Chapter 3
First Order Predicate Logic

Abstract The first order predicate logic (FOPL) is backbone of AI, as well a method of formal representation of Natural Language (NL) text. The Prolog language for AI programming has its foundations in FOPL. The chapter demonstrates how to translate NL to FOPL in the form of facts and rules, use of quantifiers and variables, syntax and semantics of FOPL, and conversion of predicate expressions to clause forms. This is followed with unification of predicate expressions using instantiations and substitutions, compositions of substitutions, unification algorithm and its analysis. The resolution principle is extended to FOPL, a simple algorithm of resolution is presented, and use of resolution is demonstrated for theorem proving. The interpretation and inferences of FOPL expressions are briefly discussed, along with the use of Herbrand's universe and Herbrand's theorem. At the end, the most general unifier (mgu) and its algorithms are presented, and chapter is concluded with summary.

Keywords First Order Predicate Logic (FOPL) · Natural language · Quantifiers · Syntax and semantics of FOPL · Unification · Most general unifier · Resolution theorem · Theorem proving · Herbrand's universe · Herbrand's theorem

3.1 Introduction

This chapter presents a formulation of first-order logic which is best suited as a basic theoretical instrument—a computer based theorem proving program. As per the requirements of theory, an inference method should be *sound*—allows only logical consequences of premises deducible from the premises. In addition, it should be *effective*—algorithmically decidable whether a claimed application of the inference principle is really an application of it. When the inference principle is performed by computer, the complexity of the inference principle is not an issue. However, for more powerful principles, usage of combinatorial information processing for single application may become dominant.

The system described in the following is an inference principle—the *resolution principle*, is a machine-oriented rather than human-oriented system. Resolution principle is quite powerful in psychological sense also, as it obeys a single type of

K. R. Chowdhary, *Fundamentals of Artificial Intelligence*,
https://doi.org/10.1007/978-81-322-3972-7_3

inference, which is often beyond the ability of the human to grasp. In theoretical sense, it is a single inference principle that forms the complete system of first-order logic. However, this latter property is not of much significance, but it is interesting in the sense that no any other *complete system* of first-order logic is based on just one inference principle, if ever one tries to realize a device of introducing a logical axioms, or by a schema as an inference principle. The principle advantage of using the resolution is due to its ability that allows us to avoid any major combinatorial obstacles to efficiency, which used to be a serious problem in earlier theorem-proving procedures.

Learning Outcomes of this Chapter:

1. Translate a natural language (e.g., English) sentence into predicate logic statement. [Usage]
2. Apply formal methods of symbolic predicate logic, such as calculating validity of formula and computing normal forms. [Usage]
3. Use the rules of inference to construct proofs in predicate logic. [Usage]
4. Convert a logic statement into clause form. [Usage]
5. Describe the strengths and limitations of predicate logic. [Familiarity]
6. Apply resolution to a set of logic statements to answer a query. [Usage]
7. Implement a unification-based type-inference algorithm for a simple language. [Usage]
8. Precisely specify the invariants preserved by a sound type system. [Familiarity]

3.2 Representation in Predicate Logic

The first Order Predicate Logic (FOPL) offers formal approach to reasoning that has sound theoretical foundations. This aspect is important to mechanize the automated reasoning process where inferences should be correct and *logically sound*.

The statements of FOPL are flexible enough to permit the accurate representation of *natural languages*. The words—*sentence* or *well formed formula* will be indicative of predicate statements. Following are some of the translations of English sentences into predicate logic:

- English sentence: Ram is man and Sita is women.
  Predicate form: $man(Ram) \wedge woman(Sita)$
- English sentence: Ram is married to Sita.
  Predicate form: $married(Ram, Sita)$
- English sentence: Every person has a mother.
  The above can be reorganized as: For all x, there exists a y, such that if x is person then x's mother is y.
  Predicate form: $\forall x \exists y[person(x) \Rightarrow hasmother(x, y)]$

- English sentence: If x and y are parents of a child z, and x is man, then y is not man.
  $\forall x \forall y[[parents(x, z) \wedge parents(y, z) \wedge man(x)] \Rightarrow \neg man(y)]$

We note that predicate language comprises constants {Ram, Sita}, variables $\{x, y\}$, operators $\{\Rightarrow, \wedge, \vee, \neg\}$, quantifiers $\{\exists, \forall\}$ and functions/ predicates {married(x, y), person(x)}. Unless specifically mentioned, the letters $a, b, c, \dots$ at the beginning of English alphabets shall be treated as constants to indicate names of *objects* and *entities*, and those at the end, i.e., $u, v, w, \dots$ shall be used as variables or identifiers for objects and entities.

To indicate that an expression is universally true, we use the *universal quantifier* symbol ∀, meaning 'for all'. Consider the sentence "any object that has a feathers is a bird." Its predicate formula is: $\forall x[hasfeathers(x) \Rightarrow isbird(x)]$. Then certainly, $hasfeathers(parrot) \Rightarrow isbird(parrot)$ is true. Some expressions, although not always *True*, are *True* at least for some objects: in logic, this is indicted by 'there exists', and the *existential quantifier* symbol ∃ is used for this. For example, $\exists x[bird(x)]$, when *True*, this expression means that there is at least one possible object, that when substituted in the position of x, makes the expression inside the parenthesis as *True* [1].

Following are some examples of representations of knowledge FOPL.

Example 3.1 Kinship Relations.

$$mother(namrata, priti).\text{(That is, Namrata is mother of Preeti.)}$$
$$mother(namrata, bharat).$$
$$father(rajan, priti).$$
$$father(rajan, bharat).$$
$$\forall x \forall y \forall z[father(y, x) \wedge mother(z, x) \Rightarrow spouse(y, z)].$$
$$\forall x \forall y \forall z[father(y, x) \wedge mother(z, x) \Rightarrow spouse(z, y)].$$
$$\forall x \forall y \forall z[mother(z, x) \wedge mother(z, y) \Rightarrow sibling(x, y)].$$

In above, the predicate $father(x, y)$ means x is father of y; $spouse(y, z)$ means y is spouse of z, and $sibling(x, z)$ means x is sibling of y. □

Example 3.2 Family tree.

Suppose that we represent "Sam is Bill's father" by *father(sam, bill)* and "Harry is one of Bill's ancestors" by *ancestor(harry, bill)*. Write a wff to represent "Every ancestor of Bill is either his father, his mother, or one of their ancestors".

$$\begin{aligned}\forall x \forall y[ancester(y, bill) \Rightarrow &(father(y, bill) \vee mother(y, bill))\\ &\vee ((father(x, bill) \wedge ancester(y, x))\\ &\vee ((mother(x, bill) \wedge (ancester(y, x))].\end{aligned}$$

Example 3.3 Represent the following sentences by predicate calculus wffs.

1. A computer system is intelligent if it can perform a task which, if performed by a human, requires intelligence.

$$\exists x[[(perform(human, x) \rightarrow requires(human, intelligence))$$
$$\wedge\ (perform(computer, x)] \rightarrow intelligent(computer))]$$

2. A formula whose main connective is $\Rightarrow$ is equivalent to a formula whose main connective is $\vee$.

$$\begin{aligned}\forall x \forall y[(formula(x) \wedge mainconnective(x, ' \Rightarrow'))\\ \wedge\ (formula(y) \wedge mainconnective(y, \vee))\\ \rightarrow x \equiv y].\end{aligned}$$

3. If a program cannot be told a fact, then it cannot learn that fact. $\forall x[(program(x) \wedge \neg told(x, fact)) \rightarrow \neg learn(x, fact)]$ □

Example 3.4 Blocks World.

Consider that there are physical objects, like—*cuboid*, *cone*, *cylinder* placed on the table-top, with some relative positions, as shown in Fig. 3.1. There are four blocks on the table: a, c, d are cuboid, and b is a cone. Along with these there is a robot arm, to lift one of the object having clear top.

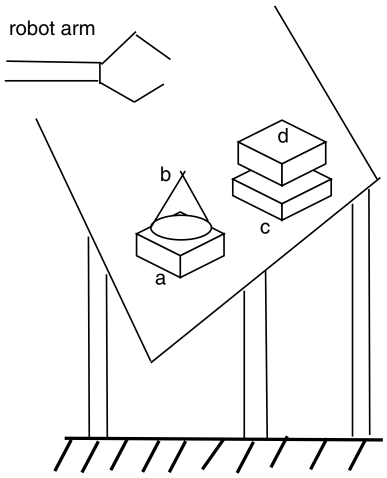


Fig. 3.1 Blocks world

Following is the set of statements about the blocks world (called knowledge base):

$cuboid(a).$
$cone(b).$
$cuboid(c).$
$cuboid(d).$
$onground(a).$
$onground(c).$
$ontop(b, a).$
$ontop(d, c).$
$toplcear(b).$
$topclear(d).$
$\forall x \forall y[topclear(x) \wedge toplcear(y) \wedge \neg cone(x) \Rightarrow puton(y, x)].$

The knowledge specified in the blocks world indicate that objects a, c, d are cuboid, b is cone, a, c are put on the ground, and b, d are on top of a, c, respectively, and the top of b, d are clear. These are called facts in knowledge representation. At the end, the rule says that there exists objects x and y such that both have their tops clear and x is not a cone, then y can be put on the object x. □

Bound and Free Variables

A variable in a wff is bound if it is within the scope of a quantifier naming the variable, otherwise the variable is free. For example, in $\forall x(p(x) \rightarrow q(x, y))$, x is bound and y is free; in $\forall x(p(x) \rightarrow q(x)) \rightarrow r(x)$, the x in $r(x)$ is free variable. In the latter case it is better to rename the variable to remove the ambiguity, hence we rephrase this statement as $\forall x(p(x) \rightarrow q(x)) \rightarrow r(z)$. An expression can be evaluated only when all the variables in that are bound.

If $F_1, F_2, \ldots, F_n$ are wffs with $\wedge, \neg$ as operators in each of F_i, then $F_1 \vee F_2 \vee \cdots \vee F_n$ is called *DNF* (disjunctive normal form). Alternatively, if operators in F_i are $\vee, \neg$ then $F_1 \wedge F_2 \wedge \cdots \wedge F_n$ is called *CNF* (conjunctive normal form). The expression F_i called a *term*, consists only literals. We will prefer the CNF for predicate expression. Thus, for an inference to be carried out, it is necessary to convert a predicate expression to *CNF*. For example, $\exists x[p(x) \Rightarrow q(x)]$ can be converted to $\neg p(a) \vee q(a)$, where a is an instance of variable x. The expression $\neg p(a) \vee q(a)$ is a term of *CNF*. A formula in *CNF*, comprising $\wedge$, $\vee$, $\neg$ along with constants, variables, and predicates, is called *clausal* or *clause* form [2].

3.3 Syntax and Semantics

Two types of semantics are defined for the programing languages: (1) *operational* semantics, and, (2) *fixpoint* semantics. The operational semantics defines input-output relation computed by a program in terms of the individual operations performed by the program inside a machine, like, basic logical and arithmetic operations. The meaning of a program is input-output relation obtained by executing the program on a machine. The other semantics—fixpoint semantics, is machine independent. It defines the meaning of a program to be the input-output relation which is the minimal fixpoint of a transformation associated with the program. The Fixpoint semantics is used to justify existing methods for proving properties of programs, and to justify new methods of proof.

We know the distinction between the *syntax* and the *semantics* from previous chapter as well from the study of programming languages. The Syntax deals with the formal part of language in abstraction from its meaning. It concerns with the definition of *well-formed formulas*. Syntax in its narrow sense and also deals with the study of axioms, rules of reference and proofs, which constitute proof theory. Semantics is concerned with the interpretation of language and includes such notions as meaning, logical implication and truth.

It is convenient to restrict attention to predicate logic programs written in *clausal form*. Such programs have an especially simple syntax but retain all the expressive power of the full predicate logic [3].

- *Atomic formula*. A string of symbols consisting of a predicate symbol of degree $n \geq 0$ followed by n terms is an *atomic formula*.
- *Clause*. A clause is a disjunction $L_1 \vee \cdots \vee L_n$ of literals L_i, each of which is *atomic formula* $P(t_1, \ldots, t_m)$ or the negation of atomic formulas, where P is a predicate symbol and t_i, are *terms*. A finite set (possibly empty) of literals is called a clause. The empty clause is denoted by: []
- *Sentence*. A sentence is a finite set of clauses.
- *Literals*. An atomic formula is a literal; and if A is an atomic formula then $\neg A$ is also literal.
  The atomic formulas are *positive* literals, and negations of atomic formulas are *negative* literals.
- *Term*. A term is either a variable or an expression like $f(t_i, \ldots, t_n)$, where f is a function symbol, t_i are terms, and constants are 0-*ary* function symbols. A variable is also a term, and a string of symbols comprising a function symbol of degree $n \geq 0$ followed by n terms is again a term.
  A set of clauses $\{C_1 \ldots, C_n\}$ is interpreted as a conjunction of clauses $C_1 \ldots C_n$. A clause C containing just the variables $x_1, \ldots, x_n$ is called universally quantified. For example,

$$\textit{for all } x_1, \ldots, x_n C \tag{3.1}$$

  is universally quantified clause.
- *Ground Literals*. A literal having no variables is called Ground Literal.
- *Ground clauses*. A clause with every member of it as a ground literal, is called a Ground Clause. Empty clause—[] is a Ground Clause.
- *Well-formed expressions*. The Terms and Literals are (the only) Well-Formed expressions.
- *Lexical Order of Well-formed expressions*. This is set of all well formed expressions ordered in lexical order. The ordering is as follows: A precedes B if A is shorter

than B. If A and B have same length, then A has the alphabetically earlier symbol in the first symbol position, at which A and B have distinct symbols.

- *Herbrand Universe*. It is set of ground terms associated with any set of S of clauses. Let F be the set of all function symbols which occur in clause set S. If F contains any function symbols of degree 0, then the functional vocabulary of S is F, otherwise, it is $\{a\} \cup F$, where a is a ground term. In this case, the Herbrand universe of S is set of all ground terms with only symbols of the functional vocabulary of S.
- *Models*. It is a set of ground literals having no complementary pair. If M is a Model and S is a set of *ground clauses*, then M is a model of S if, for all $C \in S$, C contains a member of M. In general, if S is any set of clauses, and H is the Herbrand Universe of S, then M is model of $H(S)$.
- *Satisfiability*. A set S is Satisfiable if there is a model of S, otherwise S is Unsatisfiable.

For every sentence S_1 of first order predicate logic there is always a sentence S_2 in *Clausal Form* which is satisfiable if and only if S_1 is also satisfiable. In other words, for every non-clause form sentence there is a logically equivalent clause form sentence. Due to this, all questions concerning to the *validity* or *satisfiability* of sentences in FOPL can be addressed to sentences in clausal form.

Procedure for obtaining clausal-form for any well-formed formula (*wff*) are discussed later in this chapter. In the above we have defined part of the syntax of predicate logic, which is concerned with the specification of well-formed formulas. The formalism we are going to use in the next section is based on the notions of *unsatisfiability* and *refutation* rather than upon the notions of *validity* and *proof*.

To work on the criteria of refutation and unsatisfirability, it is necessary to convert the given *wff* into clausal form.

To determine whether a finite set of sentences (S) of first-order predicate is satisfiable, it is sufficient to assume that each sentence in S is in clause form, and there is no existential quantifiers as the prefix to S. In addition, the matrix of each sentence in S is assumed to be a disjunction of formulas, each of which is either atomic formula or the negation of an atomic formula. Therefore, the syntax of S is designed such that the syntactical unit is a finite set of sentences in this special form, called *clause form*. Towards the end of conversion process, the quantifier prefix is omitted from each sentence, since it is necessary that universal quantifiers bind each variable in the sentence. The matrix of each sentence is simply a set of disjuncts and the order and multiplicity of the disjuncts are not important.

3.4 Conversion to Clausal Form

Following are the steps to convert a predicate formula into clausal-form [2].

1. Eliminate all the implications symbols using the logical equivalence: $p \rightarrow q \equiv \neg p \vee q$.

2. Move the outer negative symbol into the atom, for example, replace $\neg\forall x\ p(x)$ by $\exists x \neg p(x)$.
3. In an expression of nested quantifiers, existentially quantified variables not in the scope of universal quantifiers are replaced by constants. Replace $\exists x \forall y (f(x) \rightarrow f(y))$ by $\forall y (f(a) \rightarrow f(y))$.
4. Rename the variables if necessary. For example, in $\forall x(p(x)) \rightarrow q(x)$, rename second free variable x, as $\forall x(p(x) \rightarrow q(y))$.
5. Replace existentially quantified variables with *Skolem* functions; then eliminate corresponding quantifiers. For example, for $\forall x \exists y[\neg p(x) \vee q(y)]$, we obtain $\forall x[\neg p(x) \vee q(f(x))$. These newly created functions are celled *Skolem functions*, and the process is called *Skolemization*.
6. Move the universal quantifiers to the left of the equation. For example, substitute $\exists x[\neg p(x) \vee \forall y\ q(y)]$ by $\exists x \forall y[\neg p(x) \vee q(y)]$
7. Move the disjunctions down to the Literals, i.e., terms should be connected by conjunctions only, vertically.
8. Eliminate the conjunctions.
9. Rename the variables, if necessary.
10. Drop all the universal quantifiers, and write each term in a separate line.

The resulting sentence is a CNF, and suitable for inferencing using resolution.

Example 3.5 Convert the expression $\exists x \forall y[\forall z\ \ p(f(x), y, z) \Rightarrow (\exists u\ \ q(x, u) \wedge \exists v\ r(y, v))]$ to clausal form.

The steps discussed above are applied precisely, to get the clausal form of the predicate formula.

1. Eliminate implication.

$$\exists x \forall y[\neg \forall z\ p(f(x), y, z) \vee (\exists u\ q(x, u) \wedge \exists v\ r(y, v))]$$

2. Move negative symbols to the atom.

$$\exists x \forall y[\exists z \neg p(f(x), y, z) \vee (\exists u\ q(x, u) \wedge \exists v\ r(y, v))]$$

3. Replace existentially quantified variables not in the scope of universal quantifier to constants.

$$\forall y[\exists z \neg p(f(a), y, z) \vee (\exists u\ q(a, u) \wedge \exists v\ r(y, v))]$$

4. Rename variables (not required in this example.)
5. Replace existentially quantified variables that are functions of universal quantified variables, by Skolem functions:

$$\forall y[\neg p(f(a), y, g(y) \vee (q(a, h(y) \wedge r(y, l(y))]$$

6. Move $\forall$ to left is not required in this example.
7. Move disjunctions down to Literals.

$$\forall y[(\neg p(f(a), y, g(y)) \vee (q(a, h(y))) \wedge (\neg p(f(a), y, g(y)) \vee r(y, l(y)))]$$

8. Eliminate conjunctions.

$$\forall y[\neg p(f(a), y, g(y)) \vee (q(a, h(y)), (\neg p(f(a), y, g(y)) \vee r(y, l(y))]$$

9. Renaming variable is not required in this example.
10. Drop all universal quantifiers and write each term on separate line.

$$\neg p(f(a), y, g(y)) \vee (q(a, h(y)),$$
$$\neg p(f(a), y, g(y)) \vee r(y, l(y)).$$

Example 3.6 Convert the following wff to clause form.

$$(\forall x)(\exists y)\{[p(x, y) \Rightarrow q(y, x)] \wedge [q(y, x) \Rightarrow s(x, y)]\}$$
$$\Rightarrow (\exists x)(\forall y)[p(x, y) \Rightarrow s(x, y)]$$

For $[p(x, y) \Rightarrow q(y, x)] \wedge [q(y, x) \Rightarrow s(x, y)]$ by application of syllogism, it can be reduced to $[p(x, y) \Rightarrow s(x, y)]$. Thus, original expression reduces to:

$$\begin{aligned}
&= (\forall x)(\exists y)[p(x, y) \Rightarrow s(x, y)] \Rightarrow (\exists x)(\forall y)[p(x, y) \Rightarrow s(x, y)]\\
&= \neg(\forall x)(\exists y)[p(x, y) \Rightarrow s(x, y)] \vee (\exists x)(\forall y)[p(x, y) \Rightarrow s(x, y)]\\
&= (\exists x)\neg(\exists y)[p(x, y) \Rightarrow s(x, y)] \vee (\exists x)(\forall y)[p(x, y) \Rightarrow s(x, y)]\\
&= (\exists x)(\forall y)\neg[p(x, y) \Rightarrow s(x, y)] \vee (\exists x)(\forall y)[p(x, y) \Rightarrow s(x, y)]\\
&= (\forall y)\neg[p(a, y) \Rightarrow s(a, y)] \vee [p(a, y) \Rightarrow s(a, y)]\\
&= [p(a, y) \wedge \neg s(a, y)] \vee [\neg p(a, y) \vee s(a, y)]\\
&= T
\end{aligned}$$

3.5 Substitutions and Unification

The following definitions are concerned with the operation of *instantiation*, i.e, substitutions of terms for variables in the well-formed expressions and in sets of well-formed expressions [8].

Substitution Components

A substitution component is any expression of the form T/V, where V is any variable and T is any term different from V. The T can be any constant, variable, function, predicate, or expression.

Substitutions

A substitution is any finite set (possibly empty) of substitution components, none of the variables of which are same. If P is any set of terms, and the terms of the components of the substitution θ are all in P, we say that θ is a substitution over P. We write the substitution where components are $T_1/V_1, \ldots, T_k/V_k$ as $\theta = \{T_1/V_1, \ldots, T_k/V_k\}$, with the understanding that order of components is immaterial. We will use lowercase Greek letters θ, λ, μ denote substitutions.

Instantiations

If E is any function string of symbols, and $\theta = \{T_1/V_1, \ldots, T_k/V_k\}$ is any substitution, then the instantiation of E by θ is the operation of replacing each occurrence of variable V_i, $1 \le i \le k$, in E by term T_i. The resulting string denoted by $E\theta$ is called an instance of E by θ. That is, if E is the string $E_0V_{i_1}E_1 \ldots V_{i_n}E_n$, $n \ge 0$, then $E\theta$ is the string $E_0T_{i_1}E_1 \ldots T_{i_n}E_n$. Here, none of the substrings E_j of E contain occurrences of variables $V_1, \ldots, V_k$ after substitution. Some of E_j are possibly *null*, and each V_{i_j} is an occurrence of one of the variables $V_1, \ldots, V_k$.

3.5.1 Composition of Substitutions

If $\theta = \{T_1/V_1, \ldots, T_k/V_k\}$ and λ are any two substitutions, then the composition of θ and λ denoted by $\theta\lambda$ is union $\theta^{'} \cup \lambda^{'}$, defined as follows:

The $\theta^{'}$ is set of all components $T_i\lambda/V_i$, $1 \le i \le k$, such that $T_i\lambda$ (λ substituted in θ) is different from V_i, and $\lambda^{'}$ is set of all components of λ whose variables are not among $V_1, \ldots, V_k$.

Within a given scope, once a variable is bound, it may not be given a new binding in future unifications and inferences. If θ and λ are two substitution sets, then the composition of θ and λ, i.e., $\theta\lambda$, is obtained by applying λ to the elements of θ and adding the result to λ.

Following examples illustrate two different scenario of composition of substitutions.

Example 3.7 Find out the composition of $\{x/y, w/z\}$, $\{v/x\}$, and $\{A/v, f(B)/w\}$.

Let us assume that $\theta = \{x/y, w/z\}$, $\lambda = \{v/x\}$ and $\mu = \{A/v, f(B)/w\}$. Following are the steps:

1. To find the composition $\lambda\mu$, A is is substituted for v, and v is then substituted for x. Thus, $\lambda\mu = \{A/x, f(B)/w\}$.
2. When result of $\lambda\mu$ is substituted in θ, we get composition $\theta\lambda\mu = \{A/y, f(B)/z\}$. □

Example 3.8 Find out the composition of $\theta = \{g(x, y)/z\}$, and $\lambda = \{A/x, B/y, C/w, D/z\}$.

By composition,

$$\begin{aligned}\theta\lambda &= \{g(x, y)/z\} \circ \{A/x, B/y\}\\ &= \{g(A, B)/z, A/x, B/y, C/w\}\end{aligned}$$

The $\{D/z\}$ has not been included in the resultant substitution set, because otherwise, there will be two terms for the variable z, one $g(A, B)$ and other D. □

One of the important property of substitution is that, if E is any string, and $\sigma = \theta\lambda$, then $E\sigma = E\theta\lambda$. It is straight forward to verify that $\varepsilon\theta = \theta\varepsilon = \theta$ for any substitution θ. Also, composition enjoys the associative property $(\theta\lambda)\mu = \theta(\lambda\mu)$, so we may omit the parentheses in writing multiple compositions of substitutions. The substitutions are not in general commutative; i.e., it is generally not the case that $\theta\lambda = \lambda\theta$, because for this $E\theta\lambda$ has to be equal to $E\lambda\theta$, which is not guaranteed. However, the composition has distributive property.

The point of the composition operation on substitution is that, when E is any string, and $\sigma = \theta\lambda$, the string $E\sigma$ is just the string $E\theta\lambda$, i.e., the instance of $E\theta$ by λ.

3.5.2 Unification

If E is any set of well-formed expressions and θ is a substitution, then θ is said to unify E, or to be a unifier of E, if $E\theta$ is a singleton. Any set of well-formed expressions which has a unifier is said to be unifiable [6].

In proving theorems using quantified variables, it is often necessary to "match" certain subexpressions. For example, to apply the combination of modus ponens and universal instantiation (Eq. 3.5) to produce "$mortal(socrates)$", it was necessary to find substitution $\{socrtaes/x\}$ for x that makes *man(x)* and *man(socrates)* equal (singleton).

Unification *algorithm* determines the substitutions needed to make two predicate expressions match. For this, all the necessary condition is that variables must be universally quantified. Unless the variables in an expression are existentially quantified, they are assumed to be universally quantified. This criteria allows us full freedom choosing the substitutions. The existentially quantified variables can be eliminated by substituting them with constants or with Skolem functions that makes the sentence true. For example, in sentence,

$$\exists x\, mother(x, jill),$$

we can replace x with a constant designating jill's mother, *susan*, to get:

$$mother(susan, jill);$$

and write unifier as $\{susan/x\}$.

For perform unification, a variable can be replaced by any term, including other variable or function expressions of arbitrary complexity. This also includes function expressions that themselves contain variables. For example, the function expression, *mother(joe)*, may be substituted for x in $human(x)$ to get *human(mother(joe)).*

Example 3.9 Find out the substitution instances for$foo(x, a, zoo(y))$, given the similar predicates with literal arguments.

1. $foo(fred, a, zoo(z))$, where *fred* is substituted for x and z for y, i.e., $\lambda_1 = \{fred/x, z/y\}$. Thus,

$$foo(fred, a, zoo(z)) = foo(x, a, zoo(y))\lambda_1.$$

2. $foo(w, a, zoo(jack))$, where $\lambda_2 = \{w/x, jack/y\}$; hence

$$foo(w, a, zoo(jack)) = foo(x, a, zoo(y))\lambda_2.$$

3. $foo(z, a, zoo(moo(z)))$, where $\lambda_3 = \{z/x, moo(z)/y\}$, hence;

$$foo(z, a, zoo(moo(z))) = foo(x, a, zoo(y))\lambda_3.$$

We use the notation x/y to indicate that x is substituted for the variable y; we also call this as *bindings*, so y is bound to x. A variable cannot be unified with a term containing that variable. So $p(x)$ cannot be substituted for x, because this would create an infinite regression: $p(p(p(...x)...)$.

3.6 Resolution Principle

The resolution rule can be traced back to 1960, which was introduced by Davis and Putnam. However, this algorithm required all around ground instances for the given formula, which resulted to a combinatorial explosion. However, the source of combinatorial explosion was eliminated in 1965, when J. A. Robinson introduced an algorithm for unification. The unification allowed the instantiation of the formula during the proof "in demand", just as per the need through the newly introduced most general unifier (mgu) algorithm [8].

The resolution method for (propositional) logic due to J. A. Robinson (1965) is *sound and complete*, and a well-known procedure for checking the unsatisfiability of a set of clauses. The resolution is mathematical oriented rather than human oriented. It is quite powerful both in the psychological sense that it condones single inferences which are often beyond the ability of human to grasp, and in theoretical sense that it alone, as sole inference principle, forms a complete system of FOPL. Because of only one inference type, it allows to avoid the combinatorial obstacles to efficiency.

Let us refresh us with the terminology we discussed in the beginning of this chapter. We will designate a *literal* by symbol L, which is either a propositional

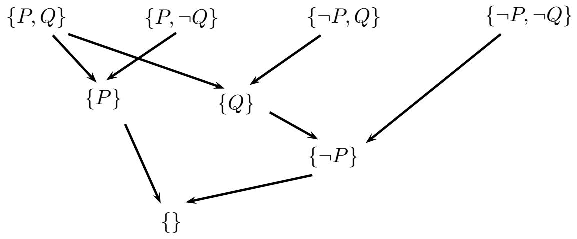


Fig. 3.2 DAG for theorem proving

symbol, P, or the negation, $\neg P$. A finite set of literals $\{L_1, \ldots, L_k\}$ is a *clause*, which is interpreted as a disjunction of literals, $L_1 \vee \cdots \vee L_k$. With $k = 0$, it is an empty clause, denoted as []. A conjunction of a set of clauses $\Gamma = \{C_1, \ldots, C_n\}$ is interpreted as $C_1 \wedge \cdots \wedge C_n$. In short, we write set of clauses as $\Gamma = C_1, \ldots, C_n$.

The resolution method is a procedure for determining whether a set of clauses Γ, is *unsatisfiable*. To find out latter, the resolution method first builds a certain kind of labeled *DAG* (Directed Acyclic Graph) whose terminal nodes are labeled with clauses in Γ and the interior nodes are labeled as per the resolution rule [5].

Consider that there are any two clauses $C = A \cup \{P\}$ and $D = B \cup \{\neg P\}$ (where P is a propositional letter, $P \notin A$ and $\neg P \notin B$). The resolvent of C and D is the clause $R = A \cup B$ obtained by canceling out P and $\neg P$, and making disjunction of the remaining part in each clause. A resolution DAG for Γ is a DAG whose terminal nodes are labeled with clauses from Γ and such that every interior node n has exactly two predecessors, n_1 and n_2 so that n is labeled with the resolvent of the clauses labeling n_1 and n_2. A resolution refutation for Γ is a resolution DAG with a single root whose label is the empty clause. Note that the root will come at the end, as shown in Fig. 3.2.

Example 3.10 Resolution refutation.

A resolution refutation for the set of clauses

$$\Gamma = \{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}\}.$$

is shown in Fig. 3.2. □

A recursive algorithm can be given for construction of a resolution DAG using any number of clauses, and to prove its correctness. That means, if the input set of clauses is unsatisfiable, the output resolution DAG is a resolution refutation. This confirms the completeness of propositional resolution constructively.

3.6.1 Theorem Proving Formalism

It is a syntactic inference procedure, when applied to clauses, determines, if the satisfied set is unsatisfiable. Proof is similar to proof by *contradiction* and deduce [] (i.e., null). If for example, we have set of clauses (axioms) $C_1, C_2, \ldots, C_n$, and we want to deduce D, i.e., D is logical consequence of of $C_1, C_2, \ldots, C_n$. For this we add $\neg D$ to the set $\{C_1, C_2, \ldots, C_n\}$, then we show that set is unsatisfiable by deducing contradiction [7].

The process of deduction using resolution is given in Algorithm 3.1. Given two clauses C_1, C_2 with no variables in common, and if l_1 is a literal in C_1 and its complement literal l_2 is a literal in C_2, then l_1, l_2 can be dropped and disjunction C is obtained from the remaining part of C_1, C_2. The C is called resolvent of C_1, C_2.

Let $C_1 = \neg P \vee Q$, and $C_2 = \neg Q \vee R$, then following can be deduced through resolution,

$$\frac{P \Rightarrow Q, Q \Rightarrow R}{P \Rightarrow R} \tag{3.2}$$

equivalently,

$$\frac{(\neg P \vee Q), (\neg Q \vee R)}{\therefore (\neg P \vee R)}. \tag{3.3}$$

It can be easily verified that $(\neg P \vee Q) \wedge (\neg Q \vee R) \models (\neg P \vee R)$, hence $(\neg P \vee Q) \wedge (\neg Q \vee R) \Rightarrow (\neg P \vee R)$ is a valid statement. Thus, $\neg P \vee R$ is inference or the resolvent. Arriving to a proof by above is called proof by *refutation.*

Resolution says that if there are axioms of the form $\neg P \vee Q$ and there is another axiom of the form $\neg Q \vee R$, then $\neg P \vee R$ logically follows; called the *resolvent.* Let us see why it is so? When $\neg P \vee Q$ is True, then either $\neg P$ is True or Q is True. For other expression, when $\neg Q \vee R$ is True, then either $\neg Q$ is True or R is True. Then we can say that $\neg P \vee R$ is certainly True. This can be generalized to two expressions, when we have any number of expressions, but two must be of opposite signs.

3.6.2 Proof by Resolution

To prove a theorem, one obvious strategy is to search forward from the axioms, using sound rules of inference. We try to prove a theorem by refutation. It requires to show that negation of a theorem cannot be True. The steps for a proof by resolution are:

1. Assume that negation of the theorem is True.
2. Try to show that axioms and assumed negation of theorem, together are True, which cannot be True.
3. Conclude that above leads to contradiction.
4. Conclude that theorem is True because its negation cannot be True.

To apply the resolution rule,

1. Find two sentences that contain the same literal, one in its positive form and one in its negative form, like,

$$CNF : summer \vee winter, \neg winter \vee cold,$$

2. use the *resolution* rule to eliminate the complement literals from both sentences to get,

$$CNF : summer \vee cold.$$

The Algorithm 3.1 is an algorithm for theorem proving through resolution-refutation, where α is the theorem to be proved, and β is set of axioms, both of these are input to the algorithm. All the inputs to algorithm are in the clause form. The algorithm returns "true" if the theorem is true, else returns "False".

Algorithm 3.1 Algorithm-Resolve(Input: α, β)

```
1: $\Gamma = \beta \cup \{\neg\alpha\}$
2: while there is a resolvable pair of clauses $C_i, C_j \in \Gamma$ do
3:    $C = resolve(C_i, C_j)$
4:    if $C = NIL$ then
5:       return "Theorem $\alpha$ is true"
6:    end if
7:    $\Gamma = \Gamma \cup \{C\}$
8: end while
9: Report that theorem is False
```

3.7 Complexity of Resolution Proof

The question is, how you can be so clever to pickup the right clauses to resolve? The answer is that you take advantage of two ideas:

1. You can be sure that every resolution involves the *negated theorem*, directly or indirectly.
2. You know where you are and where you are going, hence you can compute the difference to help you proceed with your intuition for selection of clauses.

Consider there are total n clauses, $c_1 \dots c_n$. We can try to match c_1 with $c_2 \dots c_n$, and in next level c_2 is matched with $c_3 \dots c_n$, and so on. This results to breadth first search (BFS). Consider that resolvents generated due to this matching are $c'_1 \dots c'_m$. Next all the newly generated clauses are matched with the original, and then they are merged into the original. This process is repeated until contradiction is reached, showing that theorem is proved. Since, the entire set of clauses are compared, the proof is bound to result, if it exists, at all. This gives completeness to the resolution proof.

The other alternative is, nodes which are farther and farther away are matched before those which are closer to the root. The c_1 is matched with first child c_2 out of $c_2 \ldots c_n$. Then c_2 is matched with its first child generated, and so on, resulting to the search process called DFS (depth first search).

However, the above both are brute-force algorithms, and are complex. The other methods are heuristic based. In fact, there is difficulty to express your concepts required in pure logic. One of the approaches is to use the clauses having smallest number of literals. Another, to use negated clauses.

The resolution search strategies are subject to the exponential-explosion problem. Due to this, those proofs which require long *chains of inferences*, will be exponentially expensive in time.

All resolution search strategies are subject to a version of *halting problem*, for search is not guaranteed to terminate unless there actually is a proof. In fact, all complete proof procedures for the first order predicate calculus are subject to halting problem. Complete proof procedures are said to be *semi-decidable*, because they are generated to tell you whether an expression is a theorem, only if the expression is indeed a theorem.

Theorem proving is suitable for certain problems, but not for all problems, due to the following reasons:

1. Complete theorem proving requires search, and search is inherently exponential,
2. Theorem provers may not help you to solve practical problems, even if they do their work instantaneously.

3.8 Interpretation and Inferences

A FOPL statement is made of predicates, arguments (constants or variables), functions, operators, and quantifiers. Interpretation is process of assignment of truth values (True/False) to subexpressions and atomic expressions, and computing the resultant value of any expression/statement. A statement or expression in predicate logic is also called *wwf* (well formed formula).

Consider the interpretation of predicate formula:

$$\forall x[bird(x) \rightarrow flies(x)]. \tag{3.4}$$

To find out the *satisfiability* of the formula (3.4), we need to substitute (*instantiate*) a value for x (an instance of x) and check if $flies(x)$ is true. Until, that x is found, it may require instantiation with large number of values. Similarly, to check if the Eq. (3.4) is valid, it may require infinitely large number of values in the domain of

x to be verified. If any one of that makes the formula false, the formula is not valid. Thus, checking of satisfiability as well as validity of a formula is predicate logic are complex process. The approach of *truth table* and *tableau* method we discussed in the previous chapter are applicable here also.

Given a predicate sentences of m number of predicates each having one argument, and domain size of all the arguments is n, in the worst case it will require total n^m substitutions to test for satisfiability, as well as for validity checking. However, a sentence of m propositions will require in the worst case only 2^m substitutions. Hence, satisfiability checking in predicate sentences is much more complex than that in proposition logic. It equally applies with expressions having existential quantifiers, like, $\exists x[bird(x) \rightarrow flies(x)]$.

Thus, it is only the proof methods, using which *logical deductions* can be carried out in realistic times.

Example 3.11 Given, "All men are mortal" and "Socrates is man", infer using predicate logic, that "Socrates is mortal".

The above statement can be written in predicate logic as:

$$\begin{aligned}&\forall x[man(x) \Rightarrow mortal(x)],\\ &man(socrates).\end{aligned} \tag{3.5}$$

Using a rule called *universal instantiation*, a variable can be instantiated by a constant and universal quantifier can be dropped. Hence, from (3.5) we have,

$$\begin{aligned}&man(socrates) \Rightarrow mortal(socrates),\\ &man(socrates).\end{aligned} \tag{3.6}$$

Using the rule of *modus ponens* on (3.6) we deduce "*mortal*(*socrates*)". It is also *logical consequence*. If $\Gamma = \{[man(socrates) \Rightarrow mortal(socrates)] \wedge man(socrates)\}$, and $\alpha = mortal(socrates)$, then we can say that $\Gamma \vdash \alpha$.

The set of formulas Γ is called knowledge base. To find out the result for the query "Who is man?", we must give the query

$$?man(X).$$

in Prolog (to be discussed later), which will match (called *unify* or *substitute*) $man(X)$ with $man(socrates)$ with a unification set, say, $\theta = \{socrates/X\}$. The substitution which returns $man(socrates)$ is represented by $man(X)\theta$. □

Example 3.12 Prolog Program.

The sentence in Eq. (3.5) will appear in prolog as,

$$\begin{aligned}&mortal(socrates) \text{ :- } man(socrates).\\&man(socrates).\end{aligned}$$

Here, the sign ':-'is read as 'if'. The subexpression before the sign ':-'is called 'head' or *procedure name* and the part after ':-' is called *body* of the *rule*. The sentence (3.6) can also be written in a *clause form* (to be precise, in *Horn clause* form) as,

$$\begin{aligned}&mortal(socrates) \vee \neg man(socrates).\\&man(socrates).\end{aligned} \tag{3.7}$$

3.8.1 Herbrand's Universe

Defining an operational semantics for a programming language is nothing but to define an implementation independent interpreter for it. In case of predicate logic, the proof procedure itself behaves like an interpreter. The Herbrand's Universe and Herbrand's base play an important role in interpretation of predicate language. In the following, we define the Herbrand's Universe and Herbrand's Base.

Definition 3.1 (*Herbrand's Universe*) In a predicate logic program, a Herbrand Universe **H**, is a set of ground terms that use only function symbols and constants.

Definition 3.2 (*Herbrand's Base*) A set of atomic formulas formed by predicate symbols in a program, is called Herbrand's base. The additional condition is that, arguments of these predicate symbols are in the Herbrand Universe.

For a predicate program, the Herbrand universe and Herbrand base are countably infinite if the predicate program contains a function symbol of positive arity. If the arity of function symbols is zero, then both the haerbrand's universe and base are finite [3].

In special cases, when resolution proof is used on FOPL, it reduces the expressions to propositional form. If the set of clauses is **A**, its Harbrand's universe is set of all the ground terms formed using only the function symbols and constants in **A**. For example, if **A** has constants a, b, and a unary function symbol f, then the Herbrand universe is the infinite set:

$$\{a, b, f(a), f(b), f(f(a)), f(f(b)), f(f(f(a)), \dots\}.$$

The Herbrand's base of **A** is the set of all ground clauses $c\theta$ where $c \in \mathbf{A}$ and θ is a substitution that assigns the variables in c to terms in the Herbrand's universe.

Horn clauses

The *Horn* clauses are *terms* without variables, these are constructed using constants and function symbols that occur in the set of clauses **A**. These terms form the *data structures*, which are manipulated by the program built-in in the clauses **A**. The collection of all such *terms*, determined by **A**, is called *Herbrand universe*. Every n-ary predicate symbol P occurring in **A** denotes an n-ary relation over the Herbrand universe of **A**. We call the n-tuples which belong to such relations as *input-output* tuples and the relations themselves as *input-output relations*.

In an inference system, the operational semantics determine a unique denotation for a formula P such that the n-tuple $(t_1, \dots, t_m)$ belongs to the denotation of $P \in \mathbf{A}$, iff $\mathbf{A} \vdash P(t_1, \dots, t_n)$. That is,

$$D(P) = \{(t_1, \dots, t_n) : \mathbf{A} \vdash P(t_1, \dots, t_n)\}. \tag{3.8}$$

We first pick an arbitrary constant, say, a, and then construct the variable-free terms. Formally, $D(P)$ is inductively defined as follows:

1. All constants occurring in P belong to $D(P)$; if no constant occurs in P, then $a \in D(P)$.
2. For every n-ary functional symbol p occurring in P, if $t_1, t_2, \dots, t_n \in D(P)$ then $p(t_1, t_2, \dots, t_n) \in D(P)$.

Here $X \vdash Y$ means X derives Y. For resolutions systems, if $X \vdash Y$ then there exists a refutation of the sentence in clausal form with atoms as X and $\overline{Y}$.

In goal oriented inference systems, the procedure calls are replaced by procedure bodies. Such inference systems correspond to standard notion of operational semantics. In theoretical sense, any inference system, based on predicate logic represents an abstract machine, that generates only those derivatives which are determined by this inference system.

For predicate logic, the corresponding programs compute the relations represented by predicate symbols in the set of clauses **A**. These relations may be in the form of predicates or functions. However, these function or predicate symbols are not treated as functions computed by the program, but they result into some *data structures*, and these data structures are actually the input and output objects of the relations / functions being computed.

Definition 3.3 Herbrand's Structure.

Let P be a formula in Skolem form (when a constant, say, a is substituted for x in $\exists x\ p(f(x), b)$ results to Skolem form $p(f(a), b)))$. A structure $\mathscr{A} = (U_{\mathscr{A}}, I_{\mathscr{A}})$ suitable for p is a *Herbrand structure* for P if it satisfies the following conditions:

1. $U_{\mathscr{A}} = D(P)$, and
2. for every n-ry function symbol p occurring in P and every $t_1, t_2, \dots, t_n \in D(P)$: $f^{\mathscr{A}}(t_1, t_2, \dots, t_n) = f(t_1, t_2, \dots, t_n)$.

In above, $\mathscr{A} = (U_{\mathscr{A}}, I_{\mathscr{A}})$ is model with $U_{\mathscr{A}}$ as formula and $I_{\mathscr{A}}$ as its interpretation. □

Definition 3.4 (*General logic program*) It is a finite set of general rules, with both positive and negative subgoals.

A general logic program comprises rules and facts. A general rule's format is: its *head*, or *conclusion* is to the left of the symbol "$\leftarrow$," (read "if"), and its subgoals (called body) is right of the symbol "$\leftarrow$". Following is an example of rule, where $p(X)$ is head, $q(X)$ is positive subgoal, and $r(X)$ is a negative subgoal.

$$p(X) \leftarrow q(X),\ \neg\, r(X). \tag{3.9}$$

This rule may be read as "$p(X)$ *if* $q(X)$ *and not* $r(X)$." A *Horn rule* has no negative subgoals, and a *Horn logic program* is made of only Horn rules.

We will follow the conventions of *Prolog* for naming the objects: the logical variables begin with an uppercase letter, while the constants, functions, and predicates begin with a lowercase letter. For both the predicate and its relation, we will use the same symbol, for example p.

Followings may be the arguments of a predicate:

1. a constant/variable is a term;
2. a function symbol with terms as arguments, is a term.

The terms may be viewed as data structures of the program, with function symbols serving as record names. Often a constant is treated as a function symbol of arity zero.

Following are the definitions of some important terms.

Definition 3.5 (*Herbrand Instantiation*) Herbrand instantiation of a general logic program is the set of rules obtained by substituting terms in the Herbrand universe for variables, in every possible way.

Definition 3.6 An **instantiated rule** is one only, whereas "uninstantiated" logic programs are assumed to be a finite set of rules, and instantiated logic programs may be infinite in number.

Definition 3.7 (*Complement of a set*) For a set of literals L its complement is a set formed by complementing of each literal in L, represented by $\neg L$.

Further,

- p is said to be *inconsistent* with L if $p \in \neg L$,
- Sets of literals R and L are *inconsistent* if at least one literal in R is inconsistent with L, i.e., when $R \cap \neg L \neq \phi$,
- A set of literal is *inconsistent* if it is inconsistent with itself; otherwise it is *consistent*.

3.8.2 Herbrand's Theorem

Herbrands theorem is a fundamental theorem based on mathematical logic, that permits a certain type in reduction from FOPL to propositional logic [4].

In its simplest form, the Herbrand's theorem states that a formula of first-order predicate logic $\exists x\, A$, where A is quantifier free, is provable if and only if there exist ground terms $M_1, \ldots, M_n$ such that,

$$\models A[x := M_1] \vee \cdots \vee A[x := M_n]. \tag{3.10}$$

When using the classical formulation, the Herbrand's theorem relates the validity of a first-order formula in *Skolem prenex form*[1] to the validity of one of its Herbrand extensions. That means, the formula $\forall x_1 \ldots \forall x_n \psi(x_1 \ldots, x_n)$ is valid if, and only if, $\bigwedge_i^m \psi(t_{i,1}, \ldots, t_{i,n})$ is valid for some $m \geq 1$ and some collection of ground Herbrand terms $t_{i,j}$.

Since it is possible that every classical first-order formula can be reduced to this Skolem prenex form through the Skolemization while preserving its satisfiability, the Herbrand's theorem provides a way to reduce the question of validity of first-order formulas to propositional logic formula.

However, the required Herbrand's extension and the terms $t_{i,j}$ cannot be computed recursively (for otherwise first-order logic would be decidable), this result is highly useful for the automated reasoning as it gives a way to some highly efficient proof methods such as *resolution* and the *resolution refutation*.

Theorem 3.1 *A closed formula F in Skolem form is satisfiable if and only if it has a Herbrand model.*

Proof If the formula has a Herbrand model then it is satisfiable. For the other direction let $\mathscr{A} = (U_{\mathscr{A}}, I_{\mathscr{A}})$ be an arbitrary model of F. We define a Herbrand structure $\mathscr{B} = (U_{\mathscr{B}}, I_{\mathscr{B}})$ as follows:

Universe: $U_{\mathscr{B}} = D(F)$

Functional Symbols: $f^{\mathscr{B}}(t_1, t_2, \ldots, t_n) = f(t_1, t_2, \ldots, t_n)$

Predicate Symbols: $(t_1, \ldots, t_n) \in P^{\mathscr{B}}$ iff $\mathscr{A}(t_1), \ldots, \mathscr{A}(t_n) \in P^{\mathscr{A}}$.

Claim: $\mathscr{B}$ is also a model of F.

We prove a stronger assertion: For every closed form G in Skolem form such that G^* only contains atomic formulas of F^* : *if* $\mathscr{A} \models G$ then $\mathscr{B} \models G$.

By induction on the number n of universal quantifiers of G.

Basis ($n = 0$). Then G has no quantifiers at all.

It follows $\mathscr{A}(G) = \mathscr{B}(G)$, this proves the theorem. □

To perform reasoning with the Herbrand base, the unifiers are not required, and we have a *sound* and *complete* reasoning procedure, which is guaranteed to terminate. The idea used in this approach is: Herbrand's base will typically be an infinite set of propositional clauses, but it will be finite when Herbrand's universe is finite (there

[1]A string of quantifiers followed by a quantifier-free part, e.g., $\forall x_1 \ldots \forall x_n \psi(x_1 \ldots, x_n)$.

is no function symbols and only finitely many constants appear in it). Sometimes we can keep the universe finite by considering the type of the arguments (say t) and values of functions (f), and include a term like $f(t)$ in the universe only if the type of t is appropriate for the function f. For example, $f(t)$ may be, *birthday*(*john*), which produces a date.

3.8.3 The Procedural Interpretation

It is easy to procedurally interpret the sets of clauses, say, **A**, which contain at most one positive literal per clause. However, along with this any number of negative literals can also exist. Such sets of clauses are called *Horn sentences* or *Horn Clauses* or simply clauses. We distinguish three kinds of *Horn clauses* [3].

1. '[]'the *empty clause*, containing no literals and denoting the truth value *false*, is interpreted as a *halt statement*.
2. $\bar{B}_1 \vee \cdots \vee \bar{B}_n$, a clause consisting of no positive literals and $n \geq 1$ negative literals, is interpreted as a *goal statement*. Note that goal statement is negated and added into the knowledge base to obtain the proof through *resolution refutation*.
3. $A \vee \bar{B}_1 \vee \cdots \vee \bar{B}_n$, a clause consisting of exactly one positive literal and $n \geq 0$ negative literals is interpreted as a *procedure declaration* (i.e., rule in Prolog program). The positive literal A is the *procedure name* and the collective negative literals are the *procedure body*. Each negative literal B_i, in the procedure body is interpreted as a *procedure call*. When $n = 0$ the procedure declaration has an empty body and interpreted as an unqualified assertion of fact.

In the procedural interpretation, a set of procedure declarations is a program. Computation is initiated by an *initial goal* statement, which proceeds by using declared procedures to derive new goal statements (*subgoals*) B_is from old goal statements, and terminates on the derivation of the halt statement. Such derivation of goal statements is accomplished by *resolution*, which is interpreted as *procedural invocation*.

Consider that, a selected procedure call $\bar{A}_1$ inside the body of a goal statement as,

$$\bar{A}_1 \vee \cdots \vee \bar{A}_{i-1} \vee \bar{A}_i \vee \bar{A}_{i+1} \vee \cdots \vee \bar{A}_n \tag{3.11}$$

and a procedure declaration is given as,

$$A' \vee \bar{B}_1 \vee \cdots \vee \bar{B}_m, m \geq 0. \tag{3.12}$$

Suppose, the name of procedure A' matches with the procedure call A_i, i,e., some substitution θ of terms for variables makes A_i and A' identical. In such a case, the resolution derives a new goal statement by disjunction formulas (3.11) and (3.12) as given below, subject to substitution θ.

$$(\bar{A}_1 \vee \cdots \vee \bar{A}_{i-1} \vee \bar{B}_1 \vee \cdots \vee \bar{B}_m \vee \bar{A}_{i+1} \vee \cdots \vee \bar{A}_n)\theta. \tag{3.13}$$

In general, any *derivation* can be regarded as a computation, and any *refutation* (i.e. derivation of []) can be regarded as a successfully terminating computation. It is to be noted that, only goal oriented resolution derivations correspond to the standard notion of computation.

Thus, a goal-oriented derivation, from an initial set of Horn clauses **A** and from an initial goal statement (computation) $C_1 \in \mathbf{A}$, is a sequence of goal statements $C_1, \ldots, C_n$. So that each C_i contains a single selected procedure call and C_{i+1}, obtained from C_i by procedure invocation relative to the selected procedure call in C_i, using a procedure declaration in **A**.

For the implementation of above, one method is *model elimination*. Using this, the selection of procedure calls is governed by the last-in/first-out rule: a goal statement is treated as a stack of procedure calls. The selected procedure call must be at the top of the stack. The new procedure calls which by procedure invocation replace the selected procedure call are inserted at the top of the stack. This would result to a *depth-first search* procedure.

The Predicate logic is a *nondeterministic* programming language. Consequently, given a single goal statement, several procedure declarations can have a name which matches the selected procedure call. Each declaration gives rise to a new subgoal statement. A *proof procedure* which sequences the generation of derivations in the search for a refutation behaves as an *interpreter* for the program incorporated in the initial set of clauses.

The following example explains how to use procedural interpretation to append two given lists.

Example 3.13 Appending two lists [3].

Let a term $cons(x, y)$ is interpreted as a list whose first element, the *head*, is x and whose *tail* y is the rest of the list. The constant *nil* denotes the empty list. The terms u, x, y, and z are variables. The predicate *append(x,y,z)* denotes the relationship: z is obtained by appending y to x.

The following two clauses constitute a program for appending two lists.

$$append(nil, x, x). \tag{3.14}$$

$$append(cons(x, y), z, cons(x, u)) \vee \overline{append}(y, z, u). \tag{3.15}$$

The clause in statement (3.14) represents halt statement. In (3.15) there is a positive literal for procedure name, and negative literal(s) for the procedure body, both together it is procedure declaration. The positive literal means, if $cons(x, y)$ is appended with z, it results to x appended with u such that u is, y appended with z. The later part is indicated by the complementary (negative) term. Note that clausal expression (3.15) is logically equivalent to the expression $append(y, z, u) \rightarrow append(cons(x, y), z, cons(x, u))$.

Suppose it is required to compute the result of appending list $cons(b, nil)$ to the list $cons(a, nil)$. Therefore, the goal statement is,

$$append(cons(a, nil), cons(b, nil), v), \tag{3.16}$$

where v (a variable) and a, b (constants), are the "atoms" of the lists. To prove using resolution, we add the negation of the goal,

$$\overline{append}(cons(a, nil), cons(b, nil), v), \tag{3.17}$$

into the set of clauses. The program is activated by this *goal statement* to carry out the append operation. With this goal statement the program is deterministic, because only one choice is available for matching. The following computation follows with a goal directed theorem prover as interpreter: The goal statement,

$$C_1 = \overline{append}(cons(a, nil), cons(b, nil), v). \tag{3.18}$$

matches with the clause statement (3.15) with matchings: $x = a, y = nil, z = cons(b, nil)$. Also, $v = cons(x, u) = cons(a, u)$, i.e., there exists a unifier $\theta_1 = \{cons(a, w)/v\}$. The variable u has been renamed as w. On unifying clauses (3.18) and (3.15), the next computation C_2 is:

$$C_2 = \overline{append}(nil, cons(b, nil), w)\theta_1. \tag{3.19}$$

Keeping θ_1 accompanying the predicate in above is for the purpose that if C_2 is to be unified with some other predicate, the matching of the two shall be subject to the same unifier θ_1.

As next matching, C_2 can be unified with (3.14) using a new unifier $\theta_2 = \{cons(b, nil)/w\}$ to get next computation,

$$C_3 = []\theta_2. \tag{3.20}$$

The result of the computation is value of v in the substitution, i.e.,

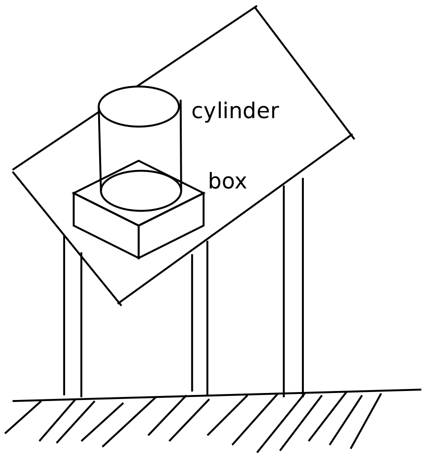
$$\begin{aligned} v &= cons(a, u)\\ &= cons(a, w)\\ &= cons(a, cons(b, nil)). \end{aligned}$$

The above result is equal to goal: $append(cons(a, nil), cons(b, nil), v)$. □

Example 3.14 Theorem proving using resolution-refutation.

Following axioms are about the observed block relationship shown in Fig. 3.3, which are already in clausal form.

Fig. 3.3 Objects on table



$$on(cylinder, box).$$

$$on(box, table).$$

It is required to be shown that object *cylinder* is above table, i.e., $above(cylinder, table)$, given the the following rules:

$\forall x \forall y[on(x, y) \rightarrow above(x, y)]$, and
$\forall x \forall y \forall z[above(x, y) \wedge above(y, z) \rightarrow above(x, z)]$.

After we have gone through the procedure for conversion to clausal form, the above axioms are transformed into clause forms.

$\neg on(u, v) \vee above(u, v)$.
$\neg above(x, y) \vee \neg above(y, z) \vee above(x, z)$.

The expression to be proved is “above(cylinder, table)”; its negation is $\neg above(cylinder, table)$. Let us list all the clauses systematically.

(1) $\neg on(u, v) \vee above(u, v)$.
(2) $\neg above(x, y) \vee \neg above(y, z) \vee above(x, z)$.
(3) $on(cylinder, box)$.
(4) $on(box, table)$.
(5) $\neg above(cylinder, table)$.

Now, we manually run the Algorithm 3.1 on the clauses (1)–(5), as well as those which would created new, to unify them according to unification Algorithm 3.2, until we reach to a null resolvent.

First we resolve clauses (2) and (5) and bind x to ‘*cylinder*’and z to ‘*table*’. Applying the resolution, we get resolvent (6). Unifier for this is $\{cylinder/x, table/z\}$.

(2) $\neg above(cylinder, y) \vee \neg above(y, table) \vee above(cylinder, table)$.
(5) $\neg above(cylinder, table)$.
(6) $\neg above(cylinder, y) \vee \neg above(y, table)$.

Next, resolve clauses (1) with (6), binding u with y and v with '*table*', we get (7). Unifier for this is $\{y/u, table/v\}$.

(1) $\neg on(y, table) \vee above(y, table)$.
(6) $\neg above(cylinder, y) \vee \neg above(y, table)$.
(7) $\neg on(y, table) \vee \neg above(cylinder, y)$.

We use (1) again with (7) with u bound to *cylinder* and v to y. Unifier for this is $\{cylinder/u, y/v\}$. On resolving we get (8).

(1) $\neg on(cylinder, y) \vee above(cylinder, y)$.
(7) $\neg on(y, table) \vee \neg above(cylinder, y)$.
(8) $\neg on(cylinder, y) \vee \neg on(y, table)$.

Next, use clause (3) and (8), binding y to *box*, with unifier $\{box/y\}$. We get (8) as resolvent.

(3) $on(cylinder, box)$.
(8) $\neg on(cylinder, box) \vee \neg on(box, table)$.
(9) $\neg on(box, table)$.

Finally, the clauses (4) and (9) are resolved to get '[]':

(4) $on(box, table)$.
(9) $\neg on(box, table)$.
(10) [].

Since we have arrived at the contradiction, it shows that negation of the theorem: $\neg above(cylinder, table)$ must be False. Hence the theorem $above(cylinder, table)$ must be True. □

3.9 Most General Unifiers

The simple approach to avoid needless search in a first-order derivation is to keep the search procedure as general as possible. Consider, for example the following two clauses, each as a literal only.

$$c_1 = p(g(x), f(x), z),$$

and

$$c_2 = \neg p(y, f(w), a).$$

They are unified by the substitution θ_1,

$$\theta_1 = \{b/x, g(b)/y, a/z, b/w\},$$

and also by θ_2,

$$\theta_2 = \{f(z)/x, g(f(z))/y, a/z, f(z)/w\}.$$

Note that a constant, or variable, or a function substitutes for a variable, and not the other way.

We may very well be able to derive the empty clause using c_1, c_2 with substitution of θ_1, followed with application of resolution. But if we cannot, we will need to consider other substitutions like θ_2.

The trouble is that both of these substitutions are overly specific. We can see that any unifier must give w the same value as x, and to y the same as $g(x)$, but we do not need to commit yet to a value for x. The substitution,

$$\theta_3 = \{g(x)/y, a/z, x/w\}$$

unifies the two literals without making an arbitrary choice that might preclude a path to the empty clause. The θ_3 is a *most general unifier* (mgu).

More precisely, an mgu θ of literals ρ_1 and ρ_2 is a unifier that has the property that for any other unifier θ', there is a further substitution θ^* such that $\theta' = \theta\theta^*$. So starting with θ, you can always get to any other unifier by applying additional substitutions. For example, given θ_3, we can get to θ_1 by further applying $\lambda = \{b/x\}$ so that $\theta_1 = \theta_3\lambda$. And, we can get to θ_2 by $\mu = \{f(z)/x\}$ so that $\theta_2 = \theta_3\mu$. Note that an mgu need not be unique. For example, $\theta_4 = \{g(w)/y, a/z, w/x\}$ is also an *mgu* for c_1 and c_2.

The key fact about mgus is that we can limits the resolution rule to mgus without loss of completeness. This helps immensely in the search since it dramatically reduces the number of resolvents that can be inferred from these two input clauses.

Example 3.15 Given a unifier, obtain a more general unifier.

Suppose you have two expressions $p(x)$ an $p(y)$. One way to unify these is to substitute any constant expression for x and y: $S = \{fred/x, fred/y\}$. But this is not the most general unifier, because if we substitute any variable for x and y, we get a more general unifier: $G = \{z/x, z/y\}$. The first unifier is a valid unifier, but it would lessen the generality of inferences that we might want to make.

$$\begin{aligned} \text{Let } E &= \{p(x), p(y)\}, \\ S &= \{fred/x, fred/y\}, \\ G &= \{z/x, z/y\}. \\ \text{Now let } S' &= \{fred/z\} \end{aligned}$$

$$\begin{aligned} \text{Then } ES &= \{p(fred), p(fred)\} \\ \text{and } GS' &= \{fred/x, fred/y\} \\ \text{and therefore } EGS' &= \{p(fred), p(fred)\} = ES. \end{aligned}$$

So, given a unifier, you can always create a more general unifier. When both of these unifiers are composed and instantiate the original expression E, you get the same instance as it was obtained with the earlier unifier.

3.9.1 *Lifting*

It is necessary to show that the general resolution principle is *sound* and *complete*. However, a technical difficulty is the completeness of the proof. Using the Herbrand's theorem and semantic trees, we can prove that there is a ground *resolution refutation* of an unsatisfiable set of clauses. But, this cannot be generalized as a proof for general resolution, because the concept of semantic trees cannot be generalized. Why it cannot be generalized, is due to the variables, which give rise to potentially infinite number of elements in the Herbrand's base, as we will show it shortly.

Fortunately, there is a technique, called, "Lifting", to prove completeness of a theorem. Following are the steps for lifting:

1. first prove the *completeness* of the system for a set of *ground* classes, then,
2. as a second step, lift the proof to non-ground case.

Example 3.16 Infinite inferences.

Let us assume that there are two non-ground clauses: 1. $p(u, a) \vee q_1(u)$ and, 2. $\neg p(v, w) \vee q_2(v, w)$. If the signature pattern contains function symbols, then these clauses have infinite *set* of instances, as follows:

$\{p(r, a) \vee q_1(r) \mid r \text{ is ground}\}$.
$\{\neg p(s, t) \vee q_2(s, t) \mid s, t \text{ are ground}\}$.

We can resolve above instances if and only if $r = s$ and $t = a$. Then we can apply the resolution refutation and obtain the inference given in the denominator of Eq. (3.21), which are infinite, due to variable s.

$$\frac{p(s, a) \vee q_1(s), \neg p(s, a) \vee q_2(s, a)}{q_1(s) \vee q_2(s, a)} \tag{3.21}$$

□

The above difficulty can be overcome by taking a ground resolution refutation and "lifting" it to a more abstract general form.

The lifting is an idea to represent infinite number of ground inferences of the form given in Eq. (3.21) by a single non-ground inferences:

$$\frac{p(u, a) \vee q_1(u),\ \neg p(v, w) \vee q_2(v, w)}{q_1(v) \vee q_2(v, a)}$$

This lifting can be done using most general unifier, we will be discussing shortly.

Example 3.17 Find out the Lifting for following clauses:

$C_1 = p(u) \vee p(f(v)) \vee p(f(w)) \vee q(u)$
$C_2 = \neg p(f(x)) \vee \neg p(z) \vee r(x)$

Using the substitution $\theta = \{f(a)/u, a/v, a/w, a/x, f(a)/z\}$, the above clauses become $C_1' = p(f(a)) \vee q(f(a))$, and $C_2' = \neg p(f(a)) \vee r(a)$. Using C_1' and C_2', it resolves to $C' = q(f(a)) \vee r(a)$. The lifting claims that there is a clause $C = q(f(x)) \vee r(x)$ which is resolvent for clauses C_1 and C_2, such that clause C' is ground instance of C. This can be realized using the *unification algorithm* to obtain a most general unifier (mgu) of clauses C_1 and C_2, the latter two clauses resolves to C, as
$\{f(x)/u, x/v, x/w, f(x)/z\}$.

3.9.2 Unification Algorithm

A unification algorithm is central to most of the theorem-proving systems. This algorithm receives as input a pair of expressions, and returns as output a set of substitutions (assignments) that make the two expressions look identical.

The unification algorithm recursively compares the structures of the clauses to be matched, working across element by element. The criteria is that,

1. the matching individuals, functions, and predicates must have the same names,
2. the matching functions and predicates must have the same number of arguments, and
3. all bindings of variables to values must be consistent throughout the whole match.

To unify two atomic formulas in an expression **A**, we need to understand the *disagreement* set.

Definition 3.8 Disagreement Set.

If **A** is any set of well-formed expressions, we call the set D the disagreement set of **A**, whenever D is the set of all well-formed subexpressions of the well-formed expressions in **A**, which begin at the first symbol position at which not all well-formed expressions in **A** have the same symbol. □

Example 3.18 Find out the disagreement set for given set of atoms.

Let the string is, $\mathbf{A} = \{p(x, h(x, y), y),\ p(x, k(y), y),\ p(x, a, b)\}$, having three predicate expressions. The disagreement set for **A** is,

$$D = \{h(x, y), k(y), a\}. \tag{3.22}$$

Once the disagreement is resolved through unification for this this symbol position, there is no disagreement at this position. The process is repeated for the new first symbol position at which all wffs in **A** do not have same symbol, and so on, until **A** becomes a singleton.

Evidently, if **A** is nonempty and is not a *singleton* (a set with exactly one element), then the disagreement set of **A** is not a singleton and nonempty. Also, if θ unifies **A**, and **A** is not singleton, the θ unifies the disagreement set **A**. □

For **A** to be a finite nonempty set of well-formed expressions for which the substitution Algorithm 3.2 terminates with "return $\sigma_{\mathbf{A}}$", the substitution $\sigma_{\mathbf{A}}$ available as output of the unification algorithm is called the most general unifier (mgu) of **A**, and **A** is said to be most generally unifiable [8, 9].

Algorithm 3.2 Unification-Algorithm (Input: **A**, Output: $\sigma_{\mathbf{A}}$)

```
1: Set $\sigma_0 = \varepsilon, k = 0$
2: while true do
3:    if $A\sigma_k$ is a singleton then
4:       Set $\sigma_{\mathbf{A}} = \sigma_k$
5:       terminate
6:    end if
7:    Let $U_k$ be the earliest and $V_k$ be the next earliest element in the disagreement set $D_k$ of $\mathbf{A}\sigma_k$
      (see Eq. 3.22)
8:    if $V_k$ is a variable, and does not occur in $U_k$ then
9:       set $\sigma_{k+1} = \sigma_k\{U_k/V_k\}$,
10:      $k = k + 1$
11:   else
12:      (A is not unifiable)
13:      exit.
14:   end if
15: end while
```

Through manually running the Algorithm 3.2 for the disagreement set in (3.22), stepwise computation for σ_k is as follows:

For $k = 0$, and $\sigma_0 = \varepsilon$,

$$\begin{aligned}\sigma_{k+1} &= \sigma_k\{k(y)/h(x, y)\}\\ \Rightarrow \sigma_1 &= \{k(y)/h(x, y)\}.\end{aligned}$$

which, in the next iteration becomes,

$$\begin{aligned}\sigma_2 &= \sigma_1\{a/k(y)\}\\ &= \{k(y)/h(x, y)\}\{a/k(y)\}.\end{aligned}$$

The same process is repeated for the disagreement set of 3rd argument in **A**, which results to substitution set as $\{b/y\}$.

$$\begin{aligned}\sigma_3 &= \sigma_2\{b/y\}\\ &= \{k(y)/k(x, y)\}\{a/k(y)\}\{b/y\}.\end{aligned}$$

On substituting these, we have,

$$\mathbf{A} = \{p(x, a, b), p(x, a, b), p(x, a, b\}.$$

which is a singleton, and σ_3 is mgu.

For obtaining the unifier σ_k, the necessary relation required between U_k and V_k is, V_k has to be a variable, and U_k can be a constant, variable, function, or predicate. V_k may even be a predicate or function with variable.

The Algorithm 3.2 always terminates for finite nonempty set of well-formed expressions, otherwise it would generate an infinite sequence of $A, A\sigma_1, A\sigma_2, \ldots,$ each of which is a finite nonempty sets of well-formed expressions, with the property that each successive set contains one less variable than its predecessor. However, this is impossible because A contains only finitely many distinct variables.

The Algorithm 3.2 runs in $O(n^2)$ time on the length of the terms, and an even better. However, there exists more complex, but linear time algorithms for same. Because, most general unifiers (mgus) greatly reduce the search, and can be calculated efficiently, almost all Resolution-based systems implementations are based on the concept of mgus.

3.10 Unfounded Sets

In the well-founded semantics, the unfounded sets provide the basis for negative conclusions. Let there is a program **P** (set of rules and facts in FOPL), its associated Herbrand base is H, and suppose its partial interpretation is I. Then, some $A \subseteq H$ is called an *unfounded set* of **P** with respect to the interpretation I, with following condition: for each instantiated rule $R \in \mathbf{P}$, at least one of the following holds: (In the rules **P** , we assume that p is a head, and q_i are the corresponding subgoals.)

1. Some positive / negative subgoal q_i of the body of the rule is false in the interpretation I,
2. Some positive subgoals q_i of the body occurs in the unfounded set A.

For rule R with respect to I, a literal that makes conditions 1 or 2 above true is called *witness of unusability*.

Intuitively, the interpretation I is intended model of **P**. The rules that satisfy condition 1 cannot be used for further derivations because their hypotheses are already known to be false.

The condition 2 in above, called *unfoundedness condition*, states that all the rules which might still be usable to derive something in A, should have an atom (i.e., a fact) in A as true. In other words, there is no single atom in A, that can be established to be true by the rules of **P** (as per the knowledge of interpretation I). Therefore, if we infer that some or all atoms in A are false, there is no way available later, using that we could infer that an atom is true [4].

Hence, the well-founded semantics uses conditions 1 and 2 to draw negative conclusions, and simultaneously infers all atoms in A to be false. The following example demonstrates the construction of unfounded set from the set of rules and facts.

Example 3.19 Unfounded set.

Assume that we have a program in predicate logic with instantiated atoms.

$$\begin{aligned}
&p(c).\\
&q(a) \leftarrow p(d).\\
&p(a) \leftarrow p(c), \neg\, p(b).\\
&q(b) \leftarrow q(a).\\
&p(b) \leftarrow \neg\, p(a).\\
&p(d) \leftarrow q(a), \neg\, q(b).\\
&p(d) \leftarrow q(b), \neg\, q(c).\\
&p(e) \leftarrow \neg\, p(d).
\end{aligned}$$

From above rules, we see that $A = \{p(d), q(a), q(b), q(c)\}$ is an unfounded set with respect to ϕ (null set). Since A is unfounded, its subsets are also unfounded. The component, $\{q(c)\}$ is unfounded due to condition (1), because there is no rule available to establish its truth. The set $\{p(d), q(a), q(b)\}$ is unfounded due to condition (2) (their subgoals or body appear in unfounded set.

There is no way available to establish $p(d)$ without first establishing $q(a)$ or $q(b)$. In other words, whether we can establish $\neg q(b)$ to support the first rule for $p(d)$ is irrelevant as far as determination of unfoundedness is the concern.

Interestingly, there is no way available to establish $q(a)$ in the absence of first establishing $p(d)$, and also there is no way available to establish $q(b)$ without first establishing $q(a)$. Further, $q(c)$ can never be proven. We note that among $p(d)$, $q(a)$, and $q(b)$ as goals, none can be proved without the other two or their negation as subgoals.

The pair $p(a), p(b)$, even though they depend on each other, but does not form an unfounded set due to the reason that the only dependence is through negation. Hence, it can be concluded that the any attempt for proof of $p(a)$ and $p(b)$ will fail, but this claim is faulty.

The difference between sets $\{p(d), q(a), q(b)\}$ and $\{p(a), p(b)\}$ is as follows: declaring any of $p(d)$, $q(a)$, or $q(b)$ false (unfounded), does not create a proof that any other element of the set is true.

Finally, consider the set $\{p(a), p(b)\}$: If any of the elements $p(a)$ or $p(b)$ is taken false, it becomes possible to prove that the other is true. And, if both are declared false together, there is an inconsistency. □

3.11 Summary

First-order logic is best suited as a basic theoretical instrument of a computer theorem proving program. From the theoretical point of view, an inference principle need only be *sound* (i.e., allow only logical consequences of premises to be deduced from them) and *effective* (i.e., it must be algorithmically decidable whether an alleged application of the inference principle is indeed an application of it). The resolution principle satisfies both.

Two types of semantics, namely, *operational* and *fixpoint*, have been defined for programing languages. The operational semantics defines input-output relation computed by a program in terms of the individual operations performed by the program inside the machine. Meaning of a program is nothing but the input-output relation obtained due to executing it in a machine.

A machine independent alternative to semantics, called *fixpoint semantics*, defines the meaning of a program as input-output relation which is the minimal fixpoint of a transformation associated with the program.

A FOPL statement is made of predicates, arguments (constants or variables), functions, operators, and quantifiers. Interpretation is process of assignment of truth values (True/False) to subexpressions and atomic expressions, and computing the resultant value of any expression/ statement.

It is easy to procedurally interpret the sets of clauses which contain at most one positive literal per clause. However, along with this any number of negative literals can also exist. Such sets of clauses are called *Horn sentences* or *Horn Clauses* or simply clauses.

The Predicate logic is a *nondeterministic* programming language. Consequently, given a single goal statement, several procedure declarations can have a name which matches the selected procedure call. Each declaration gives rise to a new subgoal statement.

A *proof procedure* which sequences the generation of derivations in the search for a refutation behaves as an *interpreter* for the program incorporated in the initial set of clauses. Defining an operational semantics for a programming language means to define an implementation independent *interpreter* for it. For predicate logic, the proof procedure behaves as such an interpreter.

The *Herbrand universe* is the set of ground terms which use the function symbols and constants that appear in the predicate logic program. The *Herbrand base* is defined as the set of atomic formulas formed by predicate symbols in the program, whose arguments are in the Herbrand universe.

Herbrands theorem is a fundamental theorem of mathematical logic, which allows a certain type of reduction of first-order logic to propositional logic.

A substitution component is any expression of the form T/V, where V is any variable and T is any term different from V, is called unifier. The V is called variable of component T/V, and T is called term of the component. A most general unifier (mgu) (i. e., simplest one) θ of literals ρ_1 and ρ_2 is a unifier that has the property that for any other unifier $\theta^{'}$, there is a further substitution θ^* such that $\theta^{'} = \theta\theta^*$.

The backbone of most theorem-proving systems is a unification algorithm. This algorithm returns a set of substitutions for a pair of input expressions. These substitutions may be assignments to variables or expressions, which make the two expressions (or variables or functions) identical or equivalent. To prove a theorem, one obvious strategy is to search forward from the axioms, using sound rules of inference. We try to prove a theorem by refutation. It requires to show that negation of a theorem cannot be True.

Exercises

1. Apply the Resolution theorem to prove:

   "Socrates is mortal", given that
   All men are mortal, and
   Socrates is man.

2. What are the other methods for automated theorem proving? Explain any three in brief.
3. Convert the following into clause form:

$$\forall x[p(x) \wedge q(x)] \Rightarrow [R(x, I) \wedge \exists y(\exists z\; r(y, z)$$

$$\Rightarrow S(x, y))] \vee \forall x\; T(x).$$

4. Show that a formula in CNF is valid if and only if each of its disjunctions contains a pair of complementary literals P and $\neg P$.
5. Prove or disprove the followings:

   a. If S is a first-order formula, then S is valid iff $S \rightarrow \perp$ is contradiction.
   b. If S is a first-order formula and x is a variable, then S is contradiction iff $\exists x S$ is a contradiction.

6. Using the resolution principle prove the validity of following formula:

$$\forall x \exists y(p(f(f(x)), y) \wedge \forall z(p(f(x), z)$$
$$\rightarrow p(x, g(x, z)))) \rightarrow \forall x \forall y\; p(x, y).$$

7. Is the predicate logic deterministic or nondetermnistic programming language? justify for yes / no.

8. Consider a set of statements of FOPL that uses two 1-place predicates: *Large* and *Small*. The set of object constants are a, b. Find out all possible models for this program. For each of the following sentences find out the models in which each of the sentence becomes true.

   a. $\forall x\, Large(x)$.
   b. $\forall x\, \neg Large(x)$.
   c. $\exists x\, Large(x)$.
   d. $\exists x\, \neg Large(x)$.
   e. $Large(a) \wedge Large(b)$.
   f. $Large(a) \vee Large(b)$.
   g. $\forall x\, [Large(x) \wedge Small(x)]$.
   h. $\forall x\, [Large(x) \vee Small(x)]$.
   i. $\forall x\, [Large(x) \Rightarrow \neg Small(x)]$.

9. Find out the clauses for the following FOPL formulas.

   a. $\exists x \forall y \exists z (P(x) \Rightarrow (Q(y) \Rightarrow R(z)))$.
   b. $\forall x \forall y ((P(x) \wedge Q(y)) \Rightarrow \exists z R(x, y, z))$.

10. Define the required predicates and represent the following sentences in FOPL.

    a. Some students opted Sanskrit in fall 2015.
    b. Every student who opts Sanskrit passes it.
    c. Only one student opted Tamil in fall 2015.
    d. The best score in Sanskrit is always higher than the best score in Tamil.
    e. There is a barber in a village who shaves every one in the village who does not shave himself / herself.
    f. A person born in country X, each of whose parents is a citizen of X or a resident of X, is also a resident of X.

11. Determine whether the expression p and q unify with each other in each of the following cases. If so, give the *mgu*, if not justify it. The lowercase letters are variables, and upper are predicate, functions, and literals.

    a. $p = f(x_1, g(x_2, x_3), x_2, b)$; $q = f(g(h(a, x_5), x_2), x_1, h(a, x_4), x_4)$.
    b. $p = f(x, f(u, x))$; $q = f(f(y, a), f(z, f(b, z)))$.
    c. $p = f(g(v), h(u, v))$; $q = f(w, j(x, y))$.

12. What can be the strategies for combination of clauses in resolution proof? For example, if there are N clauses, in how many ways they can be combined?
13. Why resolution based inference is more efficient compared modus-ponens?
14. Let Γ is knowledge base and α is inference from Γ. Give a comparison among the following inferences, in terms of their performances:

    a. Proof by Resolution, i.e., $\Gamma \vdash \alpha$,
    b. Proof by Modus poenes, i.e., $\Gamma \vdash \alpha$,
    c. Proof by Resolution Refutation, i.e., $\Gamma \cup \{\neg\alpha\} \vdash \phi$.

15. Given n number of clauses, draw a resolution proof tree to demonstrate combining them. Suggest any two strategies.
16. Given the knowledge base in clausal form, is it possible to extract answers from that making use of resolution principle? For example, finding an answer like, "Where is Tajmahal located?"
17. Represent the following set of statements in predicate logic, convert them to clause from, then apply the resolution proof to answer the question : Did Ranjana kill Lekhi?
    "Rajan owns a pat. Every pat owner is an animal lover. No animal lover ever kills an animal. Either Rajan or Ranjana killed a pat, called Lekhi."
18. Explain:
    a. Unification
    b. Skolemization
    c. Resolution principle versus resolution theorem proving.
19. Use resolution to show that the following set of clauses is unsatisfiable.
    $$\{p(a, z), \neg p(f(f(a)), a), \neg p(x, g(y)) \vee p(f(x), y)\}.$$
20. Derive $\bot$ from the following set of clauses using the resolution principle.
    $$\{p(a) \vee p(b), \neg p(a) \vee p(b), p(a) \vee \neg p(b), \neg p(a) \vee \neg p(b)\}.$$
21. Give resolution proofs for the inconsistency $\forall x\, shaves(Barber, x) \rightarrow \neg shaves(x, x)$, where *Barber* is a constant.
22. Consider ab locks-world described by facts and rules:

    Facts:

    $ontable(a), ontable(c), on(d, c), on(b, a), heavy(b),$
    $cleartop(e), cleartop(d), heavy(d), wooden(b), on(e, b).$

    Rules:

    All blocks with clear top are black.
    All wooden blocks are black.
    Every heavy and wooden block is big.
    Every big and black block is on a green block.

    Making use of resolution theorem find out the block that is on the green block.
23. Given the following knowledge base:

    If x is on top of y then y supports x.
    If x is above y and they are touching each other then x is on top of y.
    A phone is above a book.
    A phone is touching a book.

Translate the above knowledge base into clause form, and use resolution to show that the predicate "supports(book, phone)" is true.

24. How resolution can be used to show that a sentence is:
    a. Valid?
    b. Unsatisfiable?
25. "The application of resolution principle for theorem proving is a non-deterministic approach." justify this statement.
26. a. Use Herbrand's method to show that formula,

$$\forall x \; shaves(barber, x) \rightarrow \neg shaves(x, x)$$

    is unsatisfiable?
    b. What is Herband's universe for $S = \{P(a), \neg P(f(x)) \vee P(g(x))\}$?
27. Prove that $\forall x \neg p(x)$ and $\neg \exists x \, p(x)$ are equivalent statements.
28. Let S and T be unification problems. Also, let σ be a most general unifier for S and θ be a most general unifier for $\sigma(T)$. Show that $\theta\sigma$ is a most general unifier for $S \cup T$.
29. Write the axioms describing predicates: *grandchild, grandfather, grandmother, soninlaw, fatherinlaw, brother, daughter, aunt, uncle, brotherinlaw*, and *first-cousin*.
30. For each pair of atomic sentences in the following, find out the most general unifier.
    a. $knows(father(y), y)$ and $knows(x, x)$.
    b. $\{f(x, g(x)) = y, h(y) = h(v), v = f(g(z), w)\}$.
    c. $p(a, b, b)$ and $p(x, y, z)$.
    d. $q(y, g(a, b))$ and $q(g(x, x), y)$.
    e. $older(father(y), y)$ and $older(father(x), ram)$.
31. Explain what is wrong with the below given definition of set membership predicate $\in$:

$$\forall x, s : x \in \{x \mid s\}$$
$$\forall x, s : x \in s \Rightarrow \forall y : x \in \{y \mid s\}.$$

32. Consider the following riddle: "Brothers and sisters have I none, but that man's father is my father's son". Use the rules of kinship relations to show who that man is?
33. Let the following be a set of facts and rules:
    Rita, Sat, Bill, and Eden are the only members of a club.
    Rita is married to Sat.
    Bill is Eden's brother.
    Spouse of every married person in the club is also in the club.

a. Represent the above facts and rules using predicate logic.
b. Show that they do not conclude "Eden is not married."
c. Add some some more facts, and show that now the augmented set conclude that Eden is not married.

References

1. Chowdhary KR (2015) Fundamentals of discrete mathematical structures, 3rd edn. EEE, PHI India
2. Davis M, Putnam H (1960) A computing procedure for quantification theory. J ACM 7(3):201–215. https://doi.org/10.1145/321033.321034
3. Emden V, Kowalki RA (1976) The semantics of predicate logic as a programming language. J ACM 23(4):733–742
4. Av Gelder et al (1991) The well-founded semantics for general logic programs. J ACM 38(3):620–650
5. Luckham D, Nilsson NJ (1971) Extracting information from resolution trees. Artif Intell 2:27–54
6. Nilsson NJ (1980) Principles of artificial intelligence, 3rd edn. Narosa, New Delhi
7. Robinson JA (1963) Theorem-proving on the computer. J ACM 10(2):163–174
8. Robinson JA (1965) A machine-oriented logic, based on the resolution principle. J ACM 12(1):23–41
9. Stickel ME (1981) A unification algorithm for associative-commutative functions. J ACM 28(3):423–434