THE LINEAR RESOLUTION THEOREM PROVER

by

NELSON HIN-FAI CHAN
B.Sc., University Of Western Ontario, 1982

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
Department Of Computer Science

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
October 1984

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1W5

Date: October 15, 1984

Abstract

A resolution-based theorem prover (LRTP) has been built on the PROLOG/MTS system. The LRTP is designed for studying the performance of three resolution strategies, namely, linear input resolution, linear resolution, and ordered linear deduction. It allows the user to perform experiments on the three strategies in combination with others. Furthermore, the user has control over the environment in which the theorem is proved. The number of unifications involved in the search for a proof is used as a measure of performance.

Table of Contents

List of Figures

Acknowledgement

I would like to thank Dr. R. Reiter for supervising my research and for his comments on this document. Thanks also to Dr. H. Abramson for reading and commenting on the final draft.

Chapter 1. Introduction

When an AI system has a complete description of the objects, properties and relations of a problem situation, then it can answer any question by evaluation. However, there are some queries that require the system to be able to deduce answers. For instance, suppose the system was told that "All birds have wings" and "All robins are birds". If we then ask it "Do robins have wings ?", the system can answer the query by means of deduction, without having to check whether each individual robin has wings or not.

It has been a central problem in AI research to make computers deduce from given bodies of facts. Any attempts to address this problem require choosing first, a representation for the given facts and secondly, methods for drawing conclusions. McCarthy and Hayes[1969] divided the AI problem into two parts: the epistemological part and the heuristic part. The former part was defined as determining "what kinds of facts about the world are available to an observer with given opportunities to observe, how these facts can be represented in the memory of a computer, and what rules permit legitimate conclusions to be drawn from these facts" [McCarthy,1977]. The latter part deals with the issue of processing, of using the knowledge once a representation scheme has been chosen.

Throughout this article, first-order logic is used as the representation scheme, whereas the resolution principle is used

as the rule of inference.

1.1 Historical Backgrounds Of Automatic Deduction

Automatic deduction, or mechanical theorem-proving, has been an important subject in AI since its earliest day. In fact, the desire to find a general decision procedure to prove theorems dates back to Leibniz(1646-1716); it was further revived by Peano and subsequently by Hilbert. The foundation of mechanical theorem-proving was established by Herbrand in 1930, however his method was impractical to apply until the invention of computers. Gilmore[1960] was one of the persons who attempted to implement Herbrand's procedure on a computer. A few months later, his method was improved by Davis and Putnam[1960], but their improvement was still insufficient.

A major breakthrough in the development of automatic deduction techniques was made by Robinson in 1965. In his landmark paper, he introduced a simple and logically complete method for proving theorems in first-order predicate calculus. This method is much more efficient than the earlier procedures. Since then, many refinements were proposed to further improve its performance. Robinson's procedure and those refined ones are called resolution procedures since all of them are based on the same rule of inference, the resolution principle.

The early research of automatic theorem-proving was considered as exercises in expert problem-solving: the Logic Theorist proposed by Newell and Simon[1956] was regarded as an expert in propositional logic and Gelernter's

theorem-prover[1963] an expert in elementary geometery. However, with the introduction of resolution, attitudes toward automatic deduction was dramatically changed. This is because the resolution method was so powerful that it could be used to build a completely domain-independent problem-solver.

1.2 Outline

This thesis involves implementing some refined proof procedures, derived from the resolution principle, to prove theorems in first-order logic.

Since first-order logic is used as the representation scheme for this thesis, Chapter 2 is devoted to the introduction of logic. It consists of two parts: the first part decribes propositional logic and the second part first-order logic (extension of propositional logic).

Chapter 3 introduces the resolution principle. Again, it has two major parts. The first part is concerned with the resolution principle for propositional logic. And then, the principle is extended to first-order logic in the second part.

Chapter 4 introduces the concept of refutation and how various resolution strategies can be applied in a refutation system.

Chapter 5 describes the organization of the theorem-prover (LRTP) being implemented, and then a comparison between the LRTP and the Prolog system follow.

Chapter 6 concludes what has been implemented in the LRTP and gives suggestion for further research.

Chapter 2. Logic

The study of reasoning and knowledge originates with the ancient Greeks. Following their early efforts, the study was formalized in the latter half of the nineteenth century and has since developed into the philosophical and mathematical study of logic.

2.1 Propositional Logic

Propositional logic is the simplest form of logic. A proposition is a declarative sentence that can be either true or false, but not both. For example, "The book is red" and "One plus one equals two" are propositions. For convenience, symbols can be used to denote propositions and are called atoms. We can combine simple propositions to form compound propositions by using five logical connectives. These connectives are: ¬(not), ∨(or), ∧(and), -> (if..then), and <-> (if and only if). The meanings of the five connectives are as follows :

If G and H are formulas,

¬G is true iff G is false,

G∨H is true iff either G or H is true or both are true,

G∧H is true iff both G and H are true,

G->H is true iff H is true or G is false,

G<->H is true iff both G and H are true or both are false.

Note that $(G \vee H)$ and $(G \wedge H)$ are respectively, called the disjunction and the conjunction of G and H. An expression that represents a proposition or a compound proposition is called a well-formed formula.

Definition. Well-formed formulas (wff), or formulas for short, are defined recursively as follows :

1. An atom is a formula (atomic formula).
2. If G and H are formulas, then $(\neg G)$, $(G \vee H)$, $(G \wedge H)$, (G->H), and (G<->H) are formulas.
3. All formulas are generated by applying the above rules.

If A1,A2,...,An are atoms occurring in a formula G, then an interpretation of G is an assignment of truth values to A1,..,An in which every Ai is assigned either T (truth) or F (falsehood), but not both. Once the interpretation is determined, the truth value of G can be evaluated. If it is evaluated to T, then G is said to be true under (or in) the interpretation, otherwise G is said to be false under the interpretation.

In mathematics as well as in daily life, we often have to decide whether a statement follows from some other statements. This leads to the concept of "logical consequence" which is defined as follows:

Definition. Given formulas F1,F2,..,Fn and a formula G, G is said to be a logical consequence of F1,...,Fn (or G

logically follows from F1,..,Fn) if and only if for any interpretation I in which F1∧F2∧.....∧Fn is true, G is also true. F1,F2,...,Fn are called axioms (or premises) of G.

Definition. If G is a logical consequence of F1,..,Fn, the formula ((F1∧...∧Fn)->G) is called a theorem , and G is also called the conclusion of the theorem. (G is sometimes referred to as a query in the context automatic deduction). A demostration that a theorem is true is called a proof of the theorem.

Logicians seem to be particularly interested in formulas which are true under all interpretations and those which are false under all interpretations. These two kinds of formulas are respectively called valid formulas and unsatisfiable (or inconsistent) formulas. We note that a formula is valid if and only if its negation is unsatisfiable.

In the resolution methods that will be discussed in later chapters, it is necessary to transform a wff to its equivalent conjunctive normal form which is defined as follows.

Definition. A literal is an atom or the negation of an atom.

Definition. A clause is a disjunction of literals. A clause is an unit clause if and only if it consists of

only one literal.

Definition. A formula is said to be in conjunctive normal form if and only if it has the form $L1 \wedge \ldots \wedge Ln$, $n \geq 1$, where each L1,..,Ln is a clause.

We have established the background for propositional logic and now we can extend it to first-order logic.

2.2 First-order Logic

First-order logic is an extension of propositional logic in the sense that the framework of propositional logic is still retained, but the notions of proposition is extended to include predicates and quantifiers.

A predicate is a statement about specific objects (or individuals) or relation between these objects. More precisely, it is a mapping that maps a list of constants to T or F. For example, LESS is a predicate. LESS(3,5) is T but LESS(5,3) is F. Arguments of predicates are called terms , which is defined recursively as follows :

1. A constant is a term.
2. A variable is a term.
3. If f is an n-place function symbol, and t1,..,tn are terms, then f(t1,..,tn) is a term.
4. All terms are generated by applying the above rules.

We call a term containing no variables a ground term. Throughout this article, we shall only use x, y, and z to represent variables.

Having defined terms, we can now define an atom in first-order logic as P(t1,..tn), where P is predicate symbol and t1,..,tn are terms. We can build up more complex formula using the five logical connectives given in Section 2.1. Furthermore, variables in a predicate are characterized by two quantifiers, namely the universal quantifier and the existential quantifier. If x is a variable, then (x) is read as "for all x" and (Ex) is read as "there exists an x". It should be noted that first-order logic permits only quantification over individuals but not predicates and functions.

The scope of a quantifier occurring in a formula is defined as the formula to which the quantifier applies. For example, the scope of the universal quantifier and existential quantifier in the formula (x) (R(x)->(Ey)Q(x,y)) are (R(x)->(Ey)Q(x,y)) and (Q(x,y)) respectively.

A formula W is said to be closed if and only if every occurrence of a variable x in W is in the scope of (x) or (Ex). Such an x is bound by the corresponding quantifier. A variable is free if and only if it is not bound by any quantifiers.

We shall now define an interpretation of a formula in first-order logic.

Definition. An interpretation of a formula W consists of

a nonempty domain D and the following assignments:

1. Assignment of an element in D to each constant in W.
2. Assignment of a mapping, from D^n to D, to each n-place function symbol in W.
3. Assignment of a mapping, from D^n to {T,F}, to each n-place predicate symbol in W.

Having defined interpretation for first-order logic, we can extend the rest of the definitions in Section 2.1 in a similar way.

We shall close this section by giving an example to illustrate how deduction can be applied in the context of first-order logic. Suppose we have the following two formulas in our database,

(x) ROBIN(x) -> BIRD(x)

(x) BIRD(x) -> HASWINGS(x),

then from these two formulas, we can conclude, by means of logical deduction, that the following formula must also be true:

(x) ROBIN(x) -> HASWINGS(x)

2.3 Logic As A Representation Scheme

The most important feature of logic-based representations is that there is a set of infernece rules by which facts that are known to be true can be used to derive other facts that must also be true. As a result, deductions based on these rules are guaranteed correct to an extent that other representation schemes have not yet been able to do.

From the example presented at the end of the previous section, it is not hard to realize that there is a specific rule of inference that allows us to treat deductions as syntactic manipulations of logical formulas. This makes the derivation of new formulas from old easily mechanizable.

Using logic as a representation scheme facilitates the separation of representation and processing, since logic makes no commitment to the kinds of processes that will actually make deductions. In other words, logic-based representational system allows knowledge to be represented independently of its use.

Another important property of logic is its modularity. Logical assertions can be added to a database without affecting each other, whereas in some other representational systems, addition of a new fact might affect the kinds of deductions that can be made. After all, logic is a precise and natural way to express certain notions.

2.4 Descriptive Adequacy Of First-order Logic

A logic-based representation formalism allows us to express many kinds of generalization. With a knowledge base containing such generalizations and with the use of deduction, the system can manipulate expressions in the representation formalism and permit logically complex queries to be made, even when it cannot evaluate a query directly.

In order to capture these kinds of generalizations, a representational system must, at least, be powerful enough to do

the following[Barr and Feigenbaum,1982]:

1. Say that some individual possesses a certain property without specifying which individual it is: (Ex) P(x);

2. Say that all the individuals of a certain class share a certain property without specifying what those individuals of that class are: (x) P(x)->Q(x);

3. Say that at least one of the two statements is true without specifying the truth values of the statements: P∨Q;

4. Say explicitly that a statement is false, instead of simply not saying that it is true: ¬P.

Any representation formalism that can capture these notions will be at least an extension of classical first-order logic. In other words, classical first-order logic is the minimal language that possesses the necessary expressive power.

2.5 Decision Problem For First-order Logic

In propositional logic, the validity of a formula can be easily determined by the truth table method in which the truth values of the formula are evaluated for all possible interpretations.

Unfortunately, in first-order logic, one cannnot always compute whether or not a wff is valid when quantifiers occur. It was proved by Church[1936] and Turing[1936] that there is no general decision procedure to check the validity of formulas of first-order logic; for this reason, first-order logic is said to

be undecidable. However, there are proof procedures (e.g. resolution) which can verify that a formula is valid if indeed it is valid. On the other hand, when these procedures are applied to invalid formulas, they may never terminate. Thus, first-order logic is said to be semidecidable.

Chapter 3. The Resolution Principle

The resolution principle is a rule of inference that can be used to derive the logical consequences of two formulas which are related in an appropriate way. Since its application involves only syntactic manipulation of formulas, it provides us with a useful tool to prove theorems in a purely mechanical way. We shall first consider the resolution principle for propositional logic, then we shall extend it to first-order logic.

3.1 The Resolution Principle For Propositional Logic

Suppose P, Q, and R are propositions. A central rule of inference in logic, modus ponens, says that if (P->Q) and P are true then we can conclude that Q is true. An extension of this is the chain rule, which says that if (P->Q) and (Q->R) are true, then we can conclude that (P->R) is true. Expressing (P->Q), (Q->R) and (P->R) in clause form, we have ($\neg$P$\vee$Q) and ($\neg$Q$\vee$R) give rise to ($\neg$P$\vee$R). In terms of resolution, ($\neg$P$\vee$R) is the resolvent of ($\neg$P$\vee$Q) and ($\neg$Q$\vee$R), and it is formed by the disjunction of its parent clauses, namely ($\neg$P$\vee$Q) and ($\neg$Q$\vee$R), followed by the cancellation of the complementary pair Q and $\neg$Q.

In general, the resolution principle is stated as follows: "For any two clauses C1 and C2, if there is a literal L1 in C1 that is complementary to a literal L2 in C2, then delete L1 and L2 from C1 and C2 respectively, and construct the disjunction of

the remaining clauses. The constructed clause is a resolvent of C1 and C2."

An important property of a resolvent is that any resolvent of two clauses C1 and C2 is a logical consequence of C1 and C2.

3.2 The Resolution Principle For First-order Logic

In the previous section, we observe that two important processes have to be performed when applying the resolution principle to derive conclusions. The first process is the conversion of first-order logic wffs to their logically equivalent forms --- clauses. (Although not all resolution procedures require the conversion to be done, many of them work only with wffs in clause form). The next important process is to find a literal in a clause which is complementary to a literal in another clause. Both of these processes, especially the latter one, are very simple in the context of propositional logic. However, due to the existence of quantifiers, these processes become more complicated in first-order logic and they will be discussed in detail in the next two sections.

3.2.1 Conversion Of Wffs To Clauses

A clause is disjunction of literals. The clause form of wff was introduced by Davis and Putnam[1960]. It can shown that each wff in first-order logic has a unique clause form and thus, it is often referred to as the standard form of formulas. More importantly, it can be proved that if a wff logically follows from a set of wffs S, then it also logically follows from the

set of clauses obtained by converting the wffs in S to clause form.

Basically, the conversion process can be broken down into the following sequence of steps [Nilsson,1980]:

1. Eliminate implication symbols

All occurrences of the -> and <-> symbols in a wff are eliminated by making the substitution of $\neg X1 \vee X2$ for X1->X2 and $(X1 \wedge X2) \vee (\neg X1 \wedge \neg X2)$ for X1<->X2.

2. Reduce scopes of negation symbols

We make each negation symbol, ¬, to apply to at most one atomic formula. This goal can be achieved by making use of the following equivalences repeatedly:

¬(¬X) is equivalent to X

$\neg(X1 \vee X2)$ is equivalent to $\neg X1 \wedge \neg X2$

$\neg(X1 \wedge X2)$ is equivalent to $\neg X1 \vee \neg X2$

¬(Ex)P(x) is equivalent to (x)¬P(x)

¬(x)P(x) is equivalent to (Ex)¬P(x)

3. Standardize variables

Standardizing variables is to rename variables uniformly such that each quantifier has its own unique variable. For example, we write (x)P(x)->(y)Q(y) instead of (x)P(x)->(x)Q(x).

4. Eliminate existential quantifiers

To eliminate an existential quantifier with no universal quantifiers in front of it, we simply replace the variable by a constant, for example, (Ex)P(x) is replaced by P(a). That is, we instantiate the claim that an x exists by choosing a particular constant a to take its place. However, when we have a formula such as (y)(Ex)P(x,y), then we cannot replace x by an arbitrary constant because the x that exists might depend on the value of y. We let this dependence be explicitly defined by some function of y which maps each value of y into x that "exists". Such a function is called a Skolem function. Thus, we replace (x)(y)(Ez)P(x,y,z) by (x)(y)P(x,y,f(x,y)), where f is a Skolem function.

5. Convert to prenex normal form

A wff is in prenex normal form if and only if the wff consists of a string of quantifiers called a prefix followed by a quantifier-free formula called a matrix. Since we have already removed all the existential quantifiers by skolemization , so we only have to move all the universal quantifiers to the front of the wff.

6. Eliminate universal quantifiers

Since all the variables in the wffs must be bound, we are assured that all the variables remaining at this stage are universally quantified. Thus, we may eliminate the explicit occurrence of universal quantifiers and assume that all variables in the matrix are universally

quantified.

7. Put matrix in conjunctive normal form

This can be done by repeatedly applying one of the distributive laws, namely, by replacing expressions of the form $X1 \vee (X2 \wedge X3)$ by $(X1 \vee X2) \wedge (X1 \vee x3)$.

8. Eliminate $\wedge$ symbols

By repeatedly replacing expressions of the form $(X1 \wedge X2)$ with the set of wffs {X1,X2}, we will obtain a finite set of wffs, each of which is a clause.

9. Rename variables

The last step, which is also called standardizing variables apart, is to rename variable symbols so that no variable symbol appears in more than one clause.

By following steps 1-7, we can successfully convert a wff to its conjunctive normal form. By the commutative law of conjunction(step 8), we can consider a wff in conjunctive normal form as a set of clauses. Similarly, by the commutative law of disjunction, we can view a clause as a set of literals.

3.2.2 Substitution And Unification

In proving theorems involving quantified formulas, it is often necessary to make expressions identical by substituting terms for variables. The process of finding such substitution

is extremely important in AI and is called unification. (We note that unification is more general than pattern-matching because pattern-matching processes typically do not allow variables to occur in both expressions).

A substitution is a finite set of ordered pairs {t1/v1,..,tn/vn}, where every vi is a variable, every ti is a term different from vi, and no two elements in the set have the same variable after the stroke symbol. (The substitution that consists of no elements is called the empty substitution). If E is an expression and s={t1/v1,..,tm/vm} is a substitution, then Es is a substitution instance of E which is obtained by replacing simultaneously each occurrence of the variable vi in E by the term ti. For example, let E=P(x,y,z) and s={a/x, f(b)/y, c/z}, then Es=P(a,f(b),c).

The composition of two substitutions s1 and s2 is denoted by s1s2, which is that substitution obtained by applying s2 to the terms of s1 and then adding any pairs of s2 having variables not occurring among the variables of s1. Thus, {g(x,y)/z} {a/x,b/y,c/w,d/z} = {g(a,b)/z,a/x,b/y,c/w}.

A set {E1,..,Ek} of expressions is unifiable if there exists a substitution s, called a unifier, such that E1s=E2s=...Eks. Furthermore, a unifier u for a set {E1,..,Ek} of expressions is a most general unifier if and only if for each unifier s1 for the set, there is a substitution s2 such that s1=us2. For example, consider the set {P(x),P(f(y))}, the most general unifier u={f(y)/x} and if s1={f(a)/x,a/y} then s2={a/y}.

There is an algorithm, the unification algorithm[Robinson,1965], which finds the most general unifier of a set of unifiable expressions or reports failure when the expressions are not unifiable. In addition to the recursive "matching" process, the algorithm also includes a check, an occur check , to ensure that no variable can be substituted by a term containing that same variable. The reason why occur check is necessary can be illustrated by the following example. Consider two expressions, P(x) and P(f(x)), in order to unify them, we have to substitute x by f(x), which is f(f(x)), which is f(f(f(x))), and so on. As a result, x has to be substituted by some kind of infinite structure. According to the formal definition of unification, this kind of "infinite term" should never come to exist and this can be guaranteed by performing an occur check.

3.2.3 Factoring

Having introduced the notions of unification, we can now consider another important process called factoring. The objective of factoring is to remove redundant literals in a clause. Thus, if we consider a clause as a set of literals, then factoring is simply an application of unification to a clause, since there is no redundant elements in a set.

> Definition. If two or more literals (with the same sign) of a clause C have a most general unifier g, then Cg is called a factor of C. Furthermore, Cg is an unit factor of C if Cg is an unit clause.

For instance, let $C=P(x) \vee P(f(y)) \vee \neg Q(x)$. The first two literals have a most general unifier $g=\{f(y)/x\}$. Hence, $Cg=P(f(y)) \vee \neg Q(f(y))$ is a factor of C. Sometimes a clause may have more than one factor. For example, if $C=P(x) \vee P(f(x)) \vee P(f(a))$, then both $P(f(a)) \vee P(f(f(a))$ and $P(a) \vee P(f(a))$ are factors of C.

3.2.4 Definition Of Resolvent For First-order Logic

Having introduced the concepts of unification and factoring, we can now extend the resolution principle for propositional logic to first-order logic.

Definition. Let C1 and C2 be two clause (called parent clauses) with no variables in common. Let L1 and L2 be two literals in C1 and C2, respectively. If L1 and $\neg$L2 have a most general unifier g, then the clause, or set of literals

$$(C1g - L1g) \cup (C2g - L2g)$$

is called a binary resolvent of C1 and C2. The literals L1 and L2 are called literals resolved upon.

Definition. A resolvent of (parent) clauses C1 and C2 is one of the following binary resolvents:

1. a binary resolvent of C1 and C2,
2. a binary resolvent of C1 and a factor of C2,
2. a binary resolvent of a factor of C1 and C2,
4. a binary resolvent of a factor of C1 and a factor of C2.

For example, $R(g(g(a))) \vee Q(b)$ is resolvent of $P(x) \vee P(f(y)) \vee R(g(y))$ and $\neg P(f(g(a))) \vee Q(b)$. We note that the resolvent defined above still preserves the important property of being a logical consequence of its parent clauses.

The resolution principle is for generating resolvents from a set of clauses. Since a resolvent of two clauses is also a logical consequence of them, the resolution principle can be used as an inference rule for proving theorems.

Although the resolution principle is efficient and easy to apply, unlimited applications of resolution may cause many irrelevant clauses to be generated. In the next chapter, we shall examine some strategies which restrict the application of resolution so as to improve its efficiency.

Chapter 4. Resolution Refutations

If a formula G does logically follow from a set of axioms A, then the set formed by the union of {¬G} and A must be unsatisfiable. That is, they must lead to a contradiction. This contradiction is represented by an empty clause in resolution. A resolution refutation, or proof by contradiction, is a deduction of an empty clause (contradiction) using the resolution principle as a rule of inference.

4.1 Basis Of Refutations

Before introducing the resolution algorithm, we shall briefly consider some theorems which form the foundation of the algorithm. The following theorems assume that F1,..,Fn, and G are formulas.

Theorem. (Deduction Theorem)
G is a logical consequence of F1,..,Fn if and only if the formula ((F1 ∧ .. ∧ Fn)->G) is valid.

From the deduction theorem, we can derive the following theorem.

Theorem.
G is a logical consequence of F1,..,Fn if and only if ((F1 ∧ .. ∧ Fn ∧ ¬G) is inconsistent.

Theorem. (Completeness of Resolution Principle)
A set S of clauses is unsatisfiable if and only if there is a deduction of the empty clause from S.

Thus, if we let S be the set of clauses obtained by applying the procedures, described in section 3.2.1, to the formula $(F1 \wedge .. \wedge Fn \wedge \neg G)$, then we can conclude that G is a logical consequence of F1,..,Fn if and only if we can produce a deduction of the empty clause from S by applying the resolution principle. This forms the basis of the resolution refutation algorithm presented in the following section.

4.2 The Basic Algorithm For Resolution Refutation

A resolution refutation procedure is a procedure which produces refutations by applying the resolution principle to an unsatisfiable set of clauses. It involves generating new clauses, called resolvents, from the set of clauses which is obtained by converting the axioms and the negated conclusion of the theorem to clause form. These resolvents are then added to the set of clauses from which they have been derived, and new resolvents are derived. This process is repeated until an empty clause is found.

The following is a general resolution refutation algorithm[Nilsson,1980] which will generate the empty clause if the set S (the base set) of clauses is unsatisfiable.

1. CLAUSES <- S
2. While the empty clause is not in CLAUSES
3. begin
4. Select a clause Ci, from CLAUSES
5. Select a clause Cj, from CLAUSES such that there is a literal in Cj which is a complement to one in Ci
6. Compute a resolvent Rij of Ci and Cj
7. CLAUSES <- CLAUSES $\cup$ {Rij}
8. end.

Algorithm 4.1

It can be proved that the algorithm is sound (i.e. it will not indicate that nontheorems are true). Moreover, it is complete in the sense that it is guaranteed to derive the empty clause from an unsatisfiable set of clauses. However, due to the inherent semidecidable property of first-order logic, the resolution procedure may not terminate when it is applied to some satisfiable set of clauses. For example, consider $S=\{P(a), \neg P(x) \vee P(f(x))\}$, the resolution procedure will generate resolvents $P(f(a))$, $P(f(f(a)))$, $P(f(f(f(a))))$, and so on. In this case, the procedure does not terminate and it is easy to verify that S is satisfiable.

In the following section, we shall examine some resolution strategies and we shall frequently refer to algorithm 4.1.

4.3 Resolution Strategies

The resolution refutation algorithm presented in the last section is a very general one because the selection criteria are not stated in steps 4 and 5. If we perform those selections on a random basis, then the process of searching for a refutation can be extremely time-consuming. In fact, the search space generated in this manner grows exponentially with the number of clauses in S, the base set.

Ever since the introduction of the resolution principle by Robinson[1965], many resolution strategies (both complete and incomplete) have been proposed to reduce the search space by defining some selection criteria for the clauses. A resolution strategy is said to be complete if its use results in a procedure that can always derive the empty clause from an unsatisfiable set of clauses. In other words, a strategy is complete if its application preserves the completeness of the basic resolution algorithm. (The completeness of a strategy should not be confused with the logical completeness of the resolution principle).

When we consider a resolution strategy we would like it to be complete. Nevertheless, efficiency is also important in mechanical theorem-proving. Unfortunately, sometimes we can achieve only one goal at the expense of losing, or severely degrading, the other. However, if a refinement of resolution is efficient and powerful enough to prove a large class of theorems, even though it is not complete, it may still be useful. We shall look at two such strategies in Section 4.3.4.

In the following sections, we shall examine a few resolution strategies and some of them can be combined with others to further improve the performance. Examples of resolution refutations, for illustrating the use of each strategy, will be represented as a derivation graph (or refutation tree). The nodes in such a graph are labelled by clauses; initially, there is a node for every clause in the base set S. When two clauses produce a resolvent, we create a new node and label it by that resolvent. This new node have edges linking it to the pair of nodes whose labels are the parent clauses of the resolvent. (A refutation tree, which is part of a derivation graph, has a root node labelled by NIL, the empty clause).

Before we proceed to discuss various resolution strategies, we give the following definition which will be used in the discussion.

> Definition. A first-level resolvent is one between two clauses in the base set. In general, an i-th level resolvent is one whose deepest parent is an (i-1)-th level resolvent. If the empty clause is one of the i-th level resolvents, then i is the length of the proof, or the height of the refutation tree.

4.3.1 Breadth-first Strategy

The breadth-first strategy is complete and it is guaranteed to find the shortest proof if S is unsatisfiable, but

unfortunately, it is grossly inefficient.

In this strategy, all the first-level resolvents are generated first, and if the empty clause is not among them, then all the second-level resovlents will be generated, and so on. Thus, this method is also known as the level-saturation method. Figure 1 shows the refutation gragh produced by a breadth-first strategy when $S=\{\neg L(x,T) \vee H(x),\ \neg L(x,V) \vee H(x),\ L(J,T) \vee L(J,V),\ \neg H(J)\}$, and $\neg H(J)$ is the negation of the query. We note that the empty clause NIL, is among the third-level resolvents.

4.3.2 Set-of-support Strategy

The set-of-support strategy was proposed by Wos, Robinson, and Carson[1965]. In the breadth-first strategy as well as the strategies that will be discussed in the following sections, we typically do not distinguish the negated query from the axioms when we select a pair of clauses from S (in Steps 4 and 5 of algorithm 4.1). But the set of axioms is usually satisfiable, so the set-of-support strategy suggests at least one parent of each resolvent be chosen from the set of support which consists of the negated query and the clauses that are dervied from it.

It can proved that the set-of-support strategy is complete. Moreover, it is usually more efficient than the unconstrained breadth-first strategy since it reduces the number of resolvents being generated at each level, and consequently, the number of clauses that can be resolved. This can be easily verified by comparing figure 1 with figure 2. However, like most restrictive strategies, the set-of-support strategy often

increases the depth at which the empty clause is first produced.

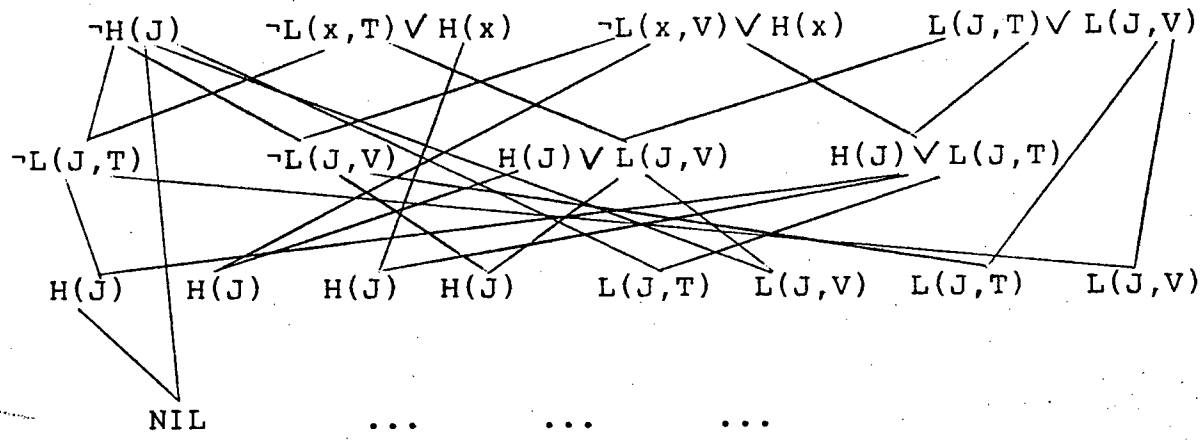

Figure 1 - Illustration of Breadth-first Strategy

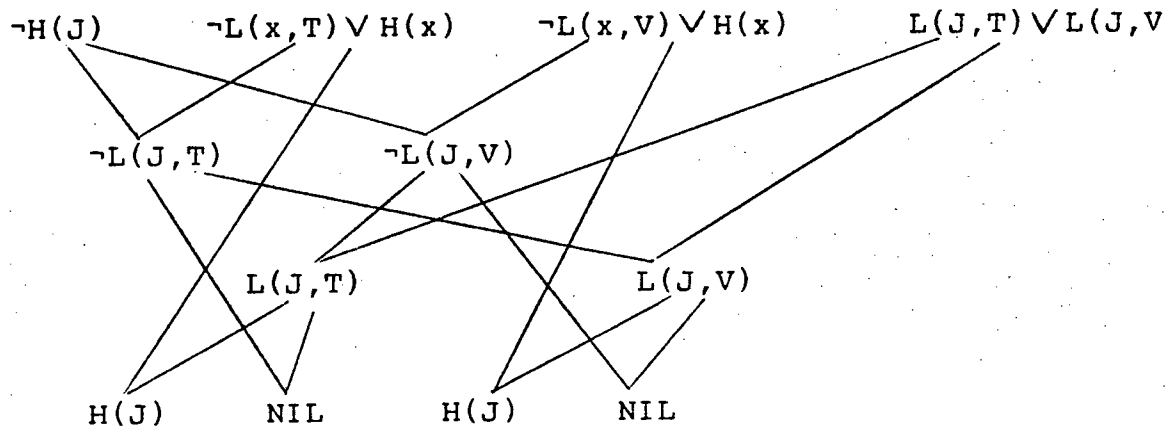

Figure 2 - Illustration of Set-of-support Strategy

4.3.3 Linear Resolution

Linear resolution was independently proposed by Loveland[1970] and Luckham[1970]. A linear resolution involves selecting a clause, called top clause, resolves it against another clause in S to obtain a resolvent, and resolves this resolvent against some other clauses in S until the empty clause is obtained. In terms of algorithm 4.1, once we have selected a top clause, then for subsequent selections in step 4, we always choose the "newest" resolvent as one of the parent clauses for further derivation until an empty clause is obtained. We shall call the top clause and all the resolvents center clauses, and the rest of the clauses involved in the proof are called side clauses.

In addition to the completeness of linear resolution, linear deduction also has a very simple structure. Figure 3 shows a linear refutation for our example problem in section 4.3.1. In fact, the refutation tree for linear resolution is so simple that it can be further reduced to a path as shown in figure 4.

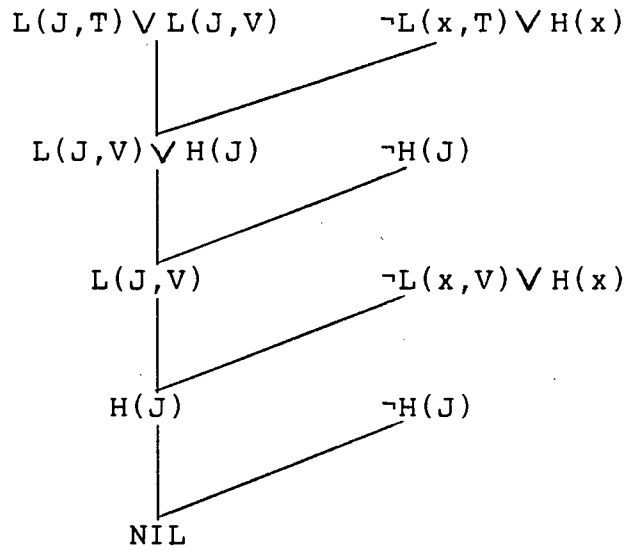

Figure 3.
Refutation tree produced by
a linear resolution

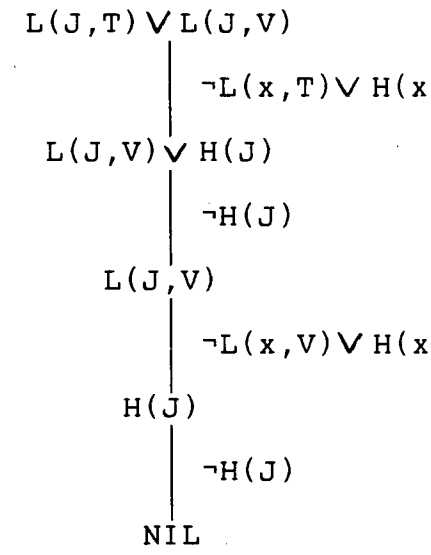

Figure 4.
Simplified form of
a linear deduction

Notice that we could also incorporate the set-of-support strategy into linear resolution by simply choosing the negated query, ¬H(J), as the top clause (Figure 5). Once we have made our choice of top clause, linear resolution allows us to remove step 4 from the algorithm, which further restricts the number of possible resolutions at any given time. Moreover, incorporation of the set-of-support strategy does not destroy the completeness of linear resolution, however, it still cannot produce the shortest proof as in the breadth-first strategy.

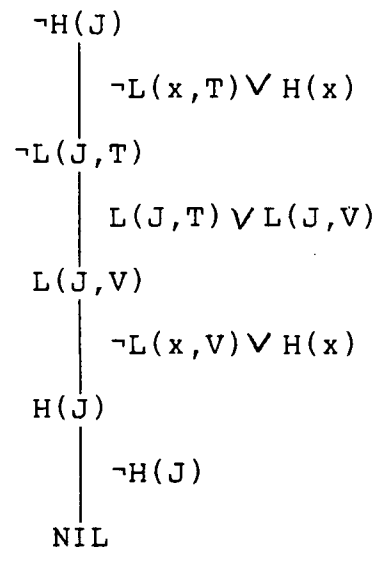

Figure 5 - Incorporation of the set-of-support into a linear resolution

4.3.4 Linear Input Resolution And Unit Resolution

All the strategies discussed in the previous sections are complete, now we shall consider two incomplete but efficient refinements of resolution: linear input resolution and unit resolution.

Linear input resolution is a subcase of linear resolution in the sense that it is the same as linear resolution except that it does not allow previously derived resolvents to be used as side clauses in a refutation. In terms of the algorithm 4.1, this simply means the removal of step 7, in addition to the modification made in the previous section. As a result, the size of CLAUSES remains constant, and thus the number of possible resolutions is greatly reduced. On the other hand, it can be shown that linear input resolution is incomplete due to

the fact that side clauses are restricted to members of the base set. Figure 3 shows a linear input refutation of our example problem.

Unit resolution is essentially an extension of the one-literal rule of Davis and Putnam[1960], and it was extensively used by Wos, Carson, and Robinson[1964]. A unit resolution is a resolution in which a resolvent is obtained by using at least one unit parent clause, or a unit factor of a parent clause. The rationale behind it is that, in order to derive an empty clause from a unsatisfiable set of clauses, one must obtain successively shorter clauses, and unit resolution provides a means for progressing toward shorter clauses rapidly. It is perhaps important to note that unit resolution is equivalent to linear input resolution since theorems that can be proved by one can also be proved by the other. Figure 6 shows a refutation produced by means of unit resolution with the set of clauses $S=\{P(x)\vee\neg Q(x,y)\vee R(f(x),y),\ \neg P(a),\ Q(a,b),\ \neg R(f(a),b)\}$.

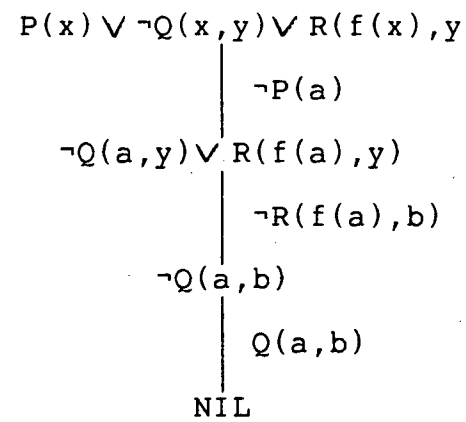

Figure 6 - Illustration of a unit resolution

4.3.5 Ordered Linear Deduction

In the last section, we observe that linear input resolution allows us to delete step 7 from algorithm 4.1 to improve efficiency. Unfortunately, the removal of step 7 destroys the completeness of linear resolution. However, linear resolution can be modified in a way that allows us to attain almost the same efficiency (by deleting step 7) while still retaining its property of completeness. This method makes use of the concept of ordered clause and the information of resolved literals.

An ordered clause is a sequence of distinct literals. As a clause, an ordered clause is also interpretated as a disjunction of all the literals in the ordered clause. The only difference is that the order of literals in a clause is immaterial, while the order of literals in an ordered clause is deliberately specified. With the concept of ordered clause in mind, we can now resolve the literals in a given clause one by one (say from left to right). Since we only consider one literal at a time, the number of clauses that can be resolved with the given clause is obviously reduced. Although the introduction of ordered clause destroys the completeness of some resolution methods, it does not affect linear resolution.

In resolution, when a resolvent is obtained, literals resolved upon are deleted. But, Loveland[1968, 1969a, b, 1972] and Kowalski and Kuehner[1971] discovered that these literals can provide information to improve linear resolution. In fact, by recording this information appropriately, we could define a

necessary and sufficient condition under which a side clause must be a center clause generated previously.

The algorithm that employs both the concept of ordered clause and the information of resolved literals is called OL-deduction (ordered linear deduction). Before presenting the precise algorithm of OL-deduction[Chang and Lee,1973], we first discuss the mechanism of recording the information of resolved literals.

Suppose $P \vee Q$ and $\neg Q \vee R$ are ordered clauses. Resolving them produces an ordered resolvent $P \vee R$. Since the literals resolved upon, namely Q and $\neg Q$, are complementary to each other, we need only record one of them, say Q. Now we can store the information by representing the ordered resolvent as $R \vee P \vee \boxed{Q}$. The framed literal, $\boxed{Q}$ in this case, does not participate in resolution, it is merely for recording that Q has been resolved upon. We shall delete a framed literal if it is not preceded by an unframed literal. Moreover, no tautology is allowed in ordered linear deduction.

Returning to the example where $S=\{\neg H(J),\ \neg L(x,T) \vee H(x),\ \neg L(x,V) \vee H(x),\ L(J,T) \vee L(J,V)\}$. An ordered linear deduction is shown in figure 7.

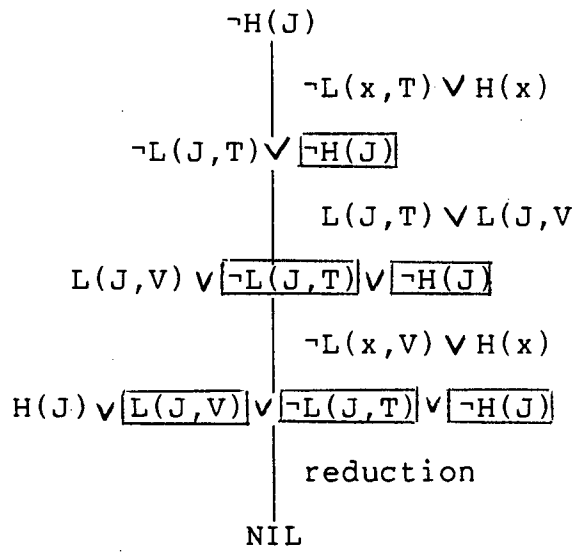

Figure 7 - Illustration of a OL-deduction

Notice that if we remove the framed literals from the first and second resolvents in figure 7, then they are exactly the same as those in figure 5. However, the last ordered resolvent $H(J) \vee \boxed{L(J,V)} \vee \boxed{\neg L(J,T)} \vee \boxed{\neg H(J)}$ is a special one because the first literal of this clause is complementary to one of its framed literals. This kind of clause is called <u>reducible ordered clause.</u> Whenever a reducible ordered clause is generated, we can be sure that the first literal can be resolved by using a center clause as the side clause (examine figure 5 to confirm this). This means that, instead of searching for that center clause to obtain the next resolvent, we could simply delete the first literal from the ordered clause to obtain it. As a consequence, we do not have to store any resolvents being generated, which obviously cuts down the number of possible resolutions; while at the same time, the completeness of linear resolution is still preserved. In terms of the algorithm 4.1,

we can delete step 7, as in the linear input strategy.

We shall discuss the details of performing step 6 after introducing the following definition.

> Definition. If two or more unframed literals (with the same sign) of an ordered clause C have a most general unifier g, an ordered factor of C can be obtained from the sequence Cg by removing all identical unframed literals except the rightmost one, and by deleting every framed literal not followed by an unframed literal in the remaining clause.

Suppose a literal L1 in a center clause C1 is complementary to a literal L2 in a side clause C2, then the ordered resolvent is computed as follows :

- obtain the clause R1, by appending C1 to C2,
- frame the literal L1 in R1 and delete L2 from R1 to obtain R2,
- obtain the ordered factor of R2 if there is one,
- if the resulting clause is reducible, reduce it by removing the leftmost literal and delete any framed literals following it.

4.4 Deletion Strategies

From algorithm 4.1, we can observe that the number of possible resolutions grows exponentially with the size of the set CLAUSES. Thus, the smaller the set CLAUSES is, the more

efficient the resolution is. Deletion strategies are strategies which eliminate two kind of clauses, namely tautologies and subsumed clauses. Although deletion strategies are complete only if they are used with the breadth-first strategy, incorporating these strategies into other resolution methods improves the performance of the methods significantly. Figure 8 shows a search tree generated by applying algorithm 4.1 to the set of clauses S={P∨Q, ¬P∨Q, ¬P∨¬Q, P∨¬Q}.

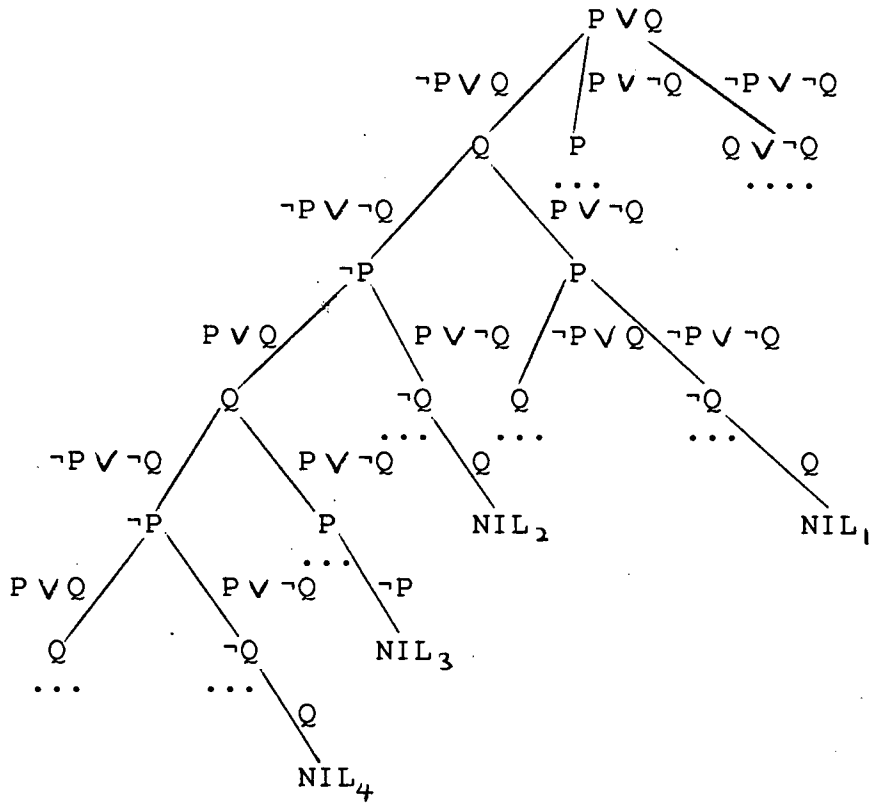

Figure 8 - Search tree generated by resolution

4.4.1 Deletion Of Tautologies

A clause is a tautology if there is a complementary pair of literals in the clause. For example, P(x)∨¬P(x)∨Q(y) is a tautology. Since tautologies are true under all

interpretations, removing them from a set will not affect the unsatisfiability of the rest of the set. By examining figure 8, one can observe that the rightmost subtree of the top clause $P \vee Q$ will not be generated if this deletion strategy is applied.

4.4.2 Subsumption

Subsumption is the process for discarding a clause that duplicates or is less general than another clause. By definition, a clause C subsumes another clause D if and only if there is a substitution s such that Cs is a subset of D, and D is called a subsumed clause. For instance, consider two clauses P(x) and P(a), P(a) is subsumed by P(x) and thus it will be deleted by subsumption. Again, refer to figure 8, we can observe that the subtree, whose root is labelled by Q at the third level, will not be generated if subsumption is applied.

The process of discarding resolvents which are subsumed by clauses in the set CLAUSES is called forward subsumption, whereas the process of discarding clauses in the set CLAUSES which are subsumed by newly generated resolvent is called backward subsumption.

The next chapter describes the implementation of the theorem-prover LRTP, and we shall see how these strategies are applied in the LRTP.

Chapter 5. The Linear Resolution Theorem Prover

The Linear Resolution Theorem Prover (LRTP), written in Prolog, was intended to be an experimental tool for studying the performance of three resolution strategies, namely, linear resolution, linear input resolution, and ordered linear deduction. In addition, it also allows the user to perform experiments on these strategies in combination with other strategies.

In the LRTP, the performance of a strategy (or a combination of strategies) is measured in terms of the number of unifications required in the search of a refutation in a given setting. The performance can be evaluated for three values, which represent the number of unifications involved in resolution and in the two deletion strategies. In addition to these statistics, the proof (if there is one) is also printed. On the other hand, if no proof can be found in the given setting, the LRTP will report failure.

5.1 Organization Of The LRTP

The LRTP is composed of a database and five modules which include the command interpreter, the translator, the refutation module, the deletion module, the unification module.

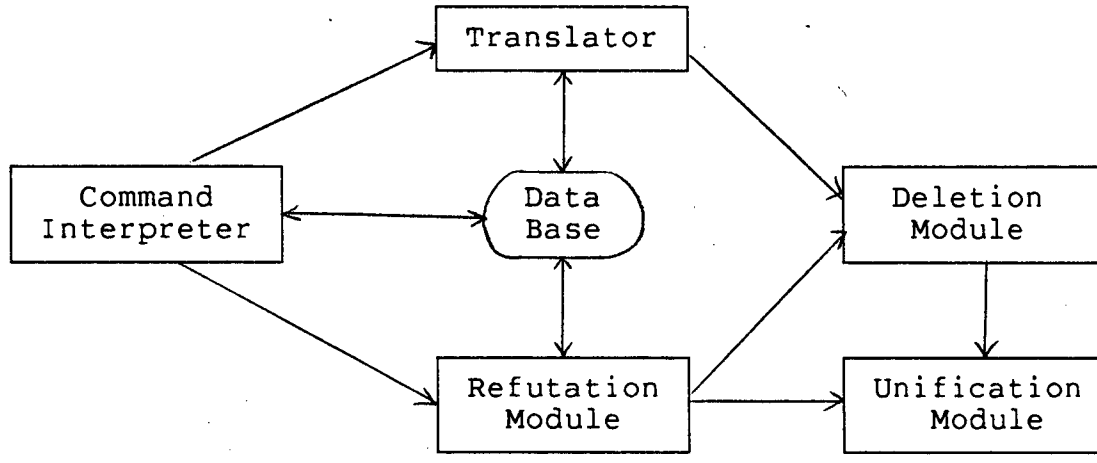

Figure 9 - Internal organization of the LRTP

The interaction between these modules and the database is shown in Figure 9. The arrows indicate the directions of access, invokation or dataflow.

5.1.1 The Database

There are two major components in the database. The first one consists of the information of the parameters required for setting up the environment in which resolutions are performed. The information is obtained from the user through interactions with the command interpreter. Details of how the parameters affect the resolutions performed in the LRTP will be discussed in section 5.1.4.

The second component consists of the "set" of clauses S which may or may not be unsatisfiable. (Notice that the LRTP makes no distinction between the negated query and the axioms). Each of the clauses in S is treated as an ordered clause, and furthermore, the "set" S itself is treated as an ordered database. In other words, the LRTP treats the set of ordered clauses as a list of ordered clauses. The ordering of the ordered clauses in the database matches the order in which they are entered by the user through the translator. It is important to note that the incorporation of the concept of ordered database will not affect the completeness of resolution if the search for refutation is aided by a backtracking mechanism (which is provided by the PROLOG/MTS interpreter). After all, treating S as an ordered database renders deterministic the selection involved in step 5 of the algorithm 4.1.

5.1.2 The Command Interpreter

The command interpreter is an interface with the user. Its basic function is to accept commands from the user and executes them, mostly by invoking other modules. Among these modules, the most important one is the refutation module which performs resolutions. By entering the appropriate commands to the command interpreter, the user can also access or modify the information stored in the database.

5.1.3 The Translator

When the user issues the command to enter clauses into the

database, the command interpreter will transfer control to the translator[Clocksin and Mellish,1981] in which a wff will be accepted and converted to its clause form using the algorithm described in section 3.2.1. When the conversion is done, the deletion strategies module will be invoked to filter any tautologies or subsumed clauses. The remaining clauses are treated as ordered clauses and are added to the ordered database. The whole process is repeated until the user has no more clauses to enter. Finally, control is transferred back to the command interpreter.

5.1.4 The Refutation Module

This module is the most important part of the LRTP. Basically, it is a refutation system which operates in an environment tailored by the users to meet their own needs. Besides searching for the proof, this module also records the number of unifications involved in factoring and resolving clauses. We shall now discuss the parameters involved in the resolution environment.

5.1.4.1 Selection Of Resolution Strategies

There are three strategies incorporated in the refutation system. These three strategies are linear resolution, linear input resolution, and ordered linear deduction. Although they are built within the same framework, each of them works independently; it is up to the user to select which one to apply.

5.1.4.2 Switches For Deletion Strategies

In addition to the selection of resolution strategies, the user has the options of combining two deletion strategies, deletion of tautologies and forward subsumption, with any one of the three resolution strategies that the user has selected. These options are provided to the user as two switches: one for forward subsumption and the other for deletion of tautologies. These switches are independent of each other and they can be turned on or off at the discretion of the user. When the switch for a deletion strategy is turned on, then that particular strategy will be applied during the search for a refutation. However, the user should be aware of the incompleteness of deletion strategies as mentioned in section 4.4.

5.1.4.3 Choice Of Top Clause

One important feature of the LRTP is that it allows the user to choose the top clause for the linear deduction. As discussed in section 4.3.3, we can combine the set-of-support strategy with linear deduction by simply choosing the top clause from the set of support. Thus, even though the LRTP itself does not distinguish between the negated query and the axioms, the user can still employ the set-of-support strategy by selecting an appropriate top clause.

5.1.4.4 Choice Of Depth Bound

Due to the semidecidable property of first-order logic, the LRTP may not terminate when it is given an invalid formula.

Furthermore, since the LRTP adopts the depth-first strategy (as is the PROLOG/MTS interpreter), it may not terminate even if the given formula is valid. As an example of this, the leftmost branch of the tree shown in figure 8 can be infinite, and as a result, an unbound depth-first search may not terminate. Thus, in order to avoid the non-termination problem, we must provide a depth bound for the tree searching. In the LRTP, the choice of the depth bound is left to the user to allow maximum flexibility. Furthermore, the depth bound is used to reduce the search space by ensuring that the number of unframed literals in a center clause C of a linear deduction is always less than the difference between the depth bound and the level of C.

5.1.4.5 Switch For Shortest Linear Refutation

By examining the search tree in figure 8, one can easily discover that there are four branches (in fact there are more) leading to the empty clause. Suppose we call the path leading to NIL_1, proof 1, and the path leading to NIL_2, proof 2, and so on. Then proof 1 and proof 2 are of length 4, whereas proof 3 and proof 4 are of length 5 and 6 respectively. Thus, if the user enters 6 for the depth bound of the search, then the LRTP will return proof 4. If the depth bound is 4 then it will return either proof 1 or proof 2, depending on the ordering of the clauses in the database. Thus, by decrementing the depth bound, one can find the shortest linear refutation. However, this requires the LRTP to repeatedly search over the same branches that have been generated before. (Notice that this shortest linear refutation is not necessarily the shortest

refutation which can only be obtained by the breadth-first strategy).

For the user's convenience, the LRTP has already automated this process and it is provided to the user as an option (switch). However, it is automated in a way that is more efficient than the one suggested in the previous paragraph. When the switch for shortest linear refutation is turned on, the LRTP will first try to find a proof within the given depth bound. If one exists, it is recorded and then, the LRTP will continue with the search, right at the point where the previous proof is found, and simultaneously setting the depth bound to one less than the length of the most recent proof obtained.

When there is more than one shortest linear refutation, the LRTP will simply return the first one that it finds. In our example, if the clause number of $P \vee \neg Q$ is smaller than that of $\neg P \vee \neg Q$, then proof 1 will be returned; otherwise proof 2 will be returned instead.

5.1.5 The Deletion Module

The deletion module can be invoked by the translator and the refutation module. When the user is entering clauses, this module will be invoked by the translator to eliminate any tautologies and subsumed clauses. The order of performing the required operations is: first, deletion of tautologies, secondly, forward subsumption, and finally, backward subsumption. We note that the application of backward subsumption implies that a clause which has already been added

to the database may still be deleted if it is subsumed by a new clause entered by the user.

During the search for a proof, the deletion module may be invoked by the refutation module, depending on the status of the switches as discussed in section 5.1.4.2. If one or both switches are on, then the deletion module will be invoked, and the number of unifications involved will be recorded as a measure of performance. Regardless of whether the switches are on or not, backward subsumption is not performed during resolution.

5.1.6 The Unification Module

Since unification done in the PROLOG/MTS interpreter does not include the occurs check, the unification is rebuilt to include this in the LRTP.

There are four occasions when unification is required. The unification module is invoked by the refutation module for factoring and resolving clauses during resolution. Moreover, unification is also involved in identifying tautologies and subsumed clauses. This implies that the unification module is invoked by the deletion module when one or both of the switches for the deletion strategies are on.

5.2 The LRTP Versus The Prolog Interpreter

Although both the LRTP and the Prolog system are resolution theorem-provers, they differ in many ways. A fundamental

difference is that the LRTP is intended as an aid to study the performance of resolution strategies, whereas the Prolog interpreter is built to provide a practical programming system. In terms of evaluating the performance of resolution strategies, the following features are lacking in the Prolog system but are provided in the LRTP.

5.2.1 Completeness

The Prolog system is based on linear input resolution which is an incomplete resolution strategy as discussed in section 4.3.4. Furthermore, the Prolog system is designed for resolution with Horn clauses only. Horn clauses[Horn,1951] are a subclass of wffs in first-order logic: it is the class of clauses with at most one unnegated literal.

On the other hand, the LRTP offers a variety of strategies which can be either complete or incomplete. Moreover, any wffs in first-order logic are legitimate expressions for the LRTP. Consequently, existential quantification is allowed in the LRTP (but not in the Prolog system).

5.2.2 Unification

As already mentioned in section 5.1.6, the unification in most Prolog implementations does not include the occurs check as the formal definition of unification requires. While the LRTP is built on top of the PROLOG/MTS system, the unification in the LRTP is rebuilt to include the occurs check for the sake of completeness.

5.2.3 Termination

Basically, Prolog adopts the unbound depth-first strategy. As a result, the resolution may not terminate. On the other hand, a depth bound is incorporated into the LRTP and consequently, termination is guaranteed.

5.2.4 Deletion Strategies

The LRTP allows any one of the three resolution strategies being implemented to combine with any one or both of the two deletion strategies, namely, deletion of tautologies and subsumption. Furthermore, these two strategies are also applied to filter any "impurities" in the wffs entered by the user. On the other hand, these strategies are not implemented in the Prolog system.

5.2.5 Measure Of Performance

As a tool for studying the performance of resolutions, the LRTP reports the number of unifications involved in searching for a refutation as a measure of performance; whereas no quantitative measure of performance is provided in the Prolog system.

5.2.6 Shortest Linear Refutation

The LRTP provides the user an option to obtain the shortest linear refutation, which is not provided in the Prolog system.

Chapter 6. Conclusion

The LRTP has been successfully implemented. It appears to be a useful experimental tool for studying the performance of the three linear resolution strategies: linear resolution, linear input resolution, and ordered linear deduction, and their combinations with other strategies, i.e. the set-of-support strategy, subsumption and deletion of tautologies. Furthermore, one can devise a large variety of experiments by manipulating the parameters of the resolution environment.

One major drawback of the LRTP is that all wffs have to be converted to clause form before resolutions can be performed. Although the clause form preserves the logical properties of the original wff, the control information for the search process is lost after the conversion. In particular, the information which guides the use of wffs in a forward-chaining or or backward-chaining manner is lost. For example, suppose we have a formula P->Q. If we use it in the forward-chaining manner, then the goal is to generate Q, given P is true. If it is used in a backward-chaining manner, as in the Prolog interpreter, then the goal is to generate P, given Q is true. In any case, we proceed in one direction only. However, if we convert it to clause form, which is $\neg P \vee Q$, then the deduction process is a bidirectional search process. As a result, the searches involved are highly redundant.

Thus, in order to perform resolution in a more efficient

way, we have to incorporate some specific control information in the resolution system to lead the search process in the "right" direction. However, incorporation of specific control information may destroy the domain-independent property of the resolution system.

BIBLIOGRAPHY

Barr, A., and Feigenbaum, E.A.(1982): The Handbook of Artifi Intelligence, Vol.1, William Kaufmann Inc., Los Altos, California.

Chang, C., and Lee, R.C.(1973): Symbolic Logic and Mechanica Theorem Proving, New York, Academic Press.

Church, A.(1936): "An unsolvable problem of number theory", Amer. J. Math. (58), pp.345-363.

Clocksin, W.F., and Mellish, C.S.(1981): Programming in Prol Springer-Verlag, New York.

Davis, M. and Putnam, H.(1960): "A computing procedure for quantification theory", J.ACM(7) No.3, pp.201-215.

Gelernter, H.(1963): "Realization of a geometry theorem-prov machine", Proceedings of an International Conference on Information Processing, Paris: UNESCO House, pp.273-282

Gilmore, P.C.(1960): "A proof method for quantification theo its justification and realization", IBM J. of Research Development (4) No.1, pp.28-35.

Goebel, R.(1980): PROLOG/MTS User's Manual, TM80-2, Dept. o Comp. Sc., Univ. of Brit. Col.

Herbrand, J.(1930): "Investigations in proof theory: the properties of the propositions", From Frege to Godel: a Source Book in Mathematical Logic (J. van Heijenoort, ed.), Havard Univ. Press, Cambridge, Massachusetts.

Horn, A.(1951): "On sentence which are true of direct unions algebras", J. of Symbolic Logic (16) No.1, pp.14-21.

Kowalski, R., and Kuehner, D.(1971): "Linear Resolution with Selection Function", Artificial Intelligence, Vol.2, pp 227-260.

Loveland, D.W.(1968): "Mechanical theorem proving by model elimination", J. ACM(15), No.2, pp.236-251.

Loveland, D.W.(1969a): "A simplified format for the model elimination theorem-proving procedure", J. ACM(16) No. pp.349-363.

Loveland, D.W.(1969b): "Theorem provers combining model elimination and resolution", Machine Intelligence, Vol. (B. Meltzer and D. Michie, eds.), American Elsevier, York, pp.73-86.

Loveland, D.W.(1970): "A linear format for resolution", Proc IRIA Symp. Automatic Demostration, Springer-Verlag, Ne York, pp.147-162.

Loveland, D.W.(1972): "A unifying view of some linear Herbra procedures", J. ACM(19), No.2, pp.366-384.

Luckham, D.(1970): "Refinements in resolution theory", Proc. IRIA Symp. Automatic Demostration, Versailes, France, 1968, Springer-Verlag, New York, pp.163-190.

McCarthy, J.(1977): "Epistemological Problems of Artificial Intelligence", IJCAI 5, pp.1038-1044.

McCarthy, J., and Hayes, P.J.(1969): "Some philosophical problems from the standpoint of artificial intelligence D. Michie and B. Meltzer (eds.), Machine Intelligence Edinburgh: Edinburgh Univ. Press, pp.463-582.

Newell, A., and Simon, H.A.(1956): "The logic theory machine IRE Transactions on Information theory(2): pp.61-79.

Nilsson, N.J.(1980): Principles of Artificial Intelligence, Alto, California, Tioga.

Robinson, J.A.(1965): "A machine oriented logic based on the resolution principle", J. ACM(12), No.1, pp.23-41.

Turing, A.M.(1936): "On computable numbers, with an applicat to the entscheindungs-problem", Proc. London Maths Soc 42, pp.230-265.

Wos, L., Carson, D., and Robinson, G.A.(1964): "The unit preference strategy in theorem proving", Proc. AFIPS 1 Fall Joint Comput. Conf.(26), pp.616-621.

Wos, L., Robinson, G.A., and Carson, D.F.(1965): "Efficiency completeness of the set of support strategy in theorem proving", J. ACM(12), No.4, pp.536-541.

Appendix A - LRTP User's Manual

1 Overview

The Linear Resolution Theorem Prover (LRTP) is built on top of the PROLOG/MTS system. It is intended to be an experimental tool for studying the performance of three resolution strategies, namely, linear resolution, ordered linear deduction, and linear input resolution. The first two are complete strategies whereas the last one is incomplete. In addition, the LRTP also allows the user to perform experiments on the three strategies in combination with others. Furthermore, the user has control over the environment in which a theorem is proved.

If a proof is found under the given setting, it will be printed together with the number of unifications involved in the search for the proof. This number is broken into three values which represent the numbers of unifications involved in resolution and in the two deletion strategies. Based on these statistics, one can evaluate the performance of a strategy (or a combination of strategies) in a designated setting. On the other hand, the LRTP will report failure if no proof can be found.

For a detailed description of the internal organization of the LRTP, the reader can refer to Chapter 5 of the thesis.

2 Using The LRTP

The LRTP is an interactive theorem prover which can be invoked by the following command:

```
$SOURCE NHFC:LRTP
```

Once the LRTP is invoked, the command interpreter will prompt the user to enter a command. (A command menu can be obtained by issuing the HELP command). All the responses to the command interpreter will be converted to uppercase and all inputs to the LRTP must end with a period.

The three commands by which the user exits the LRTP are: PROLOG, MTS, and STOP. A detailed description of the three commands can be found in section 4.4 of the manual.

The size of the workspace taken up by the LRTP is 256 MTS virtual pages (1 page = 4K bytes). This workspace contains the database of wffs in their clause form, the derivation tree constructed during the search for a proof and the theorem prover itself.

The two user interfaces of the LRTP are the translator and the command interpreter, and they will be discussed in the following two sections.

3 The Translator

When the user enters the WFF (or the PROVE command when the system is first loaded), control will be transferred to the translator. The translator is responsible for converting wffs to their clause forms and inserting them into the database if

they are neither tautologies nor subsumed clauses. When a clause is added to the database, it will be printed along with its clause number. Since a clause number indicates the current position of a clause in the database, it may be affected by backward subsumption. On the other hand, if a clause is a tautology or a subsumed clause, it will not be added to the database and a rejection message will be printed.

3.1 Clause Syntax In The LRTP

Apart from the notational differences, the concept of wffs in the LRTP is the same as in predicate calculus. The following summarizes the notational differences between the two. In this summary, F, F1, and F2 represent any wffs and x any variable.

Predicate Calculus Syntax	LRTP Syntax
¬F	-F
F1 ∧ F2	F1 $ F2
F1 ∨ F2	F1 # F2
F1 -> F2	F1 => F2
F1 <-> F2	F1 = F2
(x) F	all(*x,F)
(Ex) F	exists(*x,F)

For instance, (x)(animal(x)->(Ey)motherof(x,y)) in predicate calculus is equivalent to all(*x,animal(*x)=>exists(*y,motherof(*x,*y))) in the LRTP.

Since the LRTP does not perform any syntax checking, the user should be very careful when entering a wff (use ATTN for error recovery). In addition to the above syntax rules, each

variable in the formula must be unique and bound. Furthermore, function symbol fx and constant symbol cs (where x is an integer) should be avoided since these two types of symbols are used in skolemization and subsumption check.

Finally, when NIL (followed by a period) is entered, the translator will exit and a list of clauses in the database will be printed. Notice that there is no lowercase to uppercase conversion done in the translator, thus the atom NIL should be entered exactly as it is.

4 The Command Interpreter

The command interpreter is the major user interface through which commands are interpreted. According to the functions of the commands, we can divide them into three groups: database commands, environment commands, and exit commands. In addition to these three groups of commands, we also have the PROVE command by which a theorem can be proved.

4.1 The PROVE Command And Auto-initialization Of Parameters

Of all the LRTP commands, the PROVE command is the most important since it is the one which actually initiates the search for a proof. During the search, the LRTP frequently interrogates the values of the six parameters of the environment so as to assure that the refutation is performed in the prespecified way. The six parameters include the resolution method, the top clause, the depth bound, and switches for tautology check, subsumption check, and shortest linear

refutation.

When the user issues the PROVE command, the LRTP will check whether the six parameters have their own assigned values. If not, it will repeatedly prompt the user to assign a value for each uninitialized parameter. Once every parameter has a value, the LRTP will proceed to search for a proof in the specified environment. When the LRTP is first invoked, it is more convenient to use the PROVE command to initialize the parameters than to issue the six environment commands one by one (see Section 4.3). Furthermore, if there is no clause in the database, the PROVE command will invoke the translator to accept wffs from the user (see Section 3).

4.2 Database Commands

Database commands refer to the commands which maintain the list of clauses in the database. There are four such commands:

WFF

This command invokes the translator to convert wffs to clause forms and inserts them to the database. Details of this command can be found in Section 3.

DELETE

When this command is used, the system will prompt the user to enter a clause number. Let this number be n. The system will then remove the n-th clause (if it exists) from the database. Since a clause number indicates the current position of the clause in the database, removal of the n-th clause will cause all the

clause numbers which are greater than n to be decreased by 1. To avoid deleting a clause by mistake, the user is advised to check the clause number by the LIST command (described below) before using the DELETE command.

LIST

This command prints the list of clauses in the database. In addition to the clauses, their positions (clause numbers) in the database are also indicated.

CLEAR

This command removes all the the clauses from the database.

4.3 Environment Commands

The environment commands are used for setting the parameters of the environment in which refutation is performed. There are altogether six adjustable parameters in the environment of the LRTP, and their values can be disclosed by the ENV command. For each parameter, there is a command which allows the user to adjust its value.

METHOD

When this command is entered, the system will prompt the user to select one of the following methods for proving the theorem:

FOLR - Ordered Linear Resolution using Frames to record information of resolved literals

OLR - Ordered Linear Resolution (without frames)

LIR - Linear Input Resolution

Note that if FOLR is chosen, the switch for tautology

check will be turned on automatically since tautology is not allowed to exist in this method.

TAUT

This is the command to change the switch value for tautology check. When this command is entered, the system will prompt the user to respond with 'Y' or 'N'. Answers other than 'Y', 'YES', or 'ON' will be regarded as 'N'. Note that, however, the system will not accept 'N' when FOLR has been chosen as the resolution method.

SUBSUME

The user can use this command to change the switch value for subsumption check. Again, only 'Y', 'YES', or 'ON' are considered as affirmative answers. Forward subsumption will be performed when this switch is on.

TOP

This command causes the system to prompt the user for the top clause number. If that number corresponds to a clause in the database, the clause will be used as the top clause in the deduction.

SHORT

When this command is entered, the system will prompt the user to set the switch for obtaining the shortest linear refutation (if there is one). To turn the switch on, one has to enter 'Y', 'YES', or 'ON', any other response will turn the switch off.

STEPS

This command is used for setting the depth bound of the search for the proof.

4.4 Exit Commands

There are three commands that allow the user to exit the LRTP and they are described below:

PROLOG

When this command is used, the system will exit and return to level of the PROLOG/MTS interpreter, leaving the LRTP loaded, and the workspace intact. The LRTP can be re-entered by using LRTP as a goal clause.

MTS

This command is equivalent to the PROLOG command except that the system returns to the MTS command level rather than the PROLOG/MTS interpreter level. Consequently, the system can be re-entered with a MTS restart command.

STOP

The effect of this command is to release all workspace storage, and to unload both the LRTP and the PROLOG/MTS interpreter.

5 Sample Terminal Session

```
#$r plog:v2_prolog par=ws=256
#Execution begins
 PROLOG/MTS 0.2

 Please enter a command: {Type "HELP" for assistance}
 PROVE.

 Enter a wff : {terminate input of wffs by NIL}
 P#Q.
 The wff has been converted to:
 1: P#Q

 Enter a wff : {terminate input of wffs by NIL}
 -P#Q.
 The wff has been converted to:
 2: -P#Q

 Enter a wff : {terminate input of wffs by NIL}
 -P#-Q.
 The wff has been converted to:
 3: -P#-Q

 Enter a wff : {terminate input of wffs by NIL}
 P#-Q.
 The wff has been converted to:
 4: P#-Q

 Enter a wff : {terminate input of wffs by NIL}
 NIL.

 Clause(s) in current database :-
 1: P#Q
 2: -P#Q
 3: -P#-Q
 4: P#-Q

 Select one of the following methods:-
    FOLR - Ordered Linear Resolution with Frame
    OLR  - Ordered Linear Resolution
    LIR  - Linear Input Resolution
 OLR.

 Tautology Check ? Y/N
 Y.

 Subsumption Check ? Y/N
 N.

 Shortest linear proof ? Y/N
 N.

 Max. no. of steps :
 6.
```

```
Top clause no. :
1.
(1) P#Q has been chosen as top clause

Proof: {length of proof = 6}
(1) P#Q
 |
 |  (2) -P#Q
 |
(6) Q <==FACTOR== (5) Q#Q
 |
 |  (3) -P#-Q
 |
(7) -P
 |
 |  (1) P#Q
 |
(8) Q
 |
 |  (3) -P#-Q
 |
(9) -P
 |
 |  (4) P#-Q
 |
(10) -Q
 |
 |  (11) Q <==FACTOR== (5) Q#Q
 |
NULL

No. of unifications involved in :
 Resolution    Subsumption Checks    Tautology Checks
     72                 0                   0
```

```
Command:
SHORT.

Shortest linear proof ? Y/N
Y.

Command:
PROVE.

Proof: {length of proof = 4}
(1) P#Q
 |
 |  (2) -P#Q
 |
(6) Q <==FACTOR== (5) Q#Q
 |
 |  (3) -P#-Q
 |
(7) -P
 |
 |  (4) P#-Q
 |
(8) -Q
 |
 |  (9) Q <==FACTOR== (5) Q#Q
 |
NULL

No. of unifications involved in :
 Resolution   Subsumption Checks   Tautology Checks
     197              0                  4

Command:
STOP.
EXIT PROLOG/MTS 0.2
#Execution terminated
```

Appendix B - Program Listing Of The LRTP

```
/******************** THE  TRANSLATOR ********************/

OP('=>',RL,33).
OP(=,RL,33).
OP(#,RL,37).
OP($,RL,38).
OP(-,PREFIX,39).

/*  Convert a formula to clausal form  */

translate(*x,*x6)
   <- implout(*x,*x1) &
      negin(*x1,*x2) &
      skolem(*x2,*x3,NIL) &
      univout(*x3,*x4) &
      conjn(*x4,*x5) &
      form_list(*x5,*x6).


/*  Removing implications  */

implout((*p = *q),((*p1 $ *q1) # (-*p1 $ -*q1)))
   <- / & implout(*p,*p1) & implout(*q,*q1).

implout((*p => *q),(-*p1 # *q1))
   <- / & implout(*p,*p1) & implout(*q,*q1).

implout(all(*x,*p),all(*x,*p1))
   <- / & implout(*p,*p1).

implout(exists(*x,*p),exists(*x,*p1))
   <- / & implout(*p,*p1).

implout((*p $ *q),(*p1 $ *q1))
   <- / & implout(*p,*p1) & implout(*q,*q1).

implout((*p # *q),(*p1 # *q1))
   <- / & implout(*p,*p1) & implout(*q,*q1).

implout((-*p),(-*p1))
   <- / & implout(*p,*p1).

implout(*p,*p).


/*  Moving negation inwards  */

negin((-*p),*p1)
   <- / & neg(*p,*p1).

negin(all(*x,*p),all(*x,*p1))
   <- / & negin(*p,*p1).
```

```
negin(exists(*x,*p),exists(*x,*p1))
   <- / & negin(*p,*p1).

negin((*p $ *q),(*p1 $ *q1))
   <- / & negin(*p,*p1) & negin(*q,*q1).

negin((*p # *q),(*p1 # *q1))
   <- / & negin(*p,*p1) & negin(*q,*q1).

negin(*p,*p).


neg((-*p),*p1)
   <- / & negin(*p,*p1).

neg(all(*x,*p),exists(*x,*p1))
   <- / & neg(*p,*p1).

neg(exists(*x,*p),all(*x,*p1))
   <- / & neg(*p,*p1).

neg((*p $ *q),(*p1 # *q1))
   <- / & neg(*p,*p1) & neg(*q,*q1).

neg((*p # *q),(*p1 $ *q1))
   <- / & neg(*p,*p1) & neg(*q,*q1).

neg(*p,(not(*p))).


/*  Skolemising  */

skolem(all(*x,*p),all(*x,*p1),*vars)
   <- / & skolem(*p,*p1,*x.*vars).

skolem(exists(*x,*p),*p2,*vars)
   <- / & gensym(f,*f) &
      CONS(*f.*vars,*sk) &
      substitute(*x,*sk,*p,*p1) &
      skolem(*p1,*p2,*vars).

skolem((*p # *q),(*p1 # *q1),*vars)
   <- / & skolem(*p,*p1,*vars) &
      skolem(*q,*q1,*vars).

skolem((*p $ *q),(*p1 $ *q1),*vars)
   <- / & skolem(*p,*p1,*vars) &
      skolem(*q,*q1,*vars).

skolem(*p,*p,*).


/*  Moving universal quantifiers outwards  */
```

```
univout(all(*x,*p),*p1)
   <- / & univout(*p,*p1).

univout((*p $ *q),(*p1 $ *q1))
   <- / & univout(*p,*p1) &
      univout(*q,*q1).

univout((*p # *q),(*p1 # *q1))
   <- / & univout(*p,*p1) &
      univout(*q,*q1).

univout(*p,*p).


/*  Distributing '$' over '#'  */

conjn((*p # *q),*r)
   <- / & conjn(*p,*p1) &
      conjn(*q,*q1) &
      conjnl((*p1 # *q1),*r).

conjn((*p $ *q),(*p1 $ *q1))
   <- / & conjn(*p,*p1) &
      conjn(*q,*q1).

conjn(*p,*p).


conjnl(((*p $ *q) # *r),(*p1 $ *q1))
   <- / & conjn((*p # *r),*p1) &
      conjn((*q # *r),*q1).

conjnl((*p # (*q $ *r)),(*p1 $ *q1))
   <- / & conjn((*p # *q),*p1) &
      conjn((*p # *r),*q1).

conjnl(*p,*p).


/*  Convert to list form for resolution  */

form_list((*p $ *q),*r)
   <- / & form_list(*p,*p1) &
      form_list(*q,*q1) &
      combine(*p1,*q1,*r).

form_list((*p # *q),(*r).NIL)
   <- / & form_list(*p,(*p1).NIL) &
      form_list(*q,(*q1).NIL) &
      combine(*p1,*q1,*r).

form_list(*p,(*p.NIL).NIL).
```

```
/*  Generate a unique symbol, *symbol, by concatenating
    *letter with an integer, *integer                    */

gensym(*letter,*symbol)
   <- newsnum(*integer) &
      STRING(*integer,*intlist) &
      STRING(*symbol,*letter.*intlist).


symcount(1).

newsnum(*integer)
    <- symcount(*integer) &
       add1(*integer,*next) &
       set(symcount(*next),1).


/*  Substitute each occurrence of *x in the 3rd arg
     by *sk and return result in 4th arg                 */

substitute(*x,*sk,(*p # *q),(*p1 # *q1))
   <- / & substitute(*x,*sk,*p,*p1) &
      substitute(*x,*sk,*q,*q1).

substitute(*x,*sk,(*p $ *q),(*p1 $ *q1))
   <- / & substitute(*x,*sk,*p,*p1) &
      substitute(*x,*sk,*q,*q1).

substitute(*x,*sk,*p,*p1)
   <- sub_skel(*x,*sk,*p,*p1) & /.

substitute(*,*,*p,*p).


/*  Substitute *sk for each occurrence of *x in
    structure *p and return result in *p2     */

sub_skel(*x,*sk,*p,*p2)
    <- SKEL(*p) &
       CONS(*pred.*arglist,*p) &
       subst(*x,*sk,*arglist,*p1) &
       CONS(*pred.*p1,*p2).


/*  Construct a new list, *m, made up from elements
    of list, *l, except that any occurrences of *x
    will be replaced by *a                              */

subst(*,*,NIL,NIL).

subst(*x,*a,*var.*l,*a.*m)
   <- VAR(*var) &
      samevar(*x,*var) & / &
      subst(*x,*a,*l,*m).
```

```
subst(*x,*a,*s.*r,*s1.*r1)
   <- sub_skel(*x,*a,*s,*s1) & / &
      subst(*x,*a,*r,*r1).

subst(*x,*a,*y.*l,*y.*m)
   <- subst(*x,*a,*l,*m).


/*  Check if *x and *y refer to the same variable  */

SAME(FALSE).
samevar(*x,*y)
   <- bindtest(*x,*y) &
      SAME(TRUE) &
      set(SAME(FALSE),1).

/*  Check if *x and *y are the same by binding an atom
    to *x, if they are the same, SAME is add to the db */

bindtest(*x,*y)
   <- bind(*x) &
      ATOM(*y) &
      set(SAME(TRUE),1) &
      FAIL.

bindtest(*,*).

bind(atom).
```

```
/****************** THE  COMMAND  INTERPRETER *************/

/*  Initial values  */

CONTROL(ATTN,ON).
ERROR(N).

STATUS(CMD).
METHOD(NIL).
TAUT(NIL).
SUBSUME(NIL).
WFF(NIL).
TOP(0,NIL).
SHORT(NIL).
SHORT_STEPS(0).
STEPS(NIL).
FROM(OTHERS).
CURRNUM(0).
PROOF(NIL,0).
CLAUSE(NIL).
UNIFY_COUNT(R,0).
UNIFY_COUNT(S,0).
UNIFY_COUNT(T,0).


/*  Command interpreter  */

LRTP
   <- setup &
      READ(*input) &
      execute(*input) &
      ready_for_next &
      continue.


/*  Error recovery  */

ERROR
   <- set(ERROR(Y),1) &
      LRTP.


/*  Set up the LRTP  */

setup
   <- ERROR(N) & / &
prints('Please enter a command: {Type "HELP" for assistance}')
       & setcase(CMD).

setup
   <- set(ERROR(N),1) &
      prints('Error has been recovered, please continue') &
      ready_for_next.
```

```
/*  Convert all input to upper case when STATUS <> WFF  */

setcase(WFF)
   <- update(CONTROL(LOWER,OFF),CONTROL(LOWER,ON)) & /.

setcase(*stat)
   <- ¬equ(*stat,WFF) &
      update(CONTROL(LOWER,ON),CONTROL(LOWER,OFF)) & /.

setcase(*).


/*  Replace *oldax by *newax  */

update(*oldax,*newax)
   <- DELAX(*oldax,*index) &
      ADDAX(*newax,*index).


/*  Process input according to current STATUS  */

execute(*input)
   <- STATUS(*stat) &
      concatenate(process,*stat,*pred) &
      CONS(*pred.*input.NIL,*process) &
      *process & /.


/*  Concatenate first two strings and return a 3rd string  */

concatenate(*id1,*id2,*id3)
   <- STRING(*id1,*id1list) &
      STRING(*id2,*id2list) &
      combine(*id1list,*id2list,*id3list) &
      STRING(*id3,*id3list).


/*  Get ready to accept next input  */

ready_for_next
   <- STATUS(*stat) &
      prompt(*stat) &
      setcase(*stat).


/*  Process command  */

processCMD(HELP)
   <- describe_cmd.

processCMD(PROVE)
   <- set(FROM(PROVE),1) &
      initialize.

processCMD(*cmd)
   <- member(*cmd,WFF.METHOD.TAUT.SUBSUME.TOP.SHORT
```

```
      .STEPS.DELETE.PROLOG.NIL) &
      set(STATUS(*cmd),1).

processCMD(ENV)
   <- printENV.

processCMD(LIST)
   <- listCLAUSE.

processCMD(CLEAR)
   <- flushCLAUSE(*) &
      set(TOP(0,NIL),2) &
      set(WFF(NIL),1).

processCMD(MTS)
   <- MTS.

processCMD(STOP)
   <- STOP.

processCMD(*)
   <- print('Illegal command, type "HELP" for assistance').


/*  Translate input wffs to clausal form and
     store them in database                    */

processWFF(NIL)
   <- prints('Clause(s) in current database :-') &
      listCLAUSE &
      initialize.

processWFF(*wff)
   <- translate(*wff,*cl) &
      print('The wff has been converted to:') &
      assert(*cl).

processWFF(*)
   <- print('Illegal wff, try again').


/*  Select a method  */

processMETHOD(*method)
   <- member(*method,FOLR.OLR.LIR.NIL) &
      set(METHOD(*method),1) &
      checkMETHOD(*method) &
      initialize.

processMETHOD(*)
   <- print('This method is not available in the LRTP').


/*  Set the switch for tautology checks
     according to user response          */
```

```
processTAUT(*yes)
   <- member(*yes,Y.YES.ON.NIL) &
      set(TAUT(Y),1) &
      initialize.

processTAUT(*)
   <- ¬METHOD(FOLR) &
      set(TAUT(N),1) &
      initialize.

processTAUT(*)
   <- print('Tautology is not allowed in FOLR') &
      initialize.


/* Set the switch for subsumption checks
    according to user response                    */

processSUBSUME(*yes)
   <- member(*yes,Y.YES.ON.NIL) &
      set(SUBSUME(Y),1) &
      initialize.

processSUBSUME(*)
   <- set(SUBSUME(N),1) &
      initialize.


/*  Select top clause  */

processTOP(*top)
   <- INT(*top) &
      AXN(CLAUSE,2,CLAUSE(A,*topclause),*top) &
      printcl(*top) &
      print(' has been chosen as top clause') &
      set(TOP(*top,*topclause),2) &
      initialize.

processTOP(*)
   <- print('Clause not in database, try again').


/*  Set the switch for shortest linear refutation  */

processSHORT(*yes)
   <- member(*yes,Y.YES.ON.NIL) &
      set(SHORT(Y),1) &
      initialize.

processSHORT(*)
   <- set(SHORT(N),1) &
      initialize.


/*  Enter max. no. of steps into database  */
```

```
processSTEPS(*maxsteps)
   <- INT(*maxsteps) &
      set(STEPS(*maxsteps),1) &
      initialize.

processSTEPS(*)
 <- print('Max. no. of steps must be an integer, try again').


/*  Delete clause according to input clause no.  */

processDELETE(*clnum)
   <- INT(*clnum) &
      printcl(*clnum) &
      print(' has been deleted') &
      DELAX(CLAUSE(*,*),*clnum) &
      resetTOP(*clnum) &
      set(STATUS(CMD),1).

processDELETE(*)
   <- print('Clause not in database') &
      set(STATUS(CMD),1).



/*  Prompt user for initial values if there is one  */


INIT(WFF).
INIT(METHOD).
INIT(TAUT).
INIT(SUBSUME).
INIT(SHORT).
INIT(STEPS).

initialize
   <- FROM(OTHERS) &
      set(STATUS(CMD),1).

initialize
   <- INIT(*stat) &
      CONS(*stat.NIL.NIL,*null) &
      *null &
      set(STATUS(*stat),1).

initialize
   <- TOP(0,NIL) &
      set(STATUS(TOP),1).

initialize
   <- set(FROM(OTHERS),1) &
      set(STATUS(CMD),1) &
      getproof.
```

```
/*  Prompt user according to current STATUS  */

prompt(CMD)
   <- prints('Command:').

prompt(WFF)
 <- prints('Enter a wff : {terminate input of wffs by NIL}').

prompt(METHOD)
   <- prints('Select one of the following methods:-') &
  print('   FOLR - Ordered Linear Resolution with Frame') &
      print('   OLR  - Ordered Linear Resolution') &
      print('   LIR  - Linear Input Resolution').

prompt(TAUT)
   <- prints('Tautology Check ? Y/N').

prompt(SUBSUME)
   <- prints('Subsumption Check ? Y/N').

prompt(TOP)
   <- prints('Top clause no. :').

prompt(SHORT)
   <- prints('Shortest linear proof ? Y/N').

prompt(STEPS)
   <- prints('Max. no. of steps :').

prompt(DELETE)
   <- prints('Clause no. :').

prompt(PROLOG).


/*  Return to Prolog interpreter when STATUS=PROLOG */

continue
   <- STATUS(PROLOG) &
      set(STATUS(CMD),1).


/*  Print out the command menu  */

describe_cmd
   <- prints('Commands available:') &
      print('   PROVE   - prove a theorem') &
      print('   WFF     - enter wffs into database') &
      print('   METHOD  - select a resolution method') &
print('   TAUT    - change switch value of tautology check') &
print('   SUBSUME - change switch value of subsumption check') &
      print('   TOP     - select a top clause') &
print('   SHORT   - change switch value for shortest linear proof
 & print('   STEPS   - enter max. no. of steps (depth bound)') &
print('   ENV     - print parameter values of environment') &
```

```
      print('   LIST     - list all clauses in database') &
      print('   DELETE   - delete clause(s) in database') &
      print('   CLEAR    - clear all clauses in database') &
      print('   PROLOG   - return to Prolog interpreter') &
      print('   MTS      - return to MTS') &
      print('   STOP     - stop the theorem prover') &
      print('   HELP     - print command menu') &
      prints('Please enter one of the above commands').


/*  Print out the values of parameters in the environment  */

printENV
   <- METHOD(*method) &
      TAUT(*yesno1) &
      SUBSUME(*yesno2) &
      TOP(*top,*topclause) &
      form_clause(*topclause,*cl) &
      SHORT(*yesno3) &
      STEPS(*maxsteps) &
      println('Method = '.*method.NIL) &
      println('Tautology Check = '.*yesno1.NIL) &
      println('Subsumption Check = '.*yesno2.NIL) &
println('Top clause no. = '.*top.' ; Top clause = '.*cl.NIL) &
      println('Shortest linear proof = '.*yesno3.NIL) &
      println('Max. no. of steps = '.*maxsteps.NIL).


/*  List out all clauses in database  */

listCLAUSE
   <- AXN(CLAUSE,2,CLAUSE(*,*cl),*n) &
      display(*n,*cl) &
      FAIL.

listCLAUSE.


/*  Flush clause(s) which are of *type  */

flushCLAUSE(*type)
   <- AXN(CLAUSE,2,CLAUSE(*type,*),*clnum) &
      DELAX(CLAUSE(*type,*),*clnum) &
      FAIL.

flushCLAUSE(*).


/*  Add clauses to database  */

assert(NIL).

assert(*cl1.*cl2)
   <- screen(*cl1,0,*clist) &
      set(WFF(NON_NIL),1) &
      ADDAX(CLAUSE(A,*cl1),*n) &
```

```
      display(*n,*cl1) &
      submsg(*clist) &
      assert(*cl2).

assert(*.*cl)
   <- assert(*cl).


/*  Display a clause and its no.  */

display(*n,*cl)
   <- form_clause(*cl,*clause) &
      println(*n.': '.*clause.NIL).


/*  Print subsumption message if there is any  */

submsg(NIL) <- /.

submsg(*clist)
<- print('which subsumes the following clause(s) in database:')
    & print_delete(*clist).


/*  Print out a list of subsumed clause(s) and delete them  */

print_delete(NIL).

print_delete(*clnum.*clist)
   <- printcl(*clnum) &
      print(' {being deleted}') &
      DELAX(CLAUSE(*,*),*clnum) &
      print_delete(*clist).


/*  Update *oldax when given *newax and its *arity  */

set(*newax,*arity)
   <- CONS(*pred.*,*newax) &
      AXN(*pred,*arity,*oldax) &
      update(*oldax,*newax) & /.


/*  If METHOD=FOLR, tautology checks is compulsory  */

checkMETHOD(FOLR)
   <- set(TAUT(Y),1) &
      print('Tautology checks = Y').

checkMETHOD(*).


/*  Reset top clause to null value when it is deleted  */

resetTOP(*deleted)
   <- TOP(*deleted,*) &
```

```
      set(TOP(0,NIL),2).

resetTOP(*).


/*  Set up COUNTs ,find a proof, print the proof
    and flush all clauses except those input by user  */

getproof
   <- reset &
      findproof &
      printproof &
      flushCLAUSE(R) &
      flushCLAUSE(F).


/*  Reset UNIFY_COUNTs to zero and copy STEPS to
    SHORT_STEPS before resolving                          */

reset
   <- UNIFY_COUNT(*type,*) &
      update(UNIFY_COUNT(*type,*),
             UNIFY_COUNT(*type,0)) &
      FAIL.

reset
   <- STEPS(*maxsteps) &
      set(SHORT_STEPS(*maxsteps),1).


/*  Retrieve top clause and start proving  */

findproof
   <- TOP(*topclnum,*topcl) &
      set(PROOF(NIL,0),2) &
      factoring(*topclnum,*topcl,*cclause,*cclnums) &
      prove(*cclnums,*cclause,NIL,0).

findproof.


/*  Print out the proof  */

printproof
   <- PROOF(NIL,*) &
      NEWLINE &
      prints('Cannot be proved in given setting').

printproof
   <- PROOF(*proof,*length) &
      NEWLINE &
   println('Proof: {length of proof = '.*length.'}'.NIL) &
      printproof1(*proof) &
      print('NULL') &
      NEWLINE &
      printstatistics.
```

```
printproof1(NIL).

printproof1(*pf1.*pf2)
   <- printproof1(*pf2) &
      printclpair(*pf1).


/*  Print out unification statistics  */

printstatistics
   <- UNIFY_COUNT(R,*rcount) &
      UNIFY_COUNT(S,*scount) &
      UNIFY_COUNT(T,*tcount) &
      print('No. of unifications involved in :') &
print(' Resolution   Subsumption Checks   Tautology Checks') &
      println('     '.*rcount.'                '.*scount
            .'                    '.*tcount.NIL).


/*  Print out the two resolved clauses  */

printclpair(*cclnums.*clnums)
   <- printfactor(*cclnums) &
      printbar(2) &
      WRITECH(' | ') &
      printfactor(*clnums) &
      printbar(2).


/*  Print a clause together with its factor if there is one  */

printfactor(*facnum.*clnum)
   <- printpcl(*facnum) &
      WRITECH(' <==FACTOR== ') &
      printpcl(*clnum) &
      NEWLINE.

printfactor(*clnum)
   <- printpcl(*clnum) &
      NEWLINE.


/*  Print a clause as part of proof  */

printpcl(0)
   <- WRITECH('reduction').

printpcl(*clnum)
   <- AXN(CLAUSE,1,CLAUSE(*cl),*clnum) &
      form_clause(*cl,*clause) &
      printon('('.*clnum.')'.*clause.NIL).


/*  Print a clause  */
```

```
printcl(*clnum)
   <- AXN(CLAUSE,2,CLAUSE(*,*cl),*clnum) &
      form_clause(*cl,*clause) &
      printon('('.*clnum.')'.*clause.NIL).


/*  Form a clause from a list for output  */

form_clause(*lit.NIL,*lit1)
   <- / & checknot(*lit,*lit1).

form_clause(*lit.*cl,*lit1 # *cl1)
   <- checknot(*lit,*lit1) &
      form_clause(*cl,*cl1).

form_clause(NIL,NIL).


/*  Replace 'not' by '-'  */

checknot(not(*lit),-*lit) <- /.

checknot(*lit,*lit).


/*  Print out '|'  */

printbar(0) <- /.

printbar(*n)
   <- print(' |') &
      sub1(*n,*n1) &
      printbar(*n1).


/*  Print text utilities  */

prints(*x)
   <- NEWLINE &
      print(*x).

print(*x)
   <- WRITECH(*x) &
      NEWLINE.

println(NIL)
   <- NEWLINE.

println(*hd.*tl)
   <- WRITECH(*hd) &
      println(*tl).


printon(NIL).
```

```
printon(*hd.*tl)
   <- WRITECH(*hd) &
      printon(*tl).
```

```
/****************** THE  REFUTATION  MODULE ********************/

/*  Prove a clause (2nd arg), its no. (1st arg), and return
    the proof (3rd arg) and length (4th arg) of proof          */

prove(*,NIL,*proof,*length)
   <- SHORT_STEPS(*maxsteps) &
      LE(*length,*maxsteps) &
      set(PROOF(*proof,*length),2) &
      store_proof &
      shortest_or_1st(*length).

prove(*cclnums,*cclause,*pfsofar,*lensofar)
   <- limitcheck(*cclause,*lensofar) &
      resolve(*clnums,*cclause,*resolvent,*lensofar) &
      store(R,*resnum,*resolvent) &
      factoring(*resnum,*resolvent,*faccl,*facnums) &
      screen(*faccl,*facnums,0) &
      add1(*lensofar,*lenplus1) &
      reduce_resolvent(*facnums,*faccl,*rednum,*redcl,
      (*cclnums.*clnums),*pfsofar,*proof,*lenplus1,*newlen) &
      prove(*rednum,*redcl,*proof,*newlen).


/*  Store up all clauses involved in the proof  */

store_proof
   <- CLAUSE(*cl) &
      DELAX(CLAUSE(*cl)) &
      FAIL.

store_proof
   <- CLAUSE(*,*cl) &
      ADDAX(CLAUSE(*cl)) &
      FAIL.

store_proof.


/*  Check if shortest or first proof is required  */

shortest_or_1st(*)
   <- SHORT(N).

shortest_or_1st(*length)
   <- SHORT(Y) &
      sub1(*length,*lenless1) &
      set(SHORT_STEPS(*lenless1),1) &
      ANCESTOR(*prove,2) &
      RETRY(*prove).


/*  Suceeds iff limit not exceeded  */

limitcheck(*cclause,*lensofar)
```

```
   <- ¬equ(*cclause,NIL) &
      SHORT_STEPS(*maxsteps) &
      LT(*lensofar,*maxsteps) &
      length(*cclause,*cclen) &
      DIFF(*maxsteps,*lensofar,*stepsleft) &
      GE(*stepsleft,*cclen).


/*  Resolve a clause(2nd arg) with another clause whose
     no. is specified in 1st arg, and return resolvent
     in 3rd arg                                          */

resolve(*clnums,*lit.*cl,*resolvent,*lensofar)
   <- negate(*lit,*notlit) &
      select(*clnum,*clause) &
      factoring(*clnum,*clause,*factor,*clnums) &
      resolvable(R,*notlit,*factor,*newclause) &
      form_resolvent(*newclause,*lit.*cl,*resolvent).


/*  Store clause in database and remove it on failure  */

store(*,*,NIL) <- /.

store(*type,*clnum,*clause)
   <- ADDAX(CLAUSE(*type,*clause),*clnum) &
      set(CURRNUM(*clnum),1).

store(*type,*,*clause)
   <- CURRNUM(*clnum) &
      sub1(*clnum,*currnum) &
      set(CURRNUM(*currnum),1) &
      DELAX(CLAUSE(*type,*clause),*clnum) &
      FAIL.


/*  Factorise the clause(2nd arg) if possible and store
    the factor(3rd arg) in database if it exists        */

factoring(*clnum,*clause,*factor,(*facnum.*clnum))
   <- factor(R,*clause,*ffactor) &
      ¬equ(*clause,*ffactor) &
      last_frame(*ffactor,*factor) &
      store(F,*facnum,*factor).

factoring(*clnum,*clause,*clause,*clnum)
   <- factor(X,*clause,*clause).


/*  Reduce the resolvent if the method is FOLR  */

reduce_resolvent(*fnum,*f,*fnum,*f,*pr,*pf,*pr.*pf,*len,*len)
   <- ¬METHOD(FOLR) & /.

reduce_resolvent(*fnum,*f,*rnum,*r,*pr,*pf,(*fnum.0).*pr.*pf,
                 *len,*newlen)
```

```
   <- limitcheck(*f,*len) &
      reduce(R,*f,*r) &
      ¬equ(*f,*r) &
      store(R,*rnum,*r) &
      add1(*len,*newlen).

reduce_resolvent(*fnum,*f,*fnum,*f,*pr,*pf,*pr.*pf,*len,*len)
   <- reduce(X,*f,*f).


/*  Negate 1st arg and return in 2nd arg for resolution  */

negate(not(*lit),*lit) <- /.

negate(*lit,not(*lit)).


/*  Select a clause for resolving  */

select(*clnum,*clause)
   <- AXN(CLAUSE,2,CLAUSE(A,*clause),*clnum).

select(*clnum,*clause)
   <- METHOD(OLR) &
      AXN(CLAUSE,2,CLAUSE(R,*clause),*clnum).


/*  Resolvable iff 2nd arg is a member of 3rd arg
    and return the resolved clause in 4th arg       */

resolvable(*type,*lit,FRAMED(*).*cl,*newcl)
   <- / & resolvable(*type,*lit,*cl,*newcl).

resolvable(*type,*lit1,*lit2.*cl,*cl)
   <- unify(*type,*lit1,*lit2).

resolvable(*type,*lit,*l.*cl,*l.*newcl)
   <- resolvable(*type,*lit,*cl,*newcl).


/*  Form resolvent according to method used  */

form_resolvent(*clause,*lit.*cl,*resolvent)
   <- METHOD(FOLR) & / &
      combine(*clause,FRAMED(*lit).*cl,*rescl) &
      last_frame(*rescl,*resolvent).

form_resolvent(*clause,*lit.*cl,*resolvent)
   <- METHOD(LIR) & / &
      combine(*clause,*cl,*resolvent).

form_resolvent(*clause,*lit.*cl,*resolvent)
   <- combine(*clause,*cl,*resolvent).


/*  Merge common factor to the left  */
```

```
factor(*,NIL,NIL).

factor(*type,FRAMED(*lit).*cl,FRAMED(*lit).*newcl)
    <- / & factor(*type,*cl,*newcl).

factor(*type,*lit.*cl,*factor)
    <- member(*type,*lit,*cl) &
       factor(*type,*cl,*factor).

factor(*type,*lit.*cl,*lit.*newcl)
    <- factor(*type,*cl,*newcl).


/*  Reduce goal clause if the last literal can be resolved
    with a framed literal                                    */

reduce(*,NIL,NIL) <- /.

reduce(*type,*lit.*cl,*reduced)
    <- negate(*lit,*notlit) &
       memberf(*type,*notlit,*cl) &
       last_frame(*cl,*reduced).

reduce(*,*lit.*cl,*lit.*cl)
    <- negate(*lit,*notlit) &
       ¬memberf(X,*notlit,*cl).


/*  Remove the last literal if it is framed  */

last_frame(FRAMED(*).*cl,*newcl)
   <- / & last_frame(*cl,*newcl).

last_frame(*clause,*clause).


/*  Length returns the length(2nd arg) of a clause(1st arg).
    Framed literals are not counted                          */

length(NIL,0).

length(FRAMED(*).*cl,*len)
   <- / & length(*cl,*len).

length(*.*cl,*lenplus1)
   <- length(*cl,*len) &
      add1(*len,*lenplus1).


/*  Member succeeds iff 2nd arg is a member of 3rd arg
    and the UNIFY_COUNT is incremented according to *type;
    notice that framed literal does not affect the count  */

member(*type,*lit,FRAMED(*).*cl)
   <- / & member(*type,*lit,*cl).
```

```
member(*type,*lit1,*lit2.*)
   <- unify(*type,*lit1,*lit2).

member(*type,*lit,*.*cl)
   <- member(*type,*lit,*cl).


/*  Memberf succeeds iff the clause is reducible;
    notice non-framed literal does not affect the count  */

memberf(*type,*lit1,FRAMED(*lit2).*)
   <- unify(*type,*lit1,*lit2).

memberf(*type,*lit,*.*cl)
   <- memberf(*type,*lit,*cl).


/*  Increment UNIFY_COUNT by 1 according to *type.
    No count is incremented if *type=X              */

incrUNIFY_COUNT(X) <- /.

incrUNIFY_COUNT(*type)
   <- UNIFY_COUNT(*type,*count) &
      add1(*count,*countplus1) &
      update(UNIFY_COUNT(*type,*),
             UNIFY_COUNT(*type,*countplus1)).


/*  Combine 1st list with 2nd list to form a 3rd list  */

combine(NIL,*x,*x) <- /.

combine(*head.*tail,*x,*head.*newtail)
   <- combine(*tail,*x,*newtail).


/*  *x - 1 = *y  */

sub1(*x,*y) <- DIFF(*x,1,*y).


/* *x + 1 = *y  */

add1(*x,*y) <- DIFF(*x,'-1',*y).


/*  Succeed iff 1st arg equals to 2nd arg  */

equ(*x,*x).


/*  Succeed iff first arg is a member of second arg  */

member(*lit,*lit.*cl) <- /.
```

```
member(*lit,*.*cl)
    <- member(*lit,*cl).
```

```
/****************** THE  DELETION  MODULE ********************/


/*  Check if deletion strategies have to be applied  */

screen(NIL,*,*) <- /.

screen(*cl,*facnums,*clist)
   <- taut_test(*cl) &
      subsume_test1(*cl,*facnums) &
      subsume_test2(*cl,*clist).


/*  Test if the clause is a tautology  */

taut_test(*)
   <- TAUT(N) &
      ¬METHOD(FOLR) &
      ¬STATUS(WFF) & /.

IS_TAUT(N).
taut_test(*cl)
   <- set(IS_TAUT(N),1) &
      tautology(*cl) &
      set(IS_TAUT(Y),1) &
      FAIL.

taut_test(*)
   <- IS_TAUT(N) & /.

taut_test(*taut)
   <- IS_TAUT(Y) &
      tautmsg(*taut) & / &
      FAIL.


/*  Test if the input clause is subsumed by
     another clause in db                        */

SUBSUME_CLAUSE(NIL).
SUBSUMED_CLAUSE(NIL).

subsume_test1(*,*)
   <- SUBSUME(N) &
      ¬STATUS(WFF) & /.

subsume_test1(*fcl,*facnums)
   <- set(SUBSUME_CLAUSE(NIL),1) &
      remove_frames(*fcl,*subcl) &
      ground(*subcl) &
      set(SUBSUMED_CLAUSE(*subcl),1) &
      subsumed(*subcl,*facnums,*clnum) &
      set(SUBSUME_CLAUSE(*clnum),1) &
      FAIL.
```

```
subsume_test1(*,*)
   <- SUBSUME_CLAUSE(NIL) & /.

subsume_test1(*subcl,*)
   <- SUBSUME_CLAUSE(*clnum) &
      submsg(*subcl,*clnum) & / &
      FAIL.


/* Test if input clause subsumes any clause(s)
     in db & delete them                          */

subsume_test2(*,*)
   <- ¬STATUS(WFF) & /.

subsume_test2(*cl,*)
   <- set(SUBSUME_CLAUSE(NIL),1) &
      AXN(CLAUSE,2,CLAUSE(*,*clause),*clnum) &
      ground(*clause) &
      set(SUBSUMED_CLAUSE(*clause),1) &
      subsumed_by(*clause,*cl) &
      SUBSUME_CLAUSE(*clist) &
      set(SUBSUME_CLAUSE(*clnum.*clist),1) &
      FAIL.

subsume_test2(*,*clist)
   <- SUBSUME_CLAUSE(*clist).


/*  Succeeds iff the arg. is a tautolgy  */

tautology(FRAMED(*).*cl)
   <- / & tautology(*cl).

tautology(*lit.*cl)
   <- negate(*lit,*notlit) &
      member(T,*notlit,*cl) & /.

tautology(*.*cl)
   <- tautology(*cl).


/*  Replace all variables in the clause by distinctive
    constants (atoms starting with a letter 'c')     */

ground(NIL).

ground(*x.*y)
   <- VAR(*x) &
      gensym(c,*x) & / &
      ground(*y).

ground(FRAMED(*).*cl)
   <- / & ground(*cl).

ground(*x.*y)
```

```
   <- CONS(*.*arglist,*x) &
      ground(*arglist) &
      ground(*y) & /.


/*  Removes all frames in 1st arg
     and returns result in 2nd arg  */

remove_frames(NIL,NIL).

remove_frames(FRAMED(*).*cl,*newcl)
   <- 7 & remove_frames(*cl,*newcl).

remove_frames(*lit.*cl,*lit.*newcl)
   <- remove_frames(*cl,*newcl).


/*  Succeeds iff *subcl is subsumed by a clause
    (whose no. is *clnum) in the current database */

subsumed(*subcl,*facnums,*clnum)
   <- AXN(CLAUSE,2,CLAUSE(*,*clause),*clnum) &
      ¬itself(*clnum,*facnums) &
      subsumed_by(*subcl,*clause) & /.


/*  Succeeds iff 1st arg is subsumed by 2nd arg  */

subsumed_by(*,NIL) <- /.

subsumed_by(*lit.*,*clause)
   <- resolvable(S,*lit,*clause,*resolvent) &
      SUBSUMED_CLAUSE(*subcl) &
      subsumed_by(*subcl,*resolvent).

subsumed_by(*.*cl,*clause)
   <- subsumed_by(*cl,*clause).


/* Prevent checking subsumption with the clause itself  */

itself(*clnum,*clnum.*) <- /.

itself(*clnum,*.*clnum) <- /.

itself(*clnum,*clnum).


/*  Print message if clause is not accepted
      during input of wffs                  */

tautmsg(*taut)
   <- STATUS(WFF) &
      form_clause(*taut,*tautcl) &
      println(*tautcl.' which is a tautology '.NIL) &
      print('    -> not added into database').
```

```
tautmsg(*).


submsg(*subcl,*clnum)
   <- STATUS(WFF) &
      form_clause(*subcl,*subsumed) &
      printon(*subsumed.' which is subsumed by '.NIL) &
      printcl(*clnum) &
      prints('    -> not added into database').

submsg(*,*).
```

```
/****************** THE  UNIFICATION  MODULE ***************/

/*  Unification with occurs check  */


unify(*category,*x,*y)
   <- incrUNIFY_COUNT(*category) &
      unifiable(*x,*y) &
      equ(*x,*y) & /.


unifiable(*x,*y)
   <- VAR(*x) &
      VAR(*y) & /.

unifiable(*x,*y)
   <- VAR(*x) & / &
      occurscheck(*x,*y).

unifiable(*x,*y)
   <- VAR(*y) & / &
      occurscheck(*y,*x).

unifiable(*x,*y)
   <- SKEL(*x) &
      SKEL(*y) & / &
      CONS(*xlist,*x) &
      CONS(*ylist,*y) &
      eqlist(*xlist,*ylist).

unifiable(*,*).


occurscheck(*x,*y)
   <- SKEL(*y) & / &
      CONS(*.*arglist,*y) &
      ¬occursin(*x,*arglist).

occurscheck(*,*).


eqlist(NIL,NIL).

eqlist(*hd1.*tl1,*hd2.*tl2)
   <- unifiable(*hd1,*hd2) &
      eqlist(*tl1,*tl2).


occursin(*x,*hd.*)
   <- VAR(*hd) &
      samevar(*x,*hd).

occursin(*x,*hd.*)
   <- SKEL(*hd) &
      CONS(*.*arglist,*hd) &
```

```
      occursin(*x,*arglist).

occursin(*x,*.*tl)
   <- occursin(*x,*tl).
```