HORN CLAUSES, BACKWARD CHAINING

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

\[
\neg \text{Child}(x), \neg \text{Female}(x), \text{Girl}(x)
\]

\[
\neg \text{Girl}(x)
\]

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:

\[
\forall x_1 \ldots \forall x_n (\rho_1 \land \ldots \land \rho_n \supset \rho)
\]

For example,

\[
\forall x (\text{Child}(x) \land \text{Female}(x) \supset \text{Girl}(x))
\]

SLD resolution  SLD stands for Selected Literals, Linear pattern, Definite 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 \(c_{i+1}\) 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.

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

\[
\text{Child} \land \text{Female} \supset \text{Girl}
\]

\[
\text{Toddler} \supset \text{Child}
\]

\[
\text{Toddler}
\]

\[
\text{Female}
\]

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

procedure: SOLVE[q_1, ..., q_n ]
if n = 0 then return YES
for each clause c in KB do
  if c = [not p_1,...,not p_m, q_1] and SOLVE [p_1,...,p_m,q_2,...,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]
  SOLVE[Toddler,Female]:
    c = [Toddler]
    SOLVE[Female]:
    c = [Female]
    SOLVE[] return YES
• 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):

\[ parent(john, tom) \]

\[ father(X, Y) : \neg parent(X, Y), male(X) \]

**Clause and goal ordering**  Some clauses may cause an infinite loop: for example,

\[ p : \neg p \]

Some may cause an infinite loop with one clause ordering and goal ordering and not with another:

\[ ancestor(X, Y) : \neg ancestor(X, Z), parent(Z, Y) \]

\[ ancestor(X, Y) : \neg parent(X, Y) \]

will go into an infinite loop if asked about ancestors of a particular individual, and

\[ ancestor(X, Y) : \neg parent(X, Y) \]

\[ ancestor(X, Y) : \neg parent(Z, Y), ancestor(X, Z) \]

will not.

Sometimes one order of goals is more efficient than another:

\[ americanCousin(X, Y) : \neg american(X), cousin(X, Y) \]

vs

\[ americanCousin(X, Y) : \neg cousin(X, Y), american(X) \]

Consider a goal \( americanCousin(x, sally) \): first program will need to check all Americans and then check if any of them is Sally’s cousin; the second one checks for all cousins of Sally if they are an American.

**Backtracking**  There may be several ways to solve a goal; for example there could be lots of facts of the form \( american(john), american(tom) \), etc. When we are trying to prove \( americanCousin(X, sally) \) the first way, we try to solve \( american(X) \) using all of those facts and backtrack when we discover that e.g. Tom is not Sally’s cousin, and try to solve it using the next American.
**Cut in Prolog**  Cut is used to control backtracking.

\[ g : -t_1, t_2, \ldots, t_m, !, g_1, \ldots, g_n \]

means: if \( t_1, \ldots, t_m \) succeeded, commit, don’t backtrack if any of \( g_1, \ldots, g_n \) fails and don’t try other ways of solving \( g \). This makes reasoning a lot more efficient.

\[ answer(X, Y) : -property1(X, Y), property2(X) \]

Suppose \( property1(john, alpineClub) \) succeeds and \( property2(john) \) fails. Then we should not backtrack and try to prove \( property1(john, alpineClub) \) in some other way:

\[ answer(X, Y) : -property1(X, Y), !, property2(X) \]

**Negation as failure in Prolog**  It is a useful consequence of having procedural control of reasoning that we can define negation as failure: if you cannot derive \( g \), derive not \( g \) (provided failure to derive something can be discovered in a finite amount of time).

\[ noChildren(X) : -\text{not}(parent(X, Y)) \]

(if cannot derive \( parent(X, Y) \) for any \( Y \), derive \( noChildren(X) \)).

**A DIFFERENT WAY OF REASONING WITH HORN CLAUSES**

**Forward chaining .**

**input:** an atomic sentence \( q \)

**output:** YES if KB entails \( q \), NO otherwise

1. if \( q \) is marked as solved, return YES
2. if there is \([-p_1, \ldots, -p_n, q_1]\) in KB such that \( p_1, \ldots, p_n \) are marked as solved and \( q_1 \) is not marked solved: mark \( q_1 \) as solved and go to 1; else return NO.

Forward chaining on an example:
Given \( KB = \{[\sim Child, \sim Female, Girl], [\sim Toddler, Child], [Toddler], [Female]\} \), is \( Girl \) entailed?

mark Toddler
mark Child
mark Female
mark Girl
return YES

Propositional forward chaining is linear in the size of \( KB \). Unfortunately for a large \( KB \) it produces lots of not required consequences, goal-directed (backward chaining) inference may be faster.