G53KRR Handout on Horn clauses, SLD resolution, backward chaining

Horn clauses A Horn clause is a clause which contains at most one positive literal:

$$[\neg Child(x), \neg Female(x), Girl(x)]$$

$$[\neg Girl(x)]$$

Positive and negative Horn clauses

- Horn clauses which do contain a positive literal are called positive or definite clauses.
- Horn clauses which do not contain a positive literal are called negative clauses (or sometimes *goals*)

Note that positive Horn clauses are equivalent to FOL sentences of the form:

$$\forall x_1 \ldots \forall x_n (\rho_1 \wedge \ldots \wedge \rho_n \supset \rho)$$

For example,

$$\forall x (Child(x) \wedge Female(x) \supset Girl(x))$$

$$Female(a)$$

SLD resolution SLD stands for **S**elected **L**iterals, **L**inear pattern, **D**efinite Clauses.

An SLD derivation of a clause c from a set of clauses S is a sequence

$$c_1, \ldots, c_n$$

such that

- $c_n = c$,
- $c_1 \in S$
- and each c_{i+1} is a resolvent of c_i and some clause from S.

Note that apart from c_1 all clauses in an SLD derivation are negative clauses (because resolution always 'eats up' the only positive literal in the clause from S). If S is a set of Horn clauses and from S by resolution we can derive [ ], then we can always derive [ ] from S using SLD resolution. SLD resolution is complete for refutation for Horn clauses, but not for arbitrary clauses.

Example The process of deriving an empty clause can be seen as 'eliminating' negative literals using positive clauses in the KB. For example,

$$Child \wedge Female \supset Girl$$

$$Toddler \supset Child$$

$$Toddler$$

$$Female$$

and we want to derive $Girl$, so add a clause $\neg Girl$. First we have one goal (negative clause) $Girl$; we eliminate it against $[\neg Child, \neg Female, Girl]$ and get two new subgoals: $Child$, $Female$. Goal tree:

```
          Girl
        /______\ (joint)
       /        \
   Child      Female
     |
   Toddler
```

SLD resolution derivation:

$c_1 = \neg Girl$

$c_2 = [\neg Child, \neg Female]$ (by resolution from c_1 and $[\neg Child, \neg Female, Girl]$)

$c_3 = [\neg Toddler, \neg Female]$ (by resolution from c_2 and $[\neg Toddler, Child]$)

$c_4 = [\neg Female]$ (by resolution from c_3 and $[Toddler]$)

$c_5 = [\ ]$ ((by resolution from c_4 and $[Female]$).

Backward chaining for propositional clauses (first order requires unification):
input: a finite set of atomic sentences $q_1, \ldots, q_n$
output: YES if KB entails all of q_i, NO otherwise

```
procedure: SOLVE[q1, ..., qn ]
if n = 0 then return YES
for each clause c in KB do
     if c = [not p1,...,not pm, q1] and SOLVE [p1,...,pm,q2,...,qn]
     then return YES
end for
return NO
```

Backward chaining on an example:

```
SOLVE[Girl]
   c = [Girl, not Child,not Female] call SOLVE [Child, Female]
   c = [not Toddler, Child] call SOLVE[Toddler,Female]
   c = [Toddler] call SOLVE[Female]
   c = [Female] call SOLVE[] return YES
```

PROCEDURAL CONTROL OF REASONING. PROLOG. Procedural control in the backward chaining procedure:

- Depth first (first solve ps rather than qs)
- Left to right (solve in order as they are listed)
- Backward chaining because search from goals to facts in KB

First order case requires unification, but the order is the same. This is the execution strategy of Prolog.

Prolog Prolog is a (logic) programming language where programs consist of facts and rules (Horn clauses):

```
parent(john, tom).
father(X, Y) :- parent(X,Y), male(X).
```

This is the same as:

$$Parent(john, tom)$$

$$\forall x \forall y (Parent(x, y) \wedge Male(x) \supset Father(x, y)$$

(Note that in Prolog variables are upper case and constants and predicate names are lower case. Every clause ends with a dot, and implications are pointing right to left.)