# Inference Engine of Prolog

## Resolution Procedure for Horn Clauses

**Prolog Inference Engine**

We will discuss the inference process for Horn clauses. This is much simpler than the inferences for general clausal theories. From now on we feel free to adopt the syntax of Prolog. A program clause is of one of the forms

A :- B1,...,Bn.  
 A.

A goal clause is of the form

?- G.

where G is a conjunction of atoms of the form C1,...,Ck.

A variable is an identifier starting with an upper case letter. Any identifier that starts with a lower case letter denotes a constant.

We should take note that variables in different clauses are not related, although we tend to use some common identifiers to denote variables in writing clauses.

An example program:

grandparent(X,Z) :- parent(X,Y), parent(Y,Z).  
 parent(X,Y) :- father(X,Y).  
 parent(X,Y) :- mother(X,Y).

## **1.2 Proof-By-Contradiction**

Given a program P and an atom p(x), we want to show that P logically implies (denoted by the symbole |= ) that there is an x such that p(x).

E.g., given the program P above about grandparent relation, we want to show that john has a grandparent, i.e., (from now on, we use F for "for all" and E for "there exists")

P |= Ex grandparent(x,john)  
  
  
Denote G = Ex grandparent(x,john).

Read the logic consequence relation a few times:

Whenever P is true, G is true.

How do we show this using resolution?

We prove this by contradiction. That is, assume G is false, and show this results in a contradiction to the above statement.

Note: P is satisfiable. A trivial model is that all the instances of the head of any rule are assigned to true.

First, we place a negation in front of G

-Ex grandparent(x,john)

This is equivalent to

Fx -grandparent(x,john)

Since the qualifier is universal, just like writing a program, let's omit the qualifier Fx to have the form

-grandparent(x,john)

Denote this by

Now, we use resolution to show that our program P and -Q together is unsatisfiable.

To understand this, consider a propositional program

Example.

P:  
(1) a :- b.  
(2) b.  
  
and formula  
  
 a

We want to show that P |= a. We do this by resolution between P and the goal

(3) -a.  
  
From (1) and (3) we get  
  
(4) -b.  
  
From (2) and (4) we get CONTRADICTION.

## **1.3 Derived Goal**

Let

G: ?- C1,...,Ck

be a goal. In resolution, one is allowed to select any subgoal to resolve first. This is called a "selection rule". Without loss of any inference power, let us fix the selection rule to be the leftmost, the one used by Prolog, i.e., C1 will be resolved first before we get to C2.

Let

A :- B1,...,Bn

be a program clause such that w = unify(A,C1).

Then,

?- w(B1,...,Bn,C2,...,Ck)

is a derived goal. When a subgoal is resolved by an unconditional clause, the result is an empty goal. When all the subgoals are resolved, a proof is found.

For any goal G above, there can be a number of clauses whose head is unifiable with C1. This means a goal could have several derived goals.

## **1.4 Resolution is based on tree search**

Let's represent a goal as a node and a derived goal as a child node. This representation yields a tree describing all possible derivations for a given program and a given goal. Such a tree is potentially infinite. A leaf node is either an empty goal or a node which is not unifiable with the head of any clause. In the latter case, the goal is said to have failed.

Example.

P: p(g(X)) :- q(a).  
 p(f(X)) :- b.  
 p(X) :- l(X).  
 l(X) :- l(X).   
 q(a).  
   
 ?- p(Y)   
 / Y/g(X) | Y/f(X) \ Y/X  
 / | \  
 ?- q(a) ?- b ?- l(X)  
 / | |  
 [] fail ?- l(X)  
 ...

In each derivation, we use X/t to denote that X is bound to t by unification.

The derivation sequence from the original goal to an empty goal represents a successful proof, called a REFUTATION. All the other branches either fail or loop forever.

## **1.5 Prolog searches a tree depth first**

When there is more than one applicable clause for a goal, Prolog will try the one that appears first in the text, alternatives will be tried upon backtracking. This is the familiar depth first search strategy. For the above example, Prolog will generate the leftmost branch and an answer is returned, and if you type a semicolon ";", Prolog will backtrack and runs into an infinite loop.

To see the effect of this search strategy, let us assume the clauses are in the following order

p(f(X)) :- b.  
 p(g(X)) :- q(a).  
 p(X) :- l(X).  
 l(X) :- l(X).   
 q(a).

Prolog will first generate

?- p(Y)   
 / Y/f(X)  
 ?- b   
 /  
 fail

b fails, and backtracking occurs: Prolog goes back to the parent goal and tries other applicable clauses, in this case the second clause above. The search space that is explored to find a refutation is thus

?- p(Y)   
 / Y/f(X) \ Y/g(X)  
 ?- b ?- q(a)   
 / |  
 fail []

However, if you write clauses in the following order

p(X) :- l(X).  
 p(f(X)) :- b.  
 p(g(X)) :- q(a).  
 l(X) :- l(X).   
 q(a).

Prolog will keep exploring the infinite branch, resulting in a nonterminating computation (until running out of storage).

But why not do breadth first search? It takes too much space.

Last modified: Wednesday, 2 March 2016, 12:15 PM