5.

Reasoning with Horn Clauses

Horn clauses

Clauses are used two ways:

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

Here focus on 2nd use

Horn clause = at most one +ve literal in clause

- positive / definite clause = exactly one +ve literal

  e.g. $[\quad p_1, \quad p_2, \ldots, \quad p_n, q]$

- negative clause = no +ve literals

  e.g. $[\quad p_1, \quad p_2, \ldots, \quad p_n]$ and also [ ]

Note: $[\quad p_1, \quad p_2, \ldots, \quad p_n, q]$ is a representation for

$(\quad p_1 \lor \quad p_2 \lor \ldots \lor \quad p_n \lor q)$ or $[(p_1 \land p_2 \land \ldots \land p_n) \supset q]$

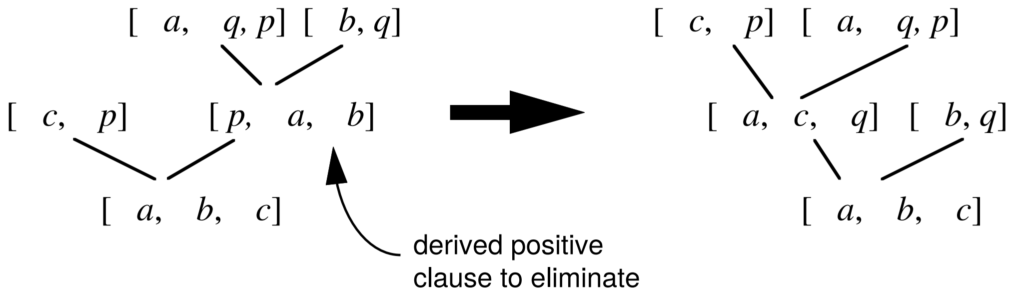
so can read as: If p_1 and p_2 and ... and p_n then q

and write as: $p_1 \land p_2 \land \ldots \land p_n \Rightarrow q$ or $q \Leftarrow p_1 \land p_2 \land \ldots \land p_n$

Resolution with Horn clauses

Only two possibilities:

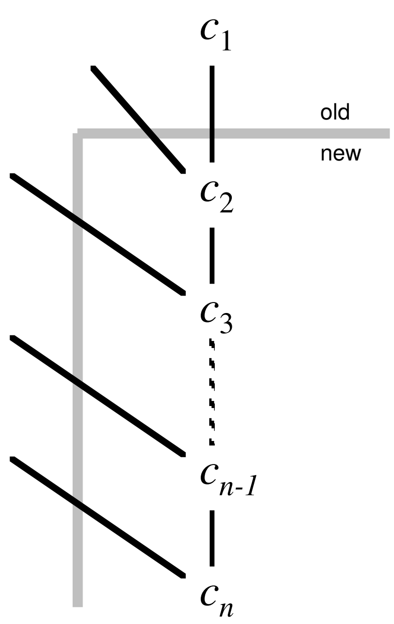
It is possible to rearrange derivations of negative clauses so that all new derived clauses are negative



Further restricting resolution

Can also change derivations such that each derived clause is a resolvent of the previous derived one (negative) and some positive 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 parent must be negative.
- Chain backwards from the final negative clause until both parents are from the original set of clauses
- Eliminate all other clauses not on this direct path



This is a recurring pattern in derivations

- See previously:
  - example 1, example 3, arithmetic example
- But not:
  - example 2, the 3 block example

SLD Resolution

An <u>SLD-derivation</u> of a clause c from a set of clauses S is a sequence of clause c_1, c_2, ... 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: $S \overset{\text{SLD}}{\rightarrow} c$

SLD means S(elected) literals
L(inear) form
D(efinite) clauses

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

In general, cannot restrict ourselves to just using SLD-Resolution

Proof: $S = \{[p, q], [p, \quad q], [\quad p, q] [\quad p, \quad q]\}$. Then $S \rightarrow []$.

Need to resolve some $[\,\rho\,]$ and $[\,\overline{\rho}\,]$ to get [].
But S does not contain any unit clauses.

So will need to derive both $[\,\rho\,]$ and $[\,\overline{\rho}\,]$ and then resolve them together.

Completeness of SLD

However, for Horn clauses, we can restrict ourselves to SLD-Resolution

Theorem: SLD-Resolution is refutation complete for Horn clauses: $H \rightarrow []$ iff $H \overset{\text{SLD}}{\rightarrow} []$

So: H is unsatisfiable iff $H \overset{\text{SLD}}{\rightarrow} []$

This will considerably simplify the search for derivations

Note: in Horn version of SLD-Resolution, each clause in the $c_1, c_2, ..., c_n$, will be negative

So clauses H must contain at least one negative clause, c_1 and this will be the only negative clause of H used.

Typical case:

- KB is a collection of positive Horn clauses
- Negation of query is the negative clause

Example 1 (again)

KB

FirstGrade
FirstGrade ⊃ Child
Child ∧ Male ⊃ Boy
Kindergarten ⊃ Child
Child ∧ Female ⊃ Girl
Female

SLD derivation

[Girl]

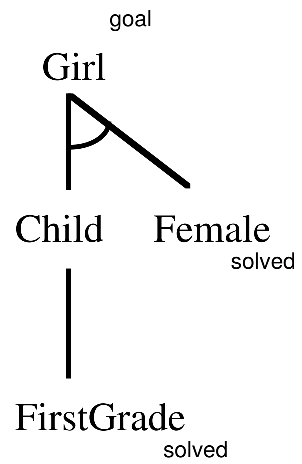
[Child, Female]

[Child]

[FirstGrade]

[]

alternate representation



Show KB ∪ { Girl} unsatisfiable

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

$\text{Append}(\text{nil},y,y)$

$\text{Append}(x,y,z) \Rightarrow \text{Append}(\text{cons}(w,x),y,\text{cons}(w,z))$

With SLD derivation, can always extract answer from proof

$H \models \exists x\, \alpha(x)$

iff

for some term t, $H \models \alpha(t)$

Different answers can be found by finding other derivations

What is the result of appending [c] to the list [a,b] ?

$\text{Append}(\text{cons}(\text{a},\text{cons}(\text{b},\text{nil})),\ \text{cons}(\text{c},\text{nil}),\ u)$ goal

$u\ /\ \text{cons}(\text{a},u')$

$\text{Append}(\text{cons}(\text{b},\text{nil}),\ \text{cons}(\text{c},\text{nil}),\ u')$

$u'\ /\ \text{cons}(\text{b},u'')$

$\text{Append}(\text{nil},\ \text{cons}(\text{c},\text{nil}),\ u'')$

solved: $u''\ /\ \text{cons}(\text{c},\text{nil})$

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

Back-chaining procedure

Solve[q_1, q_2, ..., q_n] = /* to establish conjunction of q_i */

```
If n=0 then return YES;   /* empty clause detected */
For each d ∈ KB do
    If d = [q1, ¬p1, ¬p2, ..., ¬pm]     /* match first q */
        and                              /* replace q by -ve lits */
      Solve[p1, p2, ..., pm, q2, ..., qn] /* recursively */
    then return YES
end for;            /* can't find a clause to eliminate q */
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

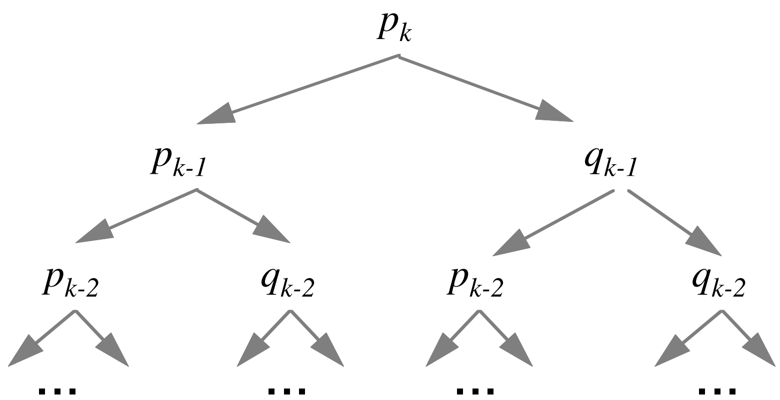
tautologous clause: $[p\ ,\quad p]$ (corresponds to Prolog program with `p :- p`).

Previous back-chaining algorithm is inefficient

Example: Consider 2n atoms, $p_0, \ldots, p_{n-1}, q_0, \ldots, q_{n-1}$ and $4n$-4 clauses

$$(p_{i-1} \Rightarrow p_i),\ (q_{i-1} \Rightarrow p_i),\ (p_{i-1} \Rightarrow q_i),\ (q_{i-1} \Rightarrow q_i).$$

With goal p_k the execution tree is like this



Solve[p_k] eventually fails after 2^k steps!

Is this problem inherent in Horn clauses?

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 $\{p_1, \; p_2, ..., \; p_n\} \in$ KB such that $p_2, ..., p_n$ 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!

Note: FirstGrade gets marked since all the negative atoms in the clause (none) are marked

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)

Query:

LessThan(zero,zero)

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

[LessThan(0,0)]

$\downarrow$ $x/0,\ y/0$

[LessThan(1,0)]

$\downarrow$ $x/1,\ y/0$

[LessThan(2,0)]

$\downarrow$ $x/2,\ y/0$

...

There is no procedure that will test for the satisfiability of first-order Horn clauses

the question is undecidable

As with non-Horn clauses, the best that we can do 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”