Declarative Programming

7: inductive reasoning

Inductive reasoning: *overview*

infer general rules from specific observations

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

$\forall\, p \in P: B \cup H \models p$

H does not "cover" any negative example given B

$\forall\, n \in N: B \cup H \not\models n$

Inductive reasoning:
relation to abduction

in inductive reasoning, the hypothesis (what has to be added to the logic program) is a set of clauses rather than a set of ground facts

given a theory T and an observation O, find an explanation E such that $T \cup E \vDash O$

Try to adapt the abductive meta-interpreter: inducible/1 defines the set of possible hypothesis

```
induce(E,H) :-
  induce(E,[],H).
induce(true,H,H).
induce((A,B),H0,H) :-
  induce(A,H0,H1),
  induce(B,H1,H).
induce(A,H0,H) :-
  clause(A,B),
  induce(B,H0,H).
```

```
induce(A,H0,H) :-
  element((A:-B),H0),
  induce(B,H0,H).
induce(A,H0,[(A:-B)|H]) :
  inducible((A:-B)),
  not(element((A:-B),H0)),
  induce(B,H0,H).
```

clause already assumed

assume clause if it's an inducible and not yet assumed

Inductive reasoning: *relation to abduction*

```
bird(tweety).
has_feathers(tweety).
bird(polly).
has_beak(polly).
```

```
inducible((flies(X):-bird(X),has_feathers(X),has_beak(X))).
inducible((flies(X):-has_feathers(X),has_beak(X))).
inducible((flies(X):-bird(X),has_beak(X))).
inducible((flies(X):-bird(X),has_feathers(X))).
inducible((flies(X):-bird(X))).
inducible((flies(X):-has_feathers(X))).
inducible((flies(X):-has_beak(X))).
inducible((flies(X):-true)).
```

enumeration of possible hypotheses

probably an overgeneralization

```
?-induce(flies(tweety),H).
H = [(flies(tweety):-bird(tweety),has_feathers(tweety))];
H = [(flies(tweety):-bird(tweety))];
H = [(flies(tweety):-has_feathers(tweety))];
H = [(flies(tweety):-true)];
No more solutions
```

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	action	hypothesis
`+ p(b, [b])`	add clause	`p(X,Y).`
`- p(x, [])`	specialize	`p(X, [V|W]).`
`- p(x, [a,b])`	specialize	`p(X, [X|W]).`
`+ p(b, [a,b])`	add clause	`p(X, [X|W]).` `p(X, [V|W]):-p(X,W).`

this negative example precludes the previous hypothesis' second argument from unifying with the empty list

Generalizing clauses: Θ-*subsumption*

$H1;...;Hn$:- $B1,...,Bm$

$H1 \vee ... \vee Hn \vee \neg B1 \vee ... \vee \neg Bm$

c1 is more general than c2

A clause c1 θ-subsumes a clause c2
⇔ ∃ a substitution θ such that $c1\theta \subseteq c2$

clauses are seen as sets of disjuncted positive (head) and negative (body) literals

```
element(X,V) :- element(X,Z)
```

θ-subsumes

```
element(X,[Y|Z]) :- element(X,Z)
```

using $\theta = \{V \rightarrow [Y|Z]\}$

```
a(X) :- b(X)
```

θ-subsumes

```
a(X) :- b(X), c(X).
```

using $\theta = id$

Generalizing clauses:

θ-*subsumption versus* ⊧

H1 is at least as general as H2 given B ⇔

H1 covers everything covered by H2 given B
$\forall p \in P: B \cup H2 \models p \Rightarrow B \cup H1 \models p$

$B \cup H1 \models H2$

clause c1 θ-subsumes c2 ⇒ c1 ⊧ c2

The reverse is not true:

```
a(X) :- b(X). % c1
p(X) :- p(X). % c2
```

c1 ⊧ c2, but there is no substitution θ such that c1θ ⊆ c2

Generalizing clauses:
testing for Θ-subsumption

A clause c1 θ-subsumes a clause c2
⇔ ∃ a substitution θ such that $c1\theta \subseteq c2$

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

```
theta_subsumes((H1:-B1),(H2:-B2)):-
  verify((ground((H2:-B2)),H1=H2,subset(B1,B2))).

verify(Goal) :-
  not(not(call(Goal))).

ground(Term):-
  numbervars(Term,0,N).
```

prove Goal, but without creating bindings

Generalizing clauses:
testing for Θ-subsumption

A clause c1 θ-subsumes a clause c2
⇔ ∃ a substitution θ such that $c1\theta \subseteq c2$

bodies are lists of atoms

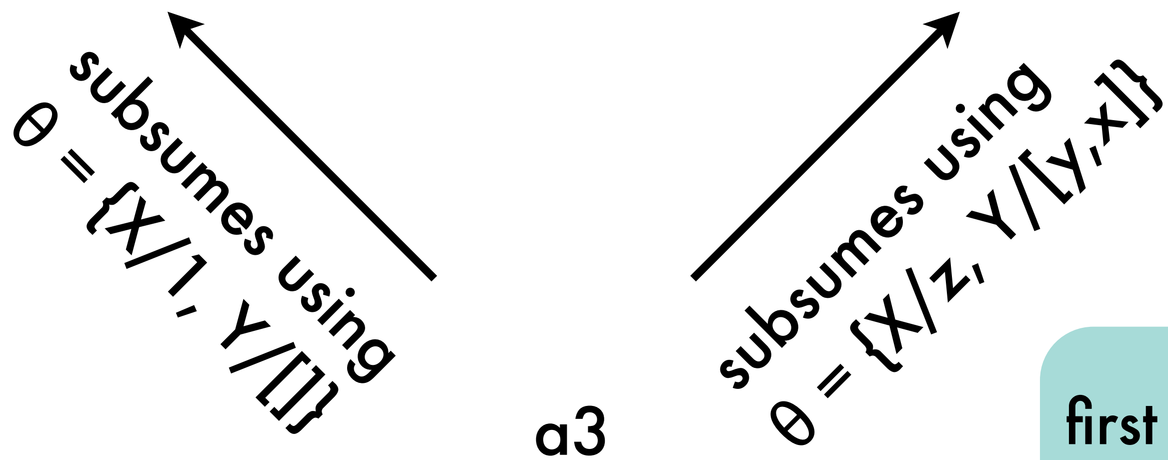
```
?- theta_subsumes((element(X,V):- []),
                  (element(X,V):- [element(X,Z)])).
yes.

?- theta_subsumes((element(X,a):- []),
                  (element(X,V):- [])).
no.
```

Generalizing clauses:
generalizing 2 atoms

A clause c1 θ-subsumes a clause c2
⇔ ∃ a substitution θ such that $c1\theta \subseteq c2$

a1 `element(1, [1]).`

`element(z, [z,y,x]).` a2



a3

`element(X, [X|Y]).`

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

happens to be the **least general** (or most specific) **generalization** because all other atoms that θ-subsume a1 and a2 also θ-subsume a3:

`element(X, [Y|Z]).`

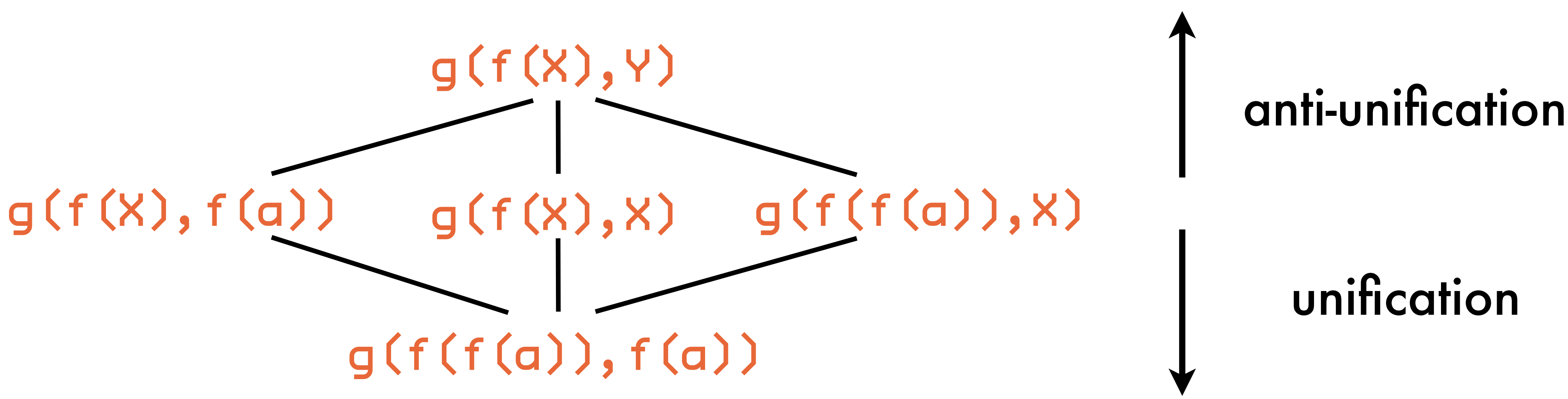
only requires second argument to be an arbitrary non-empty list

`element(X,Y).`

no restrictions on either argument

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 θ-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

θ-LGG of first two arguments

remaining arguments: inverse substitutions for each term and their accumulators

```
?- 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 θ-LGG

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

Generalizing clauses:

anti-unification computes the least-general generalization of two atoms under θ-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]).
```

same terms

not the same terms, but each has already been mapped to the same variable V in the respective inverse substitutions

equivalent compound term is constructed if both original compounds have the same functor and arity

if all else fails, map both terms to the same variable

Generalizing clauses:

anti-unification computes the least-general generalization of two atoms under θ-subsumption

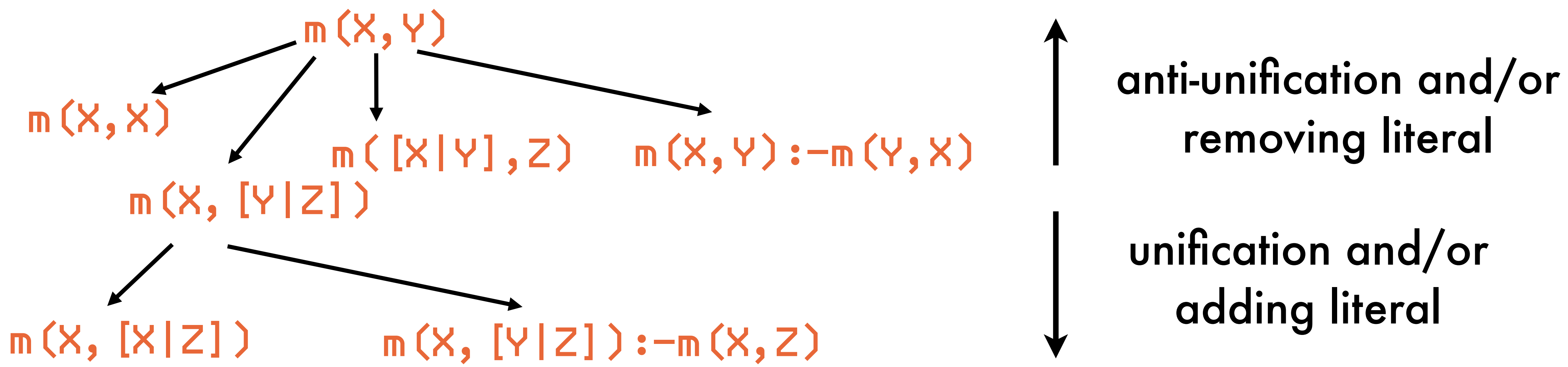
```
anti_unify_args(0,Term1,Term2,Term,S1,S1,S2,S2).
anti_unify_args(N,Term1,Term2,Term,S10,S1,S20,S2):-
  N>0,
  N1 is N-1,
  arg(N,Term1,Arg1),
  arg(N,Term2,Arg2),
  arg(N,Term,ArgN),
  anti_unify(Arg1,Arg2,ArgN,S10,S11,S20,S21),
  anti_unify_args(N1,Term1,Term2,Term,S11,S1,S21,S2).
```

anti-unify first N corresponding arguments

```
subs_lookup([T1<-V|Subs1],[T2<-V|Subs2],Term1,Term2,V) :-
  T1 == Term1,
  T2 == Term2,
  !.
subs_lookup([S1|Subs1],[S2|Subs2],Term1,Term2,V):-
  subs_lookup(Subs1,Subs2,Term1,Term2,V).
```

Generalizing clauses:

set of (equivalence classes of) clauses is a lattice



C1 is more general than C2 ⇔ for some substitution θ: C1θ ⊆ C2

greatest lower bound of two clauses (meet operation): θ-MGS

specialization = applying a substitution and/or adding a literal

least upper bound of two clauses (join operation): θ-LGG

generalization = applying an inverse substitution and/or removing a literal

Generalizing clauses:

computing the θ 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 θ least-general generalization

```
theta_lgg((H1:-B1),(H2:-B2),(H:-B)):-
  anti_unify(H1,H2,H, [],S10, [],S20),
  theta_lgg_bodies(B1,B2, [],B,S10,S1,S20,S2).
```

anti-unify heads

pair-wise anti-unification of atoms in bodies

```
theta_lgg_bodies([],B2,B,B,S1,S1,S2,S2).
theta_lgg_bodies([Lit|B1],B2, B0,B, S10,S1, S20,S2):-
  theta_lgg_literal(Lit,B2, B0,B00, S10,S11, S20,S21),
  theta_lgg_bodies(B1,B2, B00,B, S11,S1, S21,S2).
```

atom from first body

```
theta_lgg_literal(Lit1,[], B,B, S1,S1, S2,S2).
theta_lgg_literal(Lit1,[Lit2|B2],B0,B,S10,S1,S20,S2):-
  same_predicate(Lit1,Lit2),
  anti_unify(Lit1,Lit2,Lit,S10,S11,S20,S21),
  theta_lgg_literal(Lit1,B2,[Lit|B0],B, S11, S1,S21,S2).
theta_lgg_literal(Lit1,[Lit2|B2],B0,B,S10,S1,S20,S2):-
  not(same_predicate(Lit1,Lit2)),
  theta_lgg_literal(Lit1,B2,B0,B,S10,S1,S20,S2).
same_predicate(Lit1,Lit2) :-
  functor(Lit1,P,N),
  functor(Lit2,P,N).
```

atom from second body

incompatible pair

Generalizing clauses:

computing the θ 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])]
```

```
rev([2,1],[3],[1,2,3]):-rev([1],[2,3],[1,2,3])
     | |   |   |  /          |   |  |   |  /
     X Y   Z   U V           Y   X  Z   U V
     | /   |   | /           |   | /    | /
rev([a]  ,[] ,[a]    ):-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 **rlgg(e1,e2,M)**
of two positive examples e1 and e2
relative to a partial model M is defined as:
rlgg(e1, e2, M) = lgg((e1 :- Conj(M)), (e2 :- Conj(M)))

conjunction of all positive
examples plus ground facts for
the background predicates

Bottom-up induction:

relative least general generalization

M

e1
e2

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

rlgg(e1,e2,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

rlgg(e1,e2,M)

```
append([X|Y], Z, [X|U]) :- [
  append([2], [3, 4], [2, 3, 4]),
  append(Y, Z, U),
  append([V], Z, [V|Z]),
  append([K|L], [3, 4], [K, M, N|O]),
  append(L, P, Q),
  append([], [], []),
  append(R, [], R),
  append(S, P, T),
  append([A], P, [A|P]),
  append(B, [], B),
  append([a], [], [a]),
  append([C|L], P, [C|Q]),
  append([D|Y], [3, 4], [D, E, F|G]),
  append(H, Z, I),
  append([X|Y], Z, [X|U]),
  append([1, 2], [3, 4], [1, 2, 3, 4])
]
```

remaining ground facts from M (e.g., examples) are redundant: can be removed

introduces variables that do not occur in the head: can assume that hypothesis clauses are constrained

head of clause in body = tautology: restrict ourselves to strictly constrained hypothesis clauses

variables in body are **proper** subset of variables in head

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,[L|B0],B,S11,S1,S21,S2,V).
rlgg_literal(L1,[L2|B2],B0,B,S10,S1,S20,S2,V):-
  rlgg_literal(L1,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

```
var_proper_subset([],Ys) :-
  Ys \= [].
var_proper_subset([X|Xs],Ys) :-
  var_remove_one(X,Ys,Zs),
  var_proper_subset(Xs,Zs).
```

```
var_remove_one(X, [Y|Ys],Ys) :-
  X == Y.
var_remove_one(X, [Y|Ys], [Y|Zs) :-
  var_remove_one(X,Ys,Zs).
```

```
varsin(Term,Vars):-
  varsin(Term, [],V),
  sort(V,Vars).
varsin(V,Vars, [V|Vars]):-
  var(V).
varsin(Term,V0,V):-
  functor(Term,F,N),
  varsin_args(N,Term,V0,V).
```

```
varsin_args(0,Term,Vars,Vars).
varsin_args(N,Term,V0,V):-
  N>0,
  N1 is N-1,
  arg(N,Term,ArgN),
  varsin(ArgN,V0,V1),
  varsin_args(N1,Term,V1,V).
```

Bottom-up induction:
relative least general generalization - algorithm

```
?- rlgg(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

split positive from negative examples

include positive examples in background model

```
induce_rlgg(Exs,Clauses):-
  pos_neg(Exs,Poss,Negs),
  bg_model(BG),
  append(Poss,BG,Model),
  induce_rlgg(Poss,Negs,Model,Clauses).

induce_rlgg(Poss,Negs,Model,Clauses):-
  covering(Poss,Negs,Model,[],Clauses).
```

```
pos_neg([],[],[]).
pos_neg([+E|Exs],[E|Poss],Negs):-
  pos_neg(Exs,Poss,Negs).
pos_neg([-E|Exs],Poss,[E|Negs]):-
  pos_neg(Exs,Poss,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).
```

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

remove covered positive examples

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: '),
  write_ln(P),
  remove_pos(Ps,Model,Hyp,NewP).
remove_pos([P|Ps],Model,Hyp, [P|NewP]):-
  remove_pos(Ps,Model,Hyp,NewP).
```

```
covers_ex((Head:- Body),
          Example,Model):-
verify((Head=Example,
       forall(element(L,Body),
              element(L,Model)))).
```

Bottom-up induction:
main algorithm - hypothesis construction

```
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).
```

this is the only step in the algorithm that involves negative examples!

remove redundant literals and ensure that no negative examples are covered

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

```
setof0(X,G,L):-
    setof(X,G,L),!.
setof0(X,G,[]).
```

succeeds with empty list of no solutions can be found

remove redundant literals and ensure that no negative examples are covered

```
reduce((H:-B0),Negs,M,(H:-B)):-
  setof0(L,
        (element(L,B0), not(var_element(L,M))),
        B1),
  reduce_negs(H,B1,[],B,Negs,M).
```

removes literals from the body that are already in the model

```
var_element(X, [Y|Ys]):-
  X == Y.
var_element(X, [Y|Ys]):-
  var_element(X,Ys).
```

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)).
```

try to remove L from the original body

L cannot be removed

fail if the resulting clause covers a negative example

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

a negative example is covered by clause U model

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([], [3,4], [3,4])

Clauses = [(append([],X,X) :- []),
(append([X|Y],Z, [X|U]) :- [append(Y,Z,U)])]
```

Bottom-up induction:
example

```
bg_model([num(1,one),num(2,two),
          num(3,three),
          num(4,four),
          num(5,five)]).

?-induce_rlgg([
+listnum([],[]),
+listnum([2,three,4],[two,3,four]),
+listnum([4],[four]),
+listnum([three,4],[3,four]),
+listnum([two],[2]),
-listnum([1,4],[1,four]),
-listnum([2,three,4],[two]),
-listnum([five],[5,5]) ],
Clauses).
```

```
RLGG of listnum([],[]) and
        listnum([2,three,4],[two,3,four]) is too general
RLGG of listnum([2,three,4],[two,3,four]) and
        listnum([4],[four]) is
listnum([X|Xs],[Y|Ys]):-[num(X,Y),listnum(Xs,Ys)]
Covered example: listnum([2,three,4],[two,3,four])
Covered example: listnum([4],[four])
RLGG of listnum([],[]) and listnum([three,4],[3,four]) is too general
RLGG of listnum([three,4],[3,four]) and listnum([two],[2]) is
listnum([V|Vs],[W|Ws]):-[num(W,V),listnum(Vs,Ws)]
Covered example:
listnum([three,4],[3,four])
Covered example: listnum([two],[2])
Clauses =[(listnum([V|Vs],[W|Ws]):-[num(W,V),listnum(Vs,Ws)]),
          (listnum([X|Xs],[Y|Ys]):-[num(X,Y),listnum(Xs,Ys)]),listnum([],[]) ]
```