LECTURE NOTES–UNIT IV

INFERENCE IN FIRST-ORDER LOGIC

ECS302

ARTIFICIAL INTELLIGENCE

Module IV Lecture Notes [*Inference in First-Order Logic*]

Syllabus

Knowledge and Reasoning: Inference in First-Order Logic: *Propositional vs first order inference, unification and lifting, forward chaining, backward chaining, resolution.*

1. Propositional Vs First Order Inference

Earlier inference in first order logic is performed with *Propositionalization* which is a process of converting the Knowledgebase present in First Order logic into Propositional logic and on that using any inference mechanisms of propositional logic are used to check inference.

Inference rules for quantifiers:

There are some Inference rules that can be applied to sentences with quantifiers to obtain sentences without quantifiers**.** These rules will lead us to make the conversion.

Universal Instantiation (UI):

The rule says that we can infer any sentence obtained by substituting a **ground term** (a term without variables) for the variable. Let SUBST (θ*)* denote the result of applying the substitution θ to the sentence *a.* Then the rule is written

$$\frac{\forall v \;\; a}{\text{SUBST}(\{v/g\}, \alpha)}$$

For any variable v and ground term *g*.

For example, there is a sentence in knowledge base stating that all greedy kings are Evils

$$\forall x \;\; King(x) \; A \; Greedy(x) \Rightarrow Evil(x) \; .$$

For the variable x, with the substitutions like {x/John},{x/Richard}the following sentences can be inferred.

$$King(John) \; A \; Greedy(John) \Rightarrow Evil(John).$$
$$King(Richard) \; A \; Greedy(Richard) \Rightarrow Evil(Richard) \; .$$

Thus a universally quantified sentence can be replaced by the set of *all* possible instantiations.

Existential Instantiation (EI):

The existential sentence says there is some object satisfying a condition, and the instantiation process is just giving a name to that object, that name must not already belong to another object. This new name is called a **Skolem constant.** Existential Instantiation is a special case of a more general process called *"skolemization".*

For any sentence *a,* variable v, and constant symbol *k* that does not appear elsewhere in the knowledge base,

$$\frac{\exists v \ \alpha}{\text{SUBST}(\{v/k\}, \alpha)}$$

For example, from the sentence

$$\exists x \ \ Crown(x) \wedge OnHead(x, John)$$

So, we can infer the sentence

$$Crown(C_1) \wedge OnHead(C_1, John)$$

As long as C_1 does not appear elsewhere in the knowledge base. Thus an existentially quantified sentence can be replaced by one instantiation

Elimination of Universal and Existential quantifiers should give new knowledge base which can be shown to be *inferentially equivalent* to old in the sense that it is satisfiable exactly when the original knowledge base is satisfiable.

Reduction to propositional inference:

Once we have rules for inferring non quantified sentences from quantified sentences, it becomes possible to reduce first-order inference to propositional inference. For example, suppose our knowledge base contains just the sentences

$$\forall x \ \ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$$
$$King(John)$$
$$Greedy(John)$$
$$Brother(Richard, John).$$

Then we apply UI to the first sentence using all possible ground term substitutions from the vocabulary of the knowledge base-in this case, *{xl John)* and *{x/Richard)*. We obtain

$$King(John) \wedge Greedy(John) \Rightarrow Evil(John),$$
$$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$$

We discard the universally quantified sentence. Now, the knowledge base is essentially propositional if we view the ground atomic sentences-King *(John), Greedy (John),* and Brother (Richard, *John*) as proposition symbols. Therefore, we can apply any of the complete propositional algorithms to obtain conclusions such as *Evil (John).*

Disadvantage:

If the knowledge base includes a function symbol, the set of possible ground term substitutions is infinite. Propositional algorithms will have difficulty with an infinitely large set of sentences.

NOTE:

Entailment for first-order logic is *semi decidable* which means algorithms exist that say yes to every entailed sentence, but no algorithm exists that also says no to every non entailed sentence

2. Unification and Lifting

Consider the above discussed example, if we add Siblings (Peter, Sharon) to the knowledge base then it will be

$$\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$$
$$King(John)$$
$$Greedy(John)$$
$$Brother(Richard, John)$$
$$\mathbf{Siblings(Peter, Sharon)}$$

Removing Universal Quantifier will add new sentences to the knowledge base which are not necessary for the query *Evil (John)?*

$$King(John) \wedge Greedy(John) \Rightarrow Evil(John)$$
$$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$$
$$King(Peter) \wedge Greedy(Peter) \Rightarrow Evil(Peter)$$
$$King(Sharon) \wedge Greedy(Sharon) \Rightarrow Evil(Sharon)$$

Hence we need to teach the computer to make better inferences. For this purpose Inference rules were used.

First Order Inference Rule:

The key advantage of lifted inference rules over *propositionalization* is that they make only those substitutions which are required to allow particular inferences to proceed.

Generalized Modus Ponens:

If there is some substitution 8 that makes the premise of the implication identical to sentences already in the knowledge base, then we can assert the conclusion of the implication, after applying 8. This inference process can be captured as a single inference rule called General ized Modus Ponens which is a ***lifted*** version of Modus Ponens-it raises Modus Ponens from propositional to first-order logic

For atomic sentences pi, pi ', and q, where there is a substitution θ such that SUBST(θ , pi) = SUBST(θ , pi '), for all i,

$$\frac{\mathbf{p1',\ p2',\ \ldots,\ pn',\ (p1 \wedge p2 \wedge \ldots \wedge pn \Rightarrow q)}}{\mathbf{SUBST(\theta,\ q)}}$$

There are N + 1 premises to this rule, N atomic sentences + one implication.

Applying SUBST (θ, q) yields the conclusion we seek. It is a sound inference rule.

Suppose that instead of knowing Greedy (John) in our example we know that everyone is greedy:

∀y Greedy(y)

We would conclude that Evil(John).

Applying the substitution {x/John, y / John) to the implication premises King (x) and Greedy (x) and the knowledge base sentences King(John) and Greedy(y) will make them identical. Thus, we can infer the conclusion of the implication.

For our example,

p_1' is $King(John)$ p_1 is $King(x)$
p_2' is $Greedy(y)$ p_2 is $Greedy(x)$
θ is $\{x/John, y/John)$ q is $Evil(x)$
SUBST(θ, q) is $Evil(John)$.

Unification:

It is the process used to find substitutions that make different logical expressions look identical. **Unification** is a key component of all first-order Inference algorithms.

UNIFY (p, q) = θ where SUBST (θ, p) = SUBST (θ, q) θ is our unifier value (if one exists).

Ex: "Who does John know?"

UNIFY (Knows (John, x), Knows (John, Jane)) = {x/ Jane}.

UNIFY (Knows (John, x), Knows (y, Bill)) = {x/Bill, y/ John}.

UNIFY (Knows (John, x), Knows (y, Mother(y))) = {x/Bill, y/ John}

UNIFY (Knows (John, x), Knows (x, Elizabeth)) = FAIL

- The last unification fails because both use the same variable, X. X can't equal both John and Elizabeth. To avoid this change the variable X to Y (or any other value) in Knows(X, Elizabeth)

Knows(X, Elizabeth) → Knows(Y, Elizabeth)

Still means the same. This is called **standardizing apart.**

- sometimes it is possible for more than one unifier returned:

UNIFY (Knows (John, x), Knows(y, z)) =???

This can return two possible unifications: {y/ John, x/ z} which means Knows (John, z) OR {y/ John, x/ John, z/ John}. For each unifiable pair of expressions there is a single **most general unifier (MGU)**, In this case it is *{y/ John, x/z)*.

An algorithm for computing most general unifiers is shown below.

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound
          y, a variable, constant, list, or compound
          θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
      return UNIFY(ARGS[x], ARGS[y], UNIFY(OP[x], OP[y], θ))
  else if LIST?(x) and LIST?(y) then
      return UNIFY(REST[x], REST[y], UNIFY(FIRST[x], FIRST[y], θ))
  else return failure
```

```
function UNIFY-VAR(var, x, θ) returns a substitution
  inputs: var, a variable
          x, any expression
          θ, the substitution built up so far

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

Figure 2.1 *The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression, such as F (A, B), the function OP picks out the function symbol F and the function ARCS picks out the argument list (A, B).*

The process is very simple: recursively explore the two expressions simultaneously "side by side," building up a unifier along the way, but failing if two corresponding points in the structures do not match. **Occur check** step makes sure same variable isn't used twice.

Storage and retrieval

- STORE(s) stores a sentence *s* into the knowledge base
- FETCH(s) returns all unifiers such that the query q unifies with some sentence in the knowledge base.

Easy way to implement these functions is Store all sentences in a long list, browse list one sentence at a time with UNIFY on an ASK query. But this is inefficient.

To make FETCH more efficient by ensuring that unifications are attempted only with sentences that have *some* chance of unifying. (i.e. Knows(John, x) vs. Brother(Richard, John) are not compatible for unification)

- To avoid this, a simple scheme called ***predicate indexing*** puts all the *Knows* facts in one bucket and all the *Brother* facts in another.
- The buckets can be stored in a hash table for efficient access. Predicate indexing is useful when there are many predicate symbols but only a few clauses for each symbol.

But if we have many clauses for a given predicate symbol, facts can be stored under multiple index keys.

For the fact *Employs (AIMA.org, Richard),* the queries are
Employs (A IMA. org, Richard) Does AIMA.org employ Richard?
Employs (x, Richard) who employs Richard?
Employs (AIMA.org, y) whom does AIMA.org employ?
Employs Y(x), who employs whom?

We can arrange this into a **subsumption lattice,** as shown below**.**

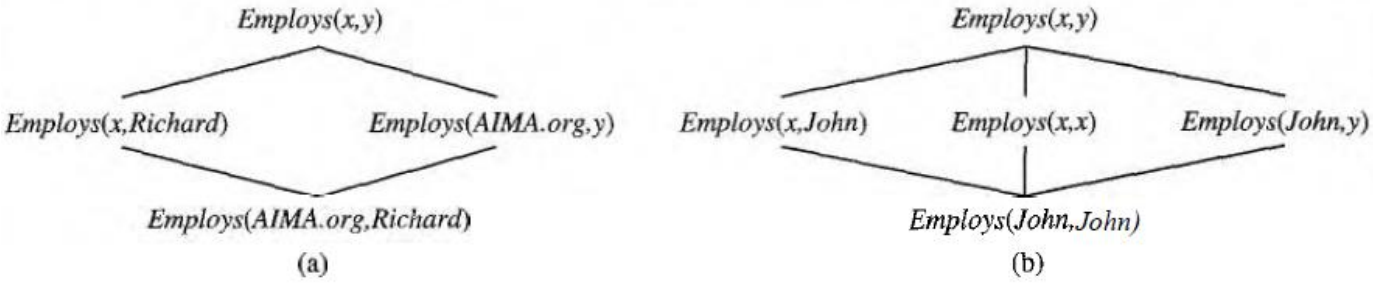


Figure 2.2 *(a) The subsumption lattice whose lowest node is the sentence Employs (AIMA.org, Richard). (b) The subsumption lattice for the sentence Employs (John, John).*

A subsumption lattice has the following properties:

- ✓ child of any node obtained from its parents by one substitution
- ✓ the "highest" common descendant of any two nodes is the result of applying their most general unifier
- ✓ predicate with n arguments contains O(2n) nodes (in our example, we have two arguments, so our lattice has four nodes)
- ✓ Repeated constants = slightly different lattice.

3. Forward Chaining

First-Order Definite Clauses:

A definite clause either is atomic or is an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal. The following are first-order definite clauses:

$$King(x) \wedge Greedy(x) \Rightarrow Evil(x) .$$
$$King(John) .$$
$$Greedy(y) .$$

Unlike propositional literals, first-order literals can include variables, in which case those variables are assumed to be universally quantified.

Consider the following problem;

"The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American."

We will represent the facts as first-order definite clauses

". . . It is a crime for an American to sell weapons to hostile nations":

$$American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x) \text{ -------- (1)}$$

"Nono . . . has some missiles." The sentence 3 x *Owns (Nono,* .rc) *A Missile* (x) is transformed into two definite clauses by Existential Elimination, introducing a new constant *M1:*

Owns (Nono, M1) ---------- (2)

Missile (Ml) ---------- (3)

"All of its missiles were sold to it by Colonel West":

***Missile (x) A Owns (Nono, x) =>Sells (West, z, Nono)* ---------- (4)**

We will also need to know that missiles are weapons:

Missile (x) => Weapon (x) ---------- (5)

We must know that an enemy of America counts as "hostile":

Enemy (x, America) =>Hostile(x) ---------- (6)

"West, who is American":

American (West) ---------- (7)

"The country Nono, an enemy of America ":

Enemy (Nono, America) ---------- (8)

A simple forward-chaining algorithm:

- Starting from the known facts, it triggers all the rules whose premises are satisfied, adding their conclusions lo the known facts
- The process repeats until the query is answered or no new facts are added. Notice that a fact is not "new" if it is just *renaming* of a known fact.

We will use our crime problem to illustrate how FOL-FC-ASK works. The implication sentences are (1), (4), (5), and (6). Two iterations are required:

- ✓ On the first iteration, rule (1) has unsatisfied premises.

  Rule (4) is satisfied with *{x/Ml),* and *Sells (West,* M1, *Nono)* is added.

  Rule (5) is satisfied with *{x/M1) and Weapon (M1)* is added.

  Rule (6) is satisfied with *{x/Nono},* and *Hostile (Nono)* is added.
- ✓ On the second iteration, rule (1) is satisfied with *{x/West, Y/MI, z /Nono),* and *Criminal (West)* is added.

It is ***sound***, because every inference is just an application of Generalized Modus Ponens, it is ***complete*** for definite clause knowledge bases; that is, it answers every query whose answers are entailed by any knowledge base of definite clauses

```
function FOL-FC-ASK(KB, a) returns a substitution or false
  inputs: KB, the knowledge base, a set of first-order definite clauses
          a, the query, an atomic sentence
  local variables: new, the new sentences inferred on each iteration

  repeat until new is empty
    new ← { }
    for each sentence r in KB do
      (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-APART(r)
      for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)
                  for some p'1, ..., p'n in KB
        q' ← SUBST(θ, q)
        if q' is not a renaming of some sentence already in KB or new then do
          add q' to new
          φ ← UNIFY(q', a)
          if φ is not fail then return φ
    add new to KB
  return false
```

Figure 3.1 *A conceptually straightforward, but very inefficient, forward-chaining algorithm. On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB.*

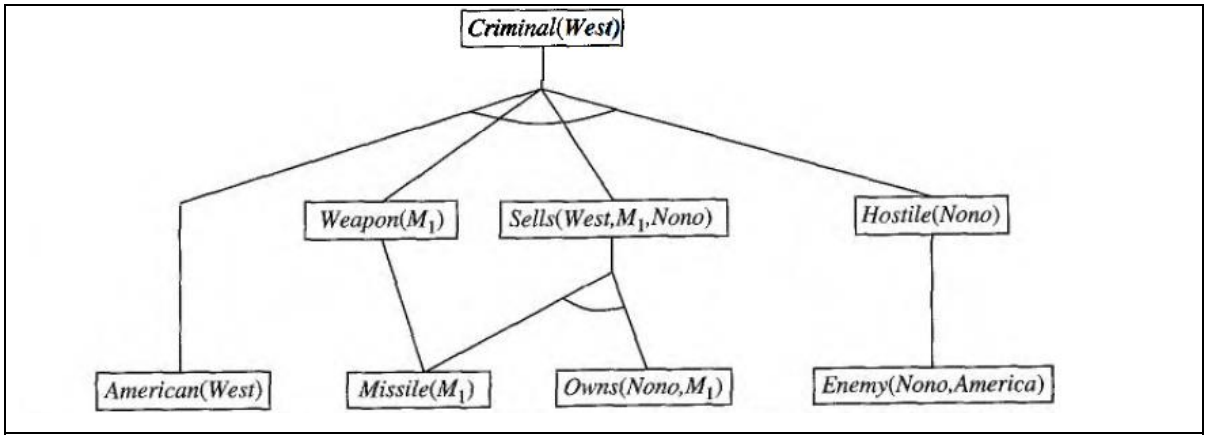


Figure 3.2 *The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.*

Efficient forward chaining:

The above given forward chaining algorithm was lack with efficiency due to the the three sources of complexities:

- ✓ Pattern Matching
- ✓ Rechecking of every rule on every iteration even a few additions are made to rules
- ✓ Irrelevant facts

1. *Matching rules against known facts:*

For example, consider this rule,

Missile(x) A Owns (Nono, x) => Sells (West, x, Nono).

The algorithm will check all the objects owned by Nono in and then for each object, it could check whether it is a missile. This is the ***conjunct ordering problem:***

"Find an ordering to solve the conjuncts of the rule premise so that the total cost is minimized". The **most constrained variable** heuristic used for CSPs would suggest ordering the conjuncts to look for missiles first if there are fewer missiles than objects that are owned by Nono.

The connection between pattern matching and constraint satisfaction is actually very close. We can view each conjunct as a constraint on the variables that it contains-for example, Missile(x) is a unary constraint on x. Extending this idea, we can express every finite-domain CSP as a single definite clause together with some associated ground facts. Matching a definite clause against a set of facts is NP-hard

2. Incremental forward chaining:

On the second iteration, the rule ***Missile (x) => Weapon (x)***

Matches against Missile (M1) (again), and of course the conclusion Weapon(x/M1) is already known so nothing happens. Such redundant rule matching can be avoided if we make the following observation:

"Every new fact inferred on iteration t must be derived from at least one new fact inferred on iteration t – 1".

This observation leads naturally to an incremental forward chaining algorithm where, at iteration t, we check a rule only if its premise includes a conjunct p, that unifies with a fact p: newly inferred at iteration t - 1. The rule matching step then fixes p, to match with p', but allows the other conjuncts of the rule to match with facts from any previous iteration.

3. Irrelevant facts:

- One way to avoid drawing irrelevant conclusions is to use backward chaining.
- Another solution is to restrict forward chaining to a selected subset of rules
- A third approach, is to rewrite the rule set, using information from the goal.so that only relevant variable bindings-those belonging to a so-called **magic** set-are considered during forward inference.

For example, if the goal is Criminal (West), the rule that concludes Criminal (x) will be rewritten to include an extra conjunct that constrains the value of x:

Magic(x) A American(z) A Weapon(y)A Sells(x, y, z) A Hostile(z) =>Criminal(x)

The fact *Magic (West)* is also added to the KB. In this way, even if the knowledge base contains facts about millions of Americans, only Colonel West will be considered during the forward inference process.

4. Backward Chaining

This algorithm work backward from the goal, chaining through rules to find known facts that support the proof. It is called with a list of goals containing the original query, and returns the set of all substitutions satisfying the query. The algorithm takes the first goal in the list and finds every clause in the knowledge base whose **head,** unifies with the goal. Each such clause creates a

new recursive call in which **body,** of the clause is added to the goal stack .Remember that facts are clauses with a head but no body, so when a goal unifies with a known fact, no new sub goals are added to the stack and the goal is solved. The algorithm for backward chaining and proof tree for finding criminal (West) using backward chaining are given below.

function FOL-BC-ASK(KB, *goals*, θ) **returns** a set of substitutions
  inputs: KB, a knowledge base
    goals, a list of conjuncts forming a query (θ already applied)
    θ, the current substitution, initially the empty substitution $\{\}$
  local variables: *answers*, a set of substitutions, initially empty

  if *goals* is empty **then return** $\{\theta\}$
  $q' \leftarrow$ SUBST(θ, FIRST(*goals*))
  for each sentence r **in** KB where STANDARDIZE-APART(^) $= (p_1 \wedge \ldots \wedge p_n \Rightarrow q)$
    and $\theta' \leftarrow$ UNIFY(q, q') succeeds
    new-goals $\leftarrow [p_1, \ldots, p_n | \text{REST}(goals)]$
    answers $\leftarrow$ FOL-BC-ASK(KB, *new-goals*, COMPOSE(θ', θ)) $\cup$ *answers*
  return *answers*

Figure 4.1 *A simple backward-chaining algorithm.*

.

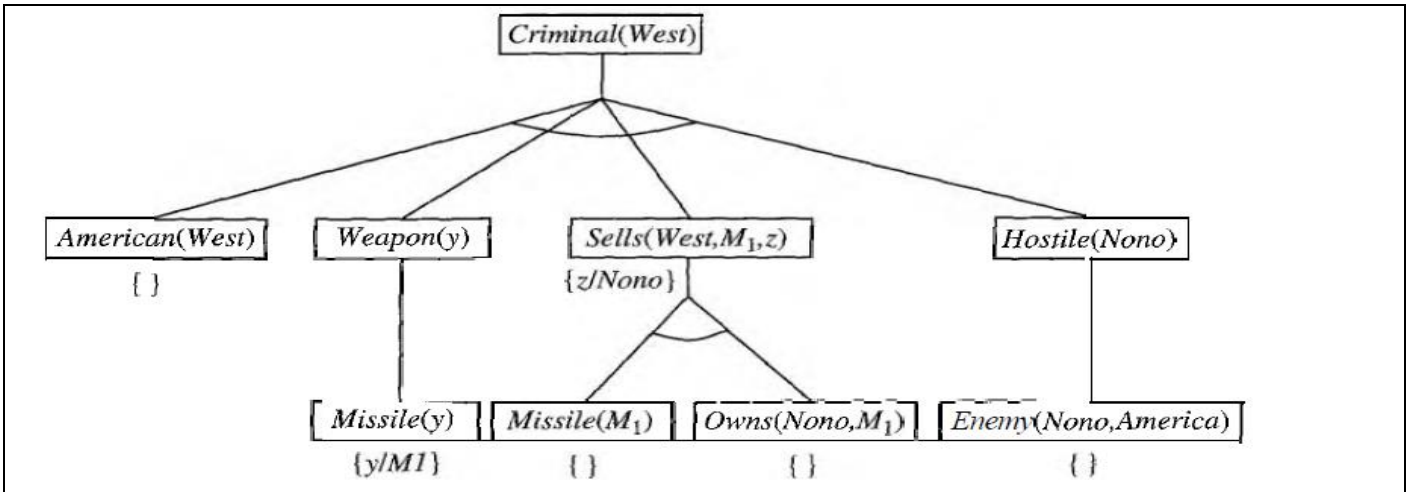


Figure 4.2 *Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove Criminal (West), we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding sub goal. Note that once one sub goal in a conjunction succeeds, its substitution is applied to subsequent sub goals.*

Logic programming:

- **Prolog** is by far the most widely used logic programming language.
- Prolog programs are sets of definite clauses written in a notation different from standard first-order logic.
- Prolog uses uppercase letters for variables and lowercase for constants.
- Clauses are written with the head preceding the body; " : -" is used for left implication, commas separate literals in the body, and a period marks the end of a sentence

```
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z)
```

Prolog includes "syntactic sugar" for list notation and arithmetic. Prolog program for append (X, Y, Z), which succeeds if list Z is the result of appending lists x and Y

```
append([],Y,Y).
append([A|X],Y,[A|Z]) :- append(X,Y,Z)
```

For example, we can ask the query append (A, B, [1, 2]): what two lists can be appended to give [1, 2]? We get back the solutions

```
A=[]     B=[1,2]
A=[1]    B=[2]
A=[1,2]  B=[]
```

- The execution of Prolog programs is done via depth-first backward chaining
- Prolog allows a form of negation called **negation as failure.** A negated goal not P is considered proved if the system fails to prove p. Thus, the sentence

Alive (X) : - not dead(X) can be read as "Everyone is alive if not provably dead."

- Prolog has an equality operator, =, but it lacks the full power of logical equality. An equality goal succeeds if the two terms are *unifiable* and fails otherwise. So X+Y=2+3 succeeds with x bound to 2 and Y bound to 3, but Morningstar=evening star fails.
- The occur check is omitted from Prolog's unification algorithm.

Efficient implementation of logic programs:

The execution of a Prolog program can happen in two modes: interpreted and compiled.

- Interpretation essentially amounts to running the FOL-BC-ASK algorithm, with the program as the knowledge base. These are designed to maximize speed.

  First, instead of constructing the list of all possible answers for each sub goal before continuing to the next, Prolog interpreters generate one answer and a "promise" to generate the rest when the current answer has been fully explored. This promise is called a **choice point.** FOL-BC-ASK spends a good deal of time in generating and composing substitutions when a path in search fails. Prolog will backup to previous choice point and unbind some variables. This is called "TRAIL". So, new variable is bound by UNIFY-VAR and it is pushed on to trail.

- Prolog Compilers compile into an intermediate language i.e., Warren Abstract Machine or WAM named after David. H. D. Warren who is one of the implementers of first prolog compiler. So, WAM is an abstract instruction set that is suitable for prolog and can be either translated or interpreted into machine language.

Continuations are used to implement choice point's continuation as packaging up a procedure and a list of arguments that together define what should be done next whenever the current goal succeeds.

- Parallelization can also provide substantial speedup. There are two principal sources of parallelism

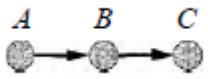
1. The first, called **OR-parallelism,** comes from the possibility of a goal unifying with many different clauses in the knowledge base. Each gives rise to an independent branch in the search space that can lead to a potential solution, and all such branches can be solved in parallel.
2. The second, called **AND-parallelism,** comes from the possibility of solving each conjunct in the body of an implication in parallel. AND-parallelism is more difficult to achieve, because solutions for the whole conjunction require consistent bindings for all the variables.

Redundant inference and infinite loops:

Consider the following logic program that decides if a path exists between two points on a directed graph.

```
path(X,Z) :- link(X,Z).
path(X,Z) :- path(X,Y), link(Y,Z)
```

A simple three-node graph, described by the facts link (a, b) and link (b, c)



It generates the query path (a, c)

Hence each node is connected to two random successors in the next layer.

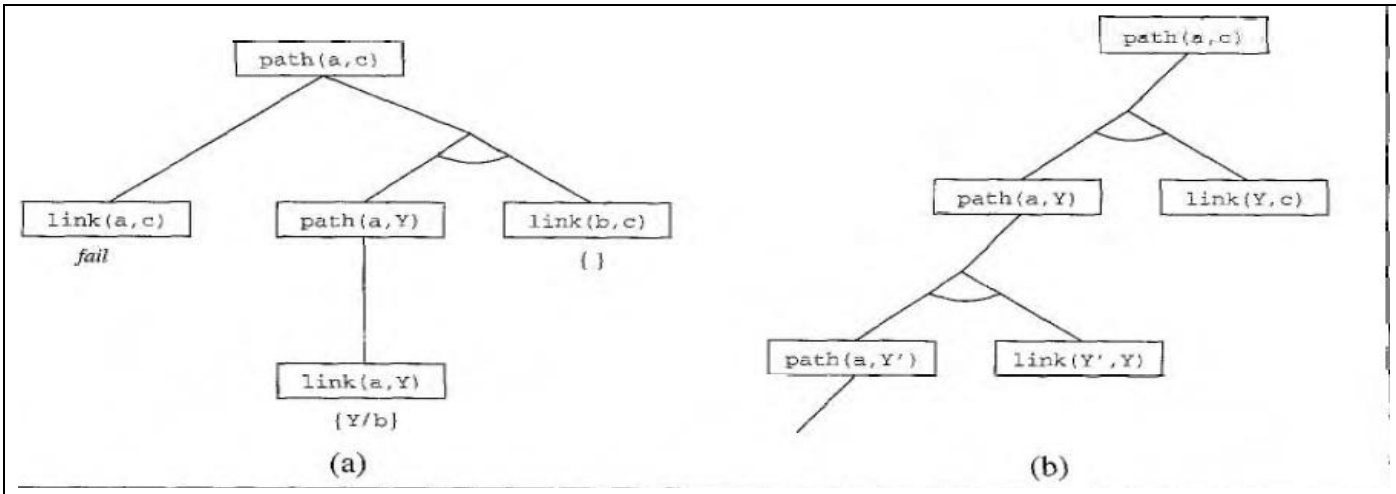


Figure 4.3 (a) Proof that a path exists from A to C. (b) Infinite proof tree generated when the clauses are in the "wrong" order.

Constraint logic programming:

The Constraint Satisfaction problem can be solved in prolog as same like backtracking algorithm. Because it works only for finite domain CSP's in prolog terms there must be finite number of solutions for any goal with unbound variables.

```
triangle(X,Y,Z) :-
    X>=0, Y>=0, Z>=0, X+Y>=Z, Y+Z>=X, X+Z>=Y.
```

- If we have a query, triangle (3, 4, and 5) works fine but the query like, triangle (3, 4, Z) no solution.
- The difficulty is variable in prolog can be in one of two states i.e., Unbound or bound.
- Binding a variable to a particular term can be viewed as an extreme form of constraint namely "equality".CLP allows variables to be constrained rather than bound.

The solution to triangle (3, 4, Z) is Constraint 7>=Z>=1.

5. Resolution

As in the propositional case, first-order resolution requires that sentences be in **conjunctive normal form** (CNF) that is, a conjunction of clauses, where each clause is a disjunction of literals.

Literals can contain variables, which are assumed to be universally quantified. Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence. We will illustrate the procedure by translating the sentence

"Everyone who loves all animals is loved by someone," or

$$\forall x \ [\forall y \ Animal(y) \Rightarrow Loves(x, y)] \Rightarrow [\exists y \ Loves(y, x)]$$

The steps are as follows:

- Eliminate implications:

$$ti\, x \ [\neg \forall y \ \neg Animal(y) \vee Loves(x, y)] \ v \ [\exists y \ Loves(y, x)]$$

- Move Negation inwards: In addition to the usual rules for negated connectives, we need rules for negated quantifiers. Thus, we have

$$\neg \forall x \ p \quad \text{becomes} \quad 3\, x \ \neg p$$
$$\neg \exists x \ p \quad \text{becomes} \quad \forall x \ \neg p$$

Our sentence goes through the following transformations:

$$\forall x \ [\exists y \ \neg(\neg Animal(y) \vee Loves(x,y))] \vee [\exists y \ Loves(y,x)] .$$
$$\forall x \ [\exists y \ \neg\neg Animal(y) \wedge \neg Loves(x,y)] \vee [\exists y \ Loves(y,x)].$$
$$\forall x \ [\exists y \ Animal(y) \wedge \neg Loves(x,y)] \vee [\exists y \ Loves(y,x)] .$$

- Standardize variables: For sentences like $(\forall x \ P(x)) \vee (\exists x \ Q(x))$ which use the same variable name twice, change the name of one of the variables. This avoids confusion later when we drop the quantifiers. Thus, we have

$$\forall x \ [\exists y \ Animal(y) \wedge \neg Loves(x,y)] \vee [\exists z \ Loves(z,x)]$$

- Skolemize: Skolemization is the process of removing existential quantifiers by elimination. Translate *3 x P(x)* into *P(A)*, where *A* is a new constant. If we apply this rule to our sample sentence, however, we obtain

$$\forall x \ [Animal(A) \wedge \neg Loves(x,A)] \vee Loves(B,x)$$

Which has the wrong meaning entirely: it says that everyone either fails to love a particular animal *A* or is loved by some particular entity *B*. In fact, our original sentence allows each person to fail to love a different animal or to be loved by a different person.

Thus, we want the Skolem entities to depend on *x:*

$$\forall x \ [Animal(F(x)) \wedge \neg Loves(x,F(x))] \vee Loves(G(x),x)$$

Here *F* and *G* are Skolem functions. The general rule is that the arguments of the Skolem function are all the universally quantified variables in whose scope the existential quantifier appears.

- Drop universal quantifiers: At this point, all remaining variables must be universally quantified. Moreover, the sentence is equivalent to one in which all the universal quantifiers have been moved to the left. We can therefore drop the universal quantifiers

$$[Animal(F(x)) \wedge \neg Loves(x,F(x))] \vee Loves(G(x),x)$$

- Distribute V over A

$$[Animal(F(x)) \vee Loves(G(x),x)] \wedge [\neg Loves(x,F(x)) \vee Loves(G(x),x)].$$

This is the CNF form of given sentence.

The resolution inference rule:

The resolution rule for first-order clauses is simply a lifted version of the propositional resolution rule. Propositional literals are complementary if one is the negation of the other; first-order literals are complementary if one ***unifies with*** the negation of the other. Thus we have

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\text{SUBST}(\theta, \ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)}$$

Where UNIFY (l_i, m_j) == θ.

For example, we can resolve the two clauses

$$[Animal(F(x)) \vee Loves(G(x), x)] \quad \text{and} \quad [\neg Loves(u, v) \vee \neg Kills(u, v)]$$

By eliminating the complementary literals *Loves (G(x), x)* and ¬*Loves (u, v),* with unifier θ = *{u/G(x), v/x),* to produce the resolvent clause

$$[Animal(F(x)) \vee \neg Kills(G(x), x)] .$$

Example proofs:

Resolution proves that KB /= *a* by proving *KB* A *la* unsatisfiable, i.e., by deriving the empty clause. The sentences in CNF are

$$\neg American(x) \vee \neg Weapon(y) \vee \neg Sells(x, y, z) \vee \neg Hostile(z) \vee Criminal(x)$$
$$\neg Missile(x) \vee \neg Owns(Nono, x) \vee Sells(West, x, Nono) .$$
$$\neg Enemy(x, America) \vee Hostile(x) .$$
$$\neg Missile(x) \vee Weapon(x) .$$
$$Owns(Nono, M_1) . \qquad Missile(M_1) .$$
$$American(West) . \qquad Enemy(Nono, America) .$$

The resolution proof is shown in below figure;

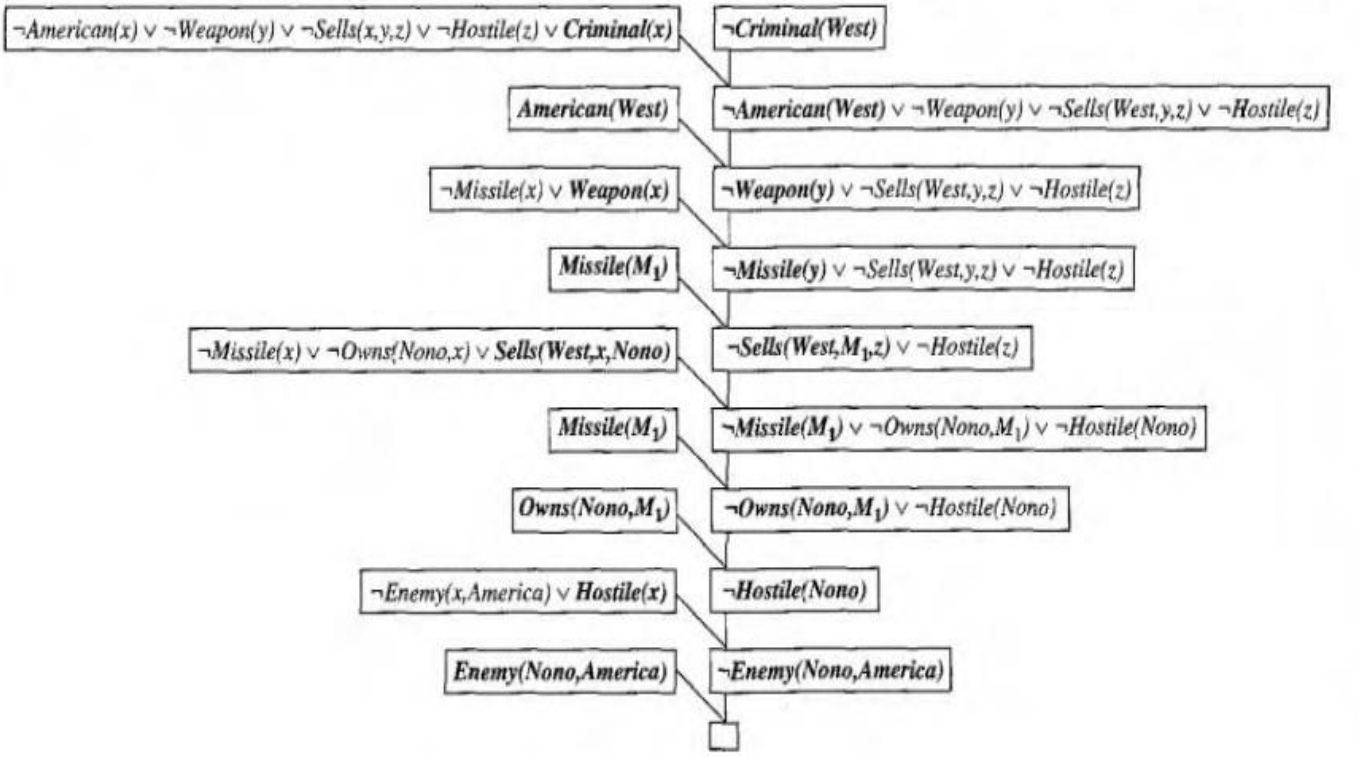


***Figure 5.1** A resolution proof that West is a criminal.*

Notice the structure: single "spine" beginning with the goal clause, resolving against clauses from the knowledge base until the empty clause is generated. Backward chaining is really just a special case of resolution with a particular control strategy to decide which resolution to perform next.

Completeness of resolution:

Resolution is **refutation-complete,** which means that if a set of sentences is unsatisfiable, then resolution will always be able to derive a contradiction. Resolution cannot be used to generate all logical consequences of a set of sentences, but it can be used to establish that a given sentence is entailed by the set of sentences.

- So we have to prove that "if S ***is an unsatisfiable set of clauses, then the application of finite number of resolution steps to S will yield a contradiction"***

The basic structure of the proof is shown below

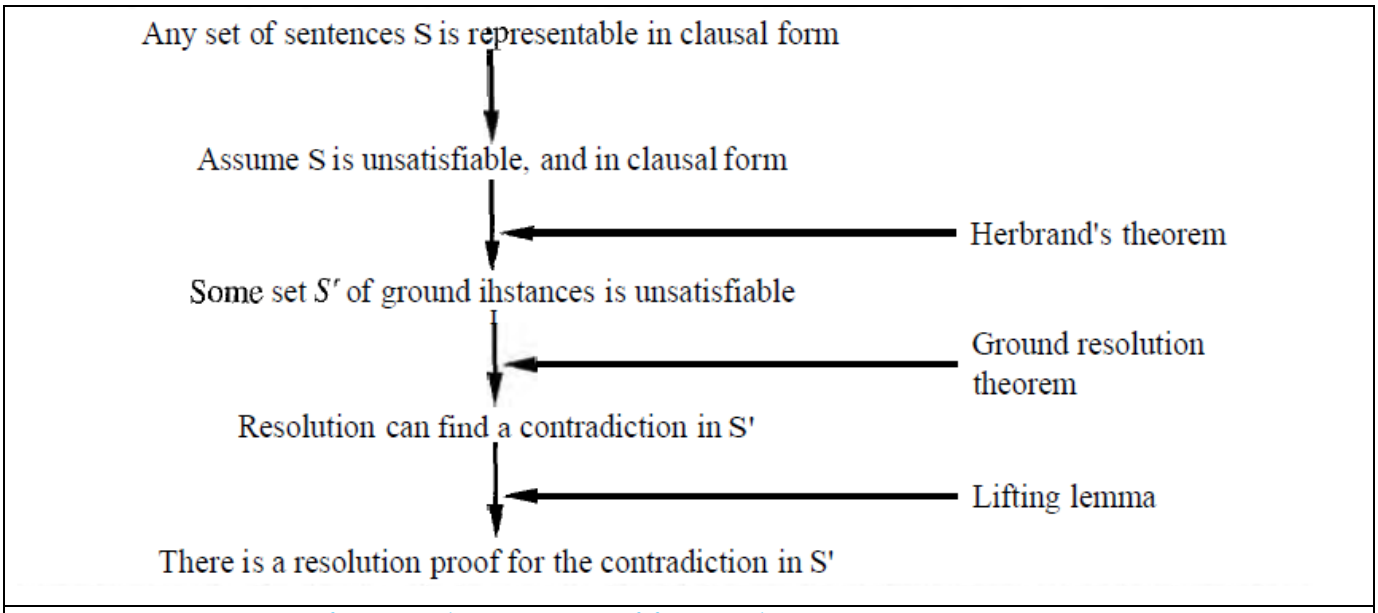


Figure 5.2 Structure of a completeness proof for resolution

It proceeds as follows:

- ✓ First, we observe that if S is unsatisfiable, then there exists a particular set of ***ground instances*** of the clauses of S such that this set is also unsatisfiable (Herbrand's theorem).
- ✓ We then appeal to the **ground resolution theorem** given in Chapter 7, which states that propositional resolution is complete for ground sentences.
- ✓ We then use a **lifting lemma** to show that, for any propositional resolution proof using the set of ground sentences, there is a corresponding first-order resolution proof using the first-order sentences from which the ground sentences were obtained.

Herbrand's theorem states that "If a set *S* of clauses is unsatisfiable, then there exists a finite subset of Hs (S) that is also unsatisfiable."Where Hs(S) is called the Herbrand base of S, The saturation of a set S of clauses with respect to its Herbrand universe.

Let S' be this finite subset of ground sentences. Now, we can appeal to the ground resolution theorem to show that the resolution closure *RC(S')* contains the empty clause.

The next step is to show that there is a resolution proof using the clauses of S itself, which are not necessarily ground clauses. Robinson's basic lemma implies the following fact:

Let C1 and C2 be two clauses with no shared variables, and let Ci and Ch be
Ground instances of C1 and C2. If C' is a resolvent of Ci and Ci, then there exists
a clause C such that (1) C is a resolvent of Cl and C2 and (2) C' is a ground instance of C.

This is called a lifting lemma because it lifts a proof step from ground clauses up to general first-order clauses. From the lifting lemma, it is easy to derive a similar statement about any sequence of applications of the resolution rule:

For any clause C' in the resolution closure of S there is a clause C in the resolution closure of *S*, such that C' is a ground instance of C and the derivation of C is the same length as the derivation of C'.

From this fact, it follows that if the empty clause appears in the resolution closure of *St*, it must also appear in the resolution closure of S. we have shown that if S is unsatisfiable, then there is a finite derivation of the empty clause using the resolution rule

Dealing with equality:

There are three distinct approaches that can be taken

- The first approach is to axiomatize equality-to write down sentences about the equality relation in the knowledge base. We need to say that equality is reflexive, symmetric, and transitive, and we also have to say that we can substitute equals for equals in any predicate or function. A standard inference procedure such as resolution can perform tasks requiring equality reasoning

- Another way to deal with equality is with an additional inference rule. The simplest rule, *demodulation,* takes a unit clause $x = y$ and substitutes y for any term that unifies with x in some other clause. More formally, we have

Demodulation: For any terms *x, y,* and *z,* where *UNIFY (X, Z)* = ***19*** and *mn[z]* is a literal containing *z:*

$$\frac{x = y, \qquad m_1 \vee \cdots \vee m_n[z]}{m_1 \vee \cdots \vee m_n[\text{SUBST}(\theta, y)]}$$

Paramodulation: For any terms *x, y,* and *z,* where *UNIFY(X, Z)* = θ ,

$$\frac{\ell_1 \vee \cdots \vee \ell_k \vee x = y, \qquad m_1 \vee \cdots \vee m_n[z]}{\text{SUBST}(\theta, \ell_1 \vee \ldots \vee \ell_k \vee m_1 \vee \ldots \vee m_n[y])}$$

Unlike demodulation, paramodulation yields a complete inference procedure for first-order logic with equality.

- A third approach handles equality reasoning entirely within an extended unification algorithm. That is, terms are unifiable if they are *provably* equal under some substitution, where "provably" allows for some amount of equality reasoning.

Resolution strategies:

We examine strategies that help find proofs *efficiently*

a) Unit preference:

This strategy prefers to1 do resolutions where one of the sentences is a single literal (also known as a **unit clause).** The idea behind the strategy is that we are trying to produce an empty clause, so it might be a good idea to prefer inferences that produce shorter clauses.

b) Set of support:

The set-of-support strategy starts by identifying a subset of the sentences called the **set of support.** Every resolution combines a sentence from the set of support with another sentence and adds the resolvent into the set of support. If the set of support is small relative to the whole knowledge base, the search space will be reduced dramatically.

c) Input resolution:

In the **input resolution** strategy, every resolution combines one of the input sentences (from the KB or the query) with some other sentence. In Horn knowledge bases, Modus Ponens is a kind of input resolution strategy, because it combines an implication from the original KB with some other sentences

Subsumption:

The **subsumption** method eliminates all sentences that are subsumed by (i.e., more specific than) an existing sentence in the KB. For example, if *P(x)* is in the KB, then there is no sense in adding *P(A)*. Subsumption helps keep the KB small, and thus helps keep the search space small.

Theorem Provers:

There are differ from Logic Programming languages in two ways:

- Most logic programming languages handle only Horn Clauses, whereas theorem provers accept for first-order logic.
- Prolog programs intertwine logic + control.
- In logic programming instead of using A:-B, C if we use A: - C, B it effects the execution of program. But in most theorem provers it doesn't effect the result.

Design a Theorem Prover:

Here we consider OTTER theorem prover. In preparing a problem for OTTER, the user must divide the knowledge into four parts

- A set of clauses known as the **set of support** (or sos), which defines the important facts about the problem
- A set of **usable axioms** that are outside the set of support. These provide background knowledge about the problem area.
- A set of equations known as **rewrites** or **demodulators.** For example, the demodulator x + 0 = x says that every term of the form x + *0* should be replaced by the term *x*.
- A set of parameters and clauses that defines the control strategy. The user specifies a heuristic function to control the search and a filtering function to eliminate some sub goals as uninteresting

At each step, OTTER moves the "lightest" clause in the set of support to the usable list and adds to the set of support some immediate consequences of resolving the lightest clause with elements of the usable list. OTTER halts when it has found a refutation or when there are no more clauses in the set of support. The algorithm is shown below:

```
procedure OTTER(sos, usable)
   inputs: sos, a set of support—clauses defining the problem (a global variable)
           usable, background knowledge potentially relevant to the problem

   repeat
      clause ← the lightest member of sos
      move clause from sos to usable
      PROCESS(INFER(clause, usable), sos)
   until sos = [] or a refutation has been found
```

```
function INFER(clause, usable) returns clauses

   resolve clause with each member of usable
   return the resulting clauses after applying FILTER
```

```
procedure PROCESS(clauses, sos)

   for each clause in clauses do
      clause ← SIMPLIFY(clause)
      merge identical literals
      discard clause if it is a tautology
      sos ← [clause | sos]
      if clause has no literals then a refutation has been found
      if clause has one literal then look for unit refutation
```

Figure 5.3 Sketch of the OTTER theorem prover. Heuristic control is applied in the selection of the "lightest" clause and in the FILTER function that eliminates uninteresting clauses from consideration.

Extending Prolog [PTTP: Prolog Technology Theorem prover]

To get Sound and Completeness to Prolog (PTTP).There are five significant changes.

- ✓ The occurs check is put back into the unification routine to make it sound.
- ✓ The depth-first search is replaced by an iterative deepening search
- ✓ Negated literals (such as ¬*P(x))* are allowed
- ✓ A clause with n atoms is stored as n different rules. For example, A <=B ∧ *C* would also be stored as ¬*B* <=*C* ∧¬*A* and as ¬C<=B ∧¬*A*. This technique, known as LOCKING
- ✓ Inference is made complete (even for non-Horn clauses) by the addition of the linear input resolution rule

The main drawback of PTTP is that the user has to relinquish all control over the search for solutions. Each inference rule is used by the system both in its original form and in the contrapositive form. This can lead to unintuitive searches.

For example, consider the rule,

$$(f(x,y) = f(a,b)) \Leftarrow (x = a) \wedge (y = b)$$

As a Prolog rule, this is a reasonable way to prove that two *f* terms are equal. But PTTP would also generate the contrapositive:

$$(x \neq a) \Leftarrow (f(x,y) \neq f(a,b)) \wedge (y = b)$$

It seems that this is a wasteful way to prove that any two terms x and *a* are different.