Horn clauses

Clauses are used two ways:

- as disjunctions: \((\text{rain} \lor \text{sleet})\)
- as implications: \((\neg \text{child} \lor \neg \text{male} \lor \text{boy})\)

Here focus on 2nd use

Horn clause = at most one +ve literal in clause

- positive / definite clause = exactly one +ve literal
  \([-\neg p_1, \neg p_2, ..., \neg p_n, q]\)
- negative clause = no +ve literals
  \([-\neg p_1, \neg p_2, ..., \neg p_n]\)

Note

\([-\neg p_1, \neg p_2, ..., \neg p_n, q]\) is a representation for

\((-\neg p_1 \lor -\neg p_2 \lor ... \lor -\neg p_n \lor q)\) or

\([p_1 \land p_2 \land ... \land p_n \supset q]\)

So can read as

If \(p_1\) and \(p_2\) and ... and \(p_n\) then \(q\)

and write sometimes as

\(p_1 \land p_2 \land ... \land p_n \Rightarrow q\)

Resolution with Horn clauses

Only two possibilities:

\[
\begin{array}{c}
\text{Neg} \quad \text{Pos} \\
\text{Neg} \quad \text{Pos} \\
\end{array}
\]

It is possible to rearrange derivations (of negative clauses) so that all new derived clauses are negative clauses

\[
\begin{array}{c}
[\alpha, \neg q, p] \\
[\gamma, \neg p] \\
[\alpha, \neg q, p] \\
\end{array} \quad \begin{array}{c}
[\beta, q] \\
[\alpha, \neg q, p] \\
[\beta, q] \\
\end{array}
\]

Can also change derivations such that each derived clause is a resolvent of the previous derived one (-ve) and some +ve clause in the original set of clauses

Since each derived clause is negative, one parent must be positive (and so from original set) and one negative.

Continue working backwards until both parents of derived clause are from the original set of clauses

Eliminate all other clauses not on direct path
SLD Resolution

Recurring pattern in derivations:

See previously:
- example 1
- example 3
- arithmetic example

But not:
- example 2
- 3 block example

An SLD-derivation of a clause \(c\) from a set of clauses \(S\) is a sequence of clause \(c_1, c_2, \ldots, c_n\) such that \(c_n = c\), and

1. \(c_1 \in S\)
2. \(c_{i+1}\) is a resolvent of \(c_i\) and a clause in \(S\)

Write: \(S \vdash c\)

Note: SLD derivation is just a special form of derivation and where we leave out the elements of \(S\) (except \(c_1\))

SLD means:
- (elected) literals
- (linear) form
- (definite) clauses

Completeness of SLD

In general, cannot restrict Resolution steps to always use a clause that is in the original set

Proof:
\[ S = \{[p, q], [p, \neg q], [\neg p, q], [\neg p, \neg q]\} \]
then \(S \vdash \square\).

Need to resolve some \([l]\) and \([\neg l]\) to get \([\square]\). But \(S\) does not contain any unit clauses.

So will need to derive both \([l]\) and \([\neg l]\) and then resolve them together.

But can do so for Horn clauses...

Theorem: for Horn clauses, \(H \vdash \square\) iff \(H \vdash_{SLD} \square\)

So: \(H\) is unsatisfiable iff \(H \vdash_{SLD} \square\)

This will considerably simplify the search for derivations

Note: in Horn version of SLD-Resolution, each clause \(c_1, c_2, \ldots, c_n\) will be negative

So clauses \(H\) must always contain at least one negative clause, \(c_1\).
Example 1 (again)

KB:
- FirstGrade
- FirstGrade → Child
- Child ∧ Male → Boy
- Kindergarten → Child
- Child ∧ Female → Girl
- Female

Show KB ∪ {¬Girl} unsatisfiable

![Goal tree diagram]

A goal tree whose nodes are atoms, whose root is the atom to prove, and whose leaves are in the KB.

Prolog

Horn clauses form the basis of Prolog

Append(nil, y, y)
Append(x, y, z) ⇒ Append(cons(w, x), y, cons(w, z))

Append(cons(a, cons(b, nil)), cons(c, nil), u)
good

| u / cons(a, u')

Append(cons(b, nil), cons(c, nil), u')

| u' / cons(b, u'')

Append(nil, cons(c, nil), u'')

solved: u'' / cons(c, nil)

So goal succeeds with u = cons(a, cons(b, cons(c, nil))))
that is: Append([a b], [c], [a b c])

With SLD derivation, can always extract answer from proof

H ⊨ ∃x α(x) iff for some term t, H ⊨ α(t)

Different answers can be found by finding other derivations
**Back-chaining procedure**

Satisfiability of a set of Horn clauses with exactly one negative clause

\[ \text{Solve}\{q_1, q_2, \ldots, q_n\} = \]

\[ \text{if } n=0 \text{ then return } \text{YES}; \]

\[ \text{for each } d \in \text{KB} \text{ do} \]

\[ \text{if } d = [q_1, \neg p_1, \neg p_2, \ldots, \neg p_m] \]

\[ \text{and return } \text{YES} \]

\[ \text{end for; return NO} \]

Depth-first, left-right, back-chaining

- depth-first because attempt \( p_i \) before trying \( q_i \)
- left-right because try \( q_i \) in order, 1, 2, 3, ...
- back-chaining because search from goal \( q \) to facts in KB \( p \)

This is the execution strategy of Prolog

First-order case requires unification etc.

**Problems with back-chaining**

Can go into infinite loop

Tautologous clause: \([p, \neg p]\)

Corresponds to Prolog program with \( p :- p. \)

Previous back-chaining algorithm is inefficient

Example: consider \( 2n \) atoms: \( p_1, \ldots, p_n, q_1, \ldots, q_n \)

And \( 4n - 4 \) clauses:

\( p_i \Rightarrow p_{i+1}, \quad q_i \Rightarrow p_{i+1}, \)

\( p_i \Rightarrow q_{i+1}, \quad q_i \Rightarrow q_{i+1}. \)

With goal \( p_n \) has execution tree like this

\[ \text{search eventually fails after } 2^n \text{ steps!} \]
Forward-chaining

Simple procedure to determine if Horn KB \( \models q \).

main idea: mark atoms as solved

1. If \( q \) is marked as solved, then return YES
2. Is there a \( \{p_1, \neg p_2, \ldots, \neg p_n\} \in KB \) such that
   \( p_2, \ldots, p_n \) are marked as solved, but the
   positive lit \( p_1 \) is not marked as solved?
   no: return NO
   yes: mark \( p_1 \) as solved, and go to 1.

FirstGrade example:
Marks: FirstGrade, Child, Female, Girl
then done!

Observe:
- only letters in KB can be marked, so at most a linear
  number of iterations
- not goal-directed, so not always desirable

A similar procedure with better data structures
will run in linear time overall

First-order undecidability

Even with just Horn clauses, in the first-order
case we still have the possibility of generating an
infinite branch of resolvents

KB: LessThan(succ(x),y) \( \Rightarrow \) LessThan(x,y)
Q: LessThan(zero,zero)

As with full Resolution, there is no way to detect
when this will happen

So there is no procedure
that will test for satisfiability
of first-order Horn clauses
the question is undecidable

As with full clauses, the best that can be
expected is to give control of the deduction to the
user
to some extent this is what is done in Prolog,
but we will see more in "Procedural Control"