Quick Tutorial in First Order Logic

January 1996

J. N. Hooker
Graduate School of Industrial Administration
Carnegie Mellon University
Pittsburgh, PA 15213 USA

This tutorial and other papers
are available through
http://www.gsia.cmu.edu/afs/andrew/gsia/45-760/jnh.html

WHAT IS FIRST-ORDER LOGIC?

- **First order logic** is a powerful formal language that contains such **quantifiers** as 'for all' and 'for some' and expresses properties of and relations among objects.

- It was invented by Aristotle and is the most intensively studied formal logic *(Frege, Gödel, Tarski, Quine, etc.)*.

- Subsets (**fragments**) of it form the basis of logic programming languages

  e.g., PROLOG, PROLOG III and soon IV, Siemens PROLOG

  and are used in some constraint programming packages

  e.g., CHIP, DECISION POWER.

- It can be used as a powerful modeling language. Some of its features underlie most existing languages

  including such algebraic languages as GAMS and AMPL.

WHAT IS LOGIC PROGRAMMING?

- It is analogous to mathematical programming:
  - Constraints are written in formal logic rather than with inequalities, etc.
  - Whereas math programming uses numerical computation to deduce consequences from the constraints, logic programming uses logical inference.
- Logic programming is much more general because it can use any predicate that is defined by constraints (rules, propositions) in the program.
- But it needs help from math programming when the predicates are mathematical (math operations, greater than, etc.).

- The following logic program finds ancestors. 'ancestor' and 'parent' are **predicates**.

  ancestor(X,Y) if parent(X,Y).
  ancestor(X,Z) if parent(X,Y), ancestor(Y,Z).
  parent(a,b).
  parent(b,c).
  goal: ancestor(X,c).

  The goal asks whether c has ancestors. This is a satisfiability problem in first order logic. $\vee$ means or, $\wedge$ means and, $\neg$ means not.

  $$\begin{aligned}&\forall x \forall y(\text{ancestor}(x, y) \vee \neg \text{parent}(x, y))\\ &\wedge \forall x \forall y \forall z(\text{ancestor}(x, z) \vee \neg \text{parent}(x, y) \vee \neg \text{ancestor}(y, z))\\ &\wedge \text{parent}(a, b) \wedge \text{parent}(b, c)\\ &\wedge \neg \exists x \text{ancestor}(x, c)\end{aligned}$$

  Because this is unsatisfiable, c has ancestors, and the satisfiability algorithm should find one (a, b).

- Historically, logic programming treats the rules as both declarative and procedural. Besides stating the problem, the rules show how to search for a solution (e.g. an ancestor) by posing goals and subgoals.

e.g., in the program

1. ancestor(X,Y) if parent(X,Y).
2. ancestor(X,Z) if parent(X,Y), ancestor(Y,Z).
3. parent(a,b).
4. parent(b,c).

goal: ancestor(X,c).

Use rule 2 to obtain the goal ancestor(X,c).
This poses two subgoals parent(X,Y), ancestor(Y,c).
Rule 3 allows subgoal parent(X,Y) to succeed with (X,Y)=(a,b).
Use rule 1 to obtain subgoal ancestor(Y,c), which is now ancestor(b,c).
This poses subgoal parent(b,c), which succeeds (rule 4).
So goal ancestor(X,c) succeeds with X=a.

- Resolution methods (discussed later) are also used to check for satisfiability; e.g., SLD resolution (Linear resolution with a Selection function for Definite clauses).
- However, resolution is very slow, and an alternate approach (partial instantiation) is suggested here.

FORMULAS

- A **ground level** formula might be:

$$P(a) \supset Q(a, b)$$

where

$a =$ **constant**

$P =$ **1-place predicate**

e.g., $P(a) = a$ *is a father*

$Q =$ 2-place predicate (**relation**)

e.g., $Q(a, b) = a$ *is father of* b

So the formula could say

If a *is a father, then he is the father of* b*.*

- **Quantified** logic contains

universal quantifier $\forall$ (for all)

existential quantifier $\exists$ (there exists)

e.g.,

$$\forall x(P(x) \supset \exists y Q(x, y))$$

For everything x *that is a father, there is something* y *of which it is a father.*

$x =$ **variable** (**bound** by $\forall$)

$y =$ variable bound by $\exists$

A variable is **free** if it is not bound by a quantifier.

- The variables of **first order** logic can be replaced by constants only (not by predicates).
- The formula

$$P(a) \supset Q(a, b)$$

is an **instance** of

$$P(x) \supset Q(x, y)$$

(in fact a **ground instance** because it contains no variables). The latter is a **generalization** of the former.

- **Functions** are indispensable in logic programming. For instance,

$$f(t, a)$$

This might be interpreted as adding an individual a to list t. So the list a, b, c, d could be represented

$$f(f(f(f(0, a), b), c), d),$$

where 0 is the empty list.

INTERPRETATIONS

- The idea of an **interpretation** is useful for defining truth and falsehood of formulas. An interpretation for a formula specifies:
  - The **domain** *(the set of constants that can instantiate variables in the formula)*;
  - Which ground instances of the formula *(using constants in the domain)* are true.
- Truth for a quantified formula is defined in the obvious way.

  For example, consider an interpretation of

  $$\forall x(P(x) \supset \exists y Q(x, y))$$

  in which

  – the domain is $\{a, b, c\}$

  – the true ground instances of P are

  $$P(a), P(b)$$

  (i.e., a and b are fathers).

  – the true ground instances of Q are

  $$Q(a, b), Q(b, c)$$

  (i.e., a is father of b, and b of c).

Then

$$\forall x(P(x) \supset \exists y Q(x,y))$$

is true if **all** substitutions for x from $\{a, b, c\}$ result in true formulas:

$$\begin{array}{l} P(a) \supset \exists y Q(a,y) \\ P(b) \supset \exists y Q(b,y) \\ P(c) \supset \exists y Q(c,y) \end{array}$$

The first formula, for instance, says that if a is a father, he is someone's father. Because the interpretation makes $P(a)$ true, the formula is true only if

$$\exists y Q(a,y)$$

is true. This is true if **some** substitution for y from $\{a, b, c\}$ makes a true formula. The substitution $y = b$ does.

A **model** for a formula is an interpretation in which it the formula is true.

- A formula is **satisfiable** if it has a model. A formula is **valid** if it is true in any interpretation.

  The formula

  $$\forall x(P(x) \supset \exists y Q(x, y))$$

  is not valid because it is not true in every interpretation; for instance, the interpretation above except that $P(c)$ is true rather than false.

DECIDABILITY

- First order logic is **complete** because there is a finite set of axioms from which any valid formula can be deduced *(as proved by Gödel)*.

- Despite this, first order logic is **undecidable** because there is no procedure that will always decide in finite time whether a given formula is satisfiable.

  The basic reason is that a satisfiable formula may have no finite model. For instance, the Schönfinkel-Bernays formula

  $$\forall(x,y,z)\exists w(\neg P(x,x) \wedge [(P(x,y) \wedge P(y,z)) \\ \supset P(x,z)] \wedge P(x,w))$$

  says basically that everything has a successor other than itself. The formula is satisfiable (e.g., natural numbers) but has no model with a finite domain.

- First order logic is **semidecidable**, meaning that unsatisfiability can always be detected by a finite procedure. *This is basically due to Herbrand's compactness theorem.*

SKOLEM NORMAL FORM

- Normal forms are useful algorithmically and were important historically for showing semidecidability.
- A formula is in **prenex form** if all the quantifiers precede everything else.

The formula

$$\forall x(P(x) \supset \exists y Q(x,y))$$

can be written in prenex form:

$$\forall x \exists y(P(x) \supset Q(x,y))$$

The rules for transforming to prenex form are

$$\begin{array}{rcl} \neg\forall xA & \equiv & \exists x\neg A \\ \neg\exists xA & \equiv & \forall x\neg A \\ (\forall xA \vee B) & \equiv & \forall x(A \vee B) \\ (\exists xA \vee B) & \equiv & \exists x(A \vee B) \\ (\exists xA \wedge B) & \equiv & \exists x(A \wedge B), \end{array}$$

where x does not occur free in B, plus,

$$\begin{array}{rcl} (\forall xA \wedge \forall xB) & \equiv & \forall x(A \wedge B) \\ (\exists xA \vee \exists xB) & \equiv & \exists x(A \vee B), \end{array}$$

and finally (to **standardize apart**),

$$\forall xA \equiv \forall yA'$$

where y does not occur free in A, and A' is A with every occurrence of x replaced by y.

- A formula is **Skolemized** to eliminate existential quantifiers.
  - *For formulas in the $\exists\forall$ fragment* (all $\exists$'s precede all $\forall$'s in prenex form): Variables bound by an $\exists$ are replaced by a constant that occurs nowhere else in the formula.

    e.g.,

    $$\exists x \exists y (P(x,y) \wedge \neg P(x,a))$$

    is Skolemized to obtain

    $$P(b,c) \wedge \neg P(b,a)$$

    The $\exists\forall$ fragment is decidable.
  - *For other formulas:* When $\forall x$ precedes $\exists y$, y is replaced by a **Skolem function** $g(x)$ to indicate that different instantiations may be required for different values of x.

    e.g.,

    $$\exists x \forall y \exists z (P(x,y) \wedge \neg P(x,z))$$

    in Skolemized form is,

    $$\forall y (P(a,y) \wedge \neg P(a,g(y)))$$

- A formula in **Skolem normal form** can be written,

$$\forall x_1 C_1 \wedge \ldots \wedge \forall x_m C_m,$$

where each C_i is a quantifier-free clause, x_i a vector of variables, and the C_i's are **standardized apart** (no variables in common). A **clause** is a disjunction of **literals** (variables or their negations).

e.g., the formula

$$\forall y[P(a,y) \vee (P(a,g(y)) \wedge P(y,b))]$$

in Skolem normal form is

$$\forall y(P(a,y) \wedge P(a,g(y))) \wedge \forall z(P(a,z) \wedge P(z,b)).$$

HERBRAND'S THEOREM

- Basic idea: a formula in Skolem normal form is equivalent to the conjunction of the ground instances of its clauses—*using an appropriate domain.*

  The formula

  $$\forall x \forall y (P(x, y) \vee P(a, x)) \wedge \forall z \neg P(z, b)$$

  is in Skolem normal form. Using the domain $\{a, b\}$, its ground instances are

  $$\begin{array}{l} P(b, a) \vee P(a, a) \\ P(b, b) \vee P(a, a) \\ P(b, a) \vee P(a, b) \\ P(b, b) \vee P(a, b) \\ \neg P(a, a) \\ \neg P(b, a). \end{array}$$

  Their conjunction is equivalent to the original formula, even though only the small domain $\{a, b\}$ is used. This is part of Herbrand's theorem.

  In general, the appropriate domain (the **Herbrand universe**) is the set of constants in the formula, or $\{a\}$ if there are no constants.

- There is also a **compactness** property: if a formula is unsatisfiable, a finite conjunction of ground instances is unsatisfiable. *For unsatisfiable formulas, this reduces the satisfiability problem to the propositional (ground level) satisfiability problem.*

  e.g., the Herbrand universe for the formula

  $$\forall y P(a, y) \wedge \forall z \neg P(a, g(z))$$

  is $\{a, g(a), g(g(a)), \ldots\}$. Because the formula is unsatisfiable, there is an unsatisfiable finite conjunction of ground instances of its clauses, namely

  $$P(a, g(a)) \wedge \neg P(a, g(a)).$$

- The ideas can be formalized as follows.
  - A **term** is defined as follows:
    1. Every constant is a term.
    2. For every n-place function f and vector t of terms, $f(t)$ is a term. (A **constant term** contains no variables.)
  - The **Herbrand universe** for a formula F is the set of all terms built up from constants and functions in F (if F contains no constants, use a).
  - The **Herbrand base** consists of all ground instances of predicates that appear in F.
  - An **Herbrand interpretation** assigns a truth value to each element of the Herbrand base.
  - An **Herbrand ground instance** of F is obtained by instantiating from the Herbrand universe.
  - **Herbrand's Theorem**: A formula in Skolem normal form is unsatisfiable iff it is false in all its Herbrand interpretations. Furthermore, it is unsatisfiable iff some finite conjunction of Herbrand ground instances is unsatisfiable (compactness).

UNIFIERS

- A **substitution** σ in a formula F is a mapping that replaces each variable with a term; the result is $F\sigma$, which is an **instance** of F. If σ substitutes variables for variables, $F\sigma$ is a **variant** of F.

  $P(y)$ and $P(f(a))$ are instances of $P(x)$.

- σ is a **unifier** for $P(t)$ and $P(t')$ if $P(t)\sigma = P(t')\sigma$. σ is a **most general unifier** (mgu) if $P(t)\tau$ is an instance of $P(t)\sigma$ for every unifier τ.

  The formulas

  $$P(x, f(x), b) \quad P(a, f(a), y)$$

  are unified by the substitution $(x, y) = (a, b)$, which is a most general unifier. The formulas

  $$P(x, f(x)) \quad P(a, f(b))$$

  cannot be unified.

- There is a linear-time algorithm for finding an mgu.

  To unify $P(t)$ and $P(t')$, first standardize them apart. Let the **disagreement set** D of $P(t)$, $P(t')$ contain the first pair of corresponding terms that differ.

  For example, the disagreement set of

  $$P(x, f(g(a)), y), \quad P(x, f(z), b)$$

  is $\{g(a), z\}$.

  The algorithm is,

  1. Let $k = 0$ and let σ_0 be the identity substitution.
  2. If $P(t)\sigma_k = P(t')\sigma_k$ then stop; σ_k is an mgu. Otherwise let D_k be the disagreement set of $P(t)\sigma_k, P(t')\sigma_k$.
  3. If D_k passes the **occur check** (i.e., one element of D_k is a variable v that does not occur in the other element t of D_k), then: let σ_{k+1} be the substitution that applies σ_k and subsequently replaces all occurrences of v with t; increment k; go to 2. Otherwise stop; there is no unifier.

As presented the algorithm is exponential, but some tricks make it linear.

RESOLUTION

- Logic programming software and theorem provers traditionally use some form of **resolution** to check for satisfiability.
- Resolution for a conjunction of ground clauses (propositional logic) was introduced by Quine in the 50's.

  Find two clauses in the formula for which exactly one atom occurs negated in one clause and posited in the other.

  $$\begin{array}{lll} P(a) \vee Q(b) & & \\ \neg P(a) & & \vee \neg R(c) \vee S(d) \end{array}$$

  The **resolvent** is

  $$Q(b) \vee \neg R(c) \vee S(d)$$

  Conjoin the resolvent to the formula and repeat. If an empty clause is generated, the formula is unsatisfiable.

Resolution is dreadfully slow. Branching methods are much faster.

- Robinson generalized resolution to first-order logic.

  Given a formula in Skolem normal form, find two clauses containing unifiable atoms that are of opposite sign.

  $$\begin{array}{ll} \forall(x,y)(& P(a,x) \vee Q(x,y)) \\ \forall(z,w)(& \neg P(z,b) \qquad\qquad\qquad \vee \neg R(z,w)) \end{array}$$

  Let σ be an mgu of $P(a,x), P(z,b)$. The resolvent is

  $$\forall(x,y,z,w)\ \ (Q(x,y) \vee \neg R(z,w))\sigma,$$

  or

  $$\forall(y,w)\ \ (Q(b,y) \vee \neg R(a,w)).$$

  Conjoin the resolvent to the formula and repeat.

PARTIAL INSTANTIATION

- By Herbrand's theorem, one can in principle check a formula in Skolem normal form for satisfiability by checking the conjunction of all its Herbrand ground instances *(if there are finitely many)*. This allows one to use a fast propositional satisfiability algorithm, but there are usually too many ground instances.

- An approach (proposed by R. Jeroslow) is to check **partial instantiations** of the formula for satisfiability; i.e., conjunctions of certain instances in which not all variables are instantiated. The satisfiability check is fast because the partial instantiations are treated as ground level formulas.

Because the Herbrand universe for the formula

$$\forall x P(a,x) \wedge \forall y \neg P(b,y)$$

is $\{a,b\}$, the formula is equivalent to

$$P(a,a) \wedge P(a,b) \wedge \neg P(b,a) \wedge \neg P(b,b)$$

Rather than instantiate both x and y, begin by instantiating neither.

$$\underset{T}{P(a,x)} \wedge \neg \underset{F}{P(b,y)} \, .$$

A **variant-independent** satisfying valuation is shown (i.e., variants get the same truth value). This defines a satisfying valuation for all Herbrand ground instances: just let each predicate **inherit** the truth value of the predicate it instantiates.

$$\underset{T}{P(a,a)} \wedge \neg \underset{T}{P(a,b)} \wedge \neg \underset{F}{P(b,a)} \wedge \neg \underset{F}{P(b,b)}$$

- The inherited valuation can be **blocked**: the same predicate can inherit both truth values.

For example,

$$\forall x \forall y (P(x, y) \vee P(a, x)) \wedge \forall z \neg P(z, b)$$

has ground level equivalent

$$
\begin{aligned}
&(P(a, a) \vee P(a, a)) \wedge \\
&(P(a, b) \vee P(a, a)) \wedge \\
&(P(b, a) \vee P(a, b)) \wedge \\
&(P(b, b) \vee P(a, b)) \wedge \\
&\neg P(a, b) \wedge \neg P(b, b)
\end{aligned}
$$

An initial partial instantiation is satisfied by the valuation

$$C_1 \wedge C_2 \;=\; (\underset{T}{P(x_1, y_1)} \vee \underset{T}{P(a, x_1)}) \wedge \underset{F}{\neg P(z_2, b)}\,.$$

(The variables have been renamed to facilitate standardizing apart.) The inherited truth values are

$$\begin{array}{c}
(\underset{T,F}{P(a,b)} \vee \underset{T}{P(a,a)}) \wedge \\
(\underset{T}{P(b,a)} \vee \underset{T,F}{P(a,b)}) \wedge \\
(\underset{T,F}{P(b,b)} \vee \underset{T,F}{P(a,b)}) \wedge \\
\underset{T,F}{\neg P(a,b)} \wedge \underset{T,F}{\neg P(b,b)}
\end{array}$$

$P(a, b)$ inherits both true and false, and similarly for $P(b, b)$.

- Checking for blockage is simplified by using **satisfiers**. One true literal (satisfier) per clause is enough.

$$C_1 \wedge C_2 \;=\; (P(x_1, y_1) \vee \underset{T}{P(a, x_1)}) \wedge \underset{F}{\neg P(z_2, b)}\,.$$

Only ground instances of satisfiers inherit truth values, which is good enough. Now, check for blockages by examining only satisfiers.

$$\begin{array}{c}
(\underset{T,F}{P(a,b)} \vee \underset{T}{P(a,a)}) \wedge \\
(P(b,a) \vee \underset{T,F}{P(a,b)}) \wedge \\
(\underset{F}{P(b,b)} \vee \underset{T,F}{P(a,b)}) \wedge \\
\underset{T,F}{\neg P(a,b)} \wedge \underset{F}{\neg P(b,b)}
\end{array}$$

$P(a, b)$ still inherits both true and false.

- Blockage can be removed by further instantiation. Blockage occurs when predicates of different signs are unifiable. Apply the mgu to obtain an instance that is conjoined to the formula. Ground level predicates now inherit truth values from the most instantiated satisfier of which they are an instance.

  Above, the predicate $P(a,b)$ inherits two truth values because $P(a,x_1)$ and $P(z_2,b)$ have the mgu $(x_1,z_2) = (b,a)$. Apply this substitution to the clauses containing $P(a,x_1)$ and $\neg P(z_2,b)$ to obtain two new clauses:

  $$\begin{aligned} C_3 &= P(b,y_3) \vee \underset{T}{P(a,b)} \, . \\ C_4 &= \underset{F}{\neg P(a,b)} \end{aligned}$$

  A satisfier mapping for $C_1 \wedge C_2 \wedge C_3 \wedge C_4$ is shown. The inherited truth values are:

  $$\begin{gathered} (\underset{F}{P(a,b)} \vee \underset{T}{P(a,a)}) \wedge \\ (\underset{\mathrm{T}}{P(b,a)} \vee \underset{F}{P(a,b)}) \wedge \\ (\underset{T,F}{P(b,b)} \vee \underset{F}{P(a,b)}) \wedge \\ \underset{F}{\neg P(a,b)} \wedge \underset{T,F}{\neg P(b,b)} \end{gathered}$$

Now there is a blockage involving $P(z_2, b)$ and $P(b, y_3)$, with mgu $(x_3, y_3) = (b, b)$. This generates two more clauses,

$$
\begin{aligned}
C_5 &= P(b, b) \vee P(a, b) \\
C_6 &= \neg P(b, b)
\end{aligned}
$$

Because $C_1 \wedge \ldots \wedge C_6$ is unsatisfiable, so is the original formula. If it were satisfiable with an unblocked variant-independent valuation, the original formula would be satisfiable.

- This method also applies to formulas with functions. It always terminates if the formula belongs to the $\exists\forall$ fragment. Otherwise termination is guaranteed if the formula is unsatisfiable.

FURTHER READING

- First order logic and logic programming: J. W. Lloyd, *Foundations of Logic Programming*, 2nd edition, Springer 1993.
- PROLOG: L. Sterling and E. Shapiro, *The Art of Prolog: Advanced Programming Techniques*, MIT Press 1986.
- Constraint programming: E. Tsang, *Foundations of Constraint Satisfaction*, Academic Press 1993.
- Example of a 1st order logic modeling language: R. Krishnan, A logic modeling language for automated model construction, *Decision Support Systems* **6** (1990) 123-152; R. Krishnan, Automated model construction: a logic based approach, *Annals of Operations Research* **21** (1989) 195-226.
- Partial instantiation: J. N. Hooker, New methods for inference in first order logic, *Annals of Operations Research* **43** (1993) 479-492; J. N. Hooker and G. Rago, Partial instantiation methods for logic programming.