Resolution

Full first-order version:

\[ \ell_1 \lor \cdots \lor \ell_k, m_1 \lor \cdots \lor m_n \]

\[(\ell_1 \lor \cdots \lor \ell_{i-1} \lor \ell_{i+1} \lor \cdots \lor \ell_k \lor m_1 \lor \cdots \lor m_{j-1} \lor m_{j+1} \lor \cdots \lor m_n)\]

where \( \text{UNIFY}(\ell_i, \neg m_j) = \theta \).

Or equivalently,

\[ L \lor \ell_i, M \lor m_j \]

\[ (L \lor M)\theta \]

where

- \( L = (\ell_1 \lor \cdots \lor \ell_{i-1} \lor \ell_{i+1} \lor \cdots \lor \ell_k) \),
- \( M = (m_1 \lor \cdots \lor m_{j-1} \lor m_{j+1} \lor \cdots \lor m_n) \), and
- \( \text{UNIFY}(\ell_i, \neg m_j) = \theta \).

Resolution: Examples

\[ \neg \text{Pet}(\text{Joe}) \lor \text{Cat}(\text{Joe}) \lor \text{Bird}(\text{Joe}) \]

\[ \text{Parrot}(x) \lor \neg \text{Bird}(x) \]  

(+)

\[ \neg \text{Pet}(\text{Joe}) \lor \text{Cat}(\text{Joe}) \lor \text{Parrot}(\text{Joe}) \]

(+) \( \text{Unify}(\text{Bird}(x), \text{Bird}(\text{Joe})) = \{x/\text{Joe}\} \)

\[ \neg \text{On}(x, y) \lor \text{Above}(x, y) \lor \text{On}(B, A) \lor \text{On}(A, B) \]

(*)

\[ \text{Above}(A, B) \lor \text{On}(B, A) \]

(*) \( \text{Unify}(\text{On}(x, y), \text{On}(A, B)) = \{x/A, y/B\} \)

\[ \neg \text{Bird}(x) \lor \text{Feathers}(x) \lor \neg \text{Feathers}(y) \lor \text{Flies}(y) \]

(†)

\[ \neg \text{Bird}(x) \lor \text{Flies}(x) \]

(†) \( \text{Unify}(\text{Feathers}(x), \text{Feathers}(y)) = \{y/x\} \)
Examples of Unification

1. $x$ and $A$ unify but $A$ and $B$ do not
2. $x$ and $F(A, H(x))$ do not unify ($x$ occurs in $F(A, H(x))$)
3. $F(H(x))$ and $G(H(x))$ do not unify (different top symbols)
4. $F(x)$ and $F(x, y)$ do not unify (different number of arguments)
5. $F(t_1, t_2, \ldots, t_n)$ and $F(s_1, s_2, \ldots, s_n)$ unify if $t_1$ and $s_1$ unify, $t_2$ and $s_2$ unify, \ldots, $t_n$ and $s_n$ unify.

Unification

- A variable unifies with any term in which it does not occur.
- A constant symbol unifies only with itself
- Two compound terms (or two atomic sentences) unify only if
  - their top symbol is the same,
  - that symbol is applied to the same number of arguments
  - the arguments unify pairwise.

Unifiers

- $\theta$ is a unifier of $t_1$ and $t_2$ if $\theta(t_1)$ is identical to $\theta(t_2)$.
- Some terms may have more than one unifier:
  - $G(x, H(z))$ and $G(A, y)$ unify with $\theta_1 := \{x/A, y/H(A), z/A\}$
  - or with $\theta_2 := \{x/A, y/H(z)\}$
- But some unifiers are more general than others. Here, $\theta_2$ above is more general than $\theta_1$.
- Formally, unifier $\sigma_2$ is more general than unifier $\sigma_1$ if there exists a substitution $\sigma_3$ such that $\sigma_1 = \sigma_2 \sigma_3$.
- When unifying two terms we are interested in their most general unifier (mgu).

The Unification Procedure

Input: a set $V$ of the form $\{t = t'\}$
Output: either FAIL or a set of the form $\{v_1 = t_1, \ldots, v_m = t_m\}$
Procedure: Repeatedly apply (in any order) the transformations rules
- Delete, Eliminate, Decompose, Orient, Occurs, Mismatch
to $V$ until no more transformations apply.

If the procedure returns $\{v_1 = t_1, \ldots, v_m = t_m\}$ on input $\{t = t'\}$, then $\{v_1/t_1, \ldots, v_m/t_m\}$ is a mgu of $t$ and $t'$.

Notation:
- $v$ denotes a variable,
- $t$ denotes a term (possibly a variable).
The Unification Procedure: Examples

**Unify:** $F(G(H(y)), H(A))$ and $F(G(x), x)$

\[
\begin{align*}
\{ F(G(H(y)), H(A)) \} & \equiv \{ F(G(x), x) \} \\
\{ G(H(y)) \} & \equiv \{ G(x), \ H(A) \equiv x \} \\
\{ H(y) \} & \equiv \{ x, \ H(A) \equiv x \} \\
\{ x \equiv H(y), \ H(A) \equiv H(y) \} & \equiv \{ x \equiv H(y), \ A \equiv y \} \\
\{ x \equiv H(y), \ y \equiv A \} & \equiv \{ x \equiv H(A), \ y \equiv A \} \\
\end{align*}
\]

\[mgu = \{ x/H(A), \ y/A \}\]

**Unify:** $F(H(y), z)$ and $F(G(x), z)$

\[
\begin{align*}
\{ F(H(y), z) \} & \equiv \{ F(G(x), z) \} \\
\{ H(y) \} & \equiv \{ z \equiv z \} \\
\end{align*}
\]

FAIL

do not unify

**Unify:** $F(y)$ and $F(H(x, y))$

\[
\begin{align*}
\{ F(y) \} & \equiv \{ F(H(x, y)) \} \\
\{ y \equiv H(x, y) \} & \equiv \{ \} \\
\end{align*}
\]

FAIL

do not unify

The Transformation Rules

**Decompose.** $U \cup \{ f(t_1, \ldots, t_n) \equiv f(s_1, \ldots, s_n) \} \rightarrow U \cup \{ t_i \equiv s_i : 1 \leq i \leq n \}$

**Orient.** $U \cup \{ t \equiv v \} \rightarrow U \cup \{ v \equiv t \}$

**Delete.** $U \cup \{ v \equiv t \} \rightarrow U$

**Eliminate.** $U \cup \{ v \equiv t \}, v \in Vars(U) \setminus Vars(t) \rightarrow U[v/t] \cup \{ v \equiv t \}$

**Mismatch.** $U \cup \{ f(t_1, \ldots, t_m) \equiv g(s_1, \ldots, s_n) \}, f, g \text{ distinct or } m \neq n \rightarrow FAIL$

**Occurs.** $U \cup \{ v \equiv t \}, v \neq t \text{ but } v \in Vars(t) \rightarrow FAIL$

**Notation:** $Vars(U), Vars(t)$ denote the set of variables in $U$, $t$, respectively; $v$ denotes a variable, $s, t$ denote terms (possibly variables); $f, g$ denote function/constant symbols.
Example Knowledge Base contd.

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

\[\text{American}(x) \land \text{Weapon}(y) \land \text{Sells}(x, y, z) \land \text{Hostile}(z) \implies \text{Criminal}(x)\]

Nono ... has some missiles, i.e.,
\[\exists x \, \text{Owns}(\text{Nono}, x) \land \text{Missile}(x) :\]

\[\text{Owns}(\text{Nono}, M_1) \land \text{Missile}(M_1)\]

... all of its missiles were sold to it by Colonel West

Example Knowledge Base

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.

Prove that Col. West is a criminal

Example Knowledge Base contd.

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

\[\text{American}(x) \land \text{Weapon}(y) \land \text{Sells}(x, y, z) \land \text{Hostile}(z) \implies \text{Criminal}(x)\]

Nono ... has some missiles, i.e.,
\[\exists x \, \text{Owns}(\text{Nono}, x) \land \text{Missile}(x) :\]

\[\text{Owns}(\text{Nono}, M_1) \land \text{Missile}(M_1)\]

... all of its missiles were sold to it by Colonel West

Example Knowledge Base contd.

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

\[\text{American}(x) \land \text{Weapon}(y) \land \text{Sells}(x, y, z) \land \text{Hostile}(z) \implies \text{Criminal}(x)\]

Nono ... has some missiles

\[\exists x \, \text{Owns}(\text{Nono}, x) \land \text{Missile}(x) :\]

\[\text{Owns}(\text{Nono}, M_1) \land \text{Missile}(M_1)\]

... all of its missiles were sold to it by Colonel West
### Example Knowledge Base contd.

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

\[
\text{American}(x) \land \text{Weapon}(y) \land \text{Sells}(x, y, z) \land \text{Hostile}(z) \implies \text{Criminal}(x)
\]

Nono ... has some missiles, i.e.,

\[
\exists x \ \text{Owns}(\text{Nono}, x) \land \text{Missile}(x):
\]

\[
\text{Owns}(\text{Nono}, M_1) \land \text{Missile}(M_1)
\]

... all of its missiles were sold to it by Colonel West

\[
\forall x \ \text{Missile}(x) \land \text{Owns}(\text{Nono}, x) \implies \text{Sells}(\text{West}, x, \text{Nono})
\]

Missiles are weapons:

\[
\text{Missile}(x) \implies \text{Weapon}(x)
\]

An enemy of America counts as “hostile”:

\[
\text{Enemy}(x, \text{America}) \implies \text{Hostile}(x)
\]

### Resolution Proof: Definite Clauses

- \(~\text{American}(x)\lor\text{Weapon}(y)\lor\text{Sells}(x,y,z)\lor\text{Hostile}(z)\lor\text{Criminal}(x)\)
- \(~\text{American}(\text{West})\lor\text{Weapon}(M_1)\lor\text{Sells}(\text{West},M_1,\text{Nono})\lor\text{Hostile}(\text{Nono})\lor\text{Criminal}(\text{West})\)
- \(~\text{Missile}(x)\lor\text{Weapon}(y)\lor\text{Sells}(\text{West},M_1,\text{Nono})\lor\text{Hostile}(\text{Nono})\)
- \(~\text{Missile}(M_1)\lor\text{Weapon}(M_1)\)
- \(~\text{Missile}(M_1)\lor\text{Sells}(\text{West},M_1,\text{Nono})\lor\text{Hostile}(\text{Nono})\)
- \(~\text{Owns}(\text{Nono},M_1)\lor\text{Hostile}(\text{Nono})\)
- \(~\text{Enemy}(\text{Nono},\text{America})\lor\text{Hostile}(\text{Nono})\)
- \(~\text{Enemy}(\text{Nono},\text{America})\lor\text{Hostile}(\text{Nono})\)
- \(~\text{Missile}(x)\implies\text{Weapon}(x)\)
- \(~\text{Sells}(\text{West},x,\text{Nono})\lor\text{Hostile}(x)\)
- \(~\text{Sells}(\text{West},y,z)\lor\text{Hostile}(y)\)
- \(~\text{Sells}(\text{West},M_1,z)\lor\text{Hostile}(\text{Nono})\)
- \(~\text{Missile}(M_1)\lor\text{Hostile}(\text{Nono})\)
- \(~\text{Owns}(\text{Nono},M_1)\lor\text{Hostile}(\text{Nono})\)
- \(~\text{Enemy}(x,\text{America})\lor\text{Hostile}(x)\)
- \(~\text{Enemy}(x,\text{America})\lor\text{Hostile}(x)\)
Resolution by Refutation

1. Obtain $CNF(KB \land \neg \alpha)$
2. Apply resolution steps to the CNF
3. If the empty clause is generated, then $KB \models \alpha$.

- A proof by contradiction is also called a refutation.
- **Theorem**: The resolution rule with factoring is a refutation-complete inference system:
  - if a set of clauses is unsatisfiable, then $\text{False}$ is provable from it by resolution.

Why Resolution by Refutation Works

- Let $\Gamma$ be a set of clauses and $\alpha$ an atomic sentence.
- To show that $\alpha$ follows from $\Gamma$, prove $\text{False}$ from $\Gamma \cup \{\neg \alpha\}$.
- This proof method is sound because:
  - False provable from $\Gamma \cup \{\neg \alpha\}$ by resolution implies $\Gamma \cup \{\neg \alpha\} \models \text{False}$
  - $\Gamma \cup \{\neg \alpha\}$ is unsatisfiable implies $\neg (\Gamma \land \neg \alpha)$ is valid
  - $\neg (\Gamma \land \neg \alpha)$ is valid implies $\neg \Gamma \lor \alpha$ is valid
  - $\neg \Gamma \lor \alpha$ is valid implies $\Gamma \Rightarrow \alpha$ is valid
  - $\Gamma \models \alpha$

Factors

- Let $\varphi$ be a CNF sentence of the form below where $\alpha_1 \lor \alpha_2$ are both positive (or negative) literals
  \[ \alpha_1 \lor \alpha_2 \lor \beta \]
- If $\varphi_1, \varphi_2$ unify with mgu $\theta$ then
  \[ \theta(\alpha_1 \lor \beta) \]
  is a factor of $\varphi$.
- **Example**: $P(x)$ is a factor of $P(x) \lor P(y)$ because $P(x), P(y)$ unify with $mgu = \{ y/x \}$

A Technicality

- The resolution rule alone is not complete.
- For example, the set $\{P(x) \lor P(y), \neg P(x) \lor \neg P(y)\}$ is unsatisfiable.
- But all we can do is start with
  \[
  \frac{P(x) \lor P(y) \quad \neg P(x) \lor \neg P(y)}{P(y) \lor \neg P(y)}
  \]
  But then, resolving $\gamma$ with $\alpha$, or $\gamma$ with $\beta$ leads nowhere. (try it!)
Some Resolution Strategies

- **Unit resolution**: Unit resolution only.
- **Input resolution**: One of the two clauses must be an input clause.
- **Set of support**: One of the two clauses must be from a designed set called set of support. New resolvent are added into the set of support.
- **Linear resolution**: The latest resolvent must be used in the current resolution.

**Note**: The first 3 strategies above are incomplete. The Unit resolution strategy is equivalent to the Input resolution strategy: a proof in one strategy can be converted into a proof in the other strategy.

Too Many Choices!

- A resolution-based inference system has just one rule to apply to build a proof.
- However, at any step there may be several possible ways to apply the resolution rule.

Several resolution strategies have been developed in order to reduce the search space.

Prolog: A Logic Programming Language

- A pure Prolog program consists of a set of definite clauses (only one positive literal). Instead of writing a clause as $A \lor \neg B_1 \lor \neg B_2 \lor \cdots \lor \neg B_n$, we write it as $A :- B_1, B_2, \ldots, B_n$
- A query is a negative clause (all literals are negative): $?- C_1, C_2, \ldots, C_m$
  which is the clause $\neg C_1 \lor \neg C_2 \lor \cdots \lor \neg C_m$.
- Horn Clauses = Definite Clauses + Negative Clauses

Logic programming

Sound bite: computation as inference on logical KBs

1. Identify problem
2. Assemble information
3. Tea break
4. Encode information in KB
5. Encode problem instance as facts
6. Ask queries
7. Find false facts

Should be easier to debug $\text{Capital(NewYork,US)}$ than $x := x + 2$!
**Prolog Systems**

- A resolution inference system on Horn clauses + bells & whistles
- Widely used in Europe, Japan (basis of 5th Generation project)
  - Compilation techniques ⇒ 60 million LIPS
- Program = set of definite clauses, which are of form:
  - head :- literal1, ..., literaln.
- Variables are identifiers starting with a capital; others are constants.
- Built-in predicates for arithmetic etc., e.g., \( X = Y \times Z + 3 \)
- Efficient unification by open coding
- Efficient retrieval of matching clauses by direct linking
- Depth-first, left-to-right resolution
- Closed-world assumption ("negation as failure") e.g., given
  \( \text{alive}(X) :- \text{not} \text{dead}(X). \text{alive}(joe) \) succeeds if
  \( \text{dead}(joe) \) fails.

**Prolog examples**

Depth-first search from a start state \( X \):

\[
\begin{align*}
\text{goal}(e). \\
\text{successor}(a,b). \\
\text{successor}(a,c). \\
\text{successor}(b,c). \\
\text{successor}(b,d). \\
\text{successor}(c,d). \\
\text{successor}(c,e). \\
\text{dfs}(X) :- \text{goal}(X). \\
\text{dfs}(X) :- \text{write}(X), \text{successor}(X,S), \text{dfs}(S).
\end{align*}
\]

?- dfs(a).

\text{a b c d e.}

No need to loop over \( S \): \text{successor} succeeds for each

\[
\begin{align*}
\text{Prolog: A Small Example}
\end{align*}
\]

1) father(bob, ken).
2) father(ken, joe).
3) father(joe, don).
4) mother(jan, don).
5) parent(X, Y) :- father(X, Y).
6) parent(X, Y) :- mother(X, Y).
7) ancestor(X, Y) :- parent(X, Z), parent(Z, Y).
8) ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
9) ancestor(X, Y) :- ancestor(X, Z), parent(Z, Y).

?- ancestor(A, don).
\( A = \text{ken}; \)
\( A = \text{bob}. \)

Resolution Proof in Prolog

\[
\begin{align*}
\text{% it is a crime to sell weapons to hostile nations:} \\
\text{criminal}(X) :- \text{american}(X), \text{weapon}(Y), \text{sells}(X,Y,Z), \text{hostile}(Z). \\
\text{% Nono ... has some missiles,} \\
\text{owns(nono,m1).} \\
\text{missile(m1).} \\
\text{% all of its missiles were sold to it by Colonel West} \\
\text{sells(west,X,nono) :- missile(X), owns(nono,X).} \\
\text{% Missiles are weapons} \\
\text{weapon(X) :- missile(X).} \\
\text{% An enemy of America counts as ‘‘hostile’’} \\
\text{hostile(X) :- enemy(X,america).} \\
\text{% The country Nono, an enemy of America ...\text{al}} \\
\text{enemy(nono,america).} \\
\text{% West, who is American ...\text{al}} \\
\text{american(west).} \\
\text{?- criminal(Who).} \\
\text{Who = west.} \\
\end{align*}
\]
An Abstract Interpreter

**Input:** Goal $G$, a list of literals, and program $P$, a list of definite clauses.

**Output:** A substitution $S$ which makes $G$ a logical consequence of $P$, or $	ext{no}$ if no such instances exist.

**Algorithm:**

1. $\text{Goals} := G$; $S := \{}$
   2. while $\text{Goals} \neq \{}$ do
      3. $A := \text{delete\_first}(\text{Goals})$
      4. Choose a (renamed) clause $A' := B_1, ..., B_n$ from $P$
         such that $A$ and $A'$ unify, with mgu $S'$.
         (If no such clauses, go back to the last choice;
         if no more choices, exit the while loop.)
      5. $\text{Goals} := \text{append}([B_1, ..., B_n], \text{Goals})$
      6. $\text{Goals} := \text{apply}(S', \text{Goals})$
      7. $S := \text{combine}(S', S)$
   8. end

   return $S$ if $\text{Goals} = \{}$, no otherwise.

Prolog examples

Appending two lists to produce a third:

```
append([], Y, Y).
append([X | L], Y, [X | Z]) :- append(L, Y, Z).
?- append([1, 2], [3, 4], C).
C = [1, 2, 3, 4].
?- append([1, 2], B, [1, 2, 3, 4]).
C = [3, 4].
?- append(A, B, [1, 2]).
A = [] B = [1, 2]
A = [1, 2] B = []
```

An Abstract Interpreter

**Input:** Goal $G$, a list of literals, and program $P$, a list of definite clauses.

**Output:** A substitution $S$ which makes $G$ a logical consequence of $P$, or $	ext{no}$ if no such instances exist.

**Algorithm:**

1. $\text{Goals} := G$; $S := \{}$
   2. while $\text{Goals} \neq \{}$ do
      3. $A := \text{delete\_first}(\text{Goals})$
      4. Choose a (renamed) clause $A' := B_1, ..., B_n$ from $P$
         such that $A$ and $A'$ unify, with mgu $S'$.
         (If no such clauses, go back to the last choice;
         if no more choices, exit the while loop.)
      5. $\text{Goals} := \text{append}([B_1, ..., B_n], \text{Goals})$
      6. $\text{Goals} := \text{apply}(S', \text{Goals})$
      7. $S := \text{combine}(S', S)$
   8. end

   return $S$ if $\text{Goals} = \{}$, no otherwise.

Search Trees in Prolog

A search tree is representation of all possible computation paths. All possible proofs of a query are represented in one search tree.

Consider, for example, the following program.

```
member(X, [X | Ys]).
member(X, [Y | Ys]) :- member(X, Ys).
```

The following is a search tree for the goal member(a,[a,b,a]).

```
member(a, [a, b, a])
\ /
success member(a, [b, a])
\ /
fail member(a, [a])
\ /
success member(a, [])
\ /
fail fail
```

Traversing Search Tree

Prolog traverses a search tree in a depth-first manner.

We may not find a proof even if one exists, because depth-first search is an incomplete search procedure.

The order of clauses and the order of literals in a clause decide the structure of the search tree.

Different goal orders result in search trees with different structures.
Consider the following example

```
q(a).
p(s(Y)) :- p(Y).
```

The results for query $p(X)$, $q(X)$ and query $q(X)$, $p(X)$ are quite different.

Changes in rule order permutes the branches of the search tree.

When writing Prolog program, make sure a successful path in the search tree appears before any infinite search tree path.
Resolution Strategies in Prolog

- Set of support: The query (negative clauses) is the only clause in the set of support.
- Input strategy: One of the two clauses must be an input clause.
- Linear strategy: Only one resolvent is saved at any time.
- The literals in the resolvent are organized as a stack: First In, Last Out.
- The clauses are treated in the order they are listed.
- The literals in one clause are treated in the order from left to right.

Note: The first three strategies are equivalent for Horn clauses.

Unification in Prolog

- A query `?- test.` will succeed for the following program:
  
  ```
  less(X, succ(X)).
  test :- less(Y, Y).
  ```

  Why? This because Prolog uses an unsound unificationalgorithm: The occur check is omitted for the purpose of efficiency.

  - occur check: If $v = ? t$ and $v$ occurs in $t$, then failure.
  - Another Example: An infinite loop for the query `test`.

  ```
  test :- leq(Y, Y).
  leq(X, s(X)) :- leq(X, X).
  ```

Using GNU Prolog

```prolog
sells(west, A, nono) :-
  missile(A),
  owns(nono, A).
enemy(nono, america).
missile(m1).
owns(nono, m1).
american(west).
criminal(A) :-
  american(A),
  weapon(B),
sells(A, B, C),
hostile(C).
(30 ms) yes
?- criminal(X).
  X = west
  yes
```

Using GNU Prolog

The download and documentation can be found at

```
http://pauillac.inria.fr/~diaz-gnu-prolog/
```

It’s available on every CS Linux machine. Just type `gprolog`.

GNU Prolog 1.2.16
By Daniel Diaz
Copyright (C) 1999-2002 Daniel Diaz

```
| ?- ['c:/c145/crime.pl'].
compiling c:\c145\crime.pl for byte code...
c:\c145\crime.pl compiled, 15 lines read - 1646 bytes written
(40 ms) yes
| ?- listing.

hostile(A) :-
  enemy(A, america).
weapon(A) :-
  missile(A).
```
The Closed World Assumption in Prolog

- Consider again the following program P1:
  
  ```prolog
  s(a).
s(b).
t(c).
  ```

- Suppose we knew that all true instances of \( s(X) \) and \( t(Y) \) were logical consequences of P1. That is to say, we assume that P1 expresses completely what we know about the true instances of the relations \( s(X) \) and \( t(Y) \).

- We are assured that \( s(c) \) is not a true instance of the relation \( s(X) \).

- Hence, we conclude that \( s(c) \) is false, or equivalently, that \( \neg s(d) \) is true.

Negation as Failure in Prolog

- Prolog's negation doesn't quite implement this. Why? just because a goal \( G \) has a finitely failed search tree doesn't mean that the search tree which Prolog traverses is finitely failed (remember the literal order decides the search tree structure).

- An example:

  ```prolog
  unmarried_student(X) :- not married(X), student(X).
  student(bill).
  married(joe).
  ```

  The query `unmarried_student(X)` fails, even though \( X=\text{bill} \) is a solution. Changing the order of the goals in the body of `unmarried_student/1` avoids this difficulty.

- Negated goals must be ground for negation as failure in Prolog to work correctly.

Negative Knowledge in Prolog

- Pure Prolog can never deduce negative information. For example, given the program

  ```prolog
  s(a).
s(b).
t(c).
  ```

  we cannot deduce \( \neg s(c) \), nor \( \neg t(a) \).

- If we had the following program:

  ```prolog
  s(a).
s(b).
  not s(c).
t(c).
  ```

  then we could deduce \( \neg s(c) \) as a logical consequence. But this is not a definite program (the head literal must be positive).

Negation as Failure in Prolog

- With the Closed World Assumption (CWA), we can implement negation as failure. In this case we will allow the inference that \( \neg G \) is true if we fail to prove \( G \).

- We cannot tell, in general, whether a goal \( G \) will succeed or fail in a finite amount of time. If \( G \) has a finitely failed search tree (one which is finite and has no success branches), we can define a negation as failure inference rule. This says that we may conclude \( \neg A \) if \( A \) has a finitely failed search tree.

- A typical implementation:

  ```prolog
  not(X) :- X, !, false.
  ```
Summary

- Prolog is based on the Resolution method.
- To make it efficient, it accepts only definite clauses.
- To make it efficient, it uses an unsound unification algorithm.
- To make it efficient, it doesn’t use factoring.
- To make it efficient, it uses depth-first search.
- To allow it to handle negative knowledge, the Closed World Assumption is taken and the Negation As Failure is implemented.
- Many other techniques are also used: cut, compilation, built-in data types, ...