Hierarchical Deduction

Submmited to Prof. W. W. Bledsoe

Tie-cheng Wang

March 1984 ATP-78

This work was supported by NSF Grant MCS-8011417 and MCS-8313499

Abstract

Hierarchical deduction proposed in this paper is a resolution style theorem prover. The basic hierarchical deduction structure is shown to be a powerful decision procedure for propositional logic. The power of the prover is enhanced for proving theorems in first order logic by the constraints on framed literals and on common tails, the learning algorithm, the level subgoal reordering, the partial set of support and other constrictive strategies.

This paper also investigates some properties of so called "twin-symbols" and discusses the use of these properties for evaluating the correctness of deduction paths and for making the procedure concentrate on the important subgoals.

A summary of the implementation result obtained by computer runs of the hierarchical prover which succeeded in proving some difficult theorems is included in the paper.

Table of Contents

1. Introduction

Many resolution methods or variants were developed during the last few years. They mainly use two types of deduction scheme. The first is called many-to-many type deduction which can be generally formalized by the following: for a given set S of clauses,

$$R^0(S) = S, R^{n+1}(S) = R(R^n(S)), \text{ for } n = 0,1,2...,$$

where R is a resolvent generator, that produces and retains (under some constraints defined by the prover) all resolvents by resolving between all (or a subset) of clauses in the current data base, $R^n(S)$. Hyperresolution [6], unit, set of support [16], locking [5] all use many-to-many type of deduction schema (we call them first type provers). The second is called one-to-one type deduction. For a given set of clauses S, a second type prover (with the one-to-one deduction schema), picks one clause in S as a goal clause, produces and retains only one resolvent by resolving this goal clause against one clause in the current data base, and then uses the newly produced resolvent as new goal clause to continue the deduction. Most of the goal oriented provers developed before, such as the input resolution [6], model elimination [8], SL-resolution [7] use one-to-one type of deduction schema.

Though it appears likely that most of the successes in proving hard theorems automatically by general theorem provers were obtained by the first type provers [12], the second type of provers have been also subjected to considerable investigations. In comparison, the goal oriented prover is more human-like, easier to incorporate with heuristics and expert knowledge, and the studying of goal oriented deduction is beneficial to develop non-resolution theorem provers [2], that emphasize also the use of the domain dependent knowledge in proving theorems.

The hierarchical deduction procedure to be described in this paper is a resolution style theorem prover, which is similar to the second type of theorem provers in that, it is a goal oriented prover, but it is also similar to the first type of provers in that, it produces and retains all legal resolvents producible from a goal clause. So we call this procedure a one-to-many type prover (with one-to-many type deduction schema).

We will show that this procedure is characterized by a number of features, among which the followings are important:

- The underlying deduction schema is a powerful decision procedure for the propositional logic;
- The power for proving theorems in first order logic can be enhanced by a series of constrictive strategies.
- As a goal-oriented deduction method, it is to some extent convenient for incorporating heuristics and expert knowledge.
- A partial set of support strategy can be designed to effectively increase its ability in proving some difficult theorems.

We will first overview the hierarchical prover, and then give a formal discussion of the underlying hierarchical structure. Then we define a version of the complete hierarchical deduction procedure, H1-DEDUC, in which the constraints on framed literals and on common tails are introduced. Following this we give a description of the so called correct reduction and the learning algorithm, and present the partial set of support strategy and locking method.

The second part of the paper is mainly devoted to the control of the hierarchical deduction. We propose a "twin" symbol match requirement as the basis to investigate some useful heuristics that depend only on the syntactical features of the resolvents. Then these heuristics are to be used in the level subgoal reordering procedure and in the evaluation routine.

Up to now our procedure has been used to prove a number of difficult theorems. Among them, theorems IMV and AM8 suggested by Prof. W.W. Bledsoe, are generally acknowledged to be hard for the mechanical prover, where

IMV: Derived from the theorem, a continuous function on a closed real interval passes each intermediate point.

Am8: Derived from the theorem, a continuous function on a closed real interval attains its minimum (or maximum) on that interval.

The clause forms of the above theorems can be found in section 15, where some of results obtained by a hierarchical prover are reported.

2. Overview of Hierarchical Deduction

Hierarchical deduction accepts well formed formula (WFF) as input and returns "Proved", if the formula is proved and is therefore a theorem, or returns "Failed" otherwise.

Preprocessing

Before the proof is started, a good deal of processing of the input WFF is done for later use by the prover. This consists mainly of following:

- Let H $\Rightarrow$ C be the formula obtained by skolemizing the input WFF. The prover first normalizes it to be two sets of clauses:

  1. The rule set, which consists of all clauses obtained from H.

  2. The goal set, which consists of all clauses obtained from C.

- A name is assigned to each of the clauses. All literals of the input clauses are indexed by 1 (some changes in the indices will be made later).

- A node named 1 is produced to hold the names of the goal clauses and the names of the rule clauses (including the names of goal clauses) and other information in the corresponding fields of the node respectively.

- Many additional things are done in this step. We will describe these in the main part of the paper.

Deduction

Hierarchical prover (H1-DEDUC) begins deduction from the first node (node 1). The general proof process is as follows:

1. A goal clause G is taken from the node along with the index of the left-most literal of G. A set of clauses which will be used as rules is obtained by retrieving the node indicated by this index.

2. All "legal" resolvents are produced by resolving each clause in the rule set with the goal clause G upon its left-most literal. The names of these resolvents are stored as goals in a newly produced node.

3. If "box" (the empty clause) is obtained, then the procedure returns "Proved" otherwise the above process is repeated by working on the new node.

Some Detail

The above is only the general outline of the procedure, much more is going on. For each of the resolvents produced, the indices of the literals inherited from the goal clause are retained but the indices of the literals inherited from a rule clause are replaced by a larger integer which is just the name of new node being produced.

Several constraints are applied for producing the "legal resolvents". For example, we use "framed literals" and "local subsumption test" to prevent redundant resolvents, and we demand the remainders (the common tails) of resolvents be "mergeable", etc.

Because of the various restrictions placed upon the process, it is sometimes the case (when the procedure has fallen in a wrong path) that no legal resolvents can be produced from a goal clause. In this case the prover backtracks to the last node where the goal clause was selected, and takes the second goal clause in the node. If all goal clauses fail at a given node, then it backtracks to the previous node, etc.

The prover has a learning algorithm by which the goal clauses that have been used are stored in some related nodes, and these clauses will be used later for checking if a newly selected goal clause is redundant.

The direction of the deduction is controlled by assigning a priority to each legal resolvent by an evaluation subroutine, and also by choosing the "important" goal literal at each step of the process. The evaluation of resolvents and the ordering of the goal literals are based on the use of certain ideas such as "twin symbols". The main idea behind the use of twin symbols, is that in any "correct" deduction path in the proof of a "generalized theorem", each twin symbol must be "matched out" at least one time.

Experiments with the prover show that the evaluation subroutine is capable of selecting useful goal clauses with a high probability.

3. Hierarchical deduction structure

In order to automatically prove hard theorems by a goal oriented theorem prover, an explosive search process seems to be generally indispensable. So, confining, with an acceptable expense, the search space for the proof to a manageable size is the key for succeeding with the proof in a reasonable time. This can be done by realizing the followings simultaneously:

1. Shortening the length of the deduction path;
2. Decreasing the branching factors effectively and efficiently.

These considerations motivated us to explore a hierarchical deduction structure. We will study it by first defining a basic hierarchical deduction, H0-DEDUC, which is a procedure only applicable for propositional logic, and later handle the general case,

Node, index, H0-resolvent

We use nodes to hold data inputted or produced by the prover. A node is named by an integer which serves as a key to locate the node. The content of a node is located in several fields, only four of which are currently mentioned. They are listed as follows:

POINTER: the parent node name of the node;
PARENT: the goal clause from which the node was produced;
RULES : a set of clauses used as rules in the deduction;
GOALS : a set of clauses from which the goal clause is selected.

We use a list to represent the content of a node. For example, there may be a node 4 with the content, (3 C0 RS GS), where 3 is a pointer, C0 is the parent, RS and GS are RULES and GOALS of node 4, respectively.

A clause is a disjunction of literals ordered from left to right. Let C be a clause, then FIRST(C) is the first (left-most) literal of C, and REST(C) is the clause obtained from C by deleting the first literal of C.

Every literal of a clause is indexed by an integer, so a clause including the indices of its literals will be written as

$${}_{i_1}L1\ {}_{i_2}L2 \ldots {}_{i_n}Ln,$$

where i_1, i_2 ... i_n are indices.

A clause C is *properly indexed* iff the indices of it are in descending order. The following are examples of properly indexed clauses:

Example 3.1

C1: ${}_0R\ {}_0L$
C2: ${}_1{\sim}R\ {}_1P$
C3: ${}_3L\ {}_2{\sim}R\ {}_1P$
C4: ${}_3{\sim}L\ {}_2{\sim}R\ {}_2H\ {}_1P$

We will use the term H-resolvent (hierarchical resolvent) to mean a resolvent obtained in the general hierarchical deduction procedure. The special form of the H-resolvent for the ground case is called H0-resolvent (ground hierarchical resolvent) which is to be defined first.

Definition: H0-resolvent. Let G, C be properly indexed ground clauses and let n be an integer. Let L_g be the first literal of G and L_c be a literal of C. If L_g and L_c are complementary, then let C' be the clause obtained from C by deleting L_c and changing all indices of the remaining literals to be the integer n, let G' be the clause obtained from G by deleting the first literal L_g and keeping the indices of the remaining literals unchanged, let H be the clause obtained by concatenating the clause C' to the left of the clause G' and then merging right. If H is not a tautology, then H is called an *H0-resolvent* of G against C in level n.

The following are two examples of H0-resolvents obtained from the clauses shown in example 3.1.

$H1 = {}_2L\ {}_1P$	H0-resolvent of C2 against C1 in level 2
$H2 = {}_2{\sim}R\ {}_2H\ {}_1P$	H0-resolvent of C4 against C3 in level 5

Note H2 is obtained by merging right from the clause, ${}_5{\sim}R\ {}_5P\ {}_2{\sim}R\ {}_2H\ {}_1P$.

H0-DEDUC

H0-DEDUC starts the deduction from the first goal clause of the first node, node 1. Once a goal clause is taken, it will be deleted from the content of the current node and will be recorded in the PARENT field of the node to be produced next.

The procedure fetches the rule clauses from the node indicated by the index of the first literal of the goal clause (The clauses stored in the GOALS field of the node are also taken as rule clauses). The name of the node to be produced next is identical to this index increased by 1. There are mainly two tasks in a round of deduction. One is to produce all H0-resolvents of the goal clause against all the rule clauses. (Note, only the first literal of the goal clause can be resolved upon in the resolutions.) Another task is to select a subset of the rule clauses as the set of rule clauses of next produced node.

The procedure works recursively. The pointer of a node is used for backtracking which happens when the GOALS field of the current node is empty. The procedure stops when "box" (the empty clause) is obtained or the procedure backtracks to node 0. The following is the precise definition of the procedure, H0-DEDUC:

PROCEDURE H0-DEDUC(node)

```
step 1  If node = 0 then exit with "FAIL"
        GS := GOALS-GET(node)[1]
        If GS = NIL then return H0-DEDUC(POINTER-GET(node))

Step 2  G := FIRST(GS)
        If G = NIL then exit with "box"
        L := FIRST(G)
        I := the index of L
        GOALS-MAKE(node, REST(GS))
        RS := GOALS-GET(I) $\cup$ RULES-GET(I)

Step 3  nextnode := I + 1
        NEWGS := {H: H is an H0-resolvent of G against C in level I+1, C$\in$RS}
        NEWRS := RS - RS^L[2] - RS^~L

Step 4  nextnode <=[3] (I G NEWRS NEWGS)
        Return H0-DEDUC(nextnode)
```

To prove a theorem (or non-theorem) of propositional logic by H0-DEDUC, the first node, node 1, inputted to the procedure should be normalized (by the prover automatically) to be a so called primary node, which is defined by the follows:

Definition: Simple ground clause. A ground clause is called a *simple ground clause* iff it is not a tautology and it contains no repetition of a literal.

Definition: Primary node. Node 1 is called a *primary* node iff all clauses in the RULES field and the GOALS field of the node are simple ground clauses, and all literals of the clauses are indexed by 1.[4]

Theorem 1: H0-DEDUC is a decision procedure for propositional logic.
The detailed rewording of this theorem as well as its proofs can be found in appendix.

Now we give an example to show the process of H0-DEDUC. We use "---------" to make a distinction between nodes.

---

[1]The functions, GOALS-GET(node), PARENT-GET(node), RULES-GET(node) and POINTER-GET(node) are used to retrieve the fields of GOALS, PARENT, RULES and POINTER respectively, and the procedure, GOALS-MAKE(node,p), PARENT-MAKE(node,C0), RULES-MAKE(node,RS) and POINTER-MAKE(node,GS) are used to assign values to the these fields.

[2]$S^{\wedge}L = \{C: C\in S \wedge L\in C\}$

[3]This is an abbreviation which indicates that the content of the "nextnode" is to be (I G NEWRS NEWGS)

[4]For the ground proof by H0-DEDUC, the goal clauses obtained from the input WFF will not included in the RULES field of the primary node.

Example 3.2

C_1:	${}_1P\ {}_1Q\ {}_1R$	Given
C_2:	${}_1P\ {}_1Q\ {}_1{\sim}R$	Given
C_3:	${}_1P\ {}_1{\sim}Q\ {}_1R$	Given
C_4:	${}_1{\sim}P\ {}_1Q$	Given
C_5:	${}_1{\sim}P\ {}_1{\sim}Q\ {}_1R$	Given
C_6:	${}_1{\sim}Q\ {}_1{\sim}R$	Given

Node 1 <= (0 NIL (C_1 C_2 C_3 C_4 C_5) (C_6))

Goal: C_6, nextnode: 2, rules: C_1,C_2,C_3,C_4,C_5

C_7:	${}_2P\ {}_1{\sim}R$	H0-resolvent of C_6 against C_2
C_8:	${}_2{\sim}P\ {}_1{\sim}R$	H0-resolvent of C_6 against C_4

Node 2 <= (1 C_6 NIL (C_7 C_8))

Goal: C_7, nextnode: 3, rules: C_8

C_9:	${}_1{\sim}R$	H0-resolvent of C_7 against C_8

Node 3 <= (2 C_7 NIL (C_9))

Goal: C_9, nextnode: 2, rules: C_1,C_2,C_3,C_4,C_5

C_{10}:	${}_2P\ {}_2Q$	H0-resolvent of C_9 against C_1
C_{11}:	${}_2P\ {}_2{\sim}Q$	H0-resolvent of C_9 against C_3
C_{12}:	${}_2{\sim}P\ {}_2{\sim}Q$	H0-resolvent of C_9 against C_5

Node 2 <= (1 C_9 (C_4) (C_{10} C_{11} C_{12}))

Goal: C_{10}, nextnode: 3, rules: C_{11},C_{12},C_4

C_{13}:	${}_2Q$	H0-resolvent of C_{10} against C_4

Node 3 <= (2 C_{10} NIL (C_{13}))

Goal: C_{13}, nextnode: 3, rules: C_{11},C_{12},C_4

C_{14}:	${}_3P$	H0-resolvent of C_{13} against C_{11}
C_{15}:	${}_3{\sim}P$	H0-resolvent of C_{13} against C_{12}

Node 3 <= (2 C_{13} NIL (C_{14} C_{15}))

Goal: C_{14}, nextnode: 4, rules: C_{15}

C_{16}:	NIL	H0-resolvent of C_{14} against C_{15}

Node 4 <= (3 C_{14} NIL (C_{16}))

Goal: C_{16}

Exit with "Box"

In comparison with linear resolution, the depth of the deduction in a proof by H0-DEDUC is usually shorter and the branching factor is usually smaller. For example, in the deduction of "box" from example 3.2, with C_6 as the top clause, only 6 goal clauses are used in H0-DEDUC, whereas 9 goal clauses are needed using linear resolution.

The reason for this is that H0-DEDUC can obtain and retain several useful resolvents from a goal clause. This results in fewer selections of the goal clauses for a proof. As was shown in Example 3.2, two useful resolvents, C_7 and C_8 were obtained from a goal clause C_6 in one round of deduction, and three useful resolvents C_{10}, C_{11}, C_{12} were obtained from the goal clause C_9, etc. In contrast, linear resolution can retain at most one resolvent from a goal clause, other resolvents as well as unused rule clauses can be only left as backup points (branches). So, each useful resolvent needed for a proof corresponds to a selected goal clause.

Note that we used the word "select" for linear resolution in the above example, because there exist wrong branches in many midsteps of a linear resolution proof. For example, at the last step before the "box" is produced, the goal clause can even see 13 rule clauses. Many of them are applicable but useless. So a careful selection of the goal clause in linear resolution is needed for almost every step.

For H0-DEDUC, the useless rule clauses can be discarded dynamically, so the branching factor is fairly small. It is shown in Example 3.2 that only one rule clause C_8 is available for goal clause C_7 in the second round of deduction, because all the rule clauses used by the last goal clause C_8 in the first round deduction are discarded by the rule cut subroutine:

$$RS = RS - RS\hat{}L - RS\hat{}\sim L$$

in constructing node 2. So only one resolvent C_9 can be obtained in the second round of deduction. This kind of situation happens in many steps. At the last step before "box" is obtained, the goal clause C_{14} can only see one rule, C_{15}, which gave "box". In fact, for this example, the procedure can deduce "box" without backtracking for any selection of goal clauses.

H0-DEDUC has been experimented with on many examples of sets of ground clauses, some of which contain up to 10 different atoms. These experiments show that H0-DEDUC can produce "box" very efficiently if the set of clauses is unsatisfiable, and needs few steps to halt (with failure) if the set is satisfiable. For the above example, if rule clause C_2 is missing, then H0-DEDUC would exit with only one resolvent C_8 produced.[5]

4. Local subsumption test

An important feature of the hierarchical deduction structure shown in the last section is that, when a ground goal clause G is resolved, the rule clauses containing a literal which is identical with or complementary to the first literal of the goal clause can be discarded in constructing the next node. But, for proving theorems of first order logic, we can not simply discard a rule clause without losing completeness. So, for the general hierarchical deduction procedure, instead of discarding the rule clauses, we will use so called "framed" literals to retain the information provided by the literals previously

[5]In order to prove a non-theorem, the set of all clauses should be included in the GOALS field of the starting node.

resolved upon and design a local subsumption test to constrain the hierarchical resolvents.

We attach to each clause a set of framed literals. The rule of recording the framed literals are as follows:

1. The set of framed literals is empty for all input clauses;
2. Let G be a goal clause, H be a resolvent of G against a rule clause C and L_g be the first literal of G. Let θ be the substitution by which H is obtained. Let FL_g be the set of framed literals of G. Then $FL_g\theta \cup \{L_g\theta\}$ is the set of framed literals of the resolvent H (i.e., add a new framed literal L to FL_g and apply the substitution θ).
3. A framed literal will be deleted from a goal clause if the index of this literal is greater than or equal to the index of the first non-framed literal of this goal clause.

The information provided by framed literals will be used by a local subsumption test. A resolvent is called *local subsumed* if an atom of its framed literals is identical to an atom of its non-framed literals. In this case, the resolvent will be discarded.

Note that only the non-framed literals form the real part of a clause, but the variables in framed literals are instantiated along with those in the non-framed literals during the resolutions.

For the ground case, we can prove that deductions obtained by using framed literals and the local subsumption test are equivalent to the deductions obtained by using the rule cut routine.

For Example 3.2, suppose the procedure uses the local subsumption test instead of rule cut, then one more resolvent can be obtained from the goal clause C_7 because the rule clause C_4 is included in the RULES field of first node 2. They look like the following:

C_4: ${}_1{\sim}P\ {}_1Q$	Rule clause in node 2
C_7: ${}_2P\ [{}_1{\sim}Q]\ {}_1{\sim}R$	Goal clause
C_{10}: ${}_3Q\ [{}_2P]\ [{}_1{\sim}Q]\ {}_1{\sim}R$	C_7 against C_4

Note C_{10} must be detected as local subsumed. So, like Example 3.2, only one resolvent C_9 is retained and C_9 must be the goal clause in the next step. Note, incidentally, that the two framed literals of C_9 will be striped when C_9 is taken as a goal in the next step because the indices of them are greater than or equal to the index 1 of the first literal of C_9.

5. The constraints on common tails

It is noted that, in the hierarchical deduction, the index of the literal of a resolvent shows the ancestors from which it comes. This property results in two constraints.

Definition: Tail index. Let C be a properly indexed clause, then each index of C which is less than the index of the first (left-most) literal of C is called a *tail index* of C.

Definition: Common tail index. Let C1 and C2 be properly indexed clauses. The index that is both a tail index of C1 and a tail index of C2 is called a *common tail index* of C1 and C2.

Using the above definitions, we can state the constraint on common tail indices as following:

> Let G be a goal clause, C be a rule clause, then any legal resolvent of G against C must not be produced by resolving upon a literal of C whose index is a common tail index of G and C.

Example 5.1. Consider following clauses

C1: ${}_2R\ {}_1L$
C2: ${}_3L\ {}_2R\ {}_1P$
C3: ${}_4{\sim}L\ {}_2R\ {}_1P$

According to the constraint on common tail indices, there is no legal resolvent of C3 against C1, because the literal L in C1 has index 1 which is a common tail index of C3 and C1. Whereas, we can obtain a legal resolvent of C3 against C2:

C5: ${}_2R\ {}_1P$

Definition: Common tails. Let C1 and C2 be properly indexed clauses. Then the subclause C1' obtained from C1 by deleting all literals in C1 with an index which is not among the common tail indices of C1 and C2 is called the *common tail* of C1 against C2.

Definition: Common tail mergeable. Let C1 and C2 be properly indexed clauses. Let C1' be the common tail of C1 against C2 and C2' be the common tail of C2 against C1. If there is a most general substitution θ, such that C1'θ is identical to C2'θ (including order and indices), then C1 and C2 is called *common tail mergeable* and θ is called the common tail unifier of C1 and C2. Otherwise C1 and C2 is *common tail unmergeable.*

The test for common tail mergeable can be done efficiently by using the standard unification algorithm.

The following are some examples, in which the clauses are taken from proofs of IMV and AM8 where the constraint on common tail mergeable is used.

Example 5.2

C_7: ${}_3P(dz)\ {}_2{\sim}P(g(z)x)\ {}_2P(g(z)h(x))\ {}_1{\sim}P(0f(x))$

C_9: ${}_3{\sim}P(yh(y))\ {}_2{\sim}P(yu)\ {}_2P(yh(u))\ {}_1P(0f(u))$

Let C_9 be the goal clause, C_7 be a rule clause. The mgu θ for unifying the first literal of C_9 with the negation of the literal P(dz) in C_7 is

$\{\theta = \{d/y, h(d)/z\}$

So the common tails of their resulting clauses are

${}_2{\sim}P(g(h(d))x)\ {}_2P(g(h(d))h(x))\ {}_1{\sim}P(0f(x))$
${}_2{\sim}P(du)\ {}_2P(dh(u))\ {}_1P(0f(u))$

Obviously, they are not unifiable, so we say that the resulting clauses of C_9 and C_7 in this resolution are common tail unmergeable.

C_3: ${}_2{\sim}P(f(x)y)\ {}_2{\sim}P(yf(k(x)))\ {}_1{\sim}P(ax)\ {}_1{\sim}P(xb))$

C_8: ${}_5P(f(l)f(k(a)))\ {}_5P(k(a)l)\ {}_1{\sim}P(aa)\ {}_1{\sim}P(ab))$

Let C_8 be the goal clause, C_3 be a rule clause. The mgu for unifying the first literal of C_8 with the negation of the literal ~P(yf(k(x) in C_3 is

$\theta = \{f(l)/y, a/x\}$

So the common tails of their resulting clauses are

$_1{\sim}P(aa)\ _1{\sim}P(ab))$
$_1{\sim}P(aa)\ _1{\sim}P(ab))$.

They are unifiable. So the resulting clauses of C_8 and C_3 in this resolution are common tail mergeable. However, the resulting clauses of C_8 and C_3 in the resolution upon the first literal of C_8 and the first literal of C_3 will be common tail unmergeable.

6. Hierarchical resolvent

Combining the constraint on the framed literal and the constraints on common tails, we now give a complete definition of the hierarchical resolvent (H-resolvent).

Definition: H-resolvent. Let G, C be properly indexed clauses with no variable in common. Let L_g be the first literal of G and L_c be a literal of C. A clause H is called an *H-resolvent* of G against C in level n iff all of following conditions are satisfied:

1. The index of L_c is not among the common tail indices of G and C;
2. L_g and $\sim L_c$ are unifiable; let θ be their most general unifier;
3. $G\theta$ and $C\theta$ are common tail mergeable; let λ be the common tail unifier;
4. Let C' be the clause obtained from C by deleting the literal L_c and changing all indices of remaining literals to be the integer n. Let G' be the clause obtained from G by deleting its first literal L_g. Let H be the clause obtained by concatenating $C'\theta\lambda$ to the left of $G'\theta\lambda$ and merging the literals of $C'\theta\lambda$ into $G'\theta\lambda$.[6] The literals inherited from clause G keep their original order as in G; but the order of the remaining literals from $C'\theta\lambda$ (those on the left not merged) is unimportant;
5. Let FL_g be the set of framed literals of G. Then the set of framed literals of H is set:

   $(FL_g \cup \{L_g\})\theta\lambda$;

   There must be no atom in the framed literals of H which is identical to an atom of a non framed literal of H;
6. H is not a tautology.

Stipulation: A clause H will lose the framed literals whose index is greater than or equal to the index of the first literal of H when H is selected by the procedure as a goal clause.

Consider the clause C_3 and C_8 in the above example (Example 5.2),

$$_6{\sim}P(f(a)f(l))\ _5P(k(a)l)\ _1{\sim}P(aa)\ _1P(ab)$$

is an H-resolvent of C_8 against C_3 in level 6.

To be complete, hierarchical deduction also needs the resolvents obtained by factoring. We design here for hierarchical deduction a special factoring algorithm which can result in fewer factors.

[6]This means the literals to be merged are only those in clause $C'\theta\lambda$, we call it restricted merging.

Definition: H-factor. Let H be an H-resolvent of a goal clause G against a rule clause C, FL_h is the set of framed literals of H. Let L_c be a literal of H coming from C, L_g be a literal of H coming from G. If L_c and L_g are unifiable and θ is the most general unifier, then the clause H' obtained from $H\theta$ by merging the literals of H' coming from the rule clause C into the literals of H' coming from the goal clause G is called an hierarchical factor (*H-factor*) of H if H' is not a tautology and no literal of H' shares a same atom with a literal in the set $FL_h\theta$. The set of framed literals of H' will be $FL_h\theta$.

By including H-factors into H-resolvents, we can make the local subsumption test more strict, such that an H-resolvent or an H-factor will be local subsumed if there is more than one literal including the framed literals sharing the same atom. And if this is done, then the tautology test is included in the local subsumption test.

7. Hierarchical deduction procedure, H1-DEDUC

Now we modify H0-DEDUC to be a complete theorem prover for the first order logic. We call this procedure H1-DEDUC. Mainly, H1-DEDUC is different from H0-DEDUC by the following:

1. The general version of hierarchical resolvent, H-resolvent is used;
2. The name (integer) of a node will be assigned by a subroutine GETNODE. For each call of it, a global variable, NODENUMBER, will increase by 1 and its value is the name of the next new node. The initial value of NODENUMBER is 0. (Like H0-DEDUC, the first node inputted to H1-DEDUC is 1 and its pointer is 0.)
3. The rule cut subroutine in H0-DEDUC, RS := RS - RS^L - RS^~L, is excluded by H1-DEDUC.
4. The goal clauses of the input set are also included in the RULES field of the first node. The resolvents newly produced are stored both as rule clauses and goal clauses in the next produced node.
5. A local depth limit described below is used to prevent infinite search.

Definition: The *local depth limit* is an additional constraint to a legal H-resolvent. This is the maximum number of the framed literals allowed for a legal H-resolvent.

For a set S of ground clauses, it is easy to verify that a local depth limit greater than the number of different atoms of the set S will not cause incompleteness. But in the case of general clauses, a small local depth limit may cause incompleteness. So, when we consider the properties of H1-DEDUC, we will suppose that the local depth limit assigned is large enough for the particular theorem being proved.[7]

The following is the definition of the procedure H1-DEDUC.

PROCEDURE H1-DEDUC(node)

```
Step 1 If node = 0 then exit with "FAIL"
       GS := GOALS-GET(node)
       IF GS = NIL then return H1-DEDUC(POINTER-GET(node))
```

[7]Of course, no depth limit is large enough to handle all theorems of first order logic. In practice we repeatedly call H1-DEDUC on a theorem with higher and higher depth limit until a proof is found, or until the search is abandoned.

```
Step 2 G := REORDER(FIRST(GS))[8]
       If G = NIL then exit with "box"
       L := FIRST(G)
       I := the index of L
       GOALS-MAKE(node REST(GS))
       RS := RULES-GET(I)

Step 3 nextnode := GETNODE()
       GS := {H: H is an H-resolvent[9] of G against C in level nextnode, C ∈ RS}

Step 4 nextnode <= (node G (GS ∪ RS)  GS)
       Return H1-DEDUC(nextnode)
```

Theorem 2: H1-DEDUC is a complete deduction procedure for first order logic.

The precise description of this theorem and the proof of this theorem are listed in Appendix.

Example 7.1 Using H2-DEDUC to prove the following set of clauses. In the listing shown below, we suppose the H-factors of the H-resolvents are not included, and suppose a local depth limit, say 3, is used to prevent infinite search.

R_1: ${}_1P(xa)$ ${}_1P(f(x)x)$	Given
R_2: ${}_1P(xa)$ ${}_1P(xf(x))$	Given
R_3: ${}_1{\sim}P(xy)$ ${}_1P(yf(y))$	Given
R_4: ${}_1{\sim}P(xy)$ ${}_1P(f(y)y)$	Given
R_5: ${}_1{\sim}P(xa)$ ${}_1{\sim}P(yx)$	Given

Node 1 <= (0 NIL (R_1 R_2 R_3 R_4 R_5) (R_5))

Goal: R_5, nextnode: 2.

C_1: ${}_2P(f(x)x)$ ${}_1{\sim}P(yx)$ [${}_1{\sim}P(xa)$]	R_5,R_1
C_2: ${}_2P(aa)$ ${}_1{\sim}P(yf(a))$ [${}_1{\sim}P(f(a)(a))$]	R_5,R_1
C_3: ${}_2P(xf(x))$ ${}_1{\sim}P(yx)$ [${}_1{\sim}P(xa)$]	R_5,R_2
C_4: ${}_2{\sim}P(xa)$ ${}_1{\sim}P(yf(a))$ [${}_1{\sim}P(f(a)(a))$]	R_5,R_4

Node 2 <= (1 R5 (C_1 C_2 C_3 C_4 R_1 R_2 R_3 R_4 R_5) (C_1 C_2 C_3 C_4))

Goal: C_1, nextnode: 3.

C_5: illegal resolvent of C_1 against C_4
/* The resulted clauses of C_1 and C_4 are common tail unmergeable. */

C_6: ${}_7P(xf(x))$ ${}_1{\sim}P(yx)$ [${}_2P(f(x)x)$] [${}_1{\sim}P(xa)$] C_1,R_3

C_7: ${}_7P(f(x)x)$ ${}_1{\sim}P(yx)$ [${}_2P(f(x)x)$] [${}_1{\sim}P(xa)$] C_1,R_4
/* C_7 is local subsumed. */

[8]The function REORDER reorders only those literals of GS which have the largest index in G. A detailed description of this function will be given in section 13. Temporarily, we let REORDER(G) = G.

[9]It is optional whether the H-factors are included in H-resolvents. But for completeness, all H-factors of an H-resolvent must be included.

C_8: $_3{\sim}P(zf(a))$ $_1{\sim}P(ya)$ $[_2P(f(a)a)]$ $[_1{\sim}P(aa)]$ C1,R5

C_9: $_3{\sim}P(xa)$ $_1{\sim}P(yx)$ $[_2P(f(x)x)]$ $[_1{\sim}P(xa)]$ C1,R5
/* C_9 is local subsumed. */

Node 3 <= (2 C_1 (C_6 C_8 C_1 C_2 C_3 C_4 R_1 R_2 R_3 R_4 R_5) (C_6 C_8))

/* C_6 and C_8 are not useful goal clauses. The procedure will
back up. We suppose the procedure has backtracked to node 2 */

Goal: C_2, nextnode: 9.

C_{10}: $_1{\sim}P(yf(a))$ $[_2P(aa)]$ $[_1{\sim}P(f(a)a)]$ C_2,C_4

/* next section, we will show that C_{10} is a correct reduction of the goal clause C_2. If a correct reduction is obtained, we can retain only this reduction resolvent in this round of deduction. */

Node 9 <= (2 C_2 (C_{10} C_1 C_2 C_3 C_4 R_1 R_2 R_3 R_4 R_5) (C_{10}))

Goal: C_9, nextnode: 10.

C_{11}: $_{10}P(f(a)a)$ $[_1{\sim}P(f(f(a))f(a))]$ C_{10},R_1
C_{12}: $_{10}P(aa)$ $[_1{\sim}P(af(a))]$ C_{10},R_2
C_{13}: $_{10}{\sim}P(xa)$ $[_1{\sim}P(af(a))]$ C_{10},R_3
C_{14}: $_{10}{\sim}P(xf(a))$ $[_1{\sim}P(f(f(a))f(a))]$ C_{10},R_4
C_{15}: $_{10}{\sim}P(f(a)a)$ $[_1{\sim}P(yf(a))]$ C_{10},R_5

Node 10 <= (9 (C_{11} C_{12} C_{13} C_{14} C_{15} R_1 R_2 R_3 R_4 R_5) (C_{11} C_{12} C_{13} C_{14} C_{15}))

Goal: C_{11}.

C_{16}: NIL C_{11},C_{15}
/* C_{16} is a correct reduction of the goal C_{11} */

Goal: C_{16}. Return "box".

Comment Several redundant resolvents were avoided by the constraints on common tail indices. For example, the goal clause C_1 could not produce any resolvent by resolving against C_2, C_3 and C_4 upon their last literals, because their indices are among the common tail indices with C_1.

There are two cases of discarding redundancy by the local subsumption test: the deletion of the resolvents C_7 and C_9; one case of discarding redundancy by the constraint of common tail mergeable: the deletion of the resolvent C_5.

Note the hierarchical deduction allows the procedure to "forget" some previous rule clauses when the goal clause obtains a reduction[10]. In this example, the goal clause C_{10} retrieved node 1 to pick the rule

[10]Reduction will be defined precisely in next section.

clauses to be used. All other resolvents produced before were "forgotten".

8. Correct reduction

In this and following sections, we will make a series of refinements for the hierarchical deduction procedure. The first refinement is related to the so-called correct reduction.

An H-resolvent H of a goal G is called a reduction of G iff H is an instance (including indices) of a proper part of G. Otherwise H is called an extension of G.

Definition: Correct reduction. A reduction H is called a *correct reduction* of a goal clause G iff H is a variant (including indices) of a proper part of G.

Because of the restricted merging used in the definition of H-resolvent, each H-resolvent of a goal G must contain a subclause which is an instance of REST(G). Thus the correct reduction of G must be a most general instance of REST(G). So we have the following conclusion:

Proposition 3: If a resolvent H is a correct reduction of a goal clause G in the hierarchical deduction, then all H-resolvents producible from G by the procedure must be subsumed by H.

For a ground proof, it is easy to verify that any reduction of a goal clause G must be a correct reduction of G. For a general proof, the correct reduction can be detected efficiently.

Once a correct reduction is obtained from a goal G, all other resolvents produced from G can be discarded without losing completeness.

In Example 7.1, the resolvent, C_{10} is in fact a correct reduction of the goal clause C_2. So we can retain only C_{10} as a legal H-resolvent in this round of deduction.

9. Learning algorithm

It is noticed that simply testing whether a newly produced resolvent is subsumed by an input clause or a resolvent produced before may not increase the efficiency of a prover, because there may be thousands of resolvents produced in proving a hard theorem even by hierarchical deduction, and there is so little probability that a resolvent is subsumed by a previously produced resolvent. And also note that the deletion of resolvents in this way may cause serious incompleteness.

However, the hierarchical structure has a feature that the resolvents grouped in the same node are similar to each other in the sense that the tails of these resolvents are all instances of a same part of a goal clause. It is based on this property that we can design an effective redundancy test method for the hierarchical deduction. This method is called a learning algorithm.

To each node, we add one more field, FAILED. The clauses stored in this field of a node are called *failed goals*. Initially, the FAILED field of the first node, node 1, is empty. Before a goal clause G is to be resolved, it is first tested to determine whether it is subsumed by a failed goal stored in the same node as that from which the rule clause for goal G is obtained (suppose this node is n). If so, the goal clause is simply discarded. Otherwise we add G into the FAILED field of the node n. The FAILED field of the node being produced from the goal G is a copy of the FAILED field of the node n.

It is noticed that the hierarchical deduction usually does not produce many subsumed resolvents. In order to avoid spending much time in the subsumption test, the procedure will not do the usually subsumption test, instead, a special equality test is used: for each failed goal clause G_f stored in the related node, the goal clause G is only tested to determine whether G_f is a variant of a part of G. If it is, delete G.

This learning algorithm increases the efficiency of the prover in some proofs, where the procedure had to backtrack from wrong paths several times. For example, in a proof of IMV by the procedure H1-DEDUC, more than 300 legal resolvents were retained before "box" was produced, among which 48 resolvents were useful. By the learning algorithm, the procedure discarded 24 unprovable goals, many of which, if used as goals, would need many steps to be "recognized" unprovable by the procedure. Because the failed goals are recorded only in some relative nodes, there are usually few failed goals applicable in the redundancy test. For IMV, the biggest number of failed goals stored in a node is less than 8. In all the examples that have been tried by our procedure, the expense of the learning algorithm is usually a small percentage of that of the whole computation.

It is well known that a proof of a theorem of first order logic can be viewed to be a proof on some set of ground clauses. We have shown, (see the proof of Theorem I.2 of appendix) by the basic hierarchical deduction H0-DEDUC, that the number of different literals contained in a node must be greater than the number in its descendant node. From this, we can deduce that, for the general case, a literal which is hard to prove in a node, must also be hard to prove in its descendant node. So if a goal clause is subsumed by a failed goal stored in the related node, then this clause will also fail. So we believe the learning algorithm (redundancy test) used in the hierarchical deduction will not cause incompleteness.

10. Partial set of support and locking

To prove theorems by a goal oriented prover, some additional constraints based on conjectures instead of completeness proofs were discussed in literature. For example, the local depth limit used in H1-DEDUC mentioned in section 7 is one of these additional constraints, which is the same as the A-literal maximum restriction used by the Model Elimination system in [10].

In this section, we propose an additional constrictive method to hierarchical deduction. The main idea is that we will "lock" some of the literals of the input clauses, such that the locked literals will never be resolved upon during the deduction. The term "locking" was introduced by Boyer in [5], but it is used by us with some changes.

First we use an example to illustrate this method.

Example 10.1. The set S of clauses given here is obtained by skolemizing the theorem AM8'. We agree that a literal that is indexed by 0 is locked (Recall all literals of input clauses are indexed by 1 in the previous discussion). A locked literal will be unlocked when it is inherited by the H-resolvent (i.e., its index will be changed from 0 to the next node name same as before). G1 is used as the first goal clause. The following listing is from the output obtained by our hierarchical prover.

R1:	$_1P(xx)$	given
R2:	$_1P(xz)\ _0{\sim}P(xy)\ _0{\sim}P(zy)$	given
R3:	$_1P(xy)\ _0P(yx)$	given
R4:	$_1P(f(x)f(y))\ _0{\sim}P(xy)\ _0{\sim}P(yx)$	given
R5:	$_1P(f(q(y))f(y))$	given
R6:	$_1P(q(x)y)\ _0{\sim}P(f(y)f(x))$	given
R7:	$_1P(f(d)f(x))\ _1P(dx)$	given

R8:	$_{1}P(h(x)x)\ _{1}P(xd)$	given
R9:	$_{0}{\sim}P(f(x)f(h(x)))\ _{1}P(xd)$	given
G1:	$_{1}{\sim}P(f(x)f(k(x)))$	given
C1:	$_{2}{\sim}P(y(f(k(x))))\ _{2}{\sim}P(f(x)y)$	G1,R2
C3:	$_{2}{\sim}P(f(x)f(q(k(x))))$	C1,R5
C7:	$_{5}P(dq(k(d)))$	C3,R7
C10:	$_{5}{\sim}P(x(q(k(x))))\ _{5}{\sim}P(q(k(x))x)$	C3,R4
C11:	$_{5}{\sim}P(q(k(d))d)$	C10,C7
C12:	$_{7}P(h(q(k(d)))q(k(d)))$	C11,R8
C13:	$_{7}{\sim}P(f(q(k(d)))f(h(q(k(d)))))$	C11,R9
C15:	$_{8}{\sim}P(f(q(k(d)))x)\ _{8}{\sim}P(xf(h(q(k(d)))))$	C13,R2
C17:	$_{8}{\sim}P(q(k(d))h(q(k(d))))\ _{8}{\sim}P(h(q(k(d)))q(k(d)))$	C13,R4
C18:	$_{9}{\sim}P(f(h(q(k(d))))f(k(d)))\ _{8}{\sim}P(h(q(k(d)))q(k(d)))$	C17,R6
C21:	$_{10}P(f(k(d))f(h(q(k(d)))))\ _{8}{\sim}P(h(q(k(d)))q(k(d)))$	C18,R3
C35:	$_{17}{\sim}P(f(q(k(d)))f(k(d)))\ _{8}{\sim}P(h(q(k(d)))q(k(d)))$	C21,C15
C36:	$_{8}{\sim}P(h(q(k(d)))q(k(d)))$	C35,R5
C37:	NIL	C36,C12

This theorem was first proved automatically, as a hard theorem, by Tyson's APRVR prover [15]. In contrast, the hierarchical deduction proved this theorem more efficiently: Total 37 resolvents were produced to obtain the proof.

It is noticed, for a goal oriented procedure, most of the resolvents are produced by resolving the goal clauses against the input clauses. In this example, if the "locking" were not used, then whenever there is a goal clause whose first literal is positive, there would be at least 4 possible resolvents due to the transitivity axiom R2 and the equality axiom R4. Actually, there is a class of theorems, which are not Horn theorems, but the sets of clauses obtained from them are almost Horn sets in the sense that, though some clauses of the sets contain more than one positive literals, most of their literals are negative. In proving these theorems by a goal oriented prover, some goal clauses containing positive literals are usually needed. If the negative literals of input clauses are not "locked", a large number of resolvents (branches) can be obtained whenever a positive goal literal is resolvent upon.

Certainly, the key problem of using locking is to preserving completeness. Unfortunately, there is no locking method generally applicable[11], so this restriction has to be implemented like the A-literal maximum restriction (local depth limit in our terminology): Lock a set of literals of input clauses first. If no proof can be obtained, release some locks, then restart the process until a proof is obtained or all locks are released.

Next, we propose a lock assignment method, and discuss the issue concerning completeness.

The idea of designing a hierarchical deduction procedure is closely related to the using set of support strategy in a goal oriented prover. Set of support strategy introduced by Wos and Robinson is acknowledged as a useful constraint for resolution style theorem provers. But, this strategy can hardly be used in a linear like resolution method. The reason is that linear resolution can retain at most one descendant clause from a goal clause; but, the set of support strategy, in order to be complete, requires that all resolvents producible from any goal clause during the deduction process be retained. For example, suppose S is a minimal unsatisfiable set of clauses, C is a goal clause and M is a model of the set M - {C}

[11] Here we mean "locking" in the restricted sense we are considering. Of course, the general locking procedure of Boyer [5] is complete.

(in this case C must be false in M), suppose, in order to use set of support, we put a restriction on linear resolution in proving S, such that each goal clause used in the deduction must be false in M. Obviously, this restriction may cause severe incompleteness if S is a non Horn set, because the procedure will behave as input resolution.

We have mentioned before, the hierarchical deduction allows the retaining of all legal resolvents producible from a goal clause. By this feature, we expect a better use of the set of support strategy. Certainly, the full use as described above may still cause incompleteness to the prover, but the situation will be better. This can be confirmed by the following proposition:

> **Proposition 4:** If S is a full literal set of ground clauses[12], and S is minimally unsatisfiable, then if M is a model of S - {C}, where C is the first goal clause used in H0-DEDUC to prove S, there is a proof by H0-DEDUC, such that each goal clause used in the deduction is false in M.

This proposition can be verified by noticing that, the first node produced by the hierarchical deduction in proving the set S described in the proposition, must contain one and only one H0-resolvent which is false in M, then this resolvent can be used as next goal to obtain a new node, which must have a similar property as its parent node.

But if S is not a full literal set, the above proposition may be not true, because, in this case, there may be more than one resolvents false in M that are produced from one goal clause, but only one of them can be used in a deduction path of the hierarchical deduction.

In order to obtain an effective restriction without severely losing completeness, we will use set of support only on the input clauses. Thus, our use of the set of support is called *partial set of support strategy.*

We use the term "setting" suggested by Loveland in [11] to describe a consistent set of literals from the Herbrand's universe of a given set of clauses. Setting is defined originally in [11] as follows:

Definition: Given a set S of clauses, a *setting* M for S is a (possibly empty) set of literal satisfying the following conditions:

1. every literal of M is an instance[13] of a literal of S or an instance of the complement of a literal of S;
2. if $L \in M$ then for every instance L_1 of L, $L_1 \in M$;
3. M does not contain a complementary pair of literals.

Given a setting M of a set S of clauses, a literal L of S is true in M iff all of the instances of L are subsumed by some elements in M; L is false in M iff all instances of the complement of L (denoted by ~L) are subsumed by some elements in M; L is called unknown to M iff L is neither true nor false in M. A clause C of S is true in M iff C contains a literal that is true in M; C is false in M iff all literals of C are false in M, otherwise C is unknown to M.

For the set of clauses obtained from AM8', shown in example 10.1, we can use the set M = {P(xy)} as a setting (The reader can find more examples concerning settings in [11]).

By using the term setting, we now describe our partial set of support strategy (for hierarchical

[12] By the full literal set, we mean that each clause contains all atoms of the whole set, such as the set {PvQ, Pv~Q, ~PvQ, ~Pv~Q}

[13] the term S-instance was used in [11]

deduction).

Given a set of clauses S and a setting M for S, the use of the partial set M of support strategy in the hierarchical deduction for proving S consists of the following additional requirements:

1. All goal clauses of the first node must not be true (i.e., false or unknown) in M.
2. No H-resolvent is produced by a goal clause G against an input clause C, such that the clause C is true in M and the literal of C resolved upon is false in M.

Because only the resolution of the goal clauses against the input clauses are restricted in the partial set of support strategy, then instead of the True/False test, we can use "locking" to efficiently implement this strategy.

Let M be a setting for a set S of clauses to be proved by hierarchical deduction. Then, for each clause C of S, if C is true in M, we lock all false literals of C. Then, the restriction of using partial set of support is equivalent to the requirement that the locked literals of S never be resolved upon during the deduction. We call this lock assignment method *full locking according to the setting* M.

A setting M for a given set S is called *reliable* if the use of full locking according to M will not cause incompleteness. Because there is no general rule guaranteeing completeness, we will discuss the reliability relatively. Generally, the fewer the number of clauses of S that are false in the setting M, the more reliable is the setting. (In this sense, the empty setting is perfectly reliable, because it results no locking at all.) A setting M for a given set S is more *effective* if the use of full locking according to M results in more literals of S being locked.

In practice, we will expect to find such a setting M for a given set S, such that all clauses of S but one are true in M (if all clauses are true in M, then we have proved a non theorem). Obviously, the setting {P(xy)} for the example 10.1 has such property. But, for some unsatisfiable sets of clauses, such a setting is hard to find or does not exist. In this case, we have to balance the considerations of both the reliability and effectiveness. Generally, a setting in which more than one input clauses are false may be unreliable. Next we will give an example (Example 10.2).

Because the goal of locking input literals is to reduce the branching factor, then the important thing is to lock the input literals whose terms are variables, such as the literals in the clauses R2, R3 and R4 of the above example. The locks for the literals with complex structure, such as the literal ~P(f(x)f(h(x))) in R9, can be released, because these literals are usually hard to unify, and a successful unification upon such literals may lead to a useful resolvent.

In summary, the use method of the partial set of support strategy can be described as following:

1. For a set S of clauses, find a setting M (possibly, with a new selection of goal clause), in which the balancing of reliability and effectiveness is considered.
2. For each clause of S, if the clause contains a literal true in M, then lock all literals of this clause which are false in M;
3. Release (optionally) some locked literals which have complex structure;
4. Try to prove the theorem. If it fails, release some locks (or redefine a setting), and then restart the proof.

It is worth noticing that the lock assignment also provides a way to incorporate expert knowledge in theorem proving by hierarchical deduction. Let us examine the following set of clauses which is Example 5 in [10].

Example 10.2 In an ordered field, if x>0, then x^{-1}>0.

R1: $_1$P(xex)
R2: $_1$~P(ee0)
R3: $_1$S(a)
R4: $_0$~P(xyz) $_1$P(m(x)m(y)z)
R5: $_1$P(xyz) $_0$~P(m(x)m(y)z)
R6: $_1$P(xm(y)m(z)) $_0$~P(xyz)
R7: $_1$P(xi(x)e) $_0$P(xx0)
R8: $_1$~S(m(x)) $_1$~S(x)
R9: $_1$~S(x) $_1$~P(xx0)
R10: $_1$P(xyz) $_0$~P(yxz)
R11: $_1$S(x) $_1$S(m(x)) $_0$P(xx0)
R12: $_0$P(xx0) $_1$~P(yy0) $_0$~P(yzx)
R13: $_1$S(x) $_1$~P(yzx) $_1$~S(y) $_1$~S(z)
G1: $_1$~S(i(a))

A setting M containing a literal S(x) or ~S(x) appears to be unreliable because there will be more than one clauses of the above set that is false in M. To be reliable, let the setting be

{S(a), S(m(x)), ~P(xx0), P(xyz)-P(uu0)},

where P(xyz)-P(uu0) means all instances of P(uu0) are excluded from the instance set of P(xyz).

According to M, we can lock the literals P(xx0) in R7, R11 and R12. The literal ~P(yxz) of R10 can be locked because of symmetry.

We can try to lock some of literals in the clauses R4, R5, R6 and R12, by assuming (guessing) that all instances of these literals to be locked, which are subsumed by P(uu0), are not needed to prove this theorem.

The resulting locks for the above set of clauses are indicated by the indices "0". The following listing is obtained from the output of our hierarchical prover in proving this theorem under these locks.

G1:	$_1$~S(i(a))	given
C1:	$_2$P(i(a)i(a)0) $_2$S(m(i(a))	G1,R11
C2:	$_2$~P(yzi(a)) $_2$~S(y) $_2$~S(z)	G1,R13
C3:	$_3$P(xx0) $_3$~P(i(a)zx) $_2$S(m(i(a))	C1,R12
C6:	$_4$~P(xi(a)y) $_3$P(yy0) $_2$S(m(i(a))	C3,R10
C8:	$_5$P(aa0) $_3$P(ee0) $_2$S(m(i(a)))	C6,R7
C9:	$_6$~S(a) $_3$P(ee0) $_2$S(m(i(a)))	C8,R9
C12:	$_3$P(ee0) $_2$S(m(i(a)))	C9,R3
C13:	$_2$S(m(i(a)))	C12,R2
C14:	$_7$~P(m(i(a))zi(a)) $_7$~S(z)	C13,C2
C19:	$_7$S(x) $_7$~P(ym(i(a))x) $_7$~S(y)	C13,R13
C27:	$_{10}$~P(xi(a)z) $_7$S(m(z)) $_7$~S(x)	C19,R6
C30:	$_{11}$P(aa0) $_7$S(m(e)) $_7$~S(a)	C27,R7
C32:	$_7$S(m(e)) $_7$~S(a)	C30,R9
C33:	$_7$S(m(e))	C32,R3
C37:	$_{12}$~P(m(i(a))m(e)i(a))	C33,C14
C46:	$_{14}$~P(i(a)ei(a))	C37,R4

```
C48: NIL                                        C46,R1
```

11. Relation to Model Elimination and SL-resolution

Several refinements have been made use of linear resolution. Among them, Loveland's Model Elimination (ME) is a well known system, where the information provided by the literals previously resolved upon (A-literals) was used (mainly for obtaining lemmas).

Kowalski and Kuehner strengthened the usage of the A-literals in their SL-resolution, where the A-literals were used both for obtaining lemmas and for restricting redundancy.

The framed literals used in our hierarchical deduction are quite similar to the A-literals used in ME and SL-resolution. But, for hierarchical deduction, framed literals are only used for restricting redundancy. It is interesting to note that, for us, the use of the framed literals was based on the basic hierarchical deduction structure, corresponding to the feature that, for the ground case, the rule cut routine can be used without losing completeness.

The main difference between hierarchical deduction and ME or SL-resolution is the deduction schema. As was pointed in the beginning of this paper, ME and SL-resolution use one-to-one type of deduction schema, whereas hierarchical deduction uses one-to-many type of deduction schema. One of our primary motivation is that, in order to make control to a goal oriented prover by using heuristics and expert knowledge (the control will be discussed in the following sections), comparisons between the possible paths (branches) are needed. It turns out, anyway, we have to produce and, so to compare, resolvents producible from each goal clause (except the case, where some particular heuristics are available). On the one hand, there may be more than one resolvent that is useful in obtaining a proof; on the other hand, the resolutions between these resolvents usually do not produce many redundant resolvents because the variables in resolvents are usually bounded by previous substitutions, and the constraints on common tails of hierarchical deduction act powerfully to restrict the redundancy producible by these resolutions. So, instead of using these resolvents for backtracking only, the hierarchical deduction retains and uses these resolvents.

Because the constraints on common tails are only used for restricting the redundancy producible by resolutions between the resolvents later produced, and since, it is noticed, the resolutions between resolvents were excluded by SL-resolution, it follows that the constraints on common tails are particular to the hierarchical deduction. Certainly, using "lemmas" (the factoring between the B-literals and the negation of A-literals in a chain), as is done in ME and SL-resolution, is an efficient way to prevent redundancy and obtain useful resolvents.

For hierarchical deduction, the deduction path needed for a proof will be further shortened. In addition, by aggregating the resolvents with similar tails into the same node, the efficient redundancy test (learning algorithm) is available.

The important benefit of the hierarchical deduction structure is the use of the partial set of support strategy (locking), because it can greatly increase the power (efficiency) of hierarchical prover for proving some non Horn theorems. Our experiments show that, some theorems, such as IMV and AM8 (the original version of AM8'), would be difficult to prove by hierarchical deduction if the locking were not used (we believe that these theorems are also difficult to ME or SL-resolution because they are similar in power to the hierarchical deduction without locking).

We have discussed the reliability of the setting selection and locking in the last section. It is worth noticing that the same locking method for the same input set of clauses will be less reliable for SL-resolution than that for hierarchical deduction. Let us examine the following unsatisfiable sets of clauses:

Example 11.1

C1: ${}_1Q\ {}_1R$	given
C2: ${}_1P\ {}_0{\sim}Q$	given
C3: ${}_1P\ {}_0{\sim}R$	given
C4: ${}_1{\sim}P$	given

Suppose that the setting M = {P,Q,R} is used, and, by full locking, the literal ~Q of C2 and ~R of C3 are locked (indicated by the index 0). Then, the hierarchical deduction can obtain "box" by following deduction path:

C5: ${}_2{\sim}Q\ [{}_1{\sim}P]$	C4,C2
C6: ${}_2{\sim}R\ [{}_1{\sim}P]$	C4,C3
C7: ${}_3R\ [{}_2{\sim}Q]\ [{}_1{\sim}P]$	C5,C1
C8: NIL	C7,C6

Now we apply SL-resolution on the same set of clause, starting from the same goal clause C4 and under the same setting and full locking.[14] There is a deduction path similar to the above until the resolvent C7 is obtained. But from goal clause C7, there will be no resolvent producible, because the literal ~R in the input clauses is locked, and, by SL-resolution, C7 can not be resolved against C6 because it is a resolvent (in fact, C6 is not in this deduction path). The other deduction paths by the SL-resolution will also fail because of the similar problem.

Example 11.2 Let us take a look again to Example 10.1 of the last section. Under the similar condition of the Example 10.1, we can obtain a similar deduction path by SL-resolution until the resolvent C21 is obtained (except that one more steps of expansion will be needed). Now let us written down some useful framed literals (some of them are still not written explicitly for simplicity) of C21.

C21: ${}_{10}$P(f(k(d))f(h(q(k(d))))) ${}_8$~P(h(q(k(d)))q(h(d)))
[${}_7$~P(f(q(k(d)))f(h(q(k(d)))))]

where the framed literal is just one of the A-literals of SL-resolution useful for obtaining the next reduction.

Suppose that the locking is not used. Then by SL-resolution, we can continue the deduction from C21 as follows:

S22: P(xf(h(q(k(d))))) ~P(xf(k(d))) [P(f(k(d))f(h(q(k(d)))))] ~P(h(q(k(d)))q(h(d))) [~P(f(q(k(d)))f(h(q(k(d)))))]	C21,R2
C35: ~P(f(q(k(d)))f(k(d))) [P(f(k(d))f(h(q(k(d)))))] ~P(h(q(k(d)))q(h(d))) [~P(f(q(k(d)))f(h(q(k(d)))))]	reduction of S22

But, if the locking is used, the expansion of C21 against the input clause R2 will be illegal because the literal ${}_0{\sim}$P(zy) of R2 is locked. So, this deduction path will fail. Note, the two terms in the literal ~P(zy) are both variables, releasing the lock of this literal will result in more redundant resolvents (chains) during deduction.

The reason that, for SL-resolution in the above two examples, the expected reductions became impossible due to locking, is that, before the reductions are to be available, the expansions producible only

[14] In the original SL-resolution, the clauses and resolvents are viewed in the reverse order to ours.

by resolving against some input clauses, such that the input clauses are true in M and the literals to be resolvend upon are false in M, are needed, but these literals are locked under the full locking according to M. In general, we have following assertion for SL-resolution:

> **assertion:** If S is a set of clauses being proved by SL-resolution, and the full locking according to a setting M for S is used, then there can be no such a reduction obtained, that the A-literal used for the reduction is true in M and this A-literal is obtained before by resolving against an input clause C upon a literal L of C such that C is true in M and L is false in M.

This assertion can be simply verified by noticing that, for the case described in the assertion, the literal L of the input clause C is locked.

However, this assertion is not generally true for the hierarchical deduction, because hierarchical deduction does not use lemmas, instead, the reductions needed for a proof are usually obtained by resolving a goal clause against a unit input clause or against a resolvent obtained in the previous steps, but, the literals of both the unit clauses and the resolvents will not be locked under any setting according to the definition of the partial set of support strategy.

12. Heuristics on twin symbols

We require that our theorem prover satisfies the following criteria:

1. The basic deduction structure of the procedure is compact and complete.
2. It is able to select promising goals with a fairly high probability, and so to obtain a proof in a reasonable time, for many theorems of moderate complexity by using heuristics depending mainly on the syntactic feature of the theorem to be proved.
3. Its ability to prove hard theorems, is enhanced by employing the heuristics, which depend on particular semantics of the theorems to be proved.

We have done a lot in the last several sections to help our theorem prover to satisfy the first criterion.

The control strategy to be discussed in this paper is mainly to satisfy our second criterion. The control of our hierarchical deduction is realized mainly by the evaluation of the resolvents and the reordering of subgoals of the goal clause. To do this, the procedure uses heuristics which depend mainly on the syntactical features of the set of clauses to be proved. So these heuristics are generally applicable.

As for the third criterion, we are investigating some effective ways to use semantics to help efficiently prove some hard theorems by hierarchical deduction, but we will describe these in some following papers.

Twin symbol

A set of clauses is constructed mainly from logic symbols and non logical symbols. There are two types of non-logical symbols. One is the so called variable, which is treated as universally quantified. another type is a called non-variable symbol consisting of predicate symbols and function symbols (including constant symbols). Among the non-variable symbols, twin symbols to be defined below are particularly interesting to us.

Definition: Twin symbol. If S is a set of clauses, a symbol occurring in S is called a *twin symbol* if it is a predicate symbol or it is a function symbol which has at least two occurrences in S.

Example 12.1 Consider the following set of clauses:

```
(1) : P(a)
(2) : Q(a)
(3) : ~P(x) ~Q(x)
```

The predicate symbol "P" and the function symbol "a" in the above example are both examples of twin symbols.

For the twin symbols of a minimally unsatisfiable set of clauses, we make following requirement:

Twin matches requirement:

Let S be a minimally unsatisfiable set of clauses, and D be any successful refutation of S. If we keep track of the different occurrences of all twin symbols in the set S in the deduction D, and record the matches between the twin symbols in all unifications occurring in the deduction D, then each occurrence of a twin symbol in S must match with a different occurrence of this twin symbol in S at least once during the deduction D.[15]

Example 12.2. For the set of clauses in the above example, suppose we mark different occurrences of twin symbols in the clauses by different indices, so to keep track of the twin symbols and their matches. Then we can obtain the following deduction:

Clause	mgu	matched symbols	rules
(1): $P_1(a_1)$			given
(2): $Q_1(a_2)$			given
(3): $\sim P_2(y) \sim Q_2(y)$			given
(4): $\sim Q_2(a_1)$	$\{a_1/y\}$	(P_1/P_2)	(3),(1)
(5): "box"	$\{\}$	$(Q_1/Q_2, a_2/a_1)$	(4),(2)

Note P_1/P_2, Q_1/Q_2 and a_1/a_2 are all the matches of the different occurrences of twin symbols.

Heuristics on twin symbols

It is well known that unnecessary hypotheses usually degrade the proofs of theorems in automatic theorem proving. So for this discussion, we suppose that there is no extra hypothesis for the theorem to be proved, and suppose that the set of clauses transformed from the theorem is minimally unsatisfiable. Then we will let the prover use the above twin symbol requirement, to control its search for a proof.

Certainly, in some cases, the theorems to be proved may contain unnecessary hypotheses, and in addition, even for a minimally unsatisfiable set of clauses, this requirement is not completely correct. But anyway, we believe that this requirement can serve as a reasonable heuristic which depends only on the syntactical structure of the set of clauses.

Obviously, the fewer the occurrences of a twin symbol, the fewer deduction paths in which one occurrence of this symbol can be matched with an another occurrence of this symbol. So there is a higher probability that a deduction path, by which the twin symbols having fewer occurrences are matched, is correct for a proof. In this sense, making the twin symbols having fewer occurrences match first, is important for finding a correct deduction path.

We call a non-variable symbol a singleton if it has only one occurrence. In comparison with a twin symbol which has more than one occurrence, a singleton contained in a goal clause is usually the hardest

[15] This requirement will not be always true unless we strengthen the condition that the set S be minimally unsatisfiable. The interesting reader may refer to ATP.79, where we have a more detailed discussion about the twin symbols.

symbol to be canceled in a deduction (by the word "cancel" we mean that the singleton will disappear from a resolvent of the goal clause) because a singleton must have fewest occurrence, 1, in the set S. We suppose that to cancel a singleton from the goal clause is also important for finding a proof.

With above discussion as background, we can suggest the following heuristics:

1. A correct deduction path should proceed with the twin symbols matched with each other. Because all twin symbols should be matched at least once in a successful deduction path, then the first match of each twin symbol can serve as a mid-station ahead of the goal "box". The harder the theorem , the more complex is its set of clauses, so the longer the deduction path that is needed for a proof. But then there are usually more twin symbols in it, so more mid-stations are usable to help construct a proof.

2. The more occurrences of twin symbols that are matched, the more promising is the resulting resolvent. The fewer the occurrences of the matched twin symbols in the original set of clauses, the more promising is the resulting resolvent. A resolvent which is obtained with new matches of occurrences of the twin symbols (they are not matched in the previous deduction) is promising.

3. Looking ahead for one or more steps, the resolvent, which if it is used in a following resolution, can lead to a promising resolvent, is promising;

4. A correct deduction path should be nearly terminated if all or most of the occurrences of the twin symbols are matched once (We prefer a shorter proof for an actual theorem).

For simplicity in our earlier prover, we do not require our procedure to record (as shown in example 12.2) accurately the matches between different occurrences of non-variable symbols, but rather record only the total number of matches of these symbol without regard to which occurrence is matched.[16]

Before the deduction is started, we let the prover assign to each non-variable symbol k of the input set S of clauses a value. We call this value the mass of the symbol k, denoted by MASS(k), which is counted as following:

$$\text{MASS(k)} = \text{number of literals in S} \ / \ \text{number of occurrences of k in S}$$

In order to make a rough record of which twin symbol has already been matched once in the previous deduction, we add a new field, TWINSYMS, to each node. Initially, all occurrences of non-variable symbols in the input set of clauses are recorded in the TWINSYMS field of the first node sorted by their masses in descending order (note if there are n occurrences of a non-variable symbol, say k, then there will be n occurrences of k in the TWINSYMS field.) The TWINSYMS field of the newly produced node is first copied from the the TWINSYMS field of the current node, and then modified by the following:

> Obtain (or retrieve) all of the matched twin symbols in the substitution by which this goal clause was obtained. For each matched symbol, delete an occurrence of it from the TWINSYMS field of the node.

Thus the twin symbols contained in the TWINSYMS field of a node must have at least one occurrence which has not been matched in the previous deduction. The information stored in the TWINSYMS fields of nodes will be used in the level subgoal reordering and evaluation subroutines, which will be discussed in the following sections.

[16] Full recording of the twin symbol matches might be required in our later prover.