CSE6390 3.0 Special Topics in AI & Interactive Systems II
Introduction to Computational Linguistics
Insructor: Nick Cercone – 3050 CSEB – nick@cse.yorku.ca
Tuesdays,Thursdays 10:00-11:30 – South Ross 104
Fall Semester, 2010

Introduction to Logic and Resolution Principle Theorem Proving

Andrew, Bernard, Claude, Donald, and Eugene have summer houses along the Atlantic coast. Each wanted to name his house after the daughter of one of his friends - that is, Anne, Belle, Cecilia, Donna, and Eve (but not necessarily in that order). To be sure that their houses would have different names the friends met to make their choices together. Claude and Bernard both wanted to name their house Donna. They drew lots and Bernard won. Claude named his house Anne. Andrew named his house Belle. Eve's father hadn't come, and Eugene phoned to tell him to name his house Cecilia. Belle's father named his house Eve.

What is the name of each friend's daughter? What is the name of his house?

Well, we know that:

- Andrew's house is Belle.
- Bernard's house is Donna.
- Claude's house is Anne.

Their daughters are not so named. Claude cannot be Donna's father and Eugene cannot be Eve's father. Belle's father, who named his villa Eve, can only be Donald or Eugene. Similarly, Eve's father is Donald or Eugene. Since he phoned to the last one, he is Donald. His house is Cecilia. Eugene is Belle's father, Andrew is Donna's father, Bernard is Anne's father, and Claude is Cecilia's father. Thus we have:

Father	**Daughter**	**House**
Andrew	Donna	Belle
Bernard	Anne	Donna
Claude	Cecilia	Anne
Donald	Eve	Cecilia
Eugene	Belle	Eve

Isn't deduction wonderful! Imagine the reasoning strategies you must develop when the situation described above becomes many times more complicated. Think of an analogous situation of sorting numbers: bubble sort is intuitive, easy to program and computationally expensive, quick sort is less intuitive but very efficient. When reasoning of the sort described above becomes more complex, our ability to reason clearly becomes clouded by detail. What is called for are procedures which we know will work more efficiently but are perhaps seemingly more abstract initially.

1.0 Logic Systems

Propositional Calculus

The simpler of the two systems which have served as a framework for most mechanical theorem proving is called the *propositional calculus*. Extensively developed by Whitehead and Russell in their early 20th century classic Principia Mathematica, this system is also known as propositional logic, sentential calculus, and (informally) as "symbolic logic".

The basic entities, or primitives, in the propositional calculus are *propositions* (sentences) which are symbolized p, q, r, s, ... A proposition symbol stands for an assertion ("The sky is blue", "It is raining", "x=y") which may be true (T) or false (F). Propositions may be combined into more complex assertions by the use of operators, analogous to the familiar arithmetic operators of addition, multiplication, and so on. These *logical connectives*, however, combine propositions into logical expressions whose truth or falsity is a function of the truth value (T or F) of each component proposition. In general, logical connectives

map combinations of n propositions onto the set {T, F}. When n=1, the only mapping of interest reverses the truth variable of a proposition. We symbolize this *negation* operator with a minus sign and read -p (or ~p) as “not p”.

When n=2 (which is as high as we need to go) there are 16 possible binary logical connectives (comprising all the distinct ways truth values can be assigned to the four possible pairs of proposition values). Table 1 shows the mappings for the *conjunction* (p&q, “P and q”), *disjunction* (pvq, “p or q”), and *implication* (p→q, “p implies q”) operators.

p	**q**	**p&q**	**pvq**	**p→q**
T	T	T	T	T
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

Table 1: Three Binary Logical Connectives

It is not difficult to show that the set of connectives in Table 1 (together with negation) is *complete* in the sense that all other binary mappings can be expressed in terms of conjunction, disjunction, implication, and negation. In fact we can dispense with implication since

$$p \rightarrow q = -pvq \qquad \{1\}$$

and with either conjunction or disjunction since

$$p\&q = -(-pv-q),\ pvq = -(-p\&-q) \qquad \{2\}$$

There are even some other binary connectives which are complete in themselves. For ease of understanding, however, we usually work with all of the connectives in Table 1.

Although the simple expressions in equations {1} and {2} clearly “make sense” we require a formal means of determining whether more complex expressions are constructed properly. We thus introduce the following recursive definition of *well-formed-formulas* (“wffs”):

1. A proposition is a wff.
2. If A and B are wffs, then so are (-A), (A&B), (AvB), and (A→B).
3. There are no other wffs.

To avoid introducing parentheses around all sub expressions, as required by this definition, we can adopt a convention which specifies a *precedence hierarchy* for logical operators. Thus, unless parentheses dictate otherwise [as in the right hand sides of both equations in {2}], all negations will be evaluated first, followed by all conjunctions, then by all disjunctions, and finally by all implications. The various instances of a given operator can be processed in a left-to-right order.

A propositional calculus formula is not especially meaningful until a truth value has been assigned to each of its component propositions. Such an assignment is called an *interpretation* of the formula. For any given interpretation, we can use the formula evaluation rules to determine if the formula as a whole has the value true or false under that interpretation. A formula that is true under all possible interpretations is a *valid* formula or a *tautology*. A formula which is true under no interpretation is an *inconsistent* formula or a *contradiction*. A *consistent (satisfiable)* formula is true under at least one interpretation.

Two formulas (A, B) are *equivalent* (A=B) if they have identical truth values under all possible interpretations. Thus equations {1} and {2} show examples of equivalent formulas. It is often useful to convert a propositional formula into an equivalent but standardized form. One common such form is called the *conjunctive normal form* [CNF]. A CNF formula consists of conjunctions of subformulas, where each subformula is a disjunction of single propositions or their negations. Any formula can be converted to an equivalent CNF expression by application of relations like those in equations {1} and {2}, together with the commutative, associative, and distributive relations that hold for logical operators.

To illustrate conversion of a formula to conjunctive normal form we begin with

$$(p\&(q \rightarrow r)) \rightarrow s.$$

Replacing both implications by their equivalents [equation {1}] gives

$$-(p\&(-qvr))vs,$$

and then

$$(-pv(q\&-r))vs,$$

having also used the fact that -(-q)=q. Distributing the disjunction with -p through the conjunction gives

$$(-pvqvs)\rightarrow(-pv-rvs),$$

where associativity of disjunction allows removal of internal parentheses. The last expression is in conjunctive normal form, consisting of a conjunction of two disjunctions of propositions and their negations.

To prove a theorem in the propositional calculus, we typically try to show that one formula follows logically from a set of other formulas. Such demonstrations are carried out within a formal deductive apparatus containing *axioms* and *rules of inference*. Axioms are usually rather "obvious" formulas which are assumed valid without proof and used as a basis for proving other results. In any deductive system with given rules of inference, a good set of axioms should be *complete* (sufficient to prove all valid formulas in the system), *consistent* (not leading to proofs of contradictory results), and *minimal* (not containing any "extra" axioms which could be proved using the others). It should be noted that for systems more complex than the propositional calculus these properties are not always obtainable.

$$((-pvq)\rightarrow(-pv-r))v\ s.$$

In Principia Mathematica**,** Whitehead and Russell were able to prove a vast number of theorems using the following five axioms (not actually a minimal set, since the fourth was later shown to be redundant).

1. (pvp)→p
2. p→(qvp)
3. (pvq)→(qvp)
4. (pv(qvr))→qv(pvr))
5. (p→q)→((rvp)→(rvq))

The other component of a deductive system is the rule of inference. One or more such procedures are needed to generate new valid formulas from existing ones. Three rules of inferences were used in Principia Mathematica: (1) *substitution*, which allows replacement of all occurrences of a proposition in a formula by any given wff; (2) *replacement*, which allows any logical connective to be replaced by its definition in terms of other connectives [definitional equivalence, as in equation {1}, can be established by testing all interpretations]; and (3) *modus ponens* or *detachment*, which allows inference of B from both A and A→B.

We may now define a *theorem* more formally. A given (well-formed) formula A is a theorem if and only if there exists a finite sequence of (well-formed) formulas A1, A2, A3, ... such the last member of the sequence is A and each other member is an axiom, a previously proved theorem, or a derivation from the previous members by some rule of inference. The sequence of formulas is called a *proof* of A. We conclude this introduction to the propositional calculus with a short proof of a simple theorem, (p→-p)→-p.

1. (pvp)→p, by axiom 1 (above)

2. (-pv-p)→-p, by substitution of -p for p

3. (p→-p)→-p, by replacement of (-AvB) with (A→B)

Predicate Calculus

We now develop the basic ideas of the (first order) *predicate calculus*, the logic system in which most current theorem proving research is carried out. We have already done quite a bit of the groundwork, since the predicate calculus is an extension of the propositional calculus which allows us to deal with individuals, relations between individuals, and properties of individuals and sets of individuals. We continue to denote propositions by p, q, r, ... and to use the same set of unary and binary logical connectives.

To those structures we add *individual constants*, denoted a, b, c, ..., which symbolically identify particular items of the *domain of discourse*, D (e.g., people, numbers, days of the week). We use the last letters of

the alphabet (z, y, x, ...) to denote *individual variables* which may range over all the individuals in D. *Functions* of one or more variables and/or constants will be denoted f, g, h, ... and will map objects or groups of objects in D into other objects in D. Thus in the domain of numbers we might represent negation by g(x), addition by f(x,y), and three way multiplication by h(x,y,z), so that h(g(2), 6, f(3,1)) would denote -48. Any expression of this sort, which evaluates to an object or set of objects in D, is known as a *term*. Defined recursively, a term is (1) a constant, (2) a variable, or (3) a function of terms.

The predicate calculus gets its name from the entities used to describe or relate terms. These *predicates* are denoted P, Q, R, ... and map terms onto the truth values T and F. Thus, if D is people, P(a) might assert that individual a has red hair, while R(c,b) might claim that b is a sibling of c. Any predicate of terms (or simple proposition) in the predicate calculus is known as an *atomic formula*.

The last group of predicate calculus entities consists of the *quantifiers*, of which there are just two. The *universal* quantifier, denoted (x) and read "for all x", when applied to a formula asserts that the formula is true for all possible substitution instances of the variable x (the entire domain D). The *existential* quantifier, denoted ($\exists$x) and read "there exists an x", asserts that the formula is true for at least one of the possible values of x. In general a quantifier does not apply to all occurrences of its variable in a formula but only to those which fall within its range or *scope* (delimited if necessary by appropriate parentheses). Such variables are said to be *bound* by the quantifier while other occurrences of the same variable may be *free* of quantification.

We can now define recursively the well-formed-formulas (wffs) of the predicate calculus, as follows:

1. *Any atomic formula is a wff.*
2. *If A and B are wffs then so are (-A), (A&B), (AvB), and (A→B).*
3. *If A is a wff and x is a (free) variable in A, then ((x)A) and (($\exists$x)A) are wffs.*
4. *There are no other wffs.*

As for the propositional calculus, a precedence hierarchy allows omission of many parentheses. To the rules given previously we need add only that quantifiers are to be evaluated first, along with negations. Thus the scope of (x) in (x)-P(x)vQ(x) is just -P(x); the x in Q(x) is a free variable.

Interpretation of predicate calculus formulas requires specification of the domain, D, an assignment of elements of D to individual constants, and assignments of "meanings" (mappings) with respect to D to all functions and predicates. For example, if D is the positive integers, F denotes equality, f is the addition function, and the constants a and b are 3 and 5 respectively, then the formula ($\exists$x)F(f(x,a),b) asserts that there is an x such that x+3=5. The formula happens to be true under the given interpretation. But just as for the propositional calculus, predicate calculus formulas are classed as valid (true for all interpretations), satisfiable (true for at least one interpretation) and inconsistent (true for no interpretations). Also as before, two predicate formulas are equivalent if and only if they have identical truth values under all interpretations.

A useful type of formula equivalent to any predicate calculus formula is its *prenex normal form*. In this form all quantifiers have been "swept" to the front of the formula, so that each of them has all the rest of the formula (called the *matrix*) as its scope. The most awkward aspect of converting formulas to prenex normal form can be "moving negation through quantifiers" where the following (sensible) equivalences apply:

$$-(x)A=(\exists x)-A,\ -(\exists x)A=(x)-A \qquad \{3\}$$

Since conversion to prenex normal form is an implicit step in preparing formulas for resolution theorem proving, we will illustrate the method next.

The notions of axiom, rule of inference, proof, and theorem carry over directly from the propositional calculus to the predicate calculus.

2.0 Clause Form

In 1965 the logician J. A. Robinson reported the development of a new inference rule for the predicate calculus. He also proved that his *resolution principle* was "sound" (producing only valid wffs) and "complete" (producing all valid wffs). While not especially convenient or intuitive for people, the resolution

principle is ideally suited to computer implementation and forms the basis for almost all current research in mechanical theorem proving.

Without going into the complex logical basis for resolution based inference [the resolution principle is based on the proof procedure of Herbrand (1930)], we can understand the central idea underlying the method in the following terms. A proof that some formula W logically follows from a set of formulas S is equivalent to the claim that every interpretation satisfying S also satisfies W. If such is the case then no interpretation can satisfy the union of S and -W. Resolution theorem proving tries to show that union is unsatisfiable by deriving a special formula called the "null" clause or resolvent from it. The method is thus a special form of "proof by contradiction".

Before resolution theorem proving techniques can be applied to a theorem, certain preliminary steps must be executed. First, if the premises and conclusion to be proved are stated in English, they must be expressed in predicate calculus notation. We will illustrate this process later. Second, the conclusion to be proved must be negated. Third, all the formulas including the negated conclusion must be converted to what is known as *clause form*. A clause is a formula in prenex normal form with no quantifiers shown because existential quantifiers have been eliminated and all variables are assumed to be universally quantified. The matrix of a clause consists solely of disjunctions of atomic formulas and their negations, known collectively as *literals*. While conversion to clause form (from more general formulas or even directly from English statements) is usually quite easy, the general algorithm has eight steps.

We now consider these steps, illustrating the operations with the unusually complex formula

$$(x)[P(x)\rightarrow[(y)Q(x,y)\&-(y)(P(y)\rightarrow R(f(x,y)))]]. \qquad \{4\}$$

The Eight-Step Algorithm

Step 1: Eliminate Implication Signs - Using equation {1}, {4} becomes

$$(x)[-P(x)v[(y)Q(x,y)\&-(y)(-P(y)vR(f(x,y)))]]$$

Step 2: Reduce Scopes of Negation Signs - We then use equations {2} and {3} to reduce the scopes of negation signs to single predicates:

$$(x)[-P(x)v[(y)Q(x,y)\&(\exists y)(P(y)\&-R(f(x,y)))]]$$

Step 3: Standardize Variables - Now we rename quantified variables, if necessary, so that each quantifier has a unique variable:

$$(x)[-P(x)v[(y)Q(x,y)\&(\exists z)(P(z)\&-R(f(x,z)))]]$$

Step 4: Eliminate Existential Quantifiers - For all such quantifiers which do not fall within the scope of universal quantifiers we may simply replace $(\exists w)P(w)$ with $P(a)$ where 'a' is a constant whose "existence" the quantifier asserts. In a case like $(v)(\exists w)Q(w)$, however, there is some (possibly distinct) w for every v, so we must write $(v)Q(h(v))$ where h is a function that selects the w which exists for each v. Constants and functions introduced in this step must be new to the formula. [The functions introduced here are called Skolem functions]. Our example becomes:

$$(x)[-P(x)v[(y)Q(x,y)\&(P(g(x))\&-R(f(x,g(x))))]]$$

Step 5: Convert to Prenex Form - This conversion is accomplished by moving all (universal) quantifiers to the front of the formula:

$$(x)(y)[-P(x)v[Q(x,y)\&P(g(x))\&-R(f(x,g(x))))]]$$

Step 6: Put Matrix in Conjunctive Normal Form - Converting from prenex form to conjunctive normal form yields

$$(x)(y)[(-P(x)vQ(x,y))\&(-P(x)vP(g(x)))\&(-P(x)v-R(f(x,g(x))))]$$

Step 7: Eliminate Universal Quantifiers - Dropping the universal quantifiers (we assume that all variables at this point are universally quantified) leaves us

$$[(-P(x)vQ(x,y))\&(-P(x)vP(g(x)))\&(-P(x)v-R(f(x,g(x))))]$$

Step 8 :Eliminate & Signs - Eliminate the conjunctions by separating the formula into distinct clauses, each of which will be a disjunction of literals:

-P(x)vQ(x) -P(x)vP(g(x)) -P(x)v-R(f(x,g(x)))

3.0 Unification Algorithm & Resolution Principle

Given a set of clauses derived from the premises and negated conclusion of a theorem, the resolution principle generates new clauses by *resolving* pairs of clauses in the set. These new clauses are added to the set and may be used in the generation of further *resolvents*. It can be shown, although we will not do so, that if the original set of clauses is unsatisfiable (the theorem is provable) resolution will eventually produce a clause containing no literals, the so-called *null resolvent*.

To produce a resolvent of two available clauses we require that at least one atomic formula appear with opposite signs in the two "parent" clauses. The resolvent then consists of a disjunction of all other literals in both parent clauses, after removal of the literal(s) differing only in sign. Thus from the clauses -P(x)vR(x) and -R(x)v Q(x) we may infer the resolvent -P(x)vQ(x) by combining the literals left after removing R(x) and -R(x). This simple example actually provides a rare demonstration of the intuitive plausibility of the resolution principle. For if we write the clauses in implication form, we are inferring P(x)?Q(x) from P(x)→R(x) and R(x)→Q(x). With more literals in each clause (and the possibility of more than one pair dropping out), it is usually much less apparent why resolvents are reasonable inferences.

Another artificially simple feature of the above example was the "nice" coincidental appearance of R(x) and -R(x) in just those forms in the parent clauses. Usually it is necessary to perform one or more *substitutions* in the parent clauses as a first stage in the resolution process. The process of finding suitable substitutions is properly termed *unification*. If a set of clauses can be unified (i.e., can produce resolvents), a procedure called the *unification algorithm* can be used to find the simplest substitution (or "most general unifier") that does the job. The details of unification are given now.

The terms of a literal can be variable letters, constant letters, or expressions consisting of function letters and terms. A *substitution instance* of a literal is obtained by substituting terms for variables in the literal. Thus four instances of P(x,f(y),b) are

P(z,f(w),b)
P(x,f(a),b)
P(g(z),f(a),b)
P(c,f(a),b)

The first instance is called an *alphabetic variant* of the original literal because we have merely substituted different variables for the variables appearing in P(x,f(y),b). The last of the four instances mentioned above is called a *ground instance* or *atom* since none of the terms in the literal contains variables.

In general, we can represent any substitution by a set of ordered pairs $\theta = \{(t_1,v_1), (t_2,v_2),..., (t_n,v_n)\}$. The pair (t_i,v_i) means that the term t_i is substituted for variable v_i throughout. We insist that a substitution be such that each occurrence of a variable have the same term substituted for it; that is $i \neq j$ implies $v_i \neq v_j$, i,j=1,...,n. The substitutions used above in obtaining the four instances of P(x,f(y),b) are

$\alpha = \{(z,x), (w,y)\}$
$\beta = \{(a,y)\}$
$\gamma = \{(g(z),x), (a,y)\}$
$\delta = \{(c,x), (a,y)\}$

To denote a substitution instance of a literal P using a substitution θ, we write P:θ. Thus P(z,f(w),b) = P(x,f(y),b):α. The composition of two substitutions α and β is denoted by α|β and is the substitution obtained by applying β to the terms of α and then adding any pairs of β having variables not occurring among the variables of α. Thus

$$\{(g(x,y),z)\}\{(a,x),(b,y),(c,w),(d,z)\}=\{(g(a,b),z),(a,x),(b,y),(c,w)\}$$

It can be shown that applying α and β successively to a literal P is the same as applying α|β to P, that is, (P:α):β = P:α:β. It can also be shown that the composition of substitutions is associative:

$$(\alpha|\beta)|\gamma = \alpha|(\beta|\gamma)$$

If a substitution θ is applied to every member of a set $\{L_i\}$ of literals, we denote the set of substitution instances by $\{L_i\}:\theta$. We say that a set $\{L_i\}$ of literals is unifiable if there exists a substitution q such that $L_1:\theta = L_2:\theta = L_3:\theta =$ etc. In such a case θ is said to be a *unifier* of $\{L_i\}$ since its use collapses the set to a singleton. For example, θ = {(a,x), (b,y)} unifies {P(x,f(y),b), P(x,f(b),b)} to yield {P(a,f(b),b)}.

Although θ = {(a,x), (b,y)} is a unifier of the set {P(x,f(y),b), P(x,f(b),b)}, in some sense it is not the simplest unifier. We note that we really did not have to substitute "a" for "x" to achieve unification. The *most-general* (or simplest) *unifier* [mgu] λ of $\{L_i\}$ has the property that if θ is any unifier of $\{L_i\}$ yielding $\{L_i\}:\theta$, then there exists a substitution δ such that $\{L_i\}:\lambda|\delta = \{L_i\}:\theta$. Furthermore, the common instance produced by a most-general unifier is unique except for alphabetic variants.

There is an algorithm called the unification algorithm that produces a most-general unifier λ for any unifiable set $\{l_i\}$ of literals and reports failure when the set is not unifiable. The general idea of how the algorithm works can be described as follows: The algorithm starts with the empty substitution and constructs, in a step-by-step process, a most general unifier if one exists. Suppose at the kth step, the substitution so far produced is λ_k. If all the literals in the set $\{L_i\}$ become identical after employing the substitution λk on each of them then $\lambda = \lambda_k$ is a most-general unifier of $\{L_i\}$. Otherwise we regard each of the literals in $\{L_i\}$: $\lambda_k k$ as a string of symbols and detect the first symbol position in which not all of the literals have the same symbol. We then construct a *disagreement set* containing the well-formed expressions from each literal that begins with this symbol position. (A well-formed expression is either a term or a literal). Thus, the disagreement set of

{P(a,f(a,**g(z)**),h(x)),P(a,f(a,**u**),g(w))} is {g(z),u}

Now the algorithm attempts to modify the substitution λ_k in such a way as to make two elements of the disagreement set equal. This can be done only if the disagreement set contains a variable that can be set equal to one of its terms. (If the disagreement set contains no variables at all, $\{L_i\}$ cannot be unified. For example, we note that at the first step of the algorithm the disagreement set may be $\{L_i\}$ itself, and then certainly then no element is a variable).

Let s_k be any variable in the disagreement set and let t_k be a term (possibly another variable) in the disagreement set such that t_k does not contain s_k. (If no such t_k exists, then again $\{L_i\}$ is not unifiable). Next we create the modified substitution $\lambda_k+1 = \lambda_k\{(t_k,s_k)\}$ and perform another step of the algorithm.

It can be proven (Robinson, 1965) that the unification algorithm finds a most-general unifier of a set of unifiable literals and reports failure when the literals are not unifiable.

As examples, we list the most common substitution instances (those obtained by the mgu) for a few sets of literals.

Set Of Literals	**Most-general Common Substitution Instances**
{P(x), P(a)}	P(a)
{P(f(x),y,g(y)), P(f(x),z,g(x))}	P(f(x),x,g(x))
{P(f(x,g(a,y)),g(a,y)),P(f(x,z),z)}	P(f(x,g(a,y)),g(a,y))

It is customary to regard clauses as sets of literals. Thus a clause containing the set $\{L_i\}$ of literals can be denoted by $\{L_i\}$.

If a subset of the literals in a clause $\{L_i\}$ is unifiable by a mgu λ, then we call the clause $\{L_i\}:\lambda$ a *factor* of $\{L_i\}$. Some example factors of the clause P(f(x)) v P(x) v Q(a,f(u)) v Q(x,f(b)) v Q(z,w) are

P(f(z)) v P(z) v Q(a,f(u)) v Q(z,f(b))

and

P(f(a)) v P(a) v Q(a,f(b))

In the first factor we unified only the last two occurrences of Q, and in the second we unified all three. Note that the two occurrences of P cannot be unified within the clause. In general, a clause may have more than one factor, but certainly it can have only finitely many.

We now consider the legal substitutions that may be made in a pair of clauses without altering their truth values. In order to avoid confusion (and possible error) from coincidentally identical variable names,

substitution should be applied to clauses which have no variable names in common. If this is not already the case we simply *rename* some or all of the variables in one of the clauses. Now since all variables are understood to be universally quantified, each specifies any object in the domain. We can therefore substitute any new or existing variable name for *all* of the occurrences of any given name in order to bring literals in the clauses into closer correspondence.

We can also substitute any constant or function for all the instances of any variable in the two clauses, since such substitutions simply limit the range to one or more of the objects for which the variable stood. We cannot however make any substitutions which would change or increase the identified set of objects, since such substitutions could alter the truth value of the clause. Thus we may not substitute variables for functions or constants, nor may we replace any constant or function with any other constant or function.

To illustrate how substitution can be used in producing resolvents, we consider the two clauses

(1) -P(a) v Q(f(x),y,c) v R(y)
(2) S(x,y) v P(x) v -Q(y,b,c).

Renaming variables, by application of "primes" to variables in (2) which also happen to appear in (1), gives us

(2a) S(x',y') v P(x') v -Q(y',b,c).

Now we can substitute a for x' in (2a) producing

(2b) S(a,y') v P(a) v -Q(y',b,c).

which can be resolved with (1) to give

(3) Q(f(x),y,c) v R(y) v S(a,y') v -Q(y',b,c)

Alternatively we might substitute "b" for "y" in (1) and "f(x)" for "y'" in (2a), giving the different resolvent

(4) -P(a) v R(b) v S(x',f(x)) v P(x')

Thus different substitutions can give different resolvents. It should also be noted that (3) and (4) can be further resolved against the original formulas, with appropriate further substitutions.

The number of possible resolvents arising from even a small set of original clauses can obviously grow very rapidly. It would therefore be quite inefficient to try to prove a theorem by generating resolvents at random and waiting for the null clause to appear. For this reason, researchers have proposed and experimented with a large number of heuristic strategies for resolution theorem proving, in order to reduce the number of resolvents generated and direct the theorem proving program along promising lines. We mention just a couple of the simpler schemes. The unit preference strategy attempts to resolve clauses with as few literal as possible (ideally one of the parent clauses should be a single literal or *unit* clause). The *set-of-support* strategy tries to identify a set of "relevant" clauses and always include at least one member of this set as a parent in every resolution. For the purposes of this introductory discussion, the examples we work with are sufficiently elementary that no further consideration of strategies is required.

In addition to the potential explosion of resolvents, another difficulty with resolution theorem proving arises when the putative theorem is in fact invalid. In such a case, the resolution method may never terminate. We can thus never be sure whether a program has not terminated because the proof is difficult to find or because no proof exists. One partial solution to this problem is to interrupt the theorem proving attempt after a fixed amount of time and spend some time attempting to prove the negation of the theorem. If this fails, the original proof attempt can be taken up where it was left off, and the alternation continued. This alternation will lead to a proof or disproof if the putative theorem is valid or unsatisfiable but not if it is invalid and satisfiable. (Incidentally, this problem of not always being able to disprove invalid formulas is not limited to the resolution methods. The predicate calculus itself is termed *undecidable* because there does not exist an effective procedure for showing that any particular formula does not logically follow from a given set of formulas).

To conclude our discussion of resolution theorem proving we work through two examples, starting with an initial English statement of the theorem and ending with the null clause. First consider the following theorem: "If there are no compassionate professors, and if all competent professors are compassionate,

then no competent professor exists". If we let S(x) indicate that x is compassionate, and P(x) that x is competent, then the predicate calculus formulas for the premise are

(1) -(∃x)S(x)
(2) (y)(P(y)?S(y)),

while the denial of the conclusion is - -(∃z) (P(z)) or just

(3) (∃z)(P(z)).

(Note that we have avoided duplication of variable names to reduce the necessity for renaming prior to substitution.) In clause form,

(1') -S(x)
(2') -P(y) v S(y)
(3') P(a)

We can also substitute any constant or function for all the instances of any variable in the two clauses, since such substitutions simply limit the range to one or more of the objects for which the variable stood. We cannot however make any substitutions which would change or increase the identified set of objects, since such substitutions could alter the truth value of the clause. Thus we may not substitute variables for functions or constants, nor may we replace any constant or function with any other constant or function.

To illustrate how substitution can be used in producing resolvents, we consider the two clauses

(1) -P(a) v Q(f(x),y,c) v R(y)
(2) S(x,y) v P(x) v -Q(y,b,c).

Renaming variables, by application of "primes" to variables in (2), which also happen to appear in (1), gives us

(2a) S(x',y') v P(x') v -Q(y',b,c).

Now we can substitute a for x' in (2a) producing

(2b) S(a,y') v P(a) v -Q(y',b,c).

which can be resolved with (1) to give

(3) Q(f(x),y,c) v R(y) v S(a,y') v -Q(y',b,c)

Alternatively, we might substitute "b" for "y" in (1) and "f(x)" for "y'" in (2a), giving the different resolvent

(4) -P(a) v R(b) v S(x',f(x)) v P(x')

Thus different substitutions can give different resolvents. It should also be noted that (3) and (4) can be further resolved against the original formulas with appropriate further substitutions.

With substitution of x for y in (2'), resolution of (1') and (2') yields just -P(x). Substituting a for x in this resolvent and using (3') as the other parent yields the null resolvent, proving the theorem.

As a second example, we will prove the somewhat more complicated theorem: "A police officer questioned everyone who knew the victim and did not have an alibi. Some of the criminals knew the victim and were questioned only by criminals. No criminal had an alibi. Therefore, some of the police officers were criminals." Let P(x) mean that x is a police officer, Q(x,y) that x questioned y, K(x) that x knew the victim, A(x) that x had an alibi, and C(x) that x is a criminal. The first premise yields the following formula

(x)((K(x) & -A(x)) ? (∃y)(P(y) & Q(y,x))),

which can be readily shown to produce the two clauses:

(1) -K(x) v A(x) v P(f(x))

(2) -K(x) v A(x) v Q(f(x),x).

The second premise produces

(∃z)(C(z) & K(z) & (w)(Q(w,z) ? C(w))),

which yields the clauses

(3) C(a) (4) K(a) (5) -Q(w,a) v C(w).

The last premise is (v)(C(v) ? -A(v)) producing clause

(6) -C(v) v -A(v).

Finally the denial of the conclusion is -(∃x)(P(x) & C(x)) or, in clause form,

(7) -P(x) v -C(x).

The proof requires eight resolution steps, in which the substitutions should be apparent from the resulting resolvents.

(8)	-A(a)	from (3) and (6)
(9)	A(a) v P(f(a))	from (1) and (4)
(10)	P(f(a))	from (8) and (9)
(11)	A(a) v Q(f(a),a)	from (2) and (4)
(12)	Q(f(a),a)	from (8) and (11)
(13)	C(f(a))	from (5) and (12)
(14)	-P(f(a))	from (7) and (13)

Finally we resolve (10) and (14) to obtain the null clause.

3.1 Answer Extraction Process and Other Refinements

Consider the following trivially simple problem: "If Marcia goes wherever John goes, and John is at school, where is Marcia?" Quite clearly the problem specifies two "facts" and then asks a question whose answer can presumably be deduced from these facts. The facts might simply be translated into the set S of wffs

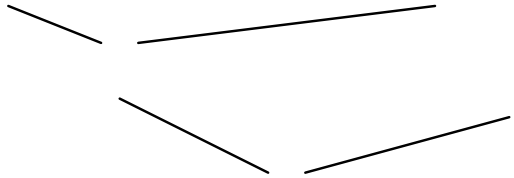
1. (x){AT(John,x) → AT(Marcia,x)} and 2. AT(John,school)

where the predicate letter AT is given the obvious interpretation. The question "where is Marcia?" could be answered if we could first prove that the wff

(∃x)AT(Marcia,x)

followed from S and could then find an instance of the x "that exists." The key idea here is to convert the question into a wff containing an existential quantifier such that the existentially quantified variable stands for an answer to the question. If the question can be answered from the facts given, the wff created in this manner will logically follow from S. After obtaining a proof, we then try to extract an instance of the existentially quantified variable to serve as an answer. In our example we will easily be able to prove that (∃x)AT(Marcia,x) follows from S. We can also show that a relatively simple process extracts the appropriate answer.

The proof is obtained in the usual manner by first negating the wff to be proved, adding this negation to the set S, converting all of the members of this enlarged set to clause form, and then, by resolution, showing that this set of clauses is unsatisfiable. A *refutation tree* for our example is shown in Figure 3-1. The wff to be proved is called the ***conjecture*** and the clauses resulting from the wffs in S are called *axioms*. Note that the negation of (∃x)AT(Marcia,x) produces (x)[-AT(Marcia,x)] whose clause form is simply -AT(Marcia,x).



Next, we must extract an answer to the question “Where is Marcia?” from this refutation tree. The process for doing so in this case is as follows:

1. *Append to each clause arising from the negation of the conjecture its own negation. Thus -AT(Marcia,x) becomes the tautology -AT(Marcia,x)vAT(Marcia,x).*
2. *Following the structure of the refutation tree, perform the same resolution as before until some clause is obtained at the root. In our example this process produces the proof tree shown in Figure 3-2 with the clause AT(Marcia,school) at the root.*
3. *Convert the clause at the root back to the conventional predicate calculus form and use it as an answer statement. This wff can then be translated back into English, say, as an answer to the question. In our example it is obvious that AT(Marcia,school) is the appropriate answer to the problem*

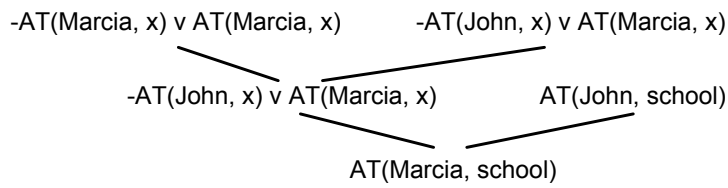


Figure 3-2. Modified Proof Tree for example problem.

We note that the answer statement has a form similar to that of the conjecture. In this case, the only difference is that in place of the existentially quantified variable in the conjecture, we have a constant (the answer) in the answer statement.

Before discussing the applications of theorem proving further, we must deal more thoroughly with the answer extraction process, particularly when the conjecture contains universal as well as existential quantifiers

Answer extraction involves converting a refutation graph (with nil at the root) to a proof graph having some statement at the root that can be used as an answer. Since the conversion involves converting every clause arising from the negation of the conjecture into a tautology, the converted proof graph is a proof that the statement at the root logically follows from the axioms plus tautologies. Hence it also follows from the axioms alone. Thus the converted proof graph itself justifies the extraction process

Although the method is simple, there are some fine points that can be clarified by considering some additional examples.

Consider the following set of wffs:

1. $(x)(y)\{P(x,y) \& P(y,z) \rightarrow G(x,z)\}$

and

2. $(y)(\square x)\{P(x,y)\}$

We might interpret these as follows:

“For all x and y if x is the parent of y and y is the parent of z, then x is the grandparent of z.”

and

“Everyone has a parent”.

Given these wffs as hypotheses, suppose we wanted to ask the question “Do there exist individuals x and y such that G(x,y)?” (That is, are there x and y such that x is the grandparent of y?)

We pose the question as a conjecture to be proved:

$$(\square x)(\square y)G(x,y)$$

The conjecture is easily proved by a resolution refutation showing the unsatisfiablility of the set of clauses obtained from the axioms and the negation of the conjecture. The refutation tree is shown in Figure 3-3.

We shall call the subset of literals in a clause that is unified during resolution the unification set.

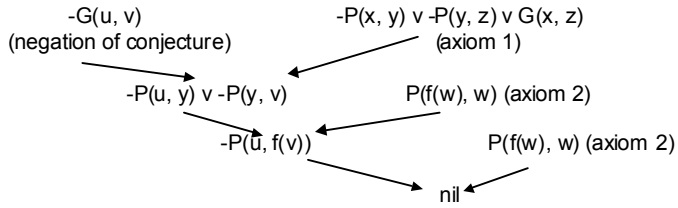


Figure 3-3. A refutation tree for example problem

Note that the clause P(f(w),w) contains a Skolem function f introduced to eliminate the existential quantifier in Axiom 2: (y)(□x){P(x,y)}. This function is defined so that (y)P(f(y),y). (The function f can be interpreted as a function that is defined to name the parent of any individual).

The modified proof tree is shown in Figure 3-4. The negation of the conjecture is transformed into a tautology, and the resolutions follow those performed on the tree of Figure 3-3. Each resolution in the modified tree uses unification sets that correspond precisely to the unification sets of the refutation tree.

The proof tree of Figure 3-4 has G(f(f(v)),v) at the root. This clause corresponds to the wff (v){G(f(f(v)),v)}, which is the answer statement. The answer provides a complete answer to the question "Are there x and y such that x is the grandparent of y?" The answer in this case involves a definitional function f: Any v and the parent of the parent of v are examples of individuals satisfying the conditions of the question. Again the answer statement has a form similar to that of the conjecture.

A problem arises when the conjecture to be proven contains universally quantified variables. The universally quantified variables become existentially quantified variables in the negation of the conjecture, causing Skolem functions to be introduced. What is to be the interpretation of these Skolem functions if they should eventually appear as terms in the answer statement?

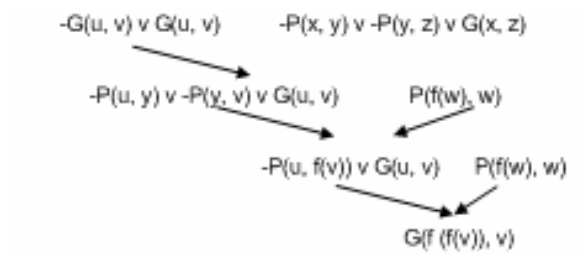


Figure 3.4 The modified proof tree for example problem

We shall illustrate this problem with another example. Let the clause form of the axioms be

* *C(x,p(x)) meaning "For all x, x is the child of p(x)". (That is, p is a function mapping a child of an individual into the individual).*

and

* *-C(x,y) v P(y,x), meaning "for all x and y, if x is the child of y, then y is the parent of x".*

Now suppose we wish to ask the question "For all x, who is the parent of x?" The conjecture corresponding to the question is

(x)(□y)P(y,x)

Converting the negation of this conjecture to clause form, we obtain

(□x)(y)-P(y,x)

and then

-P(y,a)

where 'a' is a Skolem function of no arguments (i.e. a constant) introduced to eliminate the existential quantifier occurring in the negation of the conjecture. (The negation of the conjecture alleges that some individual named a has no parent). A modified proof tree with answer statement at the root is shown in Figure 3-5.

Figure 3-5. A modified proof tree for an answer statement

Here we obtain the somewhat difficult-to-interpret answer statement P(p(a),a), containing the Skolem function a. The interpretation should be that regardless of the Skolem function a, hypothesized to “spoil” the validity of the conjecture, we have been able to prove P(p(a),a). that is, ***any*** individual a, thought to spoil the conjecture, actually satisfies the conjecture. The constant a could have been a variable without invalidating the proof shown in Figure 3-5.

It can be shown [Nilsson, 1971] that in the answer extracting process, it is correct to replace any Skolem functions in the clauses coming from the negation of the conjecture by new variables. These new variables will never be substituted for in the modified proof and will merely trickle down to occur in the final answer statement. Resolutions in the modified proof will still be limited to those defined by those unification sets corresponding to the unification sets occurring in the original refutation. Variables might be renamed during some resolutions so that, possibly, a variable used in place of a Skolem function may get renamed and thus might be the “ancestor” of several new variables in the final answer statement.

One final example will be shown for illustrative purposes. Suppose S consists of the single axiom (in clause form)

$$P(b,w,w) \vee P(a,u,u)$$

and we wish to prove the conjecture

$$(\square x)(z)(\square y)P(x,z,y)$$

A refutation tree is shown in Figure 3-6. Here, the clause coming from the negation of the conjecture contains the Skolem function g(x). In Figure3-7, we show the modified proof tree in which the variable 't' is used in place of the Skolem function g(x). Here we obtain a proof of the answer statement P(a,t,t) v P(b,z,z) that is identical (except for variable names) to the single axiom. This example illustrates how new variables introduced by renaming the variables in one clause during a resolution can finally appear in the answer statement.

Figure 3-6. A refutation proof tree.

Figure 3-7. Modified proof graph.

3.2 Refinements, Strategies, and Heuristics

Strategies and Refinements

The unrestricted application of the resolution principle will generate far too many clauses for practical computation. Consider Figure 3-8 below.

In Figure 3-8, only one deduction is shown. We could have other deductions spreading out in other directions from the original clauses. Some of the deductions will terminate with “nil” and hence lead to valid proofs of unsatisfiablility; others will not and, in fact, may never terminate. If resolution is to be made practical we need a guide to tell us which resolutions to make and which deductions to pursue.

An algorithm that selectively chooses resolutions is known as a *refinement strategy*. A rule for changing the order in which resolutions are attempted but not restricting their number will be called an 'ordering strategy'. Refinement strategies generally fall into three categories: syntactic, semantic, and ancestry strategies.

Figure 3-8. Tree of deductions.

Refinement strategies are guides for writing theorem-proving programs that compute only a restricted set of the set of all possible resolutions. To show that the resulting program is a valid theorem proving system it is necessary to show that the completeness theorem for resolution holds for the limited set of resolutions permitted by the refinement. Some refinement strategies are quite sophisticated and will be treated in separate sections. Virtually all of the ordering strategies are simple, however, and will be described here.

Unit Preference Strategy

If clauses C1 & C2, containing m and n literals, respectively, are resolved against each other, then the resolvent will contain at most (m-1)+(n-1) literals. If one of the clauses, say C1, contains only one literal, then that literal must be resolved upon and the resolvent will contain (n-1) literals. Since the goal of the program is to resolve a clause with no literals, this is a step on the right direction. The unit preference strategy requires that each step all resolutions involving clauses with only one literal (unit clauses or singletons) be computed prior to computing any other resolutions. This technique is generalized to computing resolutions involving shorter clauses rather than longer clauses.

Unit preference is an example of a syntactic ordering strategy.

Tautology and Unique Literal Elimination

Tautology and unique literal elimination are refinement strategies; their goal is the removal of irrelevant clauses from S before resolution is begun. Let S*, a subset of S be an unsatisfiable set of clauses such that every proper subset of S* is satisfiable, i.e., in order to obtain a contradiction in S*, it is necessary to use every clause in S* at least once in the proof. In general, it is more efficient to work with S* instead of S. Therefore it would be desirable to locate and remove the clauses, if any, in S-S* before beginning resolution.

A clause C is a tautology if it contains two complementary literals, for example A and -A. Since any clause contains a disjunction of literals, clauses containing tautologies may be eliminated since they cannot possibly be false for any interpretation (e.g., Av-A). Thus the unsatisfiability of S containing tautology C must be determined by the unsatisfiability of the set of clauses S - {C}.

A clause C can be dropped from S, even though it is not a tautology, if it is the only clause containing a

'unique literal', L, whose predicate, P, does not appear negated in any literal of another clause C* member of S. The reason is that no resolution involving C or any clause descended from C can ever result in the null clause. To see this suppose S is the set

C1: P(x) v -P(b) v Q(y)
C2: -P(c) v P(y)
C3: -P(b) v -P(x)

Any sequence of resolutions containing C1 must produce a clause, which contains the literal Q(y) or some instantiation of it. Because no clause in S contains an instance of the clause -Q(x), there is no way or removing Q(y) or any of its instantiations. Therefore, one might as well never begin such a sequence. The general principle is that if C contains a unique literal, then if S is unsatisfiable, the null clause can be deduced from S-{C}.

Factoring

Factoring, as discussed earlier, may reduce the length of clauses by application of an instantiation, which reduces several literals within a clause to the same literal. To illustrate, the clause

C: A(x,f(k)) v A(b,y) v A(a,f(x)) v A(x,z)

can be factored by the substitution

θ = {(b,x), (f(k),y), (f(b),z)}

to produce

C:θ A(b,f(k)) v A(a,f(b)) v A(b,f(b))

C:θ is a *factor* of C. Furthermore, factors of a clause are not necessarily unique, for example, the substitution

π = {(a,x), (f(k),y), (f(a),z)}

yields the factor

C:π A(a,f(k)) v A(b,f(k)) v A(a,f(a))

Since a clause implies its factors, S may be augmented by its factors of C. Although this increases the number of clauses of S, the added clauses will be shorter than the clauses that produced them, and may lead to shorter deductions of the null clause.

Subsumption

For any pair of clauses C, D members of S, C is said to subsume D if there is an instantiation of C, C:π, such that C:π is a subset of D. For example, if

C: A(x)

D: A(b) v P(x)

then the substitution π = {(b,x)} produces C:π A(b).

The validity of subsumption may be illustrated by an argument using the propositional calculus, in which the truth-value of a set of clauses is defined as the conjunction of the truth-value of the clauses it contains. Let C:π be (L) and D be (L,X), where X is a sequence of zero or more literals, L_1, L_2, ... Since the truth value of D is the disjunction of its literals,

D: L v X

It is elementary that

L subsumes L v X

regardless of the truth-value of X, so that C:π subsumes D. Define S*= S-{D}. Under the propositional calculus interpretation of a set, S* & C:π subsumes S.

If S is false under all assignments, then S subsumes the null clause. This means that S*&C:π subsumes the null clause, i.e., if S is truth functionally unsatisfiable, then the set S*?{C:π} is similarly unsatisfiable. It should be simpler to derive the null clause from S*→{C:π} than from S because both sets contain the same number of clauses, but C:π contains fewer literals than D.

Hyperresolution

Resolutions can be made involving several clauses at once, which is called *hyperresolution*. Suppose there exists a finite set of clauses, $\{C_1, C_2, ..., C_n\}$, and a single clause B which satisfy the following conditions:

2. *B contains the n literals L_1 ... L_n.*
3. *2.For every i, $1 \leq i \leq n$ clause C_i contains the literal $-L_i$ but does not contain the complement of any other literal which occurs in B, nor the complement of any literal which occurs in any clause C_j, $j \neq i$. The set of clauses Sa= $\{C_i\}$»{B} is called a clash. The clause R_a=(C_1 - $\{-L_i\}$, C_2-$\{-L_2\}$,...C_n - $\{-L_n\}$, B - $\{L_i\}$) is called the hyperresolvent of S_a, and may be deduced from S_a.*

In most cases we will obtain a clash only after appropriate substitutions. That is, we will be given a set S_a of clauses, which do not meet the definition of a clash, but there exists a substitution p such that S_a:π is a clash. In this case, S_a is called a latent clash. Since standard resolution involving only two clauses at a time is a special case of hyperresolution, it follows that if the null clause can be deduced from S by resolution, it can also be deduced from S, and perhaps more quickly, by hyperresolution.

As an example of hyperresolution, consider the set S_a defined by

S_a = { C1: -A(x)vP(a), C_2: P(y), C_3: -P(k)vQ(a,b), B: A(a)v-P(y)v-Q(x,y)vA(c) }

The substitutions π = {(a,x), (b,y)} produces

S_a:π = { C_1:π -A(a)vP(a), C_2:π P(b), C_3:π -P(k)vQ(a,b), B:π A(a)v-P(b)v-Q(a,b)vA(c) }

S_a:π is a clash with resolvent R_a: P(a) v -P(k) v A(c) and thus a is a latent clash,

Hyperresolution is an example of a semantic strategy. The reason is that the clash at S_a can only be satisfied by a model that contains some of the literals, and at least one from each clause of the clash, which are contained in the clash resolvent. Thus the clash resolvent points the way toward elimination of all models.

Ancestry Strategies

We can now consider some sophisticated strategies that permit savings in both the number of resolutions to be considered and the amount of computer storage required for recording clauses that have been inferred.

Set of Support

S can normally be divided into two subsets: the axioms of the system, $\{C_i\}$, and the negation of the conjecture, -T = $\{T_i\}$. It is usually reasonable to assume that the axioms are consistent, i.e., that the set $\{C_i\}$ is satisfiable. If this is the case, and if S is unsatisfiable, then the null clause can be deduced by a sequence of resolutions in which each step involves a resolution in which at least one of the clauses is in the set of support, where the *set of support* is defined as

1.All clauses in $\{T_i\}$ are in the set of support.

2.A resolvent clause is in the set of support if at least one of its parents is in the set of support.

The practical effect of the set of support strategy is that we need never to consider any resolution involving only clauses in $\{C_i\}$. This is useful since $\{C_i\}$ is often large in relation to $\{T_i\}$.

Linear Deductions

The linear deduction strategy is a quite restrictive method, which has the additional advantage of providing proofs that are quite easy to follow. Let C'_0 be a clause in a minimally satisfiable set S, and let C'_i, i>0, be the ith clause derived in a sequence of resolutions beginning with C'_0 and some other clause in S*. If the ith clause (i>0) of the deduction always has as one of its parents (called the left parent) the i-1st*

clause in the deduction, then the deduction is a linear deduction. In a linear strategy only resolutions permissible in a linear deduction are considered. The technique is illustrated by the deduction of the null clause from the set of ground clauses

S = { Q PvR -PvS -RvS -Pv-Qv-S -Qv-Rv-S }

One linear deduction of the null clause from S is

-Pv-Qv-S	resolves Q	to -Pv-S
-Pv-S	resolves PvR	to Rv-S
Rv-S	resolves –PvS	to -PvR
-PvR	resolves PvR	to R
R	resolves –RvS	to S
S	resolves -Q v -Rv-S	to -Qv-R
-Qv-R	resolves Q	to -R
-R	resolves R	to nil.

Lemma Proofs [need to fix from here on]

Lemma proofs are deductions, which do not satisfy the conditions of a linear deduction. They bring together two or more separate lines of a deduction as shown in Figure 3-9. [The proof is derived from the set S as given in the last section on linear deductions].

-P v Q v -S resolves P v R -P v S resolves P v R

-Q v R v -S resolves -Q v -R v -S R v S resolves -R v S

-Q v -S resolves Q

-S S

nil

Figure 3-9. Structure of a lemma deduction.

Subsumption and Merge Conditions

Two further refinements, called the *subsumption* and merge conditions, may be applied jointly with linearity to further restrict both the number of resolutions to be considered and the number of clauses that must be retained in memory. The combination of all three strategies is called the *merge, subsumption, linear (m.s.l.)* refinement strategy.

The subsumption condition restricts the number of resolutions that will be attempted at each round of inference. Let C'i be the clause just deduced, and let R be the set of previously resolved clauses that have been retained for use as possible right parents in a linear deduction. Thus R includes {C'j} for j<i. The left parent of C'i+1 will, of course, be C'i. The right parent, B'i, must be chosen from the set {S?R}. The subsumption condition states that if S is unsatisfiable, there exists a linear deduction of the null clause in which C'i+1 subsumes C'iin all cases in which B'i is chosen from R rather than S. Therefore, the search for candidate right parent clauses can be confined to S and some of the clauses of R. Specifically, if the subsumption condition is to be satisfied, B'i must contain only literals that are unifiable with literals in C'i, with the exception of exactly one literal, which will be the literal resolved upon, and must be unifiable with a complementary literal in C'i. Those clauses in R, which do not fulfill this condition, need not be considered as candidates for resolution.

The subsumption condition then restricts attention to a subset of R, given C'i. The *merge* condition provides a way of limiting the number of derived clauses to be placed in R in the first place. A clause C'i is a 'merge' of clauses C'i-1 and B'i-1 if there is a literal L, other than the literal resolved upon, which occurs in C'i and is an instantiation of the two literals, L1 and L2, which appear in C'i-1 and B'i-1, respectively.

The literal L is called a *merge literal*. A *merge clause* is either the clause C'i or a factor of C'i. An example of a merge resolvent is

C'i-1: Q(k,b) v -Q(a,c) v Q(k,c) B'i-1: Q(k,b) v Q(a,c) v Q(b,c)

C'i: Q(k,b) v Q(k,c) v Q(b,c)

in which Q(k,b) is a merge literal. Although the notation used to present merging refers to linear deduction, the idea of merging may equally apply to lemma structure deductions.

A-ordering and C-ordering

Two major syntactic refinement strategies are

More here

(General) Semantic Strategy

For a set of clauses S to be satisfiable means that there is some assignment of truth values to the atoms of S such that all clauses are satisfied. A particular assignment of truth-values to atoms is called a model. Semantic strategies seek to show that S is unsatisfiable by showing that there is no model which satisfies S. A relationship can be developed between model elimination and linear ancestry strategies. Nevertheless the idea of a semantic proof requires an expansion on the idea of a model first.

Much more here

Heuristics

There are a number of pragmatically useful heuristics that deliberately sacrifice completeness in the hope of obtaining a rapid solution most of the time. Most of them are simple and need only be mentioned. One, the use of analogies, requires more detail.

Much more here

References

Herbrand, J., (1930). "Recherches sur la theorie de la demonstration, Trav. Soc. Sci. Lettres Varsovie, Classe III Sci. Math. Phys., no 33.

Hunt, E. (1975). ARTIFICIAL INTELLIGENCE. Academic Press, New York.

Nilsson, Nils, (1971). Problem Solving Methods in Artificial Intelligence, McGraw Hill, New York.

Robinson, J. A., (1965). “A Machine Oriented Logic Based on the Resolution Principle”, CACM (12), 23-41.

Sampson, J. notes.