

☞ **Given**

a background theory ***Th*** (clauses)

positive examples ***Pos*** (ground facts)

negative examples ***Neg*** (ground facts)

☞ **Find** a hypothesis ***Hyp*** such that

for every $p \in Pos$: $Th \cup Hyp \models p$

(***Hyp*** covers ***p*** given ***Th***)

for every $n \in Neg$: $Th \cup Hyp \not\models n$

(***Hyp*** does not cover ***p*** given ***Th***)

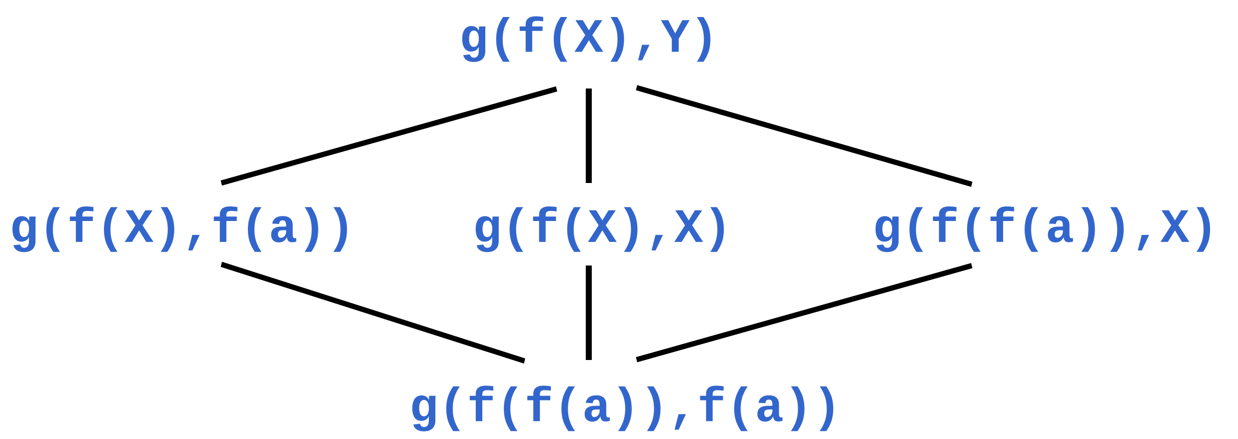
Induction

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

Induction: example

- What do the expressions **2*2=2+2** and **2*3=3+3** have **in common**?
-
```
?-anti_unify(2*2=2+2,2*3=3+3,T,[],S1,[],S2)
T = 2*X=X+X
S1 = [2<-X]
S2 = [3<-X]
```

Anti-unification

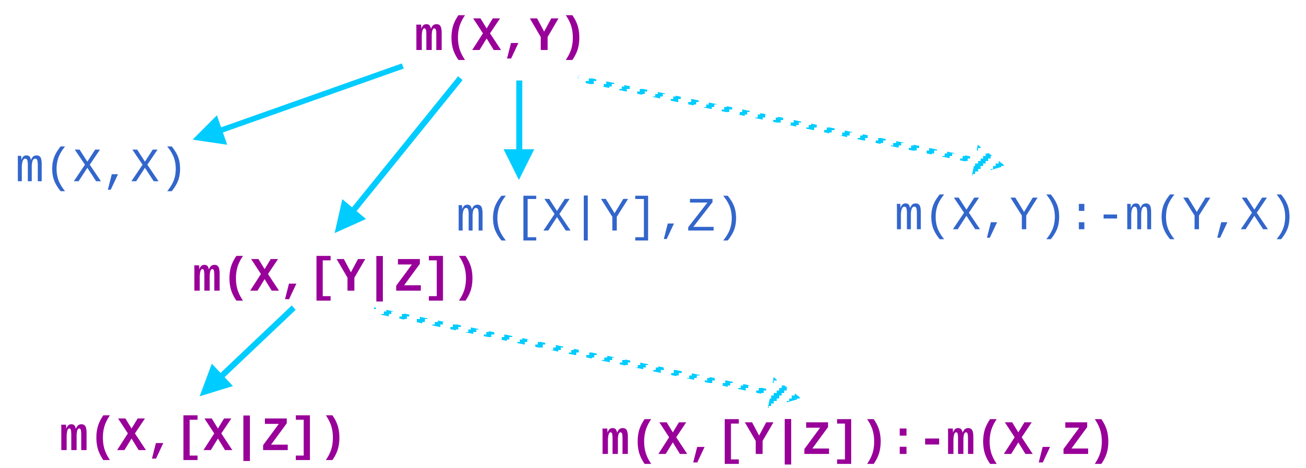


☞ The set of first-order terms is a **lattice**:

- ✓ t_1 is more general than t_2 iff for some substitution θ: $t_1\theta = t_2$
- ✓ greatest lower bound ⇒ unification, least upper bound ⇒ anti-unification
- ✓ Specialisation ⇒ applying a substitution
- ✓ Generalisation ⇒ applying an inverse substitution

Generality of terms

Generality of clauses



☞ The set of (equivalence classes of) clauses is a **lattice**:

- ✓ C_1 is more general than C_2 iff for some substitution θ: $C_1\theta \subseteq C_2$
- ✓ greatest lower bound ⇒ θ-MGS, least upper bound ⇒ θ-LGG
- ✓ Specialisation ⇒ applying a substitution and/or adding a literal
- ✓ Generalisation ⇒ applying an inverse substitution and/or removing a literal
- ✓ NB. There are infinite chains!

```
a([1,2],[3,4],[1,2,3,4]):-a([2],[3,4],[2,3,4])
a([a]  ,[]   ,[a]      ):-a([] ,[]   ,[]     )
a([A|B],C    ,[A|D]    ):-a(B  ,C    ,D      )
```

```
m(c,[a,b,c]):-m(c,[b,c]),m(c,[c])
m(a,[a,b  ]):-m(a,[a  ])
m(P,[a,b|Q]):-m(P,[R|Q]),m(P,[P])
```

θ-LGG: examples

```
rev([2,1],[3],[1,2,3]):-rev([1],[2,3],[1,2,3])
     |  |    |    |  /            |    |  |    |  /
     A  B    C    D E             B    A  C    D E
     | /     |    | /             |    | /     | /
rev([a]  ,[] ,[a]    ):-rev([] ,[a]  ,[a]    )


rev([A|B],C  ,[D|E]  ):-rev(B  ,[A|C],[D|E]  )
```

Exercise 9.3

- ***Hyp1*** is at least as general as ***Hyp2*** given ***Th*** iff
  - ✓***Hyp1*** covers everything covered by ***Hyp2*** given ***Th***
  - ✓for all ***p***: if ***Th*** ∪ ***Hyp2*** |= ***p*** then ***Th*** ∪ ***Hyp1*** |= ***p***
  - ✓***Th*** ∪ ***Hyp1*** |= ***Hyp2***

- Clause ***C1*** θ-subsumes ***C2*** iff
  - ✓there exists a substitution θ such that every literal in ***C1***θ occurs in ***C2***
  - ✓NB. if ***C1*** θ-subsumes ***C2*** then ***C1*** |= ***C2*** but not vice versa

Generality: summary

- Logical implication is **strictly stronger** than θ-subsumption
  - ✓ e.g. `list([V|W]):-list(W)` |= `list([X,Y|Z]):-list(Z)`
  - ✓ this happens when the resolution derivation requires the left-hand clause more than once
- i-LGG of definite clauses is **not unique**
  - ✓ i-LGG(`plist([A,B|C]):-list(C)`, `list([P,Q,R|S]):-list(S)`) = {`list([X|Y]):-list(Y)`, `list([X,Y|Z]):-list(V)`}
- Logical implication between clauses is undecidable, θ-subsumption is NP-complete

θ-subsumption vs. implication

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

a([a]  ,[]   ,[a]      ):-
      a([1,2],[3,4],[1,2,3,4]), a([a],[],[a]),
      a([],[],[]),              a([2],[3,4],[2,3,4]).

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

Relative least general generalisation: example

Delete ground literals and head literal from body

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

a([a]  ,[]   ,[a]      ):-
      a([1,2],[3,4],[1,2,3,4]), a([a],[],[a]),
      a([],[],[]),                a([2],[3,4],[2,3,4]).

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

Delete literals not linked to head variables

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

a([a]  ,[]   ,[a]       ):-
      a([1,2],[3,4],[1,2,3,4]), a([a],[],[a]),
      a([],[],[]),              a([2],[3,4],[2,3,4]).

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

Relative least general generalisation: example

- ☞ restrictions on existential variables
- ☞ remove as many body literals as possible

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

append([A|B],C,[A|E]):-
  append(B,C,D),append([],C,C)
```

Reducing RLGG's


```
% remove redundant literals
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).

% reduce_negs(H,B1,B0,B,N,M) <- B is a subsequence of B1
%                               such that H:-B does not
%                               cover elements of N
reduce_negs(H,[L|B0],In,B,Negs,M):-
   append(In,B0,Body),
   not covers_neg((H:-Body),Negs,M,N),!,   % remove L
   reduce_negs(H,B0,In,B,Negs,M).
reduce_negs(H,[L|B0],In,B,Negs,M):-        % keep L
   reduce_negs(H,B0,[L|In],B,Negs,M).
reduce_negs(H,[],Body,Body,Negs,M):-       % fail if clause
   not covers_neg((H:-Body),Negs,M,N).     % covers neg.ex.

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

Reducing RLGG's (cont.)

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

% covering algorithm
covering(Poss,Negs,Model,H0,H):-
   construct_hypothesis(Poss,Negs,Model,Hyp),!,
   remove_pos(Poss,Model,Hyp,NewPoss),
   covering(NewPoss,Negs,Model,[Hyp|H0],H).
covering(P,N,M,H0,H):-
   append(H0,P,H).   % add uncovered examples to hypothesis
```

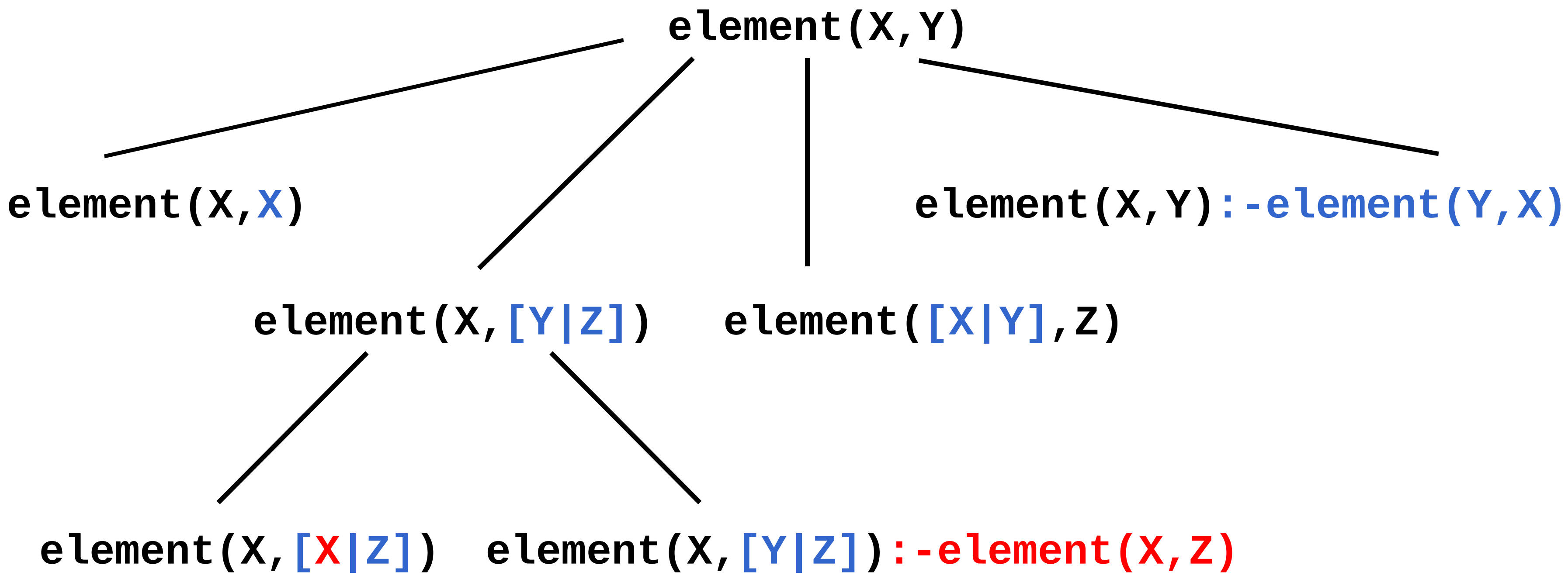
Top-level algorithm


```
% construct a clause by means of RLGG
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),!,    % no backtracking
   nl,tab(5),write(Clause),nl.
construct_hypothesis([E1,E2|Es],Negs,Model,Clause):-
   write(' too general'),nl,
   construct_hypothesis([E2|Es],Negs,Model,Clause).
```

Top-level algorithm (cont.)

```
% remove covered positive examples
remove_pos([],M,H,[]).
remove_pos([P|Ps],Model,Hyp,NewP):-
  covers_ex(Hyp,P,Model),!,
  write('Covered example: '),write(P),nl,
  remove_pos(Ps,Model,Hyp,NewP).
remove_pos([P|Ps],Model,Hyp,[P|NewP]):-
  remove_pos(Ps,Model,Hyp,NewP).
```

Top-level algorithm (cont.)



Part of the specialisation graph for `element/2`


```
literal(element(X,Y),[item(X),list(Y)]).

term(list([]),[]).
term(list([X|Y]),[item(X),list(Y)]).
```

Representation of a node in the specialisation graph:

```
a((element(X,[V|W]):-true),[item(X),item(V),list(W)])
```

Employing types


```
% specialise_clause(C,S) <- S is minimal specialisation
%                             of C under theta-subsumption
specialise_clause(Current,Spec):-
   add_literal(Current,Spec).
specialise_clause(Current,Spec):-
   apply_subs(Current,Spec).

add_literal(a((H:-true),Vars),a((H:-L),Vars)):-!,
   literal(L,LVars),
   proper_subset(LVars,Vars).  % no new variables in L
add_literal(a((H:-B),Vars),a((H:-L,B),Vars)):-
   literal(L,LVars),
   proper_subset(LVars,Vars).  % no new variables in L

apply_subs(a(Clause,Vars),a(Spec,SVars)):-
   copy_term(a(Clause,Vars),a(Spec,Vs)), % don't change
   apply_subs1(Vs,SVars).                % Clause
```

Generating the specialisation graph


```
apply_subs1(Vars,SVars):-
   unify_two(Vars,SVars).      % unify two variables
apply_subs1(Vars,SVars):-
   subs_term(Vars,SVars).      % subs. term for variable

unify_two([X|Vars],Vars):-     % not both X and Y in Vars
   element(Y,Vars),
   X=Y.
unify_two([X|Vars],[X|SVars]):-
   unify_two(Vars,SVars).

subs_term(Vars,SVars):-
   remove_one(X,Vars,Vs),
   term(Term,TVars),
   X=Term,
   append(Vs,TVars,SVars).     % TVars instead of X in Vars
```

Generating the specialisation graph (cont.)

```
% search_clause(Exs,E,C) <- C is a clause covering E and not covering
%                           negative examples (iterative deepening)
search_clause(Exs,Example,Clause):-
   literal(Head,Vars),      % root of specialisation graph
   try((Head=Example)),
   search_clause(3,a((Head:-true),Vars),Exs,Example,Clause).

search_clause(D,Current,Exs,Example,Clause):-
   write(D),write('..'),
   search_clause_d(D,Current,Exs,Example,Clause),!.
search_clause(D,Current,Exs,Example,Clause):-
   D1 is D+1,
   !,search_clause(D1,Current,Exs,Example,Clause).

search_clause_d(D,a(Clause,Vars),Exs,Example,Clause):-
   covers_ex(Clause,Example,Exs),          % goal
   not((element(-N,Exs),covers_ex(Clause,N,Exs))),!.
search_clause_d(D,Current,Exs,Example,Clause):-
   D>0,D1 is D-1,
   specialise_clause(Current,Spec),        % specialise
   search_clause_d(D1,Spec,Exs,Example,Clause).
```

Searching the specialisation graph

```
% covers_ex(C,E,Exs) <- clause C extensionally
%                        covers example E
covers_ex((Head:-Body),Example,Exs):-
  try((Head=Example,covers_ex(Body,Exs))).

covers_ex(true,Exs):-!.
covers_ex((A,B),Exs):-!,
  covers_ex(A,Exs),
  covers_ex(B,Exs).
covers_ex(A,Exs):-
  element(+A,Exs).
covers_ex(A,Exs):-
  prove_bg(A).
```

Extensional coverage

```
% covers_d(Clauses,Ex) <- Ex can be proved from Clauses and
%                         background theory (max. 10 steps)
covers_d(Clauses,Example):-
    prove_d(10,Clauses,Example).

prove_d(D,Cls,true):-!.
prove_d(D,Cls,(A,B)):-!,
    prove_d(D,Cls,A),
    prove_d(D,Cls,B).
prove_d(D,Cls,A):-
    D>0,D1 is D-1,
    copy_element((A:-B),Cls), % make copy
    prove_d(D1,Cls,B).
prove_d(D,Cls,A):-
    prove_bg(A).
```

Intensional coverage

```
induce_spec(Examples,Clauses):-
  process_examples([],[],Examples,Clauses).

% process the examples
process_examples(Clauses,Done,[],Clauses).
process_examples(Cls1,Done,[Ex|Exs],Clauses):-
  process_example(Cls1,Done,Ex,Cls2),
  process_examples(Cls2,[Ex|Done],Exs,Clauses).
```

Top-level algorithm


```
% process one example
process_example(Clauses,Done,+Example,Clauses):-
   covers_d(Clauses,Example).
process_example(Cls,Done,+Example,Clauses):-
   not covers_d(Cls,Example),
   generalise(Cls,Done,Example,Clauses).
process_example(Cls,Done,-Example,Clauses):-
   covers_d(Cls,Example),
   specialise(Cls,Done,Example,Clauses).
process_example(Clauses,Done,-Example,Clauses):-
   not covers_d(Clauses,Example).
```

Top-level algorithm (cont.)

```
generalise(Cls,Done,Example,Clauses):-
    search_clause(Done,Example,Cl),
    write('Found clause: '),write(Cl),nl,
    process_examples([Cl|Cls],[],[+Example|Done],Clauses).

specialise(Cls,Done,Example,Clauses):-
    false_clause(Cls,Done,Example,C),
    remove_one(C,Cls,Cls1),
    write('.....refuted: '),write(C),nl,
    process_examples(Cls1,[],[-Example|Done],Clauses).
```

Generalisation and specialisation