Using Rewriting Rules for Connection Graphs to Prove Theorems

C. L. Chang

IBM Research Laboratory, San Jose, CA 95193, U.S.A.

J. R. Slagle

Naval Research Laboratory, Washington, DC 20375, U.S.A.

ABSTRACT

Essentially, a connection graph is merely a data structure for a set of clauses indicating possible refutations. The graph itself is not an inference system. To use the graph, one has to introduce operations on the graph. In this paper, we shall describe a method to obtain rewriting rules from the graph, and then to show that these rewriting rules can be used to generate a refutation plan that may correspond to a large number of linear resolution refutations. Using this method, many redundant resolution steps can be avoided.

1. Introduction

The oldest technique used in mechanical theorem proving is Herbrand's method. Since this method requires to generate a very large number of ground clauses, it was abandoned in favor of two alternative approaches. One of these approaches is Robinson's resolution principle [21]. The other is to use the idea proposed by Prawitz [19, 20]. For the last decade, the resolution principle has got most of the attention of researchers, and has been playing a dominant role in the field. Many useful strategies [5] for resolution have been proposed. Recently, the attention seems to be shifted to Prawitz's idea. Compared to resolution, Prawitz's idea is relatively unexplored. There is some hope that his idea might be developed into an efficient theorem proving system.

Given a set S of clauses, Prawitz's idea is that, instead of generating the ground instances of clauses of S in some arbitrarily defined order, one should find by calculations the values that, when substituted for variables in S, give an unsatisfiable set of ground instances. Essentially, Prawitz's idea is based upon the observation that a set of clauses is unsatisfiable if and only if there is a set M of copies (variants) of clauses in S and a ground substitution θ such that $M\theta$ is truth-functionally unsatisfiable. We call θ a solution of M. This observation is actually Herbrand's theorem in a different form. Many methods [1, 3, 4, 8–10, 19, 20, 25] have been proposed to find such M and θ. In this paper, we shall propose another method for finding such M and θ. The approach we take is as follows:

(a) First, find a connection graph for a set S of clauses;

(b) Change the connection graph to a directed graph;

(c) from the directed graph, obtain a set of rewriting rules;

(d) use the rewriting rules to generate a refutation plan;

(e) Finally, use a unification algorithm to check whether the plan is acceptable or not. We note that the main difference between our method and the others is that we want to generate a plan, and then perform unification at *the last step* of a proof. In the sequel, we shall describe our method in detail.

2. Connection Graphs

The concept of a connection graph has been considered by many authors [1, 12, 13, 24, 25, 29]. Essentially, a connection graph is merely a data structure for a set of clauses indicating possible refutations. The graph itself is not an inference system. To use the graph, one has to introduce operations on the graph. For example, Andrews [1] and Shostak [24] define a criterion in terms of a connection graph for a truth-functionally unsatisfiable set of clauses; Kowalski [13] uses it for performing resolution; Sickel [25] uses it for graph-walking and graph-unrolling; and Yate et al. [29] use it for proving the completeness of linear resolution. Different operations on connection graphs lead to different theorem proving systems. In this paper, we shall use a connection graph to obtain rewriting rules. Then, finding a solution of M can be directly solved by using the rewriting rules.

In the sequel, we let S be a set of input clauses. By a copy of clause C in S, we mean that it is C itself, or a clause obtained from C by renaming variables in C. Let M be a set of zero or more copies of each clause in S. Without loss of generality, we assume that no two clauses in M have variables in common. A substitution θ is called a *solution* of M if $M\theta$ is truth-functionally unsatisfiable. A pair of literals $L1$ and $L2$ are called *potentially complementary* if $L1$ and $L2$ can be made complementary by applying some substitution, after renaming variables so that $L1$ and $L2$ share no variables.

A connection graph for a set S of clauses is constructed as follows: (This is called a clause interconnectivity graph by Sickel [25], p. 825).

(1) Exactly one copy of each clause of S is allowed to appear in the graph.

(2) For every clause, $L1 \vee \ldots \vee Lr$, in S, where $L1, \ldots, Lr$ are literals, it is represented in the graph as

FIG. 1.

(3) For every pair of potentially complementary literals, draw an edge connecting the literals. Label each such edge by a distinct name representing the most general unifier which makes the literals complementary, after variables in the literals are properly renamed. (In the subsequent section, a procedure will be given to rename variables.)

Example 1. Fig. 2 is a connection graph for the set,

$$\{P(x) \vee Q(y), \sim P(a), \sim Q(b)\}.$$

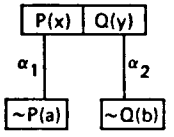


FIG. 2.

Example 2. Fig. 3 is a connection graph for the set,

$$\{\sim P(x) \vee P(f(x)), P(a), \sim P(f(f(a)))\}.$$

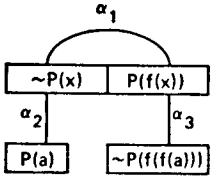


FIG. 3.

3. Combinations of Substitutions

In finding a solution of a set of clauses, it is often necessary to test whether or not we can combine substitutions. For example, $\{a/x\}$ and $\{b/y\}$ can be combined, while $\{a/x\}$ and $\{b/x\}$ can not be combined. For this purpose, we give the following definition and examples which are taken from [4] and Chapter 9 of [5].

Definition. Let $\theta_1 = \{t_{11}/v_{11}, \ldots, t_{1n_1}/v_{1n_1}\}, \ldots, \theta_r = \{t_{r_1}/v_{r_1}, \ldots, t_{rn_r}/v_{rn_r}\}$ be substitutions, $r \geq 2$. From $\theta_1, \ldots, \theta_r$ we define two expressions,

$$\text{E1} = (v_{11}, \ldots, v_{1n_1}, \ldots, v_{r1}, \ldots, v_{rn_r}), \text{ and}$$
$$\text{E2} = (t_{11}, \ldots, t_{1n_1}, \ldots, t_{r1}, \ldots, t_{rn_r}).$$

Then, $\theta_1, \ldots, \theta_r$ are said to be *consistent* if and only if E1 and E2 are unifiable. A most general unifier for {E1, E2}, denoted as $\theta_1 \cdots \theta_r$, is called a combination of $\theta_1, \ldots, \theta_r$. The substitutions $\theta_1, \ldots, \theta_r$ are said to be *inconsistent* if and only if they are not consistent.

Example 3. Consider $\theta_1 = \{a/x\}$ and $\theta_2 = \{f(a)/x\}$. For this case, we have E1 $= (x, x)$ and E2 $= (a, f(a))$. Since E1 and E2 are not unifiable, θ_1 and θ_2 are inconsistent.

Example 4. Let $\theta_1 = \{g(y)/x\}$ and $\theta_2 = \{f(x)/y\}$. For this case, we have E1 $= (x,y)$ and E2 $= (g(y), f(x))$. Since E1 and E2 are not unifiable, θ_1 and θ_2 are inconsistent.

Example 5. Let $\theta_1 = \{f(g(x1))/x3, f(x2)/x4\}$ and $\theta_2 = \{x4/x3, g(x1)/x2\}$. For this case, E1 $= (x3, x4, x3, x2)$ and E2 $= (f(g(x1)), f(x2), x4, g(x1))$. Since E1 and E2 are unifiable, θ_1 and θ_2 are consistent. The combination $\theta_1\theta_2$ is $\{f(g(x1))/x3, f(g(x1))/x4, g(x1)/x2\}$.

We note that the combination operation is associative and commutative, while the composition operation on substitutions is associative, but not commutative. In the sequel, if θ is a substitution and W is a set of substitutions, then θW is defined as

$$\theta W = \{\theta\lambda \mid \lambda \in W\}.$$

More generally, $W_1, \ldots, W_m$ are sets of substitutions, we define $W_1 \cdots W_m$ as

$$W_1 \cdots W_m = \{\lambda_1 \cdots \lambda_m \mid \lambda_1 \in W_1, \ldots, \lambda_m \in W_m\}.$$

The above concepts will be useful for giving a meaning to rewriting rules to be described in the next section. For simplicity, we shall write $\{\theta_1 \cdots \theta_n\}$, a set consisting of a combination, as $\theta_1 \cdots \theta_n$.

4. Obtaining rewriting Rules from a Connection Graph

In order to obtain rewriting rules, we first have to change an undirected connection graph into a directed one. This is done as follows:

Step 1. Choose a clause in the connection graph as a start clause. Every literal in the start clause will be labeled as a *goal* literal. (Note that a start clause is the same as a top clause in linear resolution [5].)

Step 2. For every goal literal L and every clause C in the graph, if there is an edge E connecting literal L and a literal L' of clause C, change edge E to a directed edge by pointing from literal L' to literal L. Label all the remaining literals in C as goal literals. Literal L' will be called a *premise* literal. (Note that edges within a clause are allowed.)

Step 3. Repeat Step (2) until every goal literal has been considered.

Example 6. Consider the connection graph shown in Fig. 4, which is taken from [25]. If we choose the clause consisting of literals 9 and 8 as a start clause, we obtain a directed connection graph shown in Fig. 5. However, if we choose the clause consisting of literals 1 and 2 as a start clause, we obtain a directed connection graph shown in Fig. 6. We note that edges α_1, α_2 and α_3 are bi-directional edges. This means that a literal can be a goal literal as well as a premise literal.

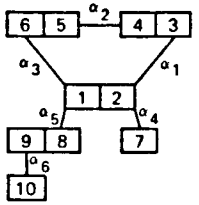


FIG. 4.

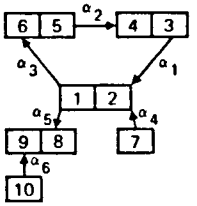


FIG. 5.

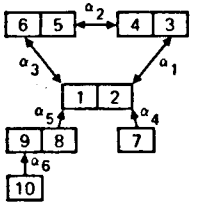


FIG. 6.

From a directed connection graph, we can obtain rewriting rules. In the following, if L is a literal in the connection graph, we shall use $W(L)$ to denote a set of substitutions obtained from proving L. Now, we obtain rewriting rules as follows:

(1) For each goal literal n, if $m_1, \ldots, m_r$ are all premise literals of n shown in Fig. 7, where $\alpha_1, \ldots, \alpha_r$ are substitutions, then we obtain a rewriting rule as

(R1) $\quad W(n) = \alpha_1 W(m_1) \cup \cdots \cup \alpha_r W(m_r)$,

where $\alpha_i W(m_i)$ is a combination of α_i and $W(m_i)$, $i = 1, \ldots, r$. The meaning of this rule is: If we know that $W(m_1), \ldots, W(m_r)$ are sets of substitutions obtained from proving literals $m_1, \ldots, m_r$, respectively, then a set of substitutions for proving literal n can be recursively described by the rule.

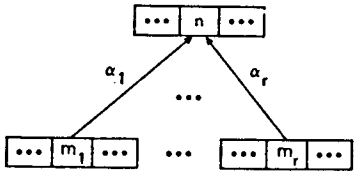


FIG. 7.

(2) For each clause of the graph shown in Fig. 8, where m_i is a premise literal, and $m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_r$, $r \geq 2$, are all the remaining literals in the clause, we obtain a rewriting rule as

(R2) $\quad W(m_i) = W(m_1) \cdots W(m_{i-1}) W(m_{i+1}) \cdots W(m_r)$.

The meaning of this rule is: If we want to use m_i as a premise literal, then all the other literals $m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_r$ have to be proved first. Suppose that $W(m_k), k = 1, \ldots, i-1, i+1, \ldots, r$, are sets of substitutions obtained from proving m_k independently. To make sure that the substitutions are consistent, we take a combination of them as shown in the rule.

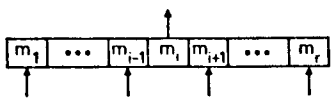


FIG. 8.

(3) For each unit clause, if literal m in the clause is used as a premise literal as shown in Fig. 9, we obtain a rewriting rule

(R3) $\quad W(m) = \{\varepsilon\}$,

where ε is the empty substitution. The meaning of this rule is: Since a unit clause by itself can be used as a premise literal, we do not need to check consistency.

FIG. 9.

(4) For a start clause as shown in Fig. 10, we obtain a rewriting rule as

(R4) $\quad T = W(m_1) \cdots W(m_r)$.

The meaning of this rule is: T is considered as a set of substitutions obtained for proving the start clause. Now, if $W(m_k)$, $k = 1, \ldots, r$ are sets of substitutions obtained independently from proving m_k, respectively, then a consistent combination of these substitutions is T, that is, a set of substitutions for proving the start clause. We note that the rewriting rules cast in a tree representation are also independently described in [25].

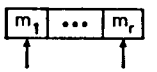


FIG. 10.

Example 7. Consider the directed connection graph shown in Fig. 5. From the edges, we obtain the rewriting rules

(1) $\quad W(9) = \alpha_6 W(10)$
(2) $\quad W(8) = \alpha_5 W(1)$
(3) $\quad W(6) = \alpha_3 W(1)$
(4) $\quad W(4) = \alpha_2 W(5)$
(5) $\quad W(2) = \alpha_1 W(3) \cup \alpha_4 W(7)$

From the clauses, we obtain the rewriting rules

(6) $\quad W(1) = W(2)$
(7) $\quad W(5) = W(6)$
(8) $\quad W(3) = W(4)$
(9) $\quad W(7) = \{\varepsilon\}$
(10) $\quad W(10) = \{\varepsilon\}$
(11) $\quad T = W(9)W(8)$

Rules (1) through (11) can be simplified as follows:

(12) $W(9) = \alpha_6 W(10) = \alpha_6\{\varepsilon\} = \sim$ from (1) and (10)

(13) $W(8) = \alpha_5 W(1)$ from (2)

(14) $T = W(9)W(8) = \alpha_6\alpha_5 W(1)$ from (11), (12) and (13)

(15) $W(1) = W(2)$ from (6)
$= \alpha_1 W(3) \cup \alpha_4 W(7)$ from (5)
$= \alpha_1 W(4) \cup \alpha_4\{\varepsilon\}$ from (8) and (9)
$= \alpha_1\alpha_2 W(5) \cup \alpha_4$ from (4)
$= \alpha_1\alpha_2 W(6) \cup \alpha_4$ from (7)
$= \alpha_1\alpha_2\alpha_3 W(1) \cup \alpha_4$ from (3)

Therefore, we have to keep only these two rules:

(16) $T = \alpha_6\alpha_5 W(1)$

(17) $W(1) = \alpha_1\alpha_2\alpha_3 W(1) \cup \alpha_4$

where α_1, α_2, α_3, α_4, α_5 and α_6 are treated as terminal symbols, and T and $W(1)$ as nonterminal symbols in a context-free grammar. (The non-terminal symbol T is the symbol for a sentence in the grammar.) Using these rules, we can generate the following sentences: $\alpha_6\alpha_5\alpha_4$, $\alpha_6\alpha_5\alpha_1\alpha_2\alpha_3\alpha_4$, ... Each of these sentences will be called a *plan*. In the next section, we shall describe how to check whether a plan is acceptable or not.

5. Renaming Variables and Performing Unifications

When we check the acceptability of a plan, we have to make sure that variables are properly renamed. That is, variables should be renamed (standardized apart) so that no two copies of clauses in the plan have variables in common. In the following, we shall give a method for standardizing apart variables in a plan. To do this, we shall label a clause by a distinct name and then refer to a literal in the clause by its position in the clause. That is, if C is a clause, then Cn is the nth literal (counting from the left) of clause C, where n is an integer. Once we have this naming scheme for literals, we can build a table for substitutions. For example, we redraw the connection graph of Fig. 4 into Fig. 11 by giving names to the clauses. From Fig. 11, we can build Table 1 for the substitutions $\alpha_1, \ldots, \alpha_6$ in the graph.

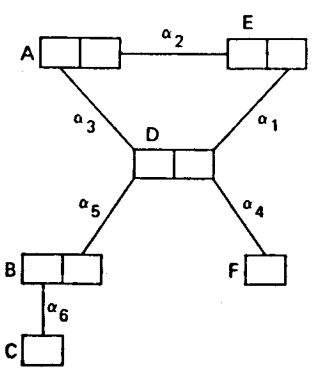


FIG. 11.

TABLE 1.

α_1	$D2$	$E2$
α_2	$A2$	$E1$
α_3	$A1$	$D1$
α_4	$D2$	$F1$
α_5	$D1$	$B2$
α_6	$B1$	$C1$

In Table 1, the first row (α_1, $D2$, $E2$) indicates that α_1 is the most general unifier that makes the 2nd literal of clause D (i.e., $D2$) and the 2nd literal of clause E (i.e., $E2$) complementary (after variables are renamed). The second row (α_2, $A2$, $E1$) indicates that α_2 is the most general unifier that makes the 2nd literal of clause A(i.e., $A2$) and the 1st literal of clause E (i.e., $E1$) complementary, and so on. Using Table 1, we can replace $\alpha_1, \ldots, \alpha_6$ in a plan by their corresponding pairs of literals. That is, we can replace α_1 by ($D2$, $E2$), α_2 by ($A2$, $E1$), and so on. For example, we know that $\alpha_6\alpha_5\alpha_4$ is a plan. This plan can be expressed as

$$(B1, C1) \quad (D1, B2) \quad (D2, F1)$$

by replacing α_6, α_5 and α_4 by ($B1$, $C1$), ($D1$, $B2$) and ($D2$, $F1$), respectively. We see that there are four clauses participating in the plan. That is, $C1$ belongs to a copy of clause C, $B1$ and $B2$ to a copy of B, $D1$ and $D2$ to a copy of D, and $F1$ to a copy of F. Once we identify which literal in the plan belongs to which copy of which clause, we then rename variables in these copies of the clauses so that they share no variables in common. After the variables are renamed, we then try to find the most general unifier θ such that $(C1, B1)\theta$, $(B2, D1)\theta$ and $(D2, F1)\theta$ are all complementary pairs of literals. If such θ can be found, then θ is a solution, and the plan is acceptable. We note that if $|L|$ denotes the atomic formula in a literal L, that is, if $|L|$ is obtained from L by deleting the negation sign (if any) from L, then θ can be obtained by unifying the two expressions, ($|B1|$, $|D1|$, $|D2|$) and ($|C1|$, $|B2|$, $|F1|$).

In general, if $(L1, M1)(L2, M2)\cdots(Lr, Mr)$ is a plan, a solution may be attempted by finding the most general unifier of two expressions,

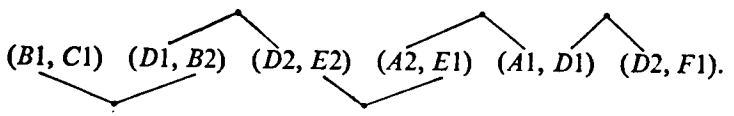
$$(|L1|, |L2|, \ldots, |Lr|) \text{ and } (|M1|, |M2|, \ldots, |Mr|).$$

The plan, $(B1, C1)(D1, B2)(D2, F1)$, considered above needs only one copy of each of clauses C, B, D and F. In general, if the same literal of a clause appears in a plan n times, then n copies of the clause may be required. For instance, in

Example 7, we know that $\alpha_6\alpha_5\alpha_1\alpha_2\alpha_3\alpha_4$ is also a plan. Expressing this plan by replacing the substitutions by the pairs of literals, we obtain

$$(B1, C1)\quad (D1, B2)\quad (D2, E2)\quad (A2, E1)\quad (A1, D1)\quad (D2, F1).$$

In this plan, since literal $D1$ occurs twice, two copies of clause D may be needed. Actually, we note that the first $D1$ and $D2$ encountered in the plan come from one copy of clause D, and $D1$ and $D2$ encountered afterwards come from another copy of clause D. Thus, the plan may look like



where literals belong to the same copy of a clause are linked together. Once we know where literals of each copy of a clause are, we can then standardize variables apart. In the following, we shall use the criterion given by Andrews [1] and Shostak [24] to link literals into copies of clauses.

In [1], Andrews defines that a connection graph is acceptable if it satisfies each of the following conditions:

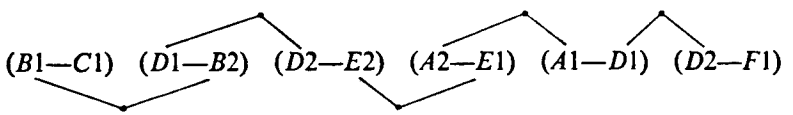
(a) Every literal in the connection graph is linked to a complementary literal:

(b) Every cycle contains a merge.

He and Shostak [24] independently prove the following theorem:

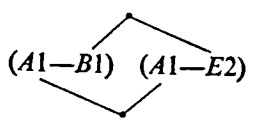
Theorem 1. *A set S of clauses is unsatisfiable iff there is an acceptable connection graph for a nonempty finite set of copies of clauses in S.*

To use the above theorem, we have to state it in terms of a plan, instead of a connection graph as follows: in a plan, for each pair of literals within a pair of parentheses, draw an edge. For all literals that belong to a copy of a clause, we draw edges from the literals to a common dot. For example, the above plan can be represented as



Let us call this a *linked plan.*

If we consider a literal or a dot as a node, clearly a linked plan can be considered as a graph. Therefore, we can conveniently talk about a cycle in a linked plan. We said that a cycle in a linked plan has a *merge* iff it contains a literal which appears in two pairs of literals in the cycle. For example, the following is a cycle



This cycle has a merge because literal $A1$ appears in the two pairs of literals, namely, $(A1—B1)$ and $(A1—B2)$.

Definition. A linked plan for a set S of clauses *is acceptable* iff it satisfies each of the following conditions:

(a) There is a substitution θ which simultaneously makes every pair of literals within a pair of parentheses complementary, after variables are standardized apart.

(b) Every cycle in the linked plan has a merge.

(c) Every bag of literals connected to a dot is a bag of all literals in one or more copies of a clause in S. For example, if $A1 \vee A2$ is a clause, the bag $(A1, A2, A1, A2)$ is such a bag, while the bag $(A1, A2, A1)$ is not. (Note that a bag is an unordered collection of elements, where the elements may be duplicated.)

Now, Theorem 1 can be stated as follows:

Theorem 2. A set of clauses *is unsatisfiable* iff there is an acceptable linked plan for S.

Example 8. Consider Fig. 12 and Table 2 for the substitutions in Fig. 12. Note that the clause $P(a)$ is the start clause. Now, from Fig. 12, we obtain the following rules:

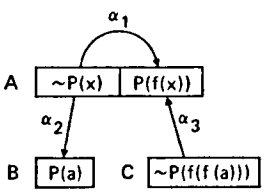


FIG. 12.

TABLE 2.

α_1	$A2$	$A1$
α_2	$B1$	$A1$
α_3	$A2$	$C1$

$$W(B1) = \alpha_2 W(A1) \tag{1}$$
$$W(A2) = \alpha_1 W(A1) \cup \alpha_3 W(C1) \tag{2}$$
$$W(A1) = W(A2) \tag{3}$$
$$W(C1) = \{\varepsilon\} \tag{4}$$
$$T = W(B1) \tag{5}$$

Then, a plan P can be generated as follows:

$$P = T$$
$$= W(B1) \qquad \text{from (5)}$$

$= \alpha_2 W(A1)$ from (1)
$= \alpha_2 W(A2)$ from (3)
$= \alpha_2\alpha_1 W(A1)$ from (2)
$= \alpha_2\alpha_1 W(A2)$ from (3)
$= \alpha_2\alpha_1\alpha_3 W(C1)$ from (2)
$= \alpha_2\alpha_1\alpha_3\{\varepsilon\}$ from (4)
$= \alpha_2\alpha_1\alpha_3$
$= (B1, A1)\quad (A2, A1)\quad (A2, C1)$ using Table 2.

From plan P, we can obtain the following linked plan

$$(B1\text{—}A1)\quad (A2\text{—}A1)\quad (A2\text{—}C1)$$

However, this linked plan is not acceptable because it has the following cycle which does not have a merge and violates Theorem 2:

$$(A2\text{—}A1)$$

Therefore, we obtain another linked plan from plan P as follows:

$$(B1\text{—}A1)\quad (A2\text{—}A1)\quad (A2\text{—}C1)$$

This linked plan does not have a cycle. Now, substituting actual literals for $A1$, $A2$, $B1$ and $C1$ in the above linked plan, we obtain

$$(P(a), \sim P(x))\quad (P(f(x)), \sim P(x))\quad (P(f(x)), \sim P(f(f(a)))).$$

Renaming the variable so that the different copies of the clause have no variables in common, we obtain

$$(P(a), \sim P(x))\quad (P(f(x)), \sim P(y))\quad (P(f(y)), \sim P(f(f(a)))).$$

From the above plan, a solution can be attempted by trying to unify the two expressions, $(P(a), P(f(x)), P(f(y)))$ and $(P(x), P(y), P(f(f(a))))$. Since $\{a/x, f(a)/y\}$ is the most general unifier of these two expressions, $\{a/x, f(a)/y\}$ is a solution for the set of the following clauses:

$$\begin{cases} P(a) \\ \sim P(x) \vee P(f(x)) \\ \sim P(y) \vee P(f(y)) \\ \sim P(f(f(a))). \end{cases}$$

6. Cyclic Rewriting Rules

In Section 4, we describe how to obtain a set of rewriting rules from a connection graph. We note that this set of rewriting rules represents a context-free grammar. We have shown that the rewriting rules can be used to generate sentences, and we call them (refutation) plans. However, in some cases, we may not be able to generate sentences, because we may not have rules which rewrite a non-terminal symbol (e.g., $W(n)$) into only terminal symbol (e.g., α_2). This is especially true when there is no unit clause in a set S of clauses. For example, consider the set of clauses, $S = \{P(x) \vee P(y), \sim P(u) \vee \sim P(v)\}$. Suppose we choose $P(x) \vee P(y)$ as the start clause. Then, we obtain a directed connection graph shown in Fig. 13.

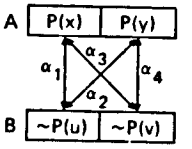


FIG. 13.

TABLE 3.

α_1	$A1$	$B1$
α_2	$A2$	$B1$
α_3	$A1$	$B2$
α_4	$A2$	$B2$

Table 3 is a table for substitutions. From Fig. 13, we can obtain the following rewriting rules:

(1) $W(A1) = \alpha_1 W(B1) \cup \alpha_3 W(B2)$,
(2) $W(B1) = \alpha_1 W(A1) \cup \alpha_2 W(A2)$,
(3) $W(A2) = \alpha_2 W(B1) \cup \alpha_4 W(B2)$,
(4) $W(B2) = \alpha_3 W(A1) \cup \alpha_4 W(A2)$,
(5) $W(A1) = W(A2)$,
(6) $W(B1) = W(B2)$,
(7) $T = W(A1)W(\mathrm{A2})$.

Now, we can obtain

(8) $W(A1) = \alpha_1 W(B1)$, from (1)
$= \alpha_1 W(B2)$, from (6)
$= \alpha_1\alpha_3 W(A1)$, from (4)
(9) $W(A2) = \alpha_2 W(B1)$, from (3)
$= \alpha_2 W(B2)$, from (6)
$= \alpha_2\alpha_4 W(A2)$. from (4)

Clearly, $W(A1)$ and $W(A2)$ can not be rewritten into strings of terminal symbols.

In cases like this, we modify rules as follows: If we can generate

$$W(n) = \alpha_1 \cdots \alpha_r W(n),$$

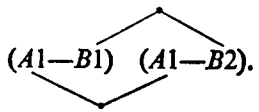
then we check whether or not $\alpha_1 \cdots \alpha_r$ corresponds to a cycle which has a merge. If yes, the rule is changed to

$$W(n) = \alpha_1 \cdots \alpha_r W(n) \cup \alpha_1 \cdots \alpha_r.$$

Now, consider Rule (8), that is,

$$W(A1) = \alpha_1 \alpha_3 W(A1).$$

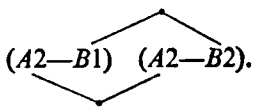
Using Table 3, we can express $\alpha_1 \alpha_3$ as



Clearly, the above graph is a cycle which has a merge $A1$. Therefore, Rule (8) is changed to

(8′) $W(A1) = \alpha_1 \alpha_3 W(A1) \cup \alpha_1 \alpha_3.$

Similarly, using Table 3, we can express $\alpha_2 \alpha_4$ as



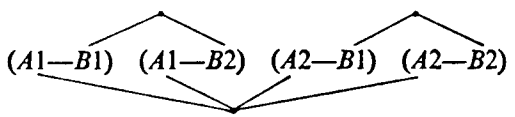
The above graph is a cycle which has a merge $A2$. Therefore, Rule (9) is changed to

(9′) $W(A2) = \alpha_2 \alpha_4 W(A2) \cup \alpha_2 \alpha_4.$

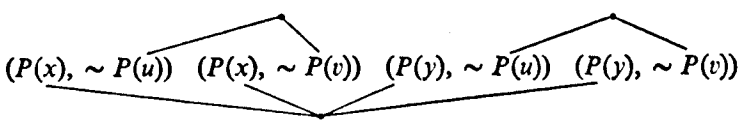
Now, we can generate a plan P as follows:

$$\begin{aligned} P &= T \\ &= W(A1)W(A2) && \text{from (7)} \\ &= \alpha_1 \alpha_3 \alpha_2 \alpha_4 && \text{from (8′) and (9′)} \end{aligned}$$

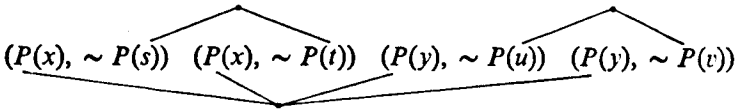
From P, we can obtain a linked plan



The above linked plan satisfies conditions (b) and (c) in the definition of an acceptable linked plan. To check condition (a), we substitute real literals for $A1$, $A2$, $B1$ and $B2$, and obtain



Renaming the variables, we obtain



Unifying the two expressions,

$(P(x), P(x), P(y), P(y))$ and
$(P(s), P(t), P(u), P(v))$,

we obtain a substitution

$$\theta = \{x/s, x/t, y/u, y/v\}.$$

This substitution is a solution of the set M of the following clauses,

$$\begin{cases} P(x) \vee P(y) \\ \sim P(s) \vee \sim P(t) \\ \sim P(u) \vee \sim P(v), \end{cases}$$

because $M\theta$ is

$$\begin{cases} P(x) \vee P(y) \\ \sim P(x) \\ \sim P(y), \end{cases}$$

and is truth-functionally unsatisfiable.

7. Relationship with Existing AI work

This section was suggested by Nilsson [17], and the material in 7.1 and 7.2. are mostly contributed by him. Some people may be more familiar with other work such as AND/OR tree type problem solving systems in artificial intelligence than they are with the resolution systems. Therefore, it would be a good idea to adapt or apply our method to the existing AI work.

7.1. Use of plans in AND/OR tree type problem solving systems

For illustration purposes, we shall consider only problem solving systems which use rules of the form

$$A1 \ \& \ A2 \ \& \cdots \& \ AN \rightarrow B,$$

where the Ai (the antecedents) and B (the consequent) are positive literals (with variables). Such rules are called *Horn* clauses. (An extension to non-Horn clauses is given in [16].) Given a goal $G1 \ \& \ G2 \ \& \cdots \& \ GM$, an AND/OR problem solving tree can be generated by splitting the Gi into AND nodes, unifying each with a rule consequent, and then splitting the (subgoal) antecedents, etc. A proof is obtained when all of the AND subgoals unify with elements of a set of fact literals say $F1, \ldots, FK$.

A simple example is shown below: Suppose we are given the following goal, rules and facts,

1) $D \& E$ goal
2) $F \& G \rightarrow D'$ rule
3) $I \& J \rightarrow E''$ rule
4) $H \rightarrow D''$ rule
5) E' fact
6) F' fact
7) G' fact
8) G'' fact
9) H' fact
10) I' fact
11) J' fact

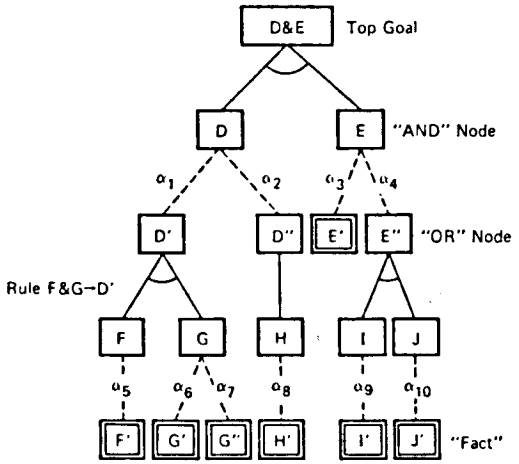


FIG. 14.

Assume only the following pairs of literals are unifiable: (D, D'), (D, D''), (E, E'), (F, F'), (G, G'), (G, G''), (H, H'), (I, I') and (J, J'). Then, starting with the goal $D \& E$, we can grow an AND/OR problem solving tree as shown in Fig. 14, where "boxed" nodes are facts, and dotted edges indicate unifiable pairs of literals. Now, to check for consistency of substitutions, we could read the following plans directly from Fig. 14:

1) $(D, D')(F, F')(G, G')(E, E')$
2) $(D, D')(F, F')(G, G'')(E, E')$
3) $(D, D'')(H, H')(E, E')$

etc.

Each plan corresponds to a possible solution tree for Fig. 14, and we must next choose a plan, rename variables in literals of the plan, and find a simultaneous unifier for all the pairs of literals in the plan.

Now, after seeing the above example, for rewriting rules R1, R2, R3 and R4 in Section 4, we can state the following analogy with AND/OR trees: R1 corresponds to OR node generation; R2 corresponds to AND node generation; R3 corresponds to "terminal nodes"; and R4 corresponds to the top AND node split.

7.2. Connection with PROLOG

The plan idea given in this paper may be also useful for PROLOG [27]. The PROLOG programming language involves an interpreter that essentially searches AND/OR trees of the type described above. The idea is this: A PROLOG program would first be converted into an AND/OR tree. Then plans would be computed. Then, these plans would be checked for consistency of substitutions. When a consistent one was found, it would be used on the "calling clauses" to produce the solution.

7.3. Applications in query transformation

A relational data base consists of a finite number of relations, each of which can be viewed as a table with a finite number of columns and rows. These relations which are explicitly stored are called *base relations*. Clearly, we can consider each row of a table a "fact" literal. Besides the base relations, there are *virtual relations* which are defined in terms of base and virtual relations. In [6, 7], virtual relations are defined by Horn clauses. For example, we may have a base relation, FATHER(x, y). Then, a virtual relation CHILD(y, x) can be defined by

$$\text{FATHER}(x, y) \rightarrow \text{CHILD}(y, x).$$

Once virtual relations are defined, a user can ask a query against base and virtual relations. In [6, 7], a query language called DEDUCE is proposed. Essentially, a DEDUCE query is similar to a formula in first order logic. A DEDUCE query may contain virtual relations. To evaluate the query, it has to be transformed into a query containing only base relations and then the transformed query is evaluated by a relational data base management system. In [7], it is shown that the rewriting rule method given in this paper can be used to transform DEDUCE queries. For detail, the reader is referred to [7].

7.4. Relationship with existing theorem proving techniques

Starting with Prawitz's idea [19, 20], we first use the concept of consistency of substitutions [4] and connection graphs [1, 12, 13, 24, 25, 29]. Then, we introduce rewriting rules, and finally come to the concept of plans. We think the plan idea is interesting, because it shows that we actually can generate a total plan and then perform unification at *the last step* for checking the consistency of substitutions. In [12], Klahr also uses plans based upon only predicate symbols. We shall now

try to relate our results to other theorem proving techniques in terms of the plan concept. For example, binary resolution involves a partial plan which uses only two literals, because only the two literals are unified. Hyper-resolution involves a partial plan using one or more pairs of literals. Davis' linked conjunct [8, 9] also involves partial plans. We think our total plan approach is better than partial plan approaches, because it eliminates redundancies. For example, suppose a solution involves the following total plan

$$(A, A') \quad (B, B') \quad (C, C').$$

Using our approach, we need only to check the consistency of substitutions for this plan. However, if we use a partial plan approach, first, we may generate the three partial plans. (A, A'), (B, B') and (C, C'). After checking the consistencies of these three partial plans, each of them may be combined with other pairs of literals to make other partial plans. For example, combining (A, A') with (B, B') and (C, C') respectively, we obtain the two partial plans,

$$(A, A')\,(B, B'),$$
$$(A, A')\,(C, C').$$

We can continue this process by testing consistencies of these partial plans and adding other pairs of literals to them. If we use the breadth-first method, eventually we may generate the following total plans:

$$(A, A')\,(B, B')\,(C, C'),$$
$$(A, A')\,(C, C')\,(B, B'),$$
$$(B, B')\,(A, A')\,(C, C'),$$
$$(B, B')\,(C, C')\,(A, A'),$$
$$(C, C')\,(A, A')\,(B, B'),$$
$$(C, C')\,(B, B')\,(A, A').$$

All of these plans are different only in their orderings of pairs of literals. For binary resolution, it is even worse, because even the ordering of literals within a pair of parentheses may make a difference. For example, some resolution strategies may consider these two partial plans (A, A') and (A', A) differently. In our total plan approach, we regard all these orderings as immaterial. Therefore, we think our approach is more efficient because it eliminates many redundant (total or partial) plans.

Of course, one may organize the generation of plans as a tree searching problem as shown in Section 7.1. As an AND/OR tree is growing, unification can be applied to partial plans so far generated. As soon as inconsistencies are detected, the partial plans could be eliminated. This tree growing method is similar to Sickel's unrolling technique [25] that checks for consistency. However, using inconsistency for pruning the tree can not eliminate the redundancies described above. That is, for the above example, no partial plans will be eliminated, because we assume that any combinations of (A, A'), (B, B') and (C, C') are consistent.

As we know, renaming of variables is a very important operation, because it is closely related to the number of copies of each clause needed in a proof. In this paper, based upon a variant of the theorem given by Andrews [1] and Shostak [24], we painstakingly describe a method for renaming variables in a plan. However, if we organize plan generation as tree growing, sometimes the variable-renaming operation can not be easily incorporated into the tree-growing process. For cases involving only Horn clauses as shown in Section 7.1, we may just use new variables for a rule (Horn clause) as soon as it is introduced for node expansion. However, for cases involving non-Horn clauses, it is not so easy, because we essentially generate an AND/OR graph, instead of an AND/OR tree. The reader may try to grow an AND/OR tree for the example in Section 6 just to convince himself that it is not obvious how variables should be *exactly* renamed. For this reason, we would prefer using rewriting rules. (A method based on formal grammars is also independently proposed in [26]. However, the concept and notation of plans, and explicit algorithms for performing the variable-renaming operation are not discussed.)

Finally, we think that the rewriting rule approach is better than the connection (AND/OR) graph approach [25], because it uses simplified rewriting rules (e.g., (16) and (17) in Example 7 of Section 4) and cyclic rewriting rules (see Section 6) to segment out the necessary subgraphs from a connection graph for possible uses in a proof.

8. Summary

We have given a method for proving theorems in first-order logic. Given a set S of clauses, our method can be described as follows:

(a) First, find a connection graph for S;

(b) Change the connection graph to a directed graph;

(c) From the directed graph, obtain a set of rewriting rules;

(d) Use the rewriting rules to generate a refutation plan;

(e) Check whether or not the plan is acceptable. If yes, we obtain a solution. Otherwise, go to Step (d) again to generate another plan.

To check a plan P, we use Theorem 2. That is, we first try to obtain a linked plan P^* from P such that every cycle (if any) in P^* has a merge; then we rename variables if necessary so that no copies of clauses share variables in common; finally, we perform unification as the last step. If a unifier can be obtained, then plan P is acceptable, and the unifier is a solution. Otherwise, it is not acceptable.

In reviewing the steps of our method, we see that Steps (a), (b) and (c) are straightforward. Step (e) is well-defined. Besides, many efficient unification algorithms [2, 11, 14, 15, 18, 22, 23] have been recently proposed. Some are in linear time and space. Therefore, Step (e) should be manageable. Step (d) is non-deterministic. In general, many plans can be generated. Which one is to be pre-

ferred, and how to organize the generation of plans must be studied in the future. Also, there is a question whether to separate Steps (d) and (e), or combine then One may argue that in Step (e) if a plan is found to be unacceptable, computational results for some partial plans should be saved because they may be useful for other plans. However, we would think that Step (e) may be so fast that it would be more advantageous to recompute than to save them. In addition, if we separate Steps (d) and (e), then parallel processing can be applied easily because plans are independently treated, and each of them can be processed independently.

REFERENCES

1. Andrews, P., Refutations by matings, *IEEE Transactions on Computers* C-25 (1976) 801–806.
2. Baxter, L. D., An efficient unification algorithm, University of Waterloo, Waterloo, Ontario, Canada (1973).
3. Chang, C. L., Theorem proving by generation of peudosemantic trees, Div. of Computer Research and Technology, National Institutes of Health, Bethesda, Maryland (1971).
4. Chang, C. L., Theorem proving with variable-constrained resolution, *Information Sci.* 4 (1972) 217–231.
5. Chang, C. L. and Lee, R. C. T., *Symbolic Logic and Mechanical Theorem Proving* (Academic Press, New York, 1973).
6. Chang, C. L., DEDUCE—A deductive query language for relational data bases, in: C. H. Chen (Ed.) *Pattern Recognition and Artificial Intelligence* (Academic Press, N.Y., 1976).
7. Chang, C. L., DEDUCE 2—Further investigations of deduction in relational data bases, IBM Research Report RJ2147, San José, California, 1978.
8. Chinlund, T. J., Davis, M., Hineman, P. G. and McIlroy, M. D., Theorem proving by matching, Bell Laboratory (1964).
9. Davis, M., Eliminating the irrelevant from mechanical proofs, *Proc. Symp. Appl. Math.* 15 (1963) 15–30.
10. Henschen, L. J. and Evangelist, W. M., Theorem proving by covering expressions, *Proc. 5th International Joint Conference on Artificial Intelligence*, MIT, Massachusetts (1977).
11. Huet, G., Algebraic aspects of unification, Presented at Automatic Theorem Proving Workshop, Oberwolfach, West Germany, 1976.
12. Klahr, P., Planning techniques for rule selection in deductive question-answering, in: Waterman and Hayes-Roth (eds.) *Pattern-Directed Inference Systems* (Academic Press, NY, 1977).
13. Kowalski, R., A proof procedure using connection graphs, *JACM* 22(4) (October 1975) 572–595.
14. Martelli, A. and Montanari, U., Unification in linear time and space: A structured presentation, Istituto Di Elaborazione Della Informazione, Consiglio Nazionale Delle Ricerche, Pisa, Italy (1976).
15. Martelli, A. and Montanari, U., Theorem proving with structure sharing and efficient unification, Istituto Di Scienze Dell'Informazione, Università Degli Studi Di Pisa, Italy (1977).
16. Nilsson, N. J., A production system for automatic deduction, Technical Note 148, Stanford Research Institute, Menolo Park, California, July 1977.
17. Nilsson, N. J. (1978): Private communications.
18. Patterson, M. S. and Wegman, M. N., Linear Unification, RC5904 (#25518), IBM Thomas J. Watson Research Center, Yorktown Heights, NY (1976).
19. Prawitz, D., An improved proof procedure, *Theoria* 26 (1960) 102–139.
20. Prawitz, D., Advances and problems in mechanical proof procedures, in: Meltzer, B. and Michie, D. (Eds.), *Machine Intelligence* 4 (American Elsevier, New York, 1969) 59–71.
21. Robinson, J. A., A Machine-oriented logic based on the resolution principle, *JACM* 12(1) (1965) 23–41.
22. Robinson, J. A., Computational logic: the unification computation, in: Meltzer, B. and Michie, D. (Eds.), *Machine Intelligence* (Edinburgh University Press, 1971) 6 63–72.
23. Robinson, J. A., Fast unification, Presented at Automatic Theorem Proving Workshop, Oberwolfach, West Germany, 1976.
24. Shostak, R., Refutation graphs, *Artificial Intelligence* 7 (1976) 51–64.
25. Sickel, S., A search technique for clause interconnectivity graphs, *IEEE Transactions on Computers* C-25 (1976) 823–834.
26. Sickel, S., Formal grammars as models of logic derivations, Proc. of IJCAI-77 (1977) 544–551.
27. Warren, D. H. and Pereira, L. M., ROLOG: The language and its implementation compared with LISP, Proc. ACM Symp. on Artificial Intelligence and Programming, University of Rochester, Rochester, NY, August 15–17, 1977 (1977) 109–115.
28. Yarmush, D. L., The linear conjunct and other algorithms for mechanical theorem proving, IMM 412, Courant Institute of Mathematic Sciences, New York University, New York, NY (July 1976).
29. Yates, R., Raphael, B. and Hart, T., Resolution graphs, *Artificial Intelligence* 1(4) (1970) 257–290.