Declarative Programming

7: inductive reasoning
Inductive reasoning: overview

Given

B: background theory (clauses of logic program)
P: positive examples (ground facts)
N: negative examples (ground facts)

Find a hypothesis H such that

H “covers” every positive example given B
∀ p ∈ P: B ∪ H ⊨ p

H does not “cover” any negative example given B
∀ n ∈ N: B ∪ H ⊭ n
Inductive reasoning: relation to abduction

Given a theory \( T \) and an observation \( O \), find an explanation \( E \) such that \( T \cup E \models O \).

Try to adapt the abductive meta-interpreter:

\[
\text{inducible/1} \text{ defines the set of possible hypothesis}
\]

\[
\begin{align*}
\text{induce}(E,H) & : - \\
& \text{induce}(E,[],H).
\end{align*}
\]

\[
\begin{align*}
\text{induce}(\text{true},H,H).
\end{align*}
\]

\[
\begin{align*}
\text{induce}((A,B),H_0,H) & : - \\
& \text{induce}(A,H_0,H_1), \text{induce}(B,H_1,H).
\end{align*}
\]

\[
\begin{align*}
\text{induce}(A,H_0,H) & : - \\
& \text{clause}(A,B), \text{induce}(B,H_0,H).
\end{align*}
\]

\[
\begin{align*}
\text{induce}(A,H_0,H) & : - \\
& \text{element}((A:-B),H_0), \text{induce}(B,H_0,H).
\end{align*}
\]

\[
\begin{align*}
\text{induce}(A,H_0,[(A:-B)|H]) & : - \\
& \text{inducible}((A:-B)), \text{not}(\text{element}((A:-B),H_0)), \text{induce}(B,H_0,H).
\end{align*}
\]
Inductive reasoning: relation to abduction

Listing all inducible hypothesis is impractical. Better to **systematically search** the **hypothesis space** (typically large and possibly infinite when functors are involved).

Avoid **overgeneralization** by including **negative examples** in search process.
Inductive reasoning:
a hypothesis search involving successive generalization and specialization steps of a current hypothesis

Ground fact for the predicate of which a definition is to be induced that is either true (+ example) or false (- example) under the intended interpretation

Example

<table>
<thead>
<tr>
<th>Action</th>
<th>Hypothesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ p(b, [b])</td>
<td>add clause p(X,Y).</td>
</tr>
<tr>
<td>- p(x, [])</td>
<td>specialize p(X, [V</td>
</tr>
<tr>
<td>- p(x, [a,b])</td>
<td>specialize p(X, [X</td>
</tr>
<tr>
<td>+ p(b, [a,b])</td>
<td>add clause p(X, [X</td>
</tr>
</tbody>
</table>
Generalizing clauses: \(\Theta\)-subsumption

A clause \(c_1\) \(\Theta\)-subsumes a clause \(c_2\) \iff \exists\ \text{a substitution } \Theta \text{ such that } c_1\Theta \subseteq c_2

\begin{align*}
\text{element}(X,V) & : \text{element}(X,Z) \\
& \Theta\text{-subsumes} \\
\text{element}(X,[Y|Z]) & : \text{element}(X,Z) \\
& \text{using } \Theta = \{V \rightarrow [Y|Z]\}
\end{align*}

\begin{align*}
\text{a}(X) & : \text{b}(X) \\
& \Theta\text{-subsumes} \\
\text{a}(X) & : \text{b}(X), \text{c}(X). \\
& \text{using } \Theta = \text{id}
\end{align*}
Generalizing clauses:

\( \theta \)-subsumption versus \( \models \)

\( \text{H1 is at least as general as H2 given B} \iff \)

\( \text{H1 covers everything covered by H2 given B} \)

\( \forall p \in P: B \cup H2 \not\models p \Rightarrow B \cup H1 \not\models p \)

\( B \cup H1 \not\models H2 \)

\( \text{clause c1 } \theta\text{-subsumes c2 } \Rightarrow c1 \not\models c2 \)

The reverse is not true:

\[
\begin{array}{l}
\text{a}(X) :- b(X). \% c1 \\
p(X) :- p(X). \% c2
\end{array}
\]

\( c1 \not\models c2 \), but there is no substitution \( \theta \) such that \( c1\theta \subseteq c2 \)
Generalizing clauses: 
**testing for θ-subsumption**

A clause \(c1\) θ-subsumes a clause \(c2\)  
\[\iff \exists \ \text{a substitution } \theta \text{ such that } c1\theta \subseteq c2\]

no variables substituted by \(\theta\) in \(c2\):  
**testing for θ-subsumption amounts to testing for subset relation (allowing unification) between a ground version of \(c2\) and \(c1\)**

\[
\text{theta_subsumes}((H1:-B1),(H2:-B2)) :- 
\text{verify}((\text{ground}((H2:-B2)),H1=H2,\text{subset}(B1,B2))).
\]

\[
\text{verify}(\text{Goal}) :- 
\text{not(not(call(Goal)))).
\]

\[
\text{ground}(\text{Term}) :- 
\text{numbervars}(\text{Term},0,N).
\]

prove Goal, but without creating bindings
Generalizing clauses: 
**testing for $\Theta$-subsumption**

A clause $c_1$ $\Theta$-subsumes a clause $c_2$ 
$\iff \exists$ a substitution $\Theta$ such that $c_1\Theta \subseteq c_2$

bodies are lists of atoms

?- theta_subsumes((element(X,V):- []),  
                 (element(X,V):- [element(X,Z)]))).
yes.

?- theta_subsumes((element(X,a):- []),  
                 (element(X,V):- [])).  
no.
Generalizing clauses: generalizing 2 atoms

A clause $c_1 \theta$-subsumes a clause $c_2$ $\iff \exists \theta$ a substitution $\theta$ such that $c_1 \theta \subseteq c_2$

\begin{align*}
\text{a1} & \quad \text{element}(1, [1]). \\
\text{a2} & \quad \text{element}(z, [z,y,x]). \\
\text{a3} & \quad \text{element}(X, [X|Y]).
\end{align*}

\begin{align*}
\theta & = \{X/1, Y/[]\} \\
\theta & = \{X/z, Y/[y,x]\}
\end{align*}

first element of second argument (a non-empty list) has to be the first argument

\begin{align*}
\text{element}(X, [X|Y]).
\end{align*}

happens to be the least general (or most specific) generalization because all other atoms that $\theta$-subsume $a_1$ and $a_2$ also $\theta$-subsume $a_3$:

\begin{align*}
\text{element}(X, [Y|Z]).
\end{align*}

only requires second argument to be an arbitrary non-empty list

no restrictions on either argument

\begin{align*}
\text{element}(X, Y).
\end{align*}
Generalizing clauses:
generalizing 2 atoms - set of first-order terms is a lattice

t1 is more general than t2 ⇔ for some substitution θ: t1θ = t2

greatest lower bound of two terms (meet operation): unification
specialization = applying a substitution

least upper bound of two terms (join operation): anti-unification

generalization = applying an inverse substitution (terms to variables)
Generalizing clauses:

anti-unification computes the least-general generalization of two atoms under $\theta$-subsumption

dual of unification

compare corresponding argument terms of two atoms,
replace by variable if they are different
replace subsequent occurrences of same term by same variable

?- anti_unify(2*2=2+2, 2*3=3+3, T, [], S1, [], S2).
T = 2*X=X+X
S1 = [2 <- X]
S2 = [3 <- X]

will not compute proper inverse substitutions: not clear which occurrences of 2 are mapped to X (all but the first)
BUT we are only interested in the $\theta$-LGG

BUT we are only interested in the $\theta$-LGG

will not compute proper inverse substitutions: not clear which occurrences of 2 are mapped to X (all but the first)

clearly, Prolog will generate a new anonymous variable (e.g., _G123) rather than X

remaining arguments: inverse substitutions for each term and their accumulators

$\theta$-LGG of first two arguments
Generalizing clauses: 

anti-unification computes the least-general generalization of two atoms under $\theta$-subsumption

:- op(600,xfx,'<-').
anti_unify(Term1,Term2,Term) :-
    anti_unify(Term1,Term2,Term,=[],S1,=[],S2).
anti_unify(Term1,Term2,Term1,S1,S1,S2,S2) :-
    Term1 == Term2,
    !.
anti_unify(Term1,Term2,V,S1,S1,S2,S2) :-
    subs_lookup(S1,S2,Term1,Term2,V),
    !.
anti_unify(Term1,Term2,Term,S10,S1,S20,S2) :-
    nonvar(Term1),
    nonvar(Term2),
    functor(Term1,F,N),
    functor(Term2,F,N),
    !,
    functor(Term,F,N),
    anti_unify_args(N,Term1,Term2,Term,S10,S1,S20,S2).
anti_unify(Term1,Term2,V,S10,[Term1<-V|S10],S20,[Term2<-V|S20]).
Generalizing clauses:
anti-unification computes the least-general generalization of two atoms under $\theta$-subsumption

\[
\text{anti\_unify\_args}(0, \text{Term}1, \text{Term}2, \text{Term}, S1, S1, S2, S2).
\]
\[
\text{anti\_unify\_args}(N, \text{Term}1, \text{Term}2, \text{Term}, S10, S1, S20, S2):=\]
\[
\begin{align*}
N &> 0, \\
N1 &\text{ is } N-1, \\
\text{arg}(N, \text{Term}1, \text{Arg}1), \\
\text{arg}(N, \text{Term}2, \text{Arg}2), \\
\text{arg}(N, \text{Term}, \text{Arg}N), \\
\text{anti\_unify}(\text{Arg}1, \text{Arg}2, \text{Arg}N, S10, S11, S20, S21), \\
\text{anti\_unify\_args}(N1, \text{Term}1, \text{Term}2, \text{Term}, S11, S1, S21, S2).
\end{align*}
\]

\[
\text{subs\_lookup}([T1<-V|Subs1], [T2<-V|Subs2], \text{Term}1, \text{Term}2, V) :-
\]
\[
\begin{align*}
\text{T1} &\text{ == } \text{Term}1, \\
\text{T2} &\text{ == } \text{Term}2, \\
!.
\end{align*}
\]
\[
\text{subs\_lookup}([S1|Subs1], [S2|Subs2], \text{Term}1, \text{Term}2, V):-
\]
\[
\text{subs\_lookup}(\text{Subs1}, \text{Subs2}, \text{Term}1, \text{Term}2, V).
\]
Generalizing clauses:
set of (equivalence classes of) clauses is a lattice

\[ C_1 \text{ is more general than } C_2 \iff \text{ for some substitution } \theta: C_1 \theta \subseteq C_2 \]

- greatest lower bound of two clauses (meet operation): \( \theta\text{-MGS} \)
  - specialization = applying a substitution and/or adding a literal
- least upper bound of two clauses (join operation): \( \theta\text{-LGG} \)
  - generalization = applying an inverse substitution and/or removing a literal

\[ \begin{align*}
  \text{m}(X, Y) & \quad \text{m}([X|Y], Z) \quad \text{m}(X, Y):=\text{m}(Y, X) \\
  \text{m}(X, [Y|Z]) & \quad \text{m}([X|Y], Z) \quad \text{m}(X, [Y|Z]):=\text{m}(X, Z)
\end{align*} \]

\[ \uparrow \quad \text{anti-unification and/or removing literal} \]
\[ \downarrow \quad \text{unification and/or adding literal} \]
Generalizing clauses: computing the \( \theta \) least-general generalization

similar to, and depends on, anti-unification of atoms

but the body of a clause is (declaratively spoken) unordered

therefore have to compare all possible pairs of atoms (one from each body)

?- theta_lgg((element(c, [b,c]):-element(c, [c]))),
   (element(d, [b,c,d]):-element(d, [c,d]),element(d, [d])),
   C).
C = element(X, [b,c|Y]):-element(X, [c|Y]),element(X, [X])

obtained by anti-unifying original heads

obtained by anti-unifying element(c, [c]) and element(d, [c,d])

obtained by anti-unifying element(c, [c]) and element(d, [d])
Generalizing clauses:
computing the $\theta$ least-general generalization

\[
\text{theta}_l\text{gg}((H1:-B1),(H2:-B2),(H:-B)) :-
\text{anti}\_\text{unify}(H1,H2,H, [], S10, [], S20),
\text{theta}_l\text{gg}\_\text{bodies}(B1,B2, [], B, S10,S1,S20,S2).
\]

\[
\text{theta}_l\text{gg}\_\text{bodies}([], B2, B, B, S1,S1,S2,S2).
\]

\[
\text{theta}_l\text{gg}\_\text{bodies}([\text{Lit}|B1], B2, B0,B, S10,S1, S20,S2) :-
\text{theta}_l\text{gg}\_\text{literal}(\text{Lit}, B2, B0,B00, S10,S11, S20,S21),
\text{theta}_l\text{gg}\_\text{bodies}(B1,B2, B00,B, S11,S1, S21,S2).
\]

\[
\text{theta}_l\text{gg}\_\text{literal}(\text{Lit1}, [], B,B, S1,S1, S2,S2).
\]

\[
\text{theta}_l\text{gg}\_\text{literal}(\text{Lit1}, [\text{Lit2}|B2], B0,B,S10,S1,S20,S2) :-
\text{same}\_\text{predicate}(\text{Lit1},\text{Lit2}),
\text{anti}\_\text{unify}(\text{Lit1},\text{Lit2},\text{Lit},S10,S11,S20,S21),
\text{theta}_l\text{gg}\_\text{literal}(\text{Lit1},B2, [\text{Lit}|B0],B, S11, S1,S21,S2).
\]

\[
\text{theta}_l\text{gg}\_\text{literal}(\text{Lit1}, [\text{Lit2}|B2], B0,B,S10,S1,S20,S2) :-
\text{not}\(\text{same}\_\text{predicate}(\text{Lit1},\text{Lit2}))\),
\text{theta}_l\text{gg}\_\text{literal}(\text{Lit1},B2,B0,B,S10,S1,S20,S2).
\]

\[
\text{same}\_\text{predicate}(\text{Lit1},\text{Lit2}) :-
\text{functor}(\text{Lit1},P,N),
\text{functor}(\text{Lit2},P,N).
\]

\[
\text{anti}\_\text{unify}\text{ heads}
\]

\[
\text{pair-wise anti-unification of atoms in bodies}
\]

\[
\text{atom from first body}
\]

\[
\text{atom from second body}
\]

\[
incompatible\ pair
\]
Generalizing clauses: computing the $\theta$ least-general generalization

?- theta_lgg((reverse([2,1],[3],[1,2,3]):-reverse([1],[2,3],[1,2,3])),
    (reverse([a],[], [a]):-reverse([], [a],[a])),
    C).
C = reverse([X|Y], Z, [U|V]) :- [reverse(Y, [X|Z], [U|V])]

\[
\text{rev}([2,1],[3],[1,2,3]) :- \text{rev}([1],[2,3],[1,2,3])
\]
\[
\quad X \quad Y \quad Z \quad U \quad V
\]
\[
\text{rev}([a],[], [a]) :- \text{rev}([], [a],[a])
\]
Bottom-up induction: specific-to-general search of the hypothesis space

generalizes positive examples into a hypothesis rather than specializing the most general hypothesis as long as it covers negative examples

relative least general generalization $\text{rlgg}(e_1, e_2, M)$ of two positive examples $e_1$ and $e_2$ relative to a partial model $M$ is defined as:

$$\text{rlgg}(e_1, e_2, M) = \text{lgg}((e_1 \ :- \ \text{Conj}(M)), \ (e_2 \ :- \ \text{Conj}(M)))$$

conjunction of all positive examples plus ground facts for the background predicates
Bottom-up induction:

relative least general generalization

\[ M \]

\[ e_1 \]
append([1,2], [3,4], [1,2,3,4]).
append([a], [], [a]).
append([], [], []).
append([2], [3,4], [2,3,4]).

\[ e_2 \]

\[ rlgg(e_1,e_2,M) \]

?– theta_lgg((append([1,2], [3,4], [1,2,3,4]) :-
[append([1,2], [3,4], [1,2,3,4]),
append([a], [], [a]), append([], [], []),
append([2], [3,4], [2,3,4]))),
(append([a], [], [a]) :-
[append([1,2], [3,4], [1,2,3,4]),
append([a], [], [a]), append([], [], []),
append([2], [3,4], [2,3,4]))),
C)\]
Bottom-up induction: relative least general generalization - need for pruning

\[ \text{rlgg}(e_1, e_2, M) \]

\[
\begin{align*}
\text{append}([X|Y], Z, [X|U]) & \leftarrow [ \\
& \quad \text{append}([2], [3, 4], [2, 3, 4]), \\
& \quad \text{append}(Y, Z, U), \\
& \quad \text{append}([V], Z, [V|Z]), \\
& \quad \text{append}([K|L], [3, 4], [K, M, N|O]), \\
& \quad \text{append}(L, P, Q), \\
& \quad \text{append}([], [], []), \\
& \quad \text{append}(R, [], R), \\
& \quad \text{append}(S, P, T), \\
& \quad \text{append}([A], P, [A|P]), \\
& \quad \text{append}(B, [], B), \\
& \quad \text{append}([a], [], [a]), \\
& \quad \text{append}([C|L], P, [C|Q]), \\
& \quad \text{append}([D|Y], [3, 4], [D, E, F|G]), \\
& \quad \text{append}(H, Z, I), \\
& \quad \text{append}([X|Y], Z, [X|U]), \\
& \quad \text{append}([1, 2], [3, 4], [1, 2, 3, 4])
\end{align*}
\]
Bottom-up induction:
relative least general generalization - algorithm

to determine vars in head (strictly constrained restriction)

rlgg(E1,E2,M,(H:- B)):-
  anti_unify(E1,E2,H,[],S10,[],S20),
  varsin(H,V),
  rlgg_bodies(M,M,[],B,S10,S1,S20,S2,V).

rlgg_bodies(B0,B1,BR0,BR,S10,S1,S20,S2,V): rlgg all literals in B0 with all literals in B1, yielding BR (from accumulator BR0) containing only vars in V

rlgg_bodies([],B2,B,B,S1,S1,S2,S2,V).
rlgg_bodies([L|B1],B2,B0,B,S10,S1,S20,S2,V):-
  rlgg Literal(L,B2,B0,B00,S10,S11,S20,S21,V),
  rlgg_bodies(B1,B2,B00,B,S11,S1,S21,S2,V).
Bottom-up induction: relative least general generalization - algorithm

rlgg_literal(L1, [], B, B, S1, S1, S2, S2, V).
rlgg_literal(L1, [L2|B2], B0, B, S10, S1, S20, S2, V):-
    same_predicate(L1, L2),
    anti_unify(L1, L2, L, S10, S11, S20, S21),
    varsin(L, Vars),
    var_proper_subset(Vars, V),
    !,
    rlgg_literal(L1, B2, B0, B, S11, S1, S21, S2, V).
rlgg_literal(L1, [L2|B2], B0, B, S10, S1, S20, S2, V):-
    rlgg_literal(L1, [L2|B2], B0, B, S10, S1, S20, S2, V).

strictly constrained (no new variables, but proper subset)

otherwise, an incompatible pair of literals
Bottom-up induction:
relative least general generalization - algorithm

\[
\text{var\_proper\_subset([], Ys)} :\= \text{Ys \not= [].}
\]
\[
\text{var\_proper\_subset([X|Xs], Ys)} :\= \\
\quad \text{var\_remove\_one(X, Ys, Zs)}, \\
\quad \text{var\_proper\_subset(Xs, Zs)}.
\]

\[
\text{varsin(Term, Vars):}\
\quad \text{varsin(Term, [], V),} \\
\quad \text{sort(V, Vars).}
\]
\[
\text{varsin(V, Vars, [V|Vars]) :}\
\quad \text{var(V).}
\]
\[
\text{varsin(Term, V0, V):}\
\quad \text{functor(Term, F, N),} \\
\quad \text{varsin\_args(N, Term, V0, V).}
\]

\[
\text{var\_remove\_one(X, [Y|Ys], Ys)} :\= \\
\quad X == Y.
\]
\[
\text{var\_remove\_one(X, [Y|Ys], [Y|Zs)} :\= \\
\quad \text{var\_remove\_one(X, Ys, Zs).}
\]
\[
\text{varsin\_args(0, Term, Vars, Vars).}
\]
\[
\text{varsin\_args(N, Term, V0, V):}\
\quad N>0, \\
\quad N1 \text{ is } N-1, \\
\quad \text{arg(N, Term, ArgN),} \\
\quad \text{varsin(ArgN, V0, V1),} \\
\quad \text{varsin\_args(N1, Term, V1, V).}
\]
Bottom-up induction:
relative least general generalization - algorithm

?- r1gg(append([1,2],[3,4],[1,2,3,4]),
    append([a],[],[a]),
    append([1,2],[3,4],[1,2,3,4]),
    append([a],[],[a]),
    append([],[],[]),
    append([2],[3,4],[2,3,4]),
    (H:- B)).
H = append([X|Y], Z, [X|U])
B = [append([2], [3, 4], [2, 3, 4]),
      append(Y, Z, U),
      append([], [], []),
      append([a], [], [a]),
      append([1, 2], [3, 4], [1, 2, 3, 4])]
Bottom-up induction: *main algorithm*

construct rlgg of two positive examples
remove all positive examples that are extensionally covered by the constructed clause

further generalize the clause by removing literals as long as no negative examples are covered
Bottom-up induction: main algorithm

\[
\text{induce}_r lgg(\text{Exs}, \text{Clauses}) :\neg \\
\quad \text{pos}_neg(\text{Exs}, \text{Poss}, \text{Negs}), \\
\quad \text{bg}_\text{model}(\text{BG}), \\
\quad \text{append}(\text{Poss}, \text{BG}, \text{Model}), \\
\quad \text{induce}_r lgg(\text{Poss}, \text{Negs}, \text{Model}, \text{Clauses}).
\]

\[
\text{induce}_r lgg(\text{Poss}, \text{Negs}, \text{Model}, \text{Clauses}) :\neg \\
\quad \text{covering}(\text{Poss}, \text{Negs}, \text{Model}, [], \text{Clauses}).
\]

\[
\text{pos}_neg([], [], []). \\
\text{pos}_neg([E|\text{Exs}], [E|\text{Poss}], \text{Negs}) :\neg \\
\quad \text{pos}_neg(\text{Exs}, \text{Poss}, \text{Negs}). \\
\text{pos}_neg([-E|\text{Exs}], \text{Poss}, [E|\text{Negs}]) :\neg \\
\quad \text{pos}_neg(\text{Exs}, \text{Poss}, \text{Negs}).
\]
Bottom-up induction:
main algorithm - covering

covering(Poss,Negs,Model,Hyp0,NewHyp) :-
    construct_hypothesis(Poss,Negs,Model,Hyp), !,
    remove_pos(Poss,Model,Hyp,NewPoss),
    covering(NewPoss,Negs,Model,[Hyp|Hyp0],NewHyp).

covering(P,N,M,H0,H) :-
    append(H0,P,H).

when no longer possible to construct new hypothesis clauses, add remaining positive examples to hypothesis

remove_pos([],M,H,[]).
remove_pos([P|Ps],Model,Hyp,NewP) :-
    covers_ex(Hyp,P,Model), !,
    write('Covered example: '), writeLn(P),
    remove_pos(Ps,Model,Hyp,NewP).
remove_pos([P|Ps],Model,Hyp,[P|NewP]) :-
    remove_pos(Ps,Model,Hyp,NewP).

construct a new hypothesis clause that covers all of the positive examples and none of the negative
remove covered positive examples

covers_ex((Head:- Body), Example,Model):-
    verify((Head=Example,
        forall(element(L,Body),
            element(L,Model)))).
Bottom-up induction:  
**main algorithm - hypothesis construction**

```prolog
construct_hypothesis([E1,E2|Es],Negs,Model,Clause):-
  write('RLGG of '), write(E1),
  write(' and '), write(E2), write(' is'),
  rlgg(E1,E2,Model,Cl),
  reduce(Cl,Negs,Model,Clause),
  !,
  nl,tab(5), write_ln(Clause).

construct_hypothesis([E1,E2|Es],Negs,Model,Clause):-
  write_ln(' too general'),
  construct_hypothesis([E2|Es],Negs,Model,Clause).
```

- **Remove redundant literals** and ensure that no negative examples are covered.
- The only step in the algorithm that involves negative examples.

If no rlgg can be constructed for these two positive examples or the constructed one covers a negative example.

Note that E1 will be considered again with another example in a different iteration of covering/5.
Bottom-up induction:  
main algorithm - hypothesis reduction

remove redundant literals and ensure that no negative examples are covered

\[
\text{reduce}((H:-B0),\text{Negs},M,(H:-B)):-
\begin{align*}
&\text{setof0}(L, \\
&\quad (\text{element}(L,B0), \text{not(var_element}(L,M)) ), \\
&\quad B1), \\
&\text{reduce_negs}(H,B1,[]B,Negs,M).
\end{align*}
\]

\[
\text{setof0}(X,G,L):-
\begin{align*}
&\text{setof}(X,G,L), !. \\
&\text{setof0}(X,G,[]).
\end{align*}
\]

succeeds with empty list of no solutions can be found

removes literals from the body that are already in the model

\[
\text{var_element}(X,[Y|Ys]):-
\begin{align*}
&X = Y. \\
&\text{var_element}(X,[Y|Ys]):-
\quad \text{var_element}(X,Ys).
\end{align*}
\]

element/2 using syntactic identity rather than unification
Bottom-up induction:
main algorithm - hypothesis reduction

B is the body of the reduced clause: a subsequence of the body of the original clause (second argument), such that no negative example is covered by model U reduced clause (H:-B)

reduce_negs(H, [L|Rest], B0, B, Negs, Model):-
  append(B0, Rest, Body),
  not(covers_neg((H:-Body), Negs, Model, N)),!
  reduce_negs(H, Rest, B0, B, Negs, Model).
reduce_negs(H, [L|Rest], B0, B, Negs, Model):-
  reduce_negs(H, Rest, [L|B0], B, Negs, Model).
reduce_negs(H, [], Body, Body, Negs, Model):-
  not(covers_neg((H:-Body), Negs, Model, N)).

a negative example is covered by clause U model

covers_neg(Clause, Negs, Model, N) :-
  element(N, Negs),
  covers_ex(Clause, N, Model).

try to remove L from the original body

L cannot be removed

fail if the resulting clause covers a negative example
Bottom-up induction:

example

?- induce_rlgg([+
append([1,2], [3,4], [1,2,3,4]),
+append([a], [], [a]),
+append([], [], []),
+append([], [1,2,3], [1,2,3]),
+append([2], [3,4], [2,3,4]),
+append([], [3,4], [3,4]),
-append([a], [b], [b]),
-append([c], [b], [c,a]),
-append([1,2], [], [1,3])], Clauses).

RLGG of append([1,2], [3,4], [1,2,3,4]) and append([a], [], [a]) is
append([X|Y],Z,[X|U]) :- [append(Y,Z,U)]
Covered example: append([1,2], [3,4], [1,2,3,4])
Covered example: append([a], [], [a])
Covered example: append([2], [3,4], [2,3,4])

RLGG of append([], [], []) and append([], [1,2,3], [1,2,3]) is
append([],X,X) :- []
Covered example: append([], [], [])
Covered example: append([], [1,2,3], [1,2,3])
Covered example: append([], [1,2,3], [1,2,3])
Covered example: append([], [3,4], [3,4])

Clauses = [(append([],X,X) :- []),
(append([X|Y],Z,[X|U]) :- [append(Y,Z,U)])]
Bottom-up induction: example

\[
\begin{align*}
\text{bg\_model}( & \text{[num}(1,\text{one}),\text{num}(2,\text{two}), \\
& \text{num}(3,\text{three}), \\
& \text{num}(4,\text{four}), \\
& \text{num}(5,\text{five})]). \\
\text{?-induce\_rlgg}([ & +\text{listnum}([\text{[]},\text{[]}]), \\
& +\text{listnum}(\text{[2,three,4],[two,3,four]}), \\
& +\text{listnum}(\text{[4],[four]}), \\
& +\text{listnum}(\text{[three,4],[3,four]}), \\
& +\text{listnum}(\text{[two],[2]}), \\
& -\text{listnum}(\text{[1,4],[1,four]}), \\
& -\text{listnum}(\text{[2,three,4],[two]}), \\
& -\text{listnum}(\text{[five],[5,5]}),] \\
& \text{Clauses}).
\end{align*}
\]

RLGG of listnum([\text{[]}],\text{[]}) and listnum([\text{2,three,4}],[\text{two,3,four}]) is too general
RLGG of listnum([\text{2,three,4}],[\text{two,3,four}]) and listnum([\text{4}],[\text{four}]) is
listnum([X|Xs],[Y|Ys]):-\text{[num}(X,Y),\text{listnum}(Xs,Ys)]
Covered example: listnum([\text{2,three,4}],[\text{two,3,four}])
Covered example: listnum([\text{4}],[\text{four}])

RLGG of listnum([\text{[]}],\text{[]}) and listnum([\text{three,4}],[\text{3,four}]) is too general
RLGG of listnum([\text{three,4}],[\text{3,four}]) and listnum([\text{two}],[\text{2}]) is
listnum([V|Vs],[W|Ws]):-\text{[num}(W,V),\text{listnum}(Vs,Ws)]
Covered example: listnum([\text{three,4}],[\text{3,four}])
Covered example: listnum([\text{two}],[\text{2}])

Clauses =\{(listnum([V|Vs],[W|Ws]):-\text{[num}(W,V),\text{listnum}(Vs,Ws))], \\
(listnum([X|Xs],[Y|Ys]):-\text{[num}(X,Y),\text{listnum}(Xs,Ys)]),\text{listnum}([\text{[]}],\text{[]})\}