sottogoal(Tail,Tail1).
```

% `parametri_ground/2` riceve in ingresso una lista di liste
%del tipo `[H|Body]` e fornisce in uscita una lista
% corrispondente alla prima che contiene il numero di
%parametri ground di ciascun elemento della prima lista

```
parametri_ground([],[]).
parametri_ground([[H,_],Tail],[Num|Tail1]):-
    H=..[_|Parametri],
    ground(Parametri,Num),
    parametri_ground(Tail,Tail1).
```

```
ground(Parametri,Num):-
    ground(Parametri, 0,Num).
ground([],Acc,Acc).
ground([H|Tail],Acc,N):-
    atomic(H),
    Acc1 is Acc +1,
    ground(Tail,Acc1,N).
ground([_|Tail],Acc,N):-
    ground(Tail,Acc,N).
```

%`sort/4` ordina la lista `L` in ordine crescente secondo i %valori contenuti nella lista `L2` e decrescente secondo i %valori della lista `L1`. Naturalmente è necessario ordinare %prima la lista `L` secondo i valori della lista `L2` e poi %ordinare il risultato secondo i valori della lista `L1`.

```
sort(L,L1,L2,L3):-
    sort_decr(L,L2,Ldecr),
    sort_decr(L1,L2,L1decr),
    sort_cresc(Ldecr,L1decr, L3).
```

%`length_and/2` calcola la lunghezza di una struttura and.

```
length_and((A,B),L):-
    length_and(B,L1),!,
    L is L1+1.
length_and(A,1).
```

Esercizio

Si scriva un meta-interprete in grado di riconoscere i predicati `is/2` e di valutare se il predicato stesso deve essere sospeso o eseguito. Si considerino solo espressioni (a sinistra di `is`) a due argomenti. Nel caso in cui i due argomenti a sinistra del predicato non siano sufficientemente istanziati, il predicato deve essere sospeso ponendolo in un'apposita lista; in caso contrario viene eseguito normalmente.

Ad esempio `A is 2+3` viene sempre eseguito mentre `A is B+C` viene sospeso se una delle variabili `B` o `C` non è istanziata.

Al termine della dimostrazione, il metainterprete deve controllare se nella lista dei predicati sospesi ne compaiono alcuni che possono essere eseguiti perché sufficientemente istanziati. Se la valutazione del goal termina con successo, ma la lista dei predicati sospesi contiene ancora predicati non sufficientemente istanziati, il metainterprete restituisce la lista dei goal sospesi.

Soluzione

Hp: si hanno a disposizione fatti del tipo `clausola(Head,Body)` dove `Body` è una lista di sottogoal. `interi/3` ha come parametri la lista dei goal da eseguire, la lista dei goal sospesi (solo `is/2`) in ingresso e la lista dei goal sospesi in uscita

```
interi([],[],_):-!.
```

Nel caso in cui si sia giunti al termine dell'esecuzione di un goal e la lista dei predicati sospesi non sia vuota, si deve controllare se tale lista contiene predicati che possono essere eseguiti. In questo caso `lista_tutti_is` fallisce facendo fallire anche `interi` che però ha un choice point che permette di eseguire i predicati divenuti sufficientemente istanziati. In caso contrario vengono stampati i goal sospesi e l'esecuzione termina.

```
interi([],L,_):-
    lista_tutti_is(L),!,
    write('Lista di goal sospesi'),nl,
    stampa(L).
interi([],L,_):-
    interi(L,[],_).
interi([Goal|Altri],L2,L1):- !,
    interi(Goal,L2,Ltemp),
    interi(Altri,Ltemp,L1).
```

Caso in cui il predicato è `is/2` ma gli argomenti sono sufficientemente istanziati.

```
interi(Goal,Lgoal,Lnuovi):-
    is_solvable(Goal),!,
    call(Goal),
    Lgoal=Lnuovi.
```

Caso in cui il predicato è `is/2` ma gli argomenti non sono sufficientemente istanziati.

```
interi(Goal,Lgoal,Lnuovi):-
    is_not_solvable(Goal),!,
    append(Lgoal,[Goal],Lnuovi).
```

Caso normale

```
interi(Goal,Lgoal,Lgoal1):-
    clausola(Goal,Body),
    interi(Body,Lgoal,Lgoal1).

is_solvable(Goal):-
    Goal=..[is,_,B],
    B=..[_,L,K],
    nonvar(L),nonvar(K),!.

is_not_solvable(Goal):-
    Goal=..[is,_,B],
    B=..[_,L,K],
    (var(L); var(K)).

lista_tutti_is([]).
lista_tutti_is([Lh|Lt]):-
    is_not_solvable(Lh),!,
    lista_tutti_is(Lt).
lista_tutti_is([Lh|_]):-
    is_solvable(Lh),fail.

stampa([]).
stampa([H|T]):-
    write(H),nl,
    stampa(T).
```

Esercizio

Si scriva un programma *preprocessore* per Prolog che trasforma un programma P in un programma P' tramite la tecnica dell'unfolding. Ogni clausola del programma P è scritta nel data base Prolog sotto forma di fatto `clausola(Head, Body)` dove `Body` è una lista di sottogoal. La trasformazione delle clausole avviene nel modo seguente:
In ogni clausola `C_old` di P, per ogni sottogoal S del body, si cerca la clausola C1 la cui testa unifica con il sottogoal e si sostituisce al sottogoal S il body di C1. La nuova clausola `C_new` viene aggiunta al data base. Si noti che il procedimento non è ricorsivo e si ferma dopo un passo di unfolding. Se si hanno più clausole C1 che unificano, verranno generate diverse alternative `C_new`*i*. Se non esiste una clausola la cui testa unifica con il sottogoal S non si aggiunge una nuova clausola `C_new`.

Esempio: Dato il programma

```
clausola(p(X),[q(X),r(X)]).
clausola(q(X),[a(X)]).
clausola(q(3),[]).
clausola(r(3),[]).
clausola(a(X),[b(X,Y),r(Y)]).
clausola(b(3,4),[]).
```

Verrà trasformato nel programma:

```
clausola_new(p(3),[a(3)]).
clausola_new(p(3),[]).
clausola_new(q(X),[b(X, Y), r(Y)]).
clausola_new(q(3),[]).
clausola_new(r(3),[]).
clausola_new(b(3,4),[]).
```

Soluzione

```
unfolding:-
    clausola(Head,Body),
    unfold(Body, Body1),
    assert(clausola1(Head,Body1)),
    fail.
unfolding.

unfold([],[]).
unfold([A|Body],Body2):-
    clausola(A,B),
    unfold(Body,Body1),
    append(B,Body1,Body2).
```

Esercizio: Valutazione parziale

Si scriva un meta-interprete che dato un programma P in Prolog puro e senza predicati ricorsivi e un goal G, trasformi P in un programma P' più efficiente e specializzato per G. P' è ottenuto da P, semplificando ogni clausola C della forma A:-Body di P in cui A ha funtore identico a G nel modo seguente:

Se G non unifica con C, C viene eliminata;

Se G unifica con C si procede all'unificazione di G con A e si va a valutare il Body di C; se la valutazione ha successo C si trasforma in G':-true, dove G' è il goal G a cui sono state applicate le opportune unificazioni; se la valutazione fallisce si possono distinguere due casi:

- si sono incontrati durante la valutazione alcuni sottogoal $B'_1,..B'_N$ per cui non esiste alcuna definizione in P; allora C si trasforma in $G':-B'_1,..B'_N$
- altrimenti si elimina la clausola C dal programma P.

Esempio:

Se il programma P è:

```
a(X,Y):-b(Y).
a(X,Y):-b(Y),c(X).
a(X,Y):-r(X).
a(2,Y).
b(1).
a(X,Y):-b(2),b(1).
r(X):-c(2).
```

La valutazione parziale di P per il goal `a(1,Z)` produrrà il nuovo programma:

```
a(1,1):-true.
a(1,1):-c(1).
a(1,Y):-c(2).
```

Soluzione

```
partial(Goal):-
    premises(Goal,[Goal],P),
    listtoand(P,AP),
    write((Goal:-AP)), nl,
    fail.
partial(_).
```

%significato degli argomenti di `premises/3` :
% il primo è il goal o i goals correnti di input
% il secondo esprime i goal di output già calcolati
% il terzo i goals di output da calcolare

```
premises((X,Y),Old,P):-!,
    premises(X,Old,P1),
    append(P1,Old,New),
    premises(Y,New,P2),
    append(P1,P2,P).
premises(true,_,[true]):-!.
premises(Goal,Z,P):-
    clause(Goal,Body),
    premises(Body,Z,P).
premises(X,Z,[X]):-
    not(definita(X)).

definita(X):-
    functor(X,Fun,Ar),
    functor(Temp,Fun,Ar),
    clause(Temp,_).
```

% listtoand trasforma una lista in una struttura a and es.
`[1,2,3]` diventa `(1,2,3)`

```
listtoand([H|T],And):-listtoand1(T,And,H).
listtoand1([],L,L).
listtoand1([H|T],(E,And),E):-
listtoand1(T,And,H).
```

Esercizio

Si scriva un meta-interprete in PROLOG che risolva programmi logici con vincoli scritti nel modo seguente:

`clausola(Head,Body,Constraints).`

dove `Constraints` è una lista di relazioni del tipo:

`A = B` o `A > B`.

A può essere:

- una costante numerica;
- una variabile legata a costante numerica;
- una variabile libera.

mentre B solo:

- una costante numerica;
- una variabile legata a costante numerica.

Body è una lista di Goals.

Il meta-interprete dovrà avere tre parametri:

`solve(Goal,Old_Constraints, New_Constraints)`

di cui:

Goal è la lista dei goals da processare;

`Old_Constraints` è l'insieme corrente di vincoli

`New_Constraints` sono i nuovi vincoli ottenuti aggiornando i vecchi vincoli con quelli contenuti nella clausola utilizzata per l'unificazione. Le unificazioni sono trattate come vincoli di uguaglianza.

Esempio di funzionamento: per il programma:

```
clausola(a(X),[b(X)],[X>5]).
clausola(b(X),[],[X=3]).
```

il meta-interprete produrrà un fallimento per il goal `a(X)` perchè i due vincoli sono incompatibili.

mentre per il programma:

```
clausola(a(X),[b(X)],[X>5]).
clausola(b(X),[],X>7).
```

il meta-interprete produrrà successo con `X>7`

(Composizione di vincoli compatibili `X>5` e `X>7`).

Soluzione

```
solve([],C,C):-!.
solve([Goal|Rest],Old_Cons,New_Cons):-
    solve(Goal,Old_Cons,Temp_Cons),!,
    solve(Rest,Temp_Cons,New_Cons).
solve(Goal,Old_Cons,New_Cons):-
    clausola(Goal,Body,Cur_cons),
    subset_c(Old_Cons,Cur_cons,Touched,[]),
    merge_c(Touched,Cur_cons,Temp_Cons,[]),
    subtract(Old_Cons,Touched,Untouched),
    append(Untouched,Temp_Cons,Partial_Cons),
    solve(Body,Partial_Cons,New_Cons).
```

%seleziona i vecchi vincoli che condividono variabili con i
%nuovi vincoli

```
subset_c(_,[],Acc,Acc).
subset_c(Old,[Currh|Currt],Touch,Acc):-
    occurrence(Old,Currh,Tou),
    append(Tou,Acc,Touched),
    subset_c(Old,Currt,Touch,Touched).

occurrence([],_,[]).
occurrence([Oldh|_],Current,[Oldh]):-
    same_variable(Oldh,Current),!.
occurrence([_|Oldt],Current,Tou):-
    occurrence(Oldt,Current,Tou).

same_variable(X=A,Y=B):- var(X),X==Y.
same_variable(X=A,Y>B):- var(X),X==Y.
same_variable(X>A,Y=B):- var(X),X==Y.
same_variable(X>A,Y>B):- var(X),X==Y.
```

% `merge_c` è una clausola tail recursive che
% ritorna la lista di vincoli aggiornata

```
merge_c(_,[],A,A):-!.
merge_c(Old_C,[Consh|Const],New_C,New1_C):-
    check_constraint(Old_C,Consh,Temp_C),
    append(Temp_C,New1_C,New2_C),
    merge_c(Old_C,Const,New_C,New2_C).
```

% check_constraint si compone di molti casi:
%1) Condizione di terminazione: quando il vincolo Consh %non trova corrispondenza nella lista dei vecchi vincoli e %non si è rilevata durante l'analisi una inconsistenza, %Consh viene aggiunto (inalterato) alla lista dei nuovi %vincoli

```
check_constraint([],Consh,[Consh]).
```

% 2) Check di due uguaglianze in cui due termini sono %uguali allo stesso valore: qualunque sia il tipo (variabile %istanziata, non istanziata o costante) di A e di X è %necessario per non avere inconsistenze che A unifichi %con X. In caso contrario si ha il fallimento

```
check_constraint([A=B|_],X=D,[A=B]):-
    A==X,B=D,!.
check_constraint([A=B|_],X=D,_):-
    A==X,!,fail.
```

% 3) Check di una disuguaglianza e un'uguaglianza: è %necessario che i vincoli siano verificati solo se il nome %delle variabili è uguale (da cui il controllo A==D). Se il %check sui vincoli ha successo viene mantenuto il vincolo %più stretto (uguaglianza)

```
check_constraint([A>B|_],D=C,[A=C]):-
    var(A),A==D,C>B,!.
check_constraint([A>B|_],D=C,_):-
    var(A),A==D,!,fail.
check_constraint([A=B|_],D>C,[A=B]):-
    var(D),A==D,B>C,!.
check_constraint([A=B|_],D>C,_):-
    var(D),A==D,!,fail.
```

% 4) Check di due disuguaglianze: è necessario che i %vincoli siano verificati solo se il nome delle variabili è %uguale (da cui il controllo A==D). Se il check sui vincoli %ha successo viene mantenuto il vincolo più stretto.

```
check_constraint([A>B|_],D>C,[A>B]):-
    var(A), A==D,B>C,!.
check_constraint([A>B|_],D>C,[A>C]):-
    var(A), A==D,C>B,!.
```

% Nel caso in cui il Vincolo considerato non trovi %corrispondenza con la testa della lista e non si siano %rilevate inconsistenze si richiama ricorsivamente %check_constraint con la coda.

```
check_constraint([_|Tail],Vincolo,New):-
    check_constraint(Tail,Vincolo,New).
```

Esercizio
Si scriva un preprocessore in grado di agire su un programma P e di trasformarlo in un programma P' secondo alcune regole contenute nel database CG sotto forma di *clausole con guardia*. In particolare, esistono due tipi di clausole con guardia: quelle che semplificano il programma e quelle che aggiungono al programma ulteriori informazioni.

Tramite il predicato `clause_simpl(Head, Guard,Body)` si recuperano dal database le clausole con guardia di semplificazione, mentre con il predicato `clause_unisci(Head,Guard,Body)` si recuperano dal database le clausole con guardia di unione. Si supponga che questi predicati abbiano successo anche con tutti i parametri non istanziati.
I parametri della `clause_simpl/3` e della `clause_unisci/3` hanno il seguente significato: `Head` è una lista di atomi, `Guard` è una lista di sottogoal da dimostrare e `Body` è una lista di fatti.

Dopo aver recuperato dal database, tramite la `clause_simpl/3`, una clausola di semplificazione il metainterprete deve cercare, per ciascun elemento della lista `Head`, se esiste una clausola nel programma P la cui testa unifica con l'elemento stesso e procede all'unificazione. Successivamente, viene verificata la guardia `Guard`. Se la guardia ha successo allora al programma P vengono tolte tutte le clausole la cui testa unificava con gli elementi di `Head` e vengono aggiunti tutti i fatti contenuti in `Body`.

Dopo aver recuperato dal database, tramite la `clause_unisci/3`, una clausola di unione il

metainterprete deve cercare, per ciascun elemento della lista `Head` se esiste una clausola nel programma P la cui testa unifica con l'elemento stesso e si procede all'unificazione. Viene poi verificata la guardia `Guard`. Se la guardia ha successo allora al programma P vengono aggiunti tutti i fatti contenuti in `Body`.

Esempio: Sia dato il programma P:

```
a(X):-b(X),c(X).
p(X):-q(X).
q(1).
b(2).
```

supponiamo che `clause_simpl/3` abbia successo in CG per la seguente istanziazione delle variabili:
`Head/[a(X)],Guard/[p(X)],Body/[a(1)]`
e che `clause_unisci/3` abbia successo per la seguente istanziazione delle variabili:
`Head/[b(X),p(2)],Guard/[q(1),b(X)],Body/[q(2)]`
`Head/[a(1)],Guard/[q(2)],Body/[a(2)]`

Il metainterprete, dopo aver recuperato la prima `clause_simpl` verifica se nel programma esiste una clausola la cui testa unifica con l'unico elemento della lista `Head`. La clausola esiste e quindi si procede alla verifica della guardia risolvendo `p(X)`. A questo punto viene ritrattata la clausola `a(X):-b(X),c(X).` e asserito il fatto `a(1)`.

Il metainterprete, dopo aver recuperato la prima `clause_unisci` verifica se nel programma esistono clausole le cui teste unificano con i due elementi della lista `Head`. Le clausole esistono, viene fatta l'unificazione e quindi si procede alla verifica della guardia risolvendo `q(1)` e `b(X)` con `X/2`. A questo punto viene asserito il fatto `q(2)`.
Allo stesso modo si procede per la terza `clause_unisci/3` tenendo in considerazione il fatto che nel programma sono state asserite delle clausole ed, in particolare, la clausola `a(1)`.

Soluzione

```
guardia:-
    clausola_simpl,
    clausola_unisci,
    fail.
guardia.

clausola_simpl:-
    clause_simpl(Head,Guardia,Body),
    controlla(Head),
    verifica(Guardia),
    sostituisci(Head,Body).

clausola_unisci:-
    clause_unisci(Head,Guardia,Body),
    controlla(Head),
    verifica(Guardia),
    unisci(Body).
```

```
controlla([]):-!.
controlla([Head1|Altre]):-
    clause(Head1,_),
    controlla(Altre).

verifica([]):-!.
verifica([Guardia|Altre]):-
    call(Guardia),
         verifica(Altre).

sostituisci(Head,Body):-
    ritratta(Head),
    asserisci(Body).

unisci(Body):-
    asserisci(Body).

ritratta([]):-!.
ritratta([H|Altre]):-
    clause(H,Body),
    retract((H:-Body)),fail.

ritratta([H|Altre]):-
    ritratta(Altre).

asserisci([]):-!.
asserisci([H|Altre]):-
    assert(H),
    asserisci(Altre).
```

Esercizio

Si scriva un metainterprete in grado di agire su un programma P formato da clausole del tipo:

```
<Head>:- <Body>.
```

dove:

`<Head>` è un atomo della forma:

`p(t1,...tn, pr)` dove `t1,...tn` sono gli argomenti, mentre `pr` è una costante intera che indica la priorità.

`<Body>` è una congiunzione di goals del tipo:

```
- maggiore(Goal,prvalue)
- minore(Goal,prvalue)
- uguale(Goal,prvalue)
- Goal
```

dove `Goal` è un goal Prolog standard.

Il significato di questi goal è il seguente:

```
- maggiore(goal(a1,...an),prvalue)
```

unifica con ogni clausola del tipo:

`goal(b1,...bn,pr)` se gli argomenti `ai` e `bi` unificano e `pr` è maggiore di `prvalue`.

Analogamente negli altri casi.

Nel caso in cui il funtore principale del goal non sia né maggiore, né minore né uguale, si dovrà procedere alla risoluzione standard ignorando `prvalue`.

Soluzione

```
priorita(true).

priorita((A,B)):-!,
   priorita(A),priorita(B).
```

% goal del tipo maggiore(Goal,prvalue),
% minore(Goal,prvalue) o uguale(Goal,prvalue)

```
priorita(Goal_prior):-
   Goal_prior=..[X,Goal,Priorita],
   (X=maggiore;X=minore;X=uguale),!,
   Goal=..[P|Args],
   append(Args,[_Priorita],ArgsP),
   GoalP=..[P|ArgsP],
   findall((GoalP,Body),
      clause(GoalP,Body),L),
   seleziona(X,L,L1,Priorita),
   member((GoalP,B),L1),
   priorita(B).
```

% goal senza priorità

```
priorita(Goal):-
   Goal=..[P|Args],
   append(Args,[_Priorita],ArgsP),
   GoalP=..[P|ArgsP],
   clause(GoalP,Body),
   priorita(Body).

seleziona(X,L,L1,Priorita):-
   X=maggiore,!,
   seleziona_max(L,L1,Priorita).

seleziona(X,L,L1,Priorita):-
   X=minore,!,
   seleziona_min(L,L1,Priorita).

seleziona(X,L,L1,Priorita):-
```

```
    X=uguale,!,
    seleziona_uguale(L,L1,Priorita).

seleziona_max([],[],_):-

seleziona_max([H|Tail],[H|Tail1],
    Priorita):-
    pr(H,P),
    P>Priorita,!,
    seleziona_max(Tail,Tail1,Priorita).

seleziona_max([H|Tail],Tail1,
    Priorita):-
    seleziona_max(Tail,Tail1,Priorita).
```

% analogamente per minore e uguale.

```
pr((H,_B),P):-
    H=..L,
    ultimo_arg(L,P).

ultimo_arg([P],P):-!.
ultimo_arg([H|T],P):-
    ultimo_arg(T,P).
```

Esercizio

Supponendo di avere un database Prolog composto da fatti del tipo:

```
regola(<nome-modulo>,Testa <- Corpo).
```

dove in Corpo possono apparire goal del tipo:

```
[<lista nome-moduli>]<<Goal
```

(che richiede la dimostrazione del `Goal` utilizzando solo le regole nei moduli della lista di moduli specificata)
si scriva un meta-interprete in grado di interpretare tali programmi e che riporti in uscita, in caso di dimostrazione con successo, anche la lista di moduli le cui regole sono state effettivamente utilizzate per la dimostrazione.

Esempio di programma:

```
regola(m1, a(X)<-[m2]<<c(X))
regola(m1, c(3)<-true)
regola(m2, c(1)<-[m3,m4]<<d(2))
regola(m3, d(3)<-true)
regola(m4, d(2)<-true)
```

La soluzione (unica) generata dal meta-interprete per il goal `[m1]<< a(X)` sarà yes con `X` legato ad `1` e con lista di moduli effettivamente usati: `[m1,m2,m4]`.

Soluzione:

```
solve(true, M, M).

solve((A,B), ModOld, ModNew):-
    solve(A, ModOld, ModTemp),
    solve(B, ModTemp, ModNew).

solve(LL<<G, ModIn, ModOut):- !,
    member(Mod, LL),
    regola(Mod, G<-Body),
    solve(Body, [Mod|ModIn], ModOut).

solve(G, ModIn, ModOut):-
    regola(X,G<-Body),
    solve(Body, [X|ModIn], ModOut)
```

Esercizio

Si supponga di avere un database Prolog composto da regole e fatti. Le regole sono del tipo:

```
regola(<nome-oggetto>,Testa if Corpo),
```

dove `if` è un opportuno operatore e dove in `Corpo` possono apparire solo goal del tipo:

```
nome-oggetto<-Goal
self <- Goal
local <- Goal
true
```

I fatti contenuti nel data base sono del tipo:

`super(<nome-og1>,<nome-og2>)` che indicano che l'oggetto con nome `<nome-og1>` è una superclasse di `<nome-og2>`.

Si scriva un meta-interprete in grado di interpretare tali programmi. Il meta interprete è composto da un predicato

```
solve(Goal, Self, OggettoCorrente)
```

e lavora nel modo seguente:

- il goal `<nome-oggetto><-Goal` è risolto utilizzando le regole in `<nome-oggetto>` e nelle sue superclassi per risolvere `Goal` (legando `Self` a `<nome-oggetto>`).

- il goal `self <-Goal` è risolto usando l'oggetto a cui è legata la variabile `Self` (e le sue superclassi) per la risoluzione di `Goal`.

- il goal `local<-Goal` è risolto utilizzando solo le regole dell'oggetto corrente, (variabile `OggettoCorrente`) cioè quello in cui si trova la clausola contenente nel `Corpo` `local<-Goal`.

Esempio di programma:

```
super(og2, og1).
super(og3, og2).
regola(og3, a(X) if  self <- d(X))
regola(og3, b(X) if  local <- f(X))
regola(og2, c(X) if og1<- a(X))
regola(og2, c(X) if og1<- b(X))
regola(og1, d(3)<-true)
regola(og1, f(2)<-true)
```

La soluzione (unica) generata dal meta-interprete per il goal `og2<-c(X)` sarà `yes` con `X` legato a `3`, (`Self` legato a `og2` e poi a `og1` e `OggettoCorrente` legato prima a `og2`, poi a `og1`, `og3` e quindi ancora a `og1`) mentre fallirà la seconda alternativa per `c(X)` poiché in `og3` (local) non esiste una definizione locale per il predicato f.

Soluzione:

```
solve(true, _, _).
solve((A,B), Self, Ogg):-
    solve(A, Self, Ogg),
    solve(B, Self, Ogg).

solve(self<-Goal, Self, Ogg):- !,
    solve(Goal, Self, Self).

solve(local<-Goal, Self, Ogg):- !,
    solve_ogg(Goal, Self, Ogg).

solve(Ogg<-Goal, Self, _Ogg1):-
    solve(Goal,Ogg,Ogg).

solve(Goal, Self, Ogg):-
    atomic(Goal),
    regola(Ogg, Goal if Body),
    solve(Body, Self,Ogg).

solve(Goal, Self, Ogg):-
    superclasse(Super,Ogg),
    solve(Goal, Self, Super).

solve_ogg(Goal, Self, Ogg):-
    atomic(Goal),
    regola(Ogg, Goal if Body),
    solve(Body, Self, Ogg).

superclasse(Super,Ogg):-
    super(Super,Ogg).

superclasse(Super,Ogg):-
    super(Super1, Ogg),
    superclasse(Super,Super1).
```

Esercizio

Si scriva un metainterprete in grado di risolvere clausole generali. Si supponga di avere un database di clausole generali del tipo:

```
clausola([A1, A2,...An,not(B1),not(B2),...,not(Bm)]).
```

dove i letterali della lista sono in or. Si supponga di dovere risolvere un goal G, sotto forma di lista di letterali in and. Per la risoluzione si utilizzi la strategia *linear-input*.

Soluzione:

%come primo passo, viene negato il goal. Poi parte la %risoluzione.

```
solve(Goal):-
    nega_goal(Goal,Goal1),
    solve1(Goal1).
```

%La `solve1` seleziona un letterale dal risolvente e cerca %se esiste una clausola nel database che contiene lo %stesso letterale negato. In tal caso cancella il letterale %dal risolvente, e crea il nuovo risolvente.
% Uso `delete1` per cancellare tutte le occorrenze di `NA`
% Si suppone di utilizzare una regola di selezione che
% risolva i goal da sinistra a destra

```
solve1([]).
solve1([A|Tail]):-
    clausola(L),
    nega(A, NA),
    member(NA, L),
    delete1(NA, L, L1),
    append(L1, Tail,L2),
    solve1(L2).
```

```
nega_goal([],[]).
nega_goal([H|Tail],[HN|Tail1]):-
    nega(H,HN),
    nega_goal(Tail,Tail1).

nega(not(X), X):-!.
nega(X, not(X)):-!.
```

Esercizio

Si scriva un metainterprete che realizzi una strategia di risoluzione abduttiva.
Si supponga di avere programmi composti da clausole del tipo:

```
clausola(<Head>, <Body>).
clausola(fail, <Body>).
```

dove `<Body>` è una lista di atomi. Le clausole del tipo `clausola(fail, <Body>).` sono dette vincoli di integrità e asseriscono che gli atomi all'interno del `Body` (supposti sempre ground) non possono essere tutti veri contemporaneamente.

Ad ogni passo di risoluzione, il metainterprete prova a risolvere il Goal corrente nel modo classico. Se nel database Prolog trova una clausola la cui testa unifica con il Goal, questo viene risolto valutando il Body della clausola. Se invece non esiste una definizione per il Goal nel database del programma, allora si controlla se il predicato è un abducibile, ossia se esiste un fatto di tipo

```
abduc(Pred).
```

che ha come argomento un predicato con lo stesso funtore e arità del Goal in esame.
Se tale predicato è abducibile, viene supposto vero ed è possibile proseguire la risoluzione previo controllo dei vincoli di integrità. Si cerca pertanto il vincolo di integrità che contiene nel Body il predicato abducibile (se esiste) e si controlla che fallisca la derivazione per almeno uno dei restanti sottogoal del vincolo di integrità.

ESEMPIO: Supponendo di avere un programma del tipo

```
clausola(p(X),[q(X)]).
clausola(r(X),[s(X)]).
clausola(fail,[q(a),r(a)]).

abduc(q(_)).
```

La query `solve(p(X))` avrà successo. Infatti, `p(X)` viene risolto normalmente valutando `q(X)`. Per `q(X)` non c'è una definizione nel programma ed è un predicato adbucibile. Pertanto il goal `q(X)` può essere considerato vero (abdotto) a patto che sia rispettato il vincolo di integrità che lo contiene nel suo `Body`. Affinchè tale vincolo sia soddisfatto, è necessario che `r(a)` fallisca. Deve pertanto partire una risoluzione per `r(a)`. `r(a)` fallisce in quanto non c'è una definizione nel programa per `s(a)` e non è un predicato adbucibile.
Se invece nel programma ci fosse stata la clausola

```
clausola(s(a),[]).
```

la query `solve(p(X))` sarebbe fallita. Essendo vero il predicato `r(a)` non era possibile abdurre `q(a)` e quindi `q(X)`.

Soluzione:

% Risoluzione normale

```
solve([]).
solve([H|Tail]):-solve(H),solve(Tail),! .
solve(Goal):-
    clausola(Goal, Body),
    solve(Body).
```

% Non è possibile risolvere il Goal con la risoluzione classica, pertanto si controlla se il predicato è abducibile e si verifica il vincolo di integrità che lo contiene nel suo Body.

```
solve(Goal):-
        not clausola(Goal, Body),
    abducibile(Goal),
    checkconstraint(Goal).

abducibile(Goal):-
    functor(Goal,F,A),
    functor(Atomo,F,A),
         abduc(Atomo).

checkconstraint(Goal):-
    clausola(fail,Lista),
    in(Goal,Lista, Lista_fallim),!,
    not solve(Lista_fallim).
checkconstraint(_).

in(Goal,[],_):-!,fail.
in(Goal,[Goal|Tail],Tail):-!.
in(Goal,[H|Tail],[H|Tail1]):-
    in(Goal,Tail,Tail1).
```

Esercizio

Si scriva un metainterprete per Prolog che applichi un procedimento induttivo. Si supponga di avere a disposizione un set di esempi positivi (predicati ground ad arità 1) che vengono dati come ingresso al meta-interprete, e un set di fatti ground (sempre ad arità 1). Per ciascun esempio positivo, il metainterprete deve costruire delle clausole la cui testa è un esempio positivo e il cui body contiene tutti quei fatti che predicano sulla medesima costante dell'esempio positivo. A questo punto generalizza sostituendo alla costante una variabile.

Ad esempio, si supponga di avere un set di esempi positivi: `E+ = [p(a), p(c), q(b)]`
e un insieme di fatti ground:

```
r(a).
m(b).
r(c).
m(c).
s(b).
```

Si supponga, per semplicità di poter utilizzare un predicato `clausola(Head,Body)` che ha successo anche se `Head` non è istanziata.

Il meta interprete deve costruire le seguenti clausole:

`p(a):- r(a).` e generalizzando `p(X):- r(X).`

`p(c):- r(c),m(c).` e generalizzando
`p(Y):- r(Y), m(Y).`

`q(b):- m(b),s(b).` e generalizzando
`q(Z):- m(Z), s(Z).`

Soluzione

```
induz([]).
induz([E|Tail]):-
    E=..[_,X], % Esempi positivi ad arità 1
    findall(H, pred_suX(H, X), Body_new),
    sost_var(E,Body_new, New_Head, New_Body),
    listtoand(New_Body, New_Body_and),
    assert((New_Head :- New_Body_and)),
    induz(Tail).

pred_suX(Head, X):-
    clausola(Head, true),
    Head=..[_,X].

sost_var(Head,Body, Head_new, Body_new):-
    Head=..[X, _],
    Head_new=..[X, Z],
    scorri_body(Body, Z, Body_new).

scorri_body([],_,[]).
scorri_body([G|Tail],Z, [G_new|Tail1]):-
    G=..[X, _],
    G_new=..[X, Z],
    scorri_body(Tail,Z,Tail1).

listtoand([H|T],And):-listtoand1(T,And,H).
listtoand1([],L,L).
listtoand1([H|T],(E,And),E):-
listtoand1(T,And,H).
```