Efficient Recursion Termination for Function-Free Horn Logic[†]

Wang-chan Wong
Lubomir Bic
Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

Technical Report #86–26

December 18, 1986

[†] This work was supported by the NSF Grant DCR-8503589: The UCI Dataflow Database Project

Contents

Abstract

We present an efficient scheme to terminate infinite recursion in function-free Horn logic. In [BW84], Brough and Walker show that a preorder linear resolution with a *goal termination* strategy is incomplete, i.e. it must miss some answers. Their theory is true if left-recursion is allowed. The crucial assumption underlying Brough and Walker's theory is that the order of literals in a clause should not be altered. This assumption, however, is not necessary in programs that do not contain any extra-logical features such as the 'cut' symbol of Prolog. This is because the order of literals does not affect the correctness of such programs, only their efficiency. In this paper, we show that left-recursion can always be eliminated. The idea is to transform loops of the input set into *safe* loops, that are left-recursion free. Consequently, the goal termination strategy guarantees to always terminate properly with all possible answers; thus, it is complete in the domain of safe loops. We further show that all rules in a safe loop can be transformed into rules that begin with a base literal. This permits the implementation of a simple scheme to carry out the goal termination strategy more efficiently. The basic idea of this scheme is to distribute the history containing all executed goals over assertions, rather than maintaining it as a centralized data structure. This reduces the amount of work performed during execution.

1. Introduction

Function-free logic programming is a powerful tool that can be used to implement first-order deductive databases. One important issue in such programs is to prevent queries involving recursive clauses from entering an infinite loop. The traditional way to handle this problem is to order clauses such that infinite loops are explicitly avoided. However, this approach is undesirable for two reasons. First, it still does not solve the problem when assertions are cyclic. Second, imposing order on clauses destroys the completeness of logic programs.

In [BW84], Brough and Walker study the problem for programs with *preorder* search strategy i.e., top-down left-to-right, as, for example, in Prolog. For such programs, there are two distinct approaches to solve the termination problem: a *goal termination* strategy and a *rule termination* strategy. With *goal termination*, a branch of a derivation tree is terminated if a goal is repeated in its own branch of the derivation tree. With *rule termination*, a branch of the derivation tree is terminated if a rule (clause) with the same instances is repeated in the existing derivation tree. Brough and Walker show that any *preorder* search strategy where termination is based on examining the current partial derivation tree, is incomplete, i.e. it must miss some answers.

The crucial assumption underlying Brough and Walker's theory is that the order of literals in a clause should not be altered. This assumption, however, is not necessary in programs that do not contain any extra-logical features such as the 'cut' symbol of Prolog. This is because the order of literals does not affect the correctness of such programs, only their efficiency.

In this paper, we present an approach that prevents infinite recursion yet always produces *all* possible answers derivable from the given program, i.e., is complete. This approach is to transform all loops into *safe* loops which are left-recursion free. After the transformation, the goal termination strategy can be applied safely and it does not require any particular ordering of clauses. Clauses of a safe loop can further be transformed into clauses that always begin with a base literal. This serves as the basis for a simple scheme that implements the goal termination strategy efficiently.

The organization of this paper is as follows: in Section 2, we introduce the notation of Horn programs that we use in subsequent discussion. In Section 3, we show the problems of preorder search strategy, define the goal termination strategy precisely and show why the goal termination strategy is incomplete. In Section 4, we present the methods to transform a set of clauses into a set of clauses that are *safe* and always begin with a base literal. In Section 5, we show the simple scheme to implement the goal termination strategy and demonstrate it with a complete example.

2. Preliminaries

In this section, we introduce the basic concepts of logic programming and specifically of SL-Resolution as in [KOW79B, KoKu71, ChLe73, WOSLB84].

2.1. Basic Definitions

Definition: A *Horn program* is a collection of *clauses* of the form

$$\neg p_0 \vee \neg p_1 \vee ... \vee \neg p_{n-1} \vee p_n$$

Each p_i is called a *literal* and has the form $p(t_1, ..., t_m)$, where p is a *predicate* symbol and $t_1, ..., t_m$ are *terms*. In a function-free Horn program, terms can only be constants or variables. The literal p_n, which is the only positive literal, is called the *head* of the clause; the remaining literals form its *body*. A clause with an empty body is called an *assertion* and is used to represent explicit facts. Clauses with a non-empty body are called *rules*. Any predicate occurring in a Horn program may be interpreted as a *relation* among the terms of that predicate.

1) $p_1(a, b)$.
2) $p_1(b, c)$.
3) $\neg p_1(X, A) \vee \neg q(A, Y) \vee q(X, Y)$.
4) $\neg p_1(X, Y) \vee q(X, Y)$.
5) $\neg p_2(X, A) \vee \neg l(A, Y) \vee d(X, Y)$.
6) $\neg p_3(X, A) \vee \neg f(A, Y) \vee l(X, Y)$.
7) $\neg d(A, Y) \vee \neg p_3(X, A) \vee f(X, Y)$.
8) $\neg p_4(X, Y) \vee d(X, Y)$.
9) *negated query* $\neg q(a, ?)$.

Figure 1

An Example of Horn Program

Definition: Relations defined by assertions are called *base* relations while relations defined as the head of a rule will be referred to as *virtual* relations. For example, p_1 in Figure 1 is a base relation and q, d, l and f are virtual relations.

Definition: A *loop* is a set of clauses which can be ordered such that the i^{th} clause contains a predicate that matches the head of the $i + 1^{st}$ clause, modulo the number of clauses forming the loop. For example, rules (5), (6) and (7) in Figure 1 form a loop because the predicate l of clause (5) matches the head of clause (6); the predicate f of clause (6) matches the head of clause (7); and the predicate d of clause (7) matches the head of clause (5).

Definition: A rule (and, consequently, the relation it defines) is *recursive* if it is part of a loop. For example, rules (5), (6) and (7) in Figure 1 are all recursive. A rule (relation) is *directly recursive* if it is part of a loop of size one; in other words, a predicate in both a negative and a positive form must exist in the same clause. For example, clause (3) in Figure 1 is directly recursive due to the predicate q.

Definition: A rule is *non-recursive* if it is not part of any loop; its body comprises only base and/or other non-recursive relations. For example, clauses (4) and (8) in Figure 1 are non-recursive.

(1) $\neg p(X, A) \vee \neg q(A, Y) \vee q(X, Y)$

(2) $\neg a(Z, X, Y) \vee a(X, Y, Z)$

p is a base relation and a and q are recursive relations.

Figure 2

Examples of Linear Recursive Rules

Definition: A recursive rule can be either *linear* or *non-linear*. In a linear recursive rule, the body has one and only one literal that is mutually recursive with its own head. For example, the two recursive rules

$$\neg p(X, A) \vee \neg q(A, Y) \vee q(X, Y)$$
$$\neg a(Z, X, Y) \vee a(X, Y, Z)$$

are both linear. The body of a non-linear rule, on the other hand, contains more than one literal that is mutually recursive with its head, as for example in

$$\neg q(X, A) \vee \neg q(A, Y) \vee q(X, Y)$$

Consequently, a loop formed by linear recursive rules is called a linear recursive loop while a loop containing a non-linear recursive rule is called a non-linear loop.

Definition: A literal which causes a loop to be entered is called the *start literal* of that loop. Hence, the head of any recursive rule is the start literal of the loop in which the rule participates. For instance, literals $d(X, Y)$, $l(X, Y)$ and $f(X, Y)$ in rules (5), (6) and (7) of Figure 1 are possible start literals.

Definition: Any negative literal inside a loop that calls upon a start literal of that loop is called an *end literal.* Start and end literals are complementary in that they have the same predicate name but opposite sign. Start literals are always positive while end literals are always negative. For example, in Figure 1, $\neg d(A, Y)$ in clause (7) is the end literal corresponding to the start literal $d(X, Y)$ of clause (5).

Definition: An *exit clause* of a loop is a non-recursive rule whose head has the same predicate name and arity as (i.e., is unifiable with) the head of some rule inside that loop. For example, clause (8) in Figure 1 is the exit clause for the loop comprising the clauses (5), (6) and (7). It is assumed that each loop always has at least one exit clause; otherwise, the loop cannot terminate.

2.2. SL-Resolution

Linear input resolution [ChLe73, WOLB84] is a special case of binary resolution. At each resolution step, a literal from the current resolvent is selected as the next goal. The clause with which this goal is to be unified must come from the original input set. The linear input resolution method does not, however, prescribe *which* of the literals comprising the resolvent should be selected as the next goal.

SL-resolution [KoKu71] stands for *L*inear resolution with a *S*election function. It is a refinement of linear input resolution, in which the rule for selecting the next goal from a resolvent is explicitly specified. Each selection rule will result in a different derivation tree. One of the most common selection rules is one that always selects the leftmost literal of the resolvent as the next goal. For example, consider the clauses in Figure 1. The query $\neg\ q(a,?)$ will generate the derivation tree shown in Figure 3. The root of the derivation tree is the negated query and each node thereafter is the result of resolving the leftmost literal of its parent with some input clause; the number of the selected clause (according to Figure 1) is recorded on the tree edge. For example, the initial query can be resolved with clause (3), which yields the resolvent $\neg\ p_1(a, A_1) \vee \neg\ q(A_1, ?)$.

In a given derivation tree, each path corresponds to a different search for a possible solution. The order in which the paths are traversed is determined by the *search strategy*. The most common search strategy is depth-first. In the remainder of this paper, we consider only the SL-resolution with the left-to-right selection rule and a depth-first search strategy; this will be referred to as *preorder*

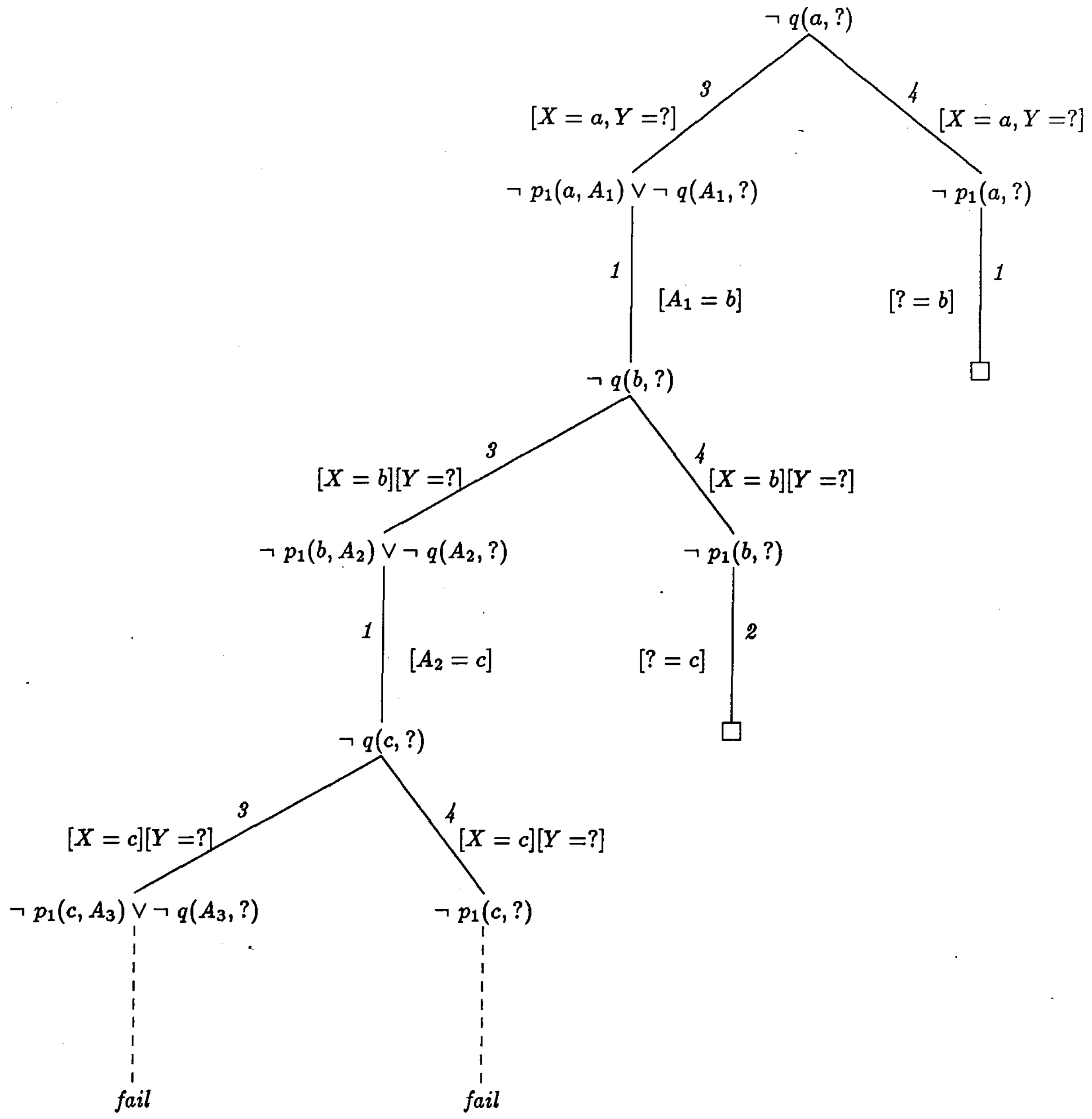


Figure 3

Example of SL-Resolution

Selecting the Leftmost Literal

SL-resolution. Algorithm A specifies this method formally. (Note that this version always generates all possible answers to a given query, rather than just the first one.) The delimiters "(*" and "*)" denote comments.

```
(1) Negate query to become the current goal $G$;
(2) Scan input clauses;
    if an unused input clause $C$ is unifiable with goal $G$
    then mark $C$ as used; (* goto step (3) *)
    else  if $G$ is the root of the derivation tree
          then terminate the process
          else  fail $G$ and backtrack to parent clause $P$ of $G$
                set the current resolvent to $P$;
                goto step (4);
(3) Resolve $G$ with $C$;
    if an empty clause is derived
    then succeed and return bindings
         fail $C$; (* in order to collect the remaining answers *)
         goto step (2); (* the backtracking step *)
    else  the residual literals of $C$ are placed at the leftmost
          position of the resolvent;
(4) Set $G$ to the leftmost literal of the current resolvent;
    Re-initialize all input clauses that have been marked in step (2);
    goto step (2);
```

Preorder SL-resolution
Algorithm A

3. Incompleteness of Preorder SL-resolution

A resolution method is complete if, for any unsatisfiable program, the resulting derivation tree contains at least one empty clause (represented by a small square at the leaf level). In that respect, SL-resolution is complete [KoKu71, LLO84]. However, it depends on the particular search strategy, whether the empty clause can be found. Under certain search strategies the SL-resolution may enter an infinite loop and thus become incomplete. In particular, the preorder SL-resolution introduced in Section 2.2, is incomplete because of the occurrence of the following

Strategy	*Complete*	*Domains in which Strategy is incomplete/complete*
Preorder	*No*	*1. Left Recursion* *a). linear recursive* *b). non-linear recursive* *2. Cyclic Assertions*
Preorder+ Goal Termination	*No*	*1. Left Recursion* *a). linear recursive* *b). non-linear recursive*
Preorder+ Goal Termination+ Transformation	*Yes*	*Safe Loops*

Table 1

Completeness of SL-resolution with Different Refinement Strategies

two special cases: (1) left recursive rules, and (2) cyclic assertions, that, in the presence of recursion, keep generating the same results indefinitely. Row 1 of Table 1 represents this situation. As indicated in the table, left recursion can further be subdivided into two cases: (a) linear recursive rules, and (b) non-linear recursive rules.

In [BW84], Brough and Walker show that a goal termination strategy can terminate properly for recursions with cyclic assertions provided that the rules are left recursion free.* The goal termination strategy is described in detail in Section 3.1. Unfortunately, preorder SL-resolution with a goal termination strategy still remains incomplete, because of the left recursion problem. Row 2 of Table 1 summaries this situation that will be discussed in detail in Section 3.2. The objective of this

* There is one special case of left recursion, called permuted tautology loop, for which the goal termination strategy does terminate properly; this will be discussed in Section 4.1

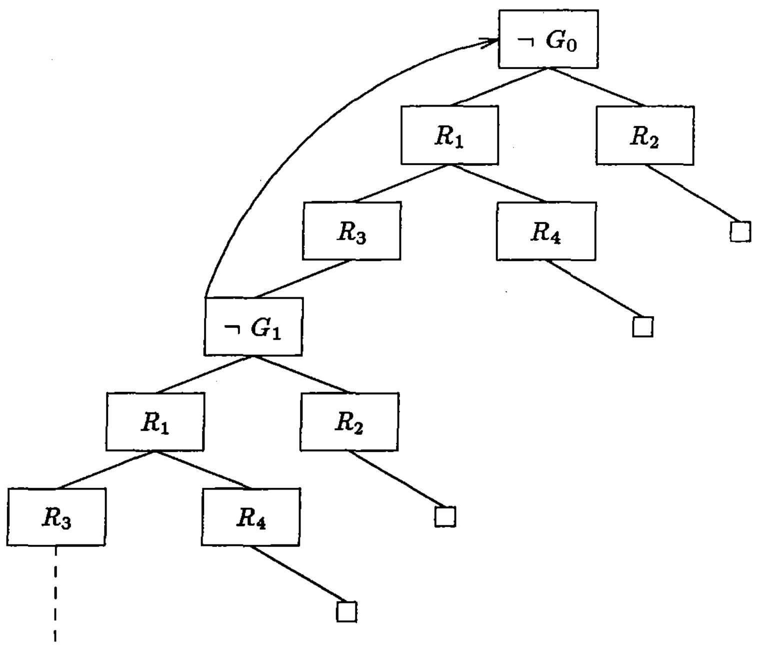


Figure 4

Derivation Tree of the Goal Termination Theorem

paper is to present a method that will transform all loops into *safe loops* which are left-recursion free. For this case, preorder SL-resolution combined with a goal termination strategy will always terminate with all possible answers; thus, it is complete. Row 3 of Table 1 represents this claim, which will be substantiated in Section 4.

3.1. The Goal Termination Strategy

In [BW84], the goal termination strategy was introduced informally. In this section, we first give a precise definition of this concept. The following theorem serves as the basis for the definition:

Theorem *Goal Termination Strategy.* If a goal G occurs as its own subgoal (i.e. within its own branch of a derivation tree) with identical bindings, the goal G can be failed.

Proof The proof is by contradiction. Consider the derivation tree in Figure 4 and assume the following:

(1) G_0 and G_1 are the same goals with identical bindings;

(2) G_1 cannot be failed (false assumption).

According to preorder SL-resolution, assumption (1) implies that the subtrees for the two goals G_0 and G_1 are identical. Assumption (2) implies that there exists a proof for G_1 that is different from any proof for G_0. Consequently, the subtrees for G_0 and G_1 must be different. This results in a contradiction and hence assumption (2) must be false. (Q.E.D)

The goal termination strategy is shown in Algorithm B – an extension of Algorithm A. A global history list H is used to hold the sequence of executed goals (with their bindings). The main distinction between these two algorithms is in step 4 which detects a repeated goal by searching the history H for the occurrence of the current goal.

3.2. Behavior of Preorder SL-resolution with Goal Termination Strategy

In this section, we show that a preorder SL-resolution combined with a goal termination strategy is still incomplete because of the left recursion problem.

First, consider the following clauses, containing cyclic assertions:

(1) $at(pencil, lamp)$.

(2) $at(lamp, radil)$.

(3) $at(radio, pencil)$.

(4) $\neg\ at(X,Z) \vee \neg\ locate(Z,Y) \vee locate(X,Y)$.

(5) $\neg\ at(X,Y) \vee locate(X,Y)$.

When the query $\neg\ locate(pencil, W)$ is asked, Algorithm A will enter an infinite branch, since the assertions (1) to (3) are cyclic. This occurs when rule (4) is applied. The algorithm will loop indefinitely without even trying rule (5), which would yield some answers. Algorithm B, on the other hand, will terminate properly,

```
(1) Negate query to become the current goal $G$;
    if $G$ is recursive then history list $H = [G.nil]$ else $H = nil$
(2) Scan input clauses;
    if an unused input clause $C$ is unifiable with goal $G$
    then mark $C$ as used; (* goto step (3) *)
    else if $G$ is the root of the derivation tree
         then terminate the process
         else fail $G$ and backtrack to parent clause $P$ of $G$
              set current resolvent to $P$
              goto step (4);
(3) Resolve $G$ with $C$;
    if an empty clause is derived
    then succeed and return bindings
         fail $C$ (* in order to collect the remaining answers *)
         goto step (2) (* the backtracking step *)
    else the residual literals of $C$ are placed at the leftmost
         position of the resolvent;
(4) Set $G$ to the leftmost literal of the current resolvent;
    if $G$ is recursive
    then if $G$ is in $H$ (* repeated goal detected *)
         then fail $G$ and backtrack to parent clause $P$ of $G$
              set $G$ to the leftmost literal of $P$
              goto step (2)
         else append $G$ to $H$
    Re-initialize all input clauses that have been marked in step (2);
    goto step (2);
```

Preorder SL-resolution with Goal Termination Strategy
Algorithm B

since the same goal repeats in the same branch of the derivation tree. That is, goal termination is sufficient in the case of cyclic assertions.

Second, consider the following clauses containing a left recursion:

(1) $\neg\, q(A,Y) \vee \neg\, p(X,A) \vee q(X,Y)$.

(2) $\neg\, p(X,Y) \vee q(X,Y)$.

(3) $p(a,b)$.

(4) query $\neg q(a,Y)$.

In this situation, even Algorithm B will enter an infinite branch when it tries to resolve the left most literal of clause (1). The remedy, however, is simple, provided the program does not contain any extra-logical features such as the 'cut' symbol of Prolog. In this case, we may simply rearrange the literals of rule (1) to avoid left recursion. This is because the order of literals in a clause does not affect the logical consequences of the resolution; only, perhaps, the efficiency of execution. Rule (1) may be rearranged as follows:

$$\neg\ p(X,A) \vee \neg\ q(A,Y) \vee q(X,Y)$$

Finally, consider a non-linear recursive rule that contains no base or non-recursive literals at all; for example, clause (1) in Figure 8. In this case, the left recursion problem cannot be avoided by simply rearranging literals. Hence, preorder SL-resolution with goal termination is incomplete (Row 2 of Table 1).

4. Preprocessing of Input Clauses

In this section, we show how to transform loops of the input set into *safe* loops, for which the strategy is complete. The first transformation is to eliminate tautology loops while the second eliminates left recursion in non-linear rules. We then show how the resulting safe loops may further be transformed such that all rules begin with a base literal. This is necessary to permit the efficient implementation of goal termination which will be described in Section 5.

4.1. Elimination of Tautology Loops

A *basic tautology loop* is defined in [SIC76] as a loop in which the start literal and the end literal are the same predicates with identical bindings. Rules (1) to (3) in the following program form a tautology loop with the start literal $q(X,Y)$ in rule (1) and the end literal $\neg\ q(A,B)$ in rule (3):

(1) $\neg\ p(X,Y) \vee q(X,Y)$.

(2) $\neg\ d(X,Y) \vee p(X,Y)$.

(3) $\neg\ q(A,B) \vee d(A,B)$.

(4) $\neg\ e(X,Y) \vee q(X,Y)$. exit clause

To detect whether a given loop is a basic tautology loop, we rewrite it as follows. For each clause i in the loop, we rename all terms such that the head of clause i matches the corresponding unifiable literal of clause $i-1$. (Recall that, for any loop, there exists an ordering of clauses, as specified in Section 2.) If, after the renaming, the terms of the start and end literals are identical, the loop is a basic tautology loop.

For example, the following three clauses are the result of rewriting clauses (1) through (3) of the above program.

(1) $\neg\ p(X,Y) \vee q(X,Y)$.

(2) $\neg\ d(X,Y) \vee p(X,Y)$.

(3) $\neg\ q(X,Y) \vee d(X,Y)$.

Clause (2) did not have to be rewritten, since its head ($p(X,Y)$) already matched the corresponding literal of clause (1). Clause (3) was rewritten by replacing A and B with X and Y, respectively. The terms of the resulting start literal ($q(X,Y)$) and end literal ($\neg q(X,Y)$) are identical; hence a tautology loop is present.

Sickel has shown that basic tautology loops never need to be traversed in the search for a proof and thus can be eliminated from the program. The elimination is accomplished by replacing each clause of the loop by the set of exit clauses unifiable with that clause.

The reasoning underlying this transformation can be illustrated using the previous example. (For a detailed proof, refer to [SIC76].) Suppose we resolve clauses (1) through (3); the resolvent will contain the pairs $q(X,Y) \vee \neg\ q(X,Y)$ which is a tautology. The loop together with its exit clause is shown in Figure 5. The only way to get out of the loop and thus to obtain any solutions is to resolve the exit clause. Since the loop by itself does not produce any solutions, it can be

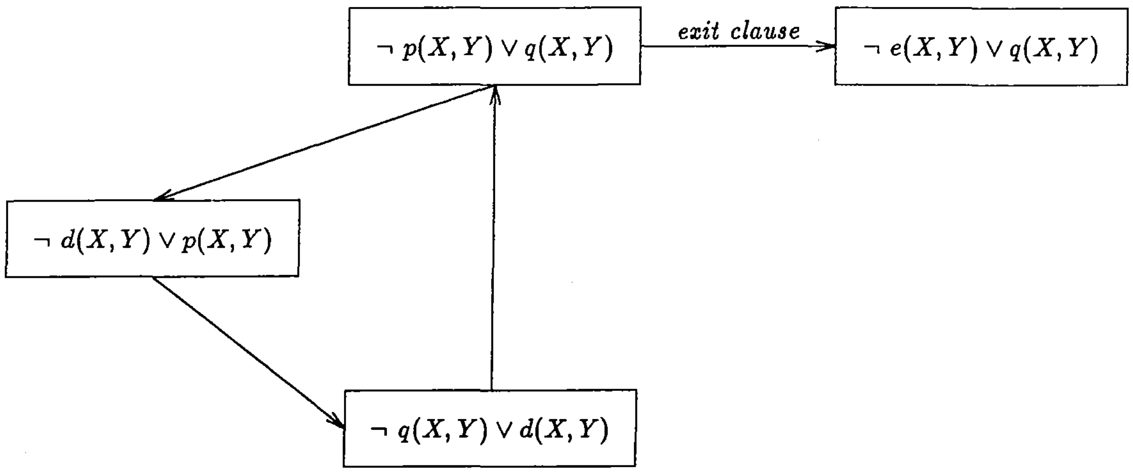

$\neg\, p(X,Y) \vee q(X,Y)$
exit clause
$\neg\, e(X,Y) \vee q(X,Y)$
$\neg\, d(X,Y) \vee p(X,Y)$
$\neg\, q(X,Y) \vee d(X,Y)$

Figure 5

Example of a Basic Tautology Loop

broken by replacing clause (1) with the exit clause (4). The resulting program is as follows:

(1) $\neg e(X,Y) \vee q(X,Y)$
(2) $\neg d(X,Y) \vee p(X,Y)$
(3) $\neg q(X,Y) \vee d(X,Y)$

Remark: The goal termination strategy is sufficient to prevent infinite recursion in the case of basic tautology loops, even if the loop contains left recursion [BW84]. Hence the elimination of basic tautology loops is not really necessary to guarantee completeness of the strategy; rather, it is performed only to avoid unnecessary computation.

The concept of the basic tautology loop can be extended further as follows:

Definition: A *permuted tautology loop* is a linear loop, in which the terms of the end literal are a permutation of the terms of the start literal (after renaming).

Consider, for example, the following two clauses:

(1) $\neg q(Z,X,Y) \vee q(X,Y,Z)$
(2) $\neg e(X,Y,Z) \vee q(X,Y,Z)$ (exit clause)

$$\begin{tikzcd}
\boxed{\neg\, q(Z,X,Y) \vee q(X,Y,Z)} \arrow[r, "\textit{exit clause}"] \arrow[d] & \boxed{\neg\, e(X,Y,Z) \vee q(X,Y,Z)} \\
\boxed{\neg\, q(Y,Z,X) \vee q(Z,X,Y)} \arrow[r, "\textit{exit clause}"] \arrow[d] & \boxed{\neg\, e(Z,X,Y) \vee q(X,Y,Z)} \\
\boxed{\neg\, q(X,Y,Z) \vee q(Y,Z,X)} \arrow[r, "\textit{exit clause}"] \arrow[uu, bend left] & \boxed{\neg\, e(Y,Z,X) \vee q(X,Y,Z)}
\end{tikzcd}$$

Figure 6

Example of a Permuted Tautology Loop

Clause (1) forms a permuted tautology loop since the terms of the end literal ($\neg q(Z,X,Y)$) are a permutation of the terms of the start literal ($q(X,Y,Z)$).

Theorem A permuted tautology loop is never required in a refutation and hence may be eliminated.

Proof The proof is based on the idea of transforming the original loop into a basic tautology loop, which may be eliminated according to the theorem given by Sickel. Assume that the terms of the end literal are a permutation of the terms of the start literal; the permutation order is r. (Interested readers are referred to the Appendix, describing how to determine the order of a permutation.)

The transformation comprises the following steps:

(1) replicate the original loop r-times;

(2) rewrite the resulting sequence of clauses as in the case of detecting a basic tautology loop; i.e., for each clause i in the loop, rename all terms such that the head of clause i matches the corresponding unifiable literal of clause $i-1$;

If a permutation's order is r and it is applied to a sequence of elements r-times, the resulting sequence is guaranteed to be identical to the original sequence [FRA82]. Since the sequence derived through the above transformation permutes the terms of the start literal r-times, the end literal of the last replication of the loop is complementary to the start literal of the first replication of the loop. Hence this new sequence forms a basic tautology loop and can be eliminated using the approach given by Sickel. (Q.E.D)

To illustrate the application of this theorem, consider again the above example of a permuted tautology loop. The permutation of the terms in rule (1) has an order of 3. By replicating clause (1) three times and renaming all terms accordingly, a basic tautology loop is derived; this is shown, together with the corresponding exit clauses, in Figure 6. By eliminating the basic tautology loop, we obtain the following loop-free set of clauses:

(1) $e(X,Y,Z) \vee q(X,Y,Z)$

(2) $e(Z,X,Y) \vee q(X,Y,Z)$

(3) $e(Y,Z,X) \vee q(X,Y,Z)$

4.2. Elimination of Non-linear Recursive Loops

Non-linear recursive loops can always be transformed into linear recursive loops as follows. If we interpret Horn clauses by means of "procedural semantics", any clause

$$\neg p_0 \vee \neg p_1 \vee \ldots \vee \neg p_{n-1} \vee p_n$$

is analogous to a call to the procedure p_n which, in turn, calls the procedures p_0 through p_{n-1}, in sequence. Hence the procedural semantics allows us to interpret any clause analogous to a production rule in a context free grammar.

It has been proven [HAR78] that for any left-recursive context free grammar G, there exists another context free grammar, $\bar{G}$, that is equivalent to G but contains no left recursion. $\bar{G}$ is derived by the following transformation:

A left-recursive context free grammar G has the form

(1) $A \rightarrow A\alpha_1 \vee \ldots \vee A\alpha_r$

(2) $A \rightarrow \beta_1 \vee \ldots \vee \beta_s$

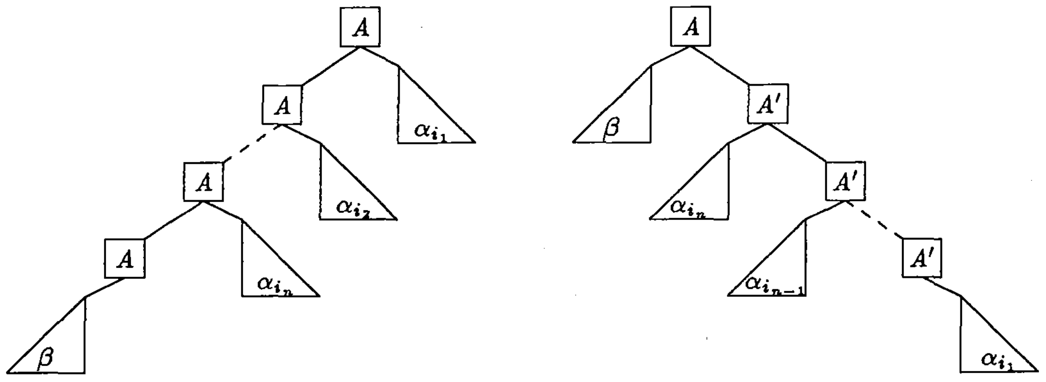


Figure 7

Elimination of Left Recursion

where α's may be terminal or non-terminal strings, and β's are always terminal strings. Figure 7(a) shows the generic derivation tree of this grammar. Each level of the tree corresponds to the execution of one of the production rules. The right subtree, α_{i_j}, always represents a finite string. The left subtree may continue to expand recursively as long as rule (1) is applied; application of rule (2) terminates the recursion.

The above left recursive grammar may be transformed into the following equivalent left recursion free grammar (any text book on compiler or automata theory, e.g. [HAR78], describes the appropriate procedures):

(1) $A \rightarrow \beta_1 A' \vee ... \vee \beta_s A'$

(2) $A \rightarrow \beta_1 \vee ... \vee \beta_s$

(3) $A' \rightarrow \alpha_1 A' \vee ... \vee \alpha_r A'$

(4) $A' \rightarrow \alpha_1 \vee ... \vee \alpha_r$

where α's and β's have the same meanings as before and A' is a new non-terminal symbol. The corresponding derivation tree is shown in Figure 7(b). Note that the leaves of the trees in both cases (a) and (b) are visited in the same order when a preoder traversal is used; hence the same strings are produced.

(1) $\neg\, a(X,W) \vee \neg\, a(W,Y) \vee a(X,Y)$

(2) $\neg\, b(X,Y) \vee a(X,Y)$

a is a recursive relation and b is a base relation.

Figure 8

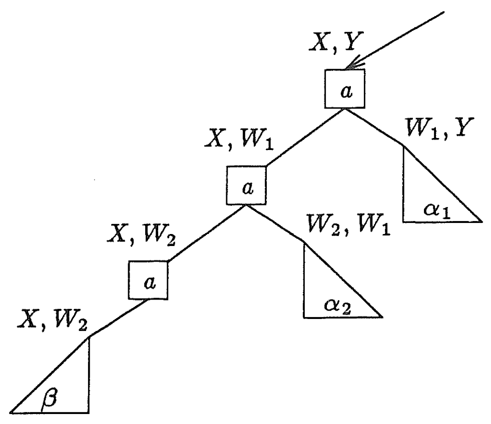
Example of a Non-linear Recursive Rule

An analogous transformation can be applied to eliminate left-recursion in Horn-clause logic by viewing each symbol as a literal. Terminal symbols represent base or nor-recursive predicates while non-terminal symbols are recursive predicates. The necessary extension is to accommodate the generation of terms for each literal.

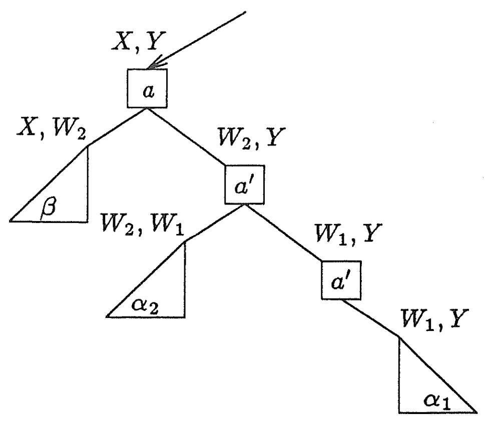
The procedure to transform a given left-recursive loop into one without left recursion is as follows. First, create a derivation tree, A, of the form shown in Figure 7(a). This is accomplished by replicating the clause with left-recursion n-times, where n is the arity of the recursive literal. This assures that all possible permutations of the terms will be included in the tree. The terms in the resulting sequence of clauses are then renamed (subscripted) such that the head of clause i matches the corresponding unifiable literal of clause $i-1$, as in the case of detecting tautology loops (Section 4.1).

As a next step, we ignore the terms of all clauses in the original loop and generate the corresponding left-recursion free loop according to the rule for eliminating left recursion in context-free grammars, given above. From this loop, we create a derivation tree, B, of the form shown in Figure 7(b).

Next, we transfer the renamed terms of the leaf nodes of tree A into the corresponding leaf nodes of tree B. We then proceed towards the root by creating consistent terms for each intermediate node. As a last step, we transform the modified tree B into clausal forms.



(a)



(b)

Figure 9

Elimination of Left Recursion for Predicate Logic

The above procedure is illustrated by showing the transformation of the left-recursive program of Figure 8 into a corresponding left-recursion free program. Let α and β correspond to the predicates a and b, respectively. A derivation tree obtained from the program by following the above procedure is shown in Figure 9 (a). This tree resembles the tree of Figure 7 (a) with $n = 2$ but, in addition to the predicate names, it also shows the renamed terms for each node. For instance, the root a has the terms (X, Y).

As a next step, we create the left-recursion free tree and proceed by filling in the terms as follows. The bindings of the leaves α_1, α_2, and β are transferred from the tree in Figure 9(a) into the tree just created; these are the terms (W_1, Y), (W_2, W_1) and (X, W_2), respectively. By proceeding towards the root, we assign consistent terms to the remaining nodes of the tree. The final tree, shown in Figure 9 (b), is then translated into the following left-recursion free clauses:

(1) $\neg b(X, W_2) \vee \neg a'(W_2, Y) \vee a(X, Y)$

(2) $\neg a(W_2, W_1) \vee \neg a'(W_1, Y) \vee a'(W_2, Y)$

(3) $\neg a(W_1, Y) \vee a'(W_1, Y)$

The subscripts can be eliminated to conform with the notation of Figure 8 by rewriting each clause with a new set of variables. One possible result of such rewriting is the following set of clauses (note that the exit clause is included):

(1) $\neg b(X,W) \vee \neg a'(W,Y) \vee a(X,Y)$
(2) $\neg a(X,D) \vee \neg a'(D,Y) \vee a'(X,Y)$
(3) $\neg a(X,Y) \vee a'(X,Y)$
(4) $\neg b(X,Y) \vee a(X,Y)$ (exit clause)

The above program is left-recursion free and is equivalent to the program in Figure 8.

4.3. A Final Transformation

The transformations presented in the preceding sections 4.1 and 4.2 eliminated tautology loops and left recursion, respectively. In order to permit an efficient implementation of goal termination, it is necessary to further transform the resulting safe clauses such that each begins with a base literal. This step is analogous to converting a context free grammar into Greibach normal form [HAR78]. This is accomplished by rewriting all clauses that do not begin with a base literal as follows:

```
for each clause $C$ that begins with a non-base literal $l_{nb}$ do
    - find all clauses $C'$ whose head has the same predicate as $l_{nb}$
    for each clause $C'$ do
        - rewrite $C'$ with the terms of $l_{nb}$
        - make a copy of $C$ and replace $l_{nb}$ with the body of $C'$
```

To illustrate this procedure, consider the final set of left-recursion clauses derived in Section 4.2. Clauses (2) and (3) begin with a non-base literal. In both cases, the clauses unifiable with the respective non-base literals $\neg a(X,D)$ and $\neg a(X,Y)$ are the clauses (1) and (4). After renaming and substitution, the resulting set of clauses is as shown below. The original clause (2) was transformed into two new clauses (2′) and (3′); similarly, clause (3) resulted in the new clauses (3′) and (4′). (Note that clause (3′) is the same for both cases.)

(1′) $\neg b(X,W) \vee \neg a'(W,Y) \vee a(X,Y)$

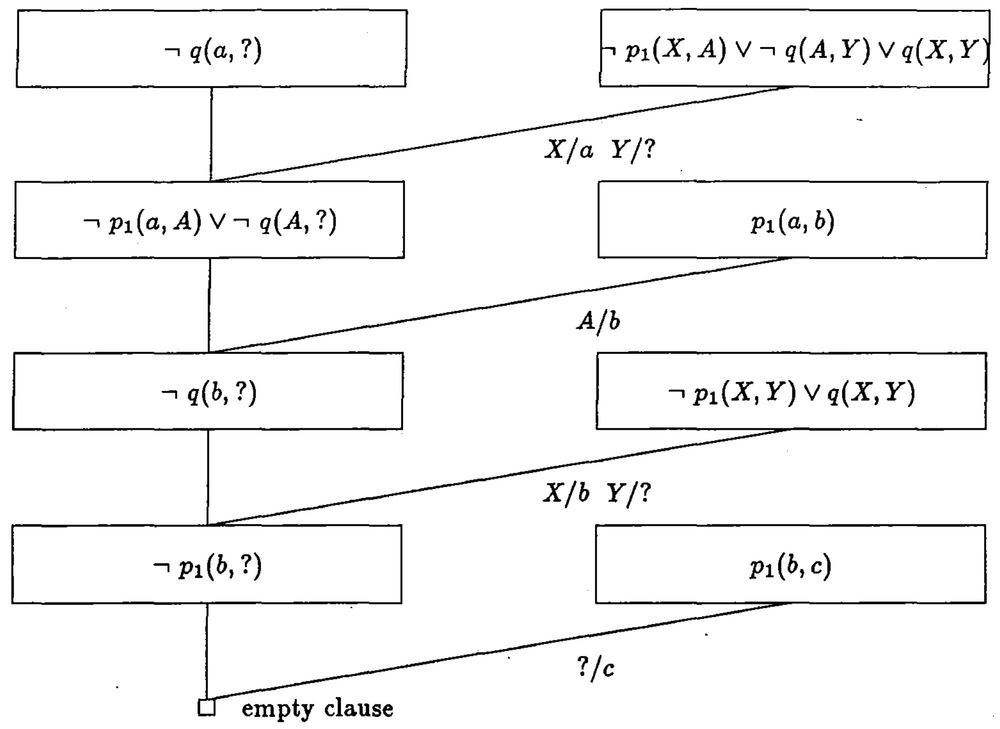


Figure 10

An Example of a Refutation Tree

(2') $\neg b(X,W) \vee \neg a'(W,D) \vee \neg a'(D,Y) \vee a'(X,Y)$

(3') $\neg b(X,W) \vee \neg a'(W,Y) \vee a'(X,Y)$

(4') $\neg b(X,Y) \vee a'(X,Y)$

(5') $\neg b(X,Y) \vee a(X,Y)$ (exit clause)

5. An Efficient Scheme to Implement the Goal Termination Strategy

In this section, we present a simple scheme that is used to implement the goal termination strategy efficiently. We further demonstrate how the scheme works using a complete example. To simplify the presentation in this section, we need to introduce the concept of a *refutation tree.* This may be used to show a particular sequence of resolution steps, instead of showing the whole derivation tree.

Figure 10 gives the refutation tree corresponding to a possible path through the derivation tree of Figure 3; the clausal form of the program is shown in Figure 1. At each step, the clause to be resolved, called the *center* clause, is shown in the left column, and the clause resolving the center clause, termed the *side* clause, is shown in the right column. Along with the resolution, we also show the substitutions used to unify the clauses.

5.1. A Distributed Goal Recording Scheme

As shown in [BW84], one way to implement the goal termination strategy is to keep a complete execution history of the given query (i.e. a list or stack of all executed goals.) By checking this history, it can be determined whether a goal has been repeated. Unfortunately, checking and maintaining the list of executed goals can be very costly in a deep recursion.

Our approach is to distribute the history list throughout the entire database such that the amount of work to check the occurrence of any particular goal is reduced. To understand the basic idea of this approach, consider the refutation tree in Figure 11. Assume that the current goal to be solved is G_i. The objective is to determine whether the same goal (with identical bindings) occurred earlier, i.e., anywhere higher in the tree. There are four possible ways to maintain the history information:

(1) The original proposal [BW84] implies that a global list is maintained, which records the occurrence of all recursive goals. Each time a recursive goal G_i is encountered, this global history list must be searched to determine if G_i occurred earlier. This corresponds to Step 4 of Algorithm B.

(2) To reduce the search, the history list can be distributed over individual clauses of the program. The first possible level of distribution is over clauses that are unifiable with G_i. Assume that C is such a clause (in Figure 11). A list recording the history relevant to the use of C is maintained with each

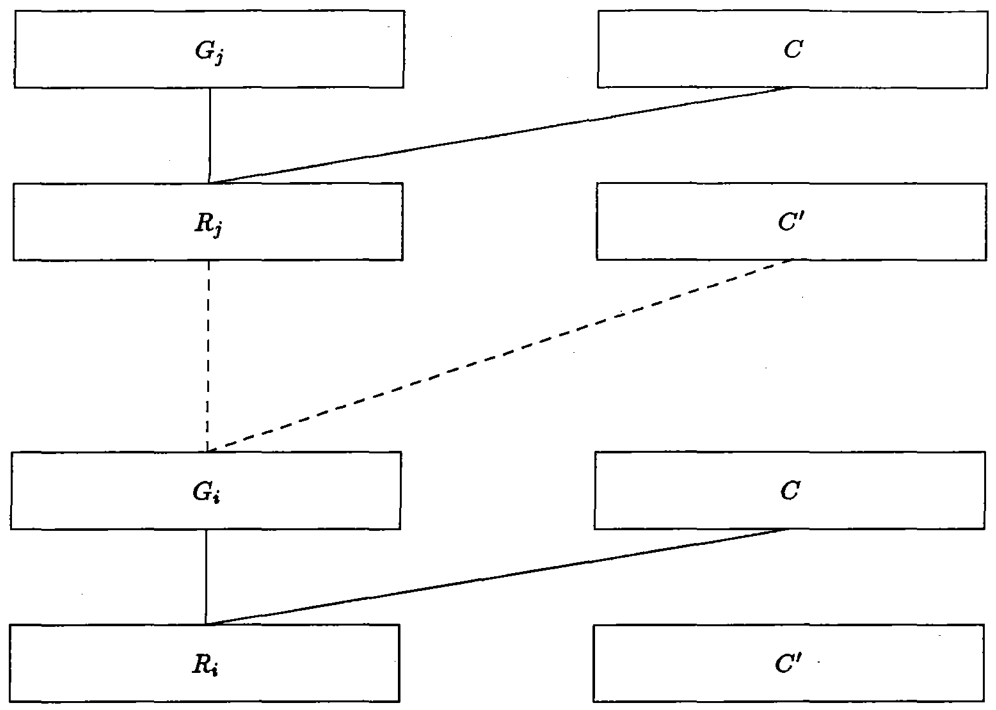


Figure 11

such clause. This list is used in the same way as the global list in case 1 above. That is, each time C is used to resolve a goal G_i, the list is searched for the occurrence of the same goal. If a match is found, the current goal G_i is failed; otherwise, G_i is included in the list and resolution continues.

The above distribution is justified by the following observation. If a goal G_j, identical to G_i, occurs in the tree, the same clause C will be used to resolve it; this is due to the preorder search strategy. Since G_i preceded G_j in the tree, it was recorded in the list associated with C at the time when G_j is encountered. Hence G_j will be failed.

This approach distributes the original history list of case 1 over the set of recursive clauses constituting the program; i.e., the encountered goals are segregated into sublists according to their predicate name and current bindings. Hence, the average number of goals that must be examined in each case is reduced in proportion to the number of recursive clauses.

(3) An even higher degree of distribution is possible by recording the goals not with the clause C itself but with next side clause C', immediately following C in the resolution (Figure 11). That is, when a recursive goal G_j is encountered, we first find a clause C to solve G_j and perform the unification; this yields a resolvent R_j. Next we find a clause C' that can be resolved with R_j; G_j is then recorded in the list associated with the clause C'.

This approach is based on the following fundamental observation. Assume that G_i is a goal identical to G_j, and G_j occurs earlier in the tree. This implies that the same clause C will be used to solve both G_i and G_j. Consequently, the first subgoals in R_i and R_j, immediately following G_i and G_j respectively, must also be identical. Hence, both subgoals will be resolved with the same clause C'. This clause is the one with which the goal G_j was previously recorded. Hence, when G_i is to be solved, this list will be examined and G_i will fail.

The advantage of this approach over the one in case 2 above is a greater distribution of the history information. In case 2, the history list was distributed over all recursive clauses C. In case 3, it is distributed over clauses C'. Since, for each recursive clause C, there is at least one clause C' that can be used as the next side clause, the history list will, on the average, be distributed over a larger number of clauses.

(4) A final improvement can be achieved by influencing the number of clauses C' over which the history list will be distributed. That can be attained by transforming all clauses of the input set such that each begins with a base literal. Section 4.3 described a procedure to accomplish this transformation. The reason why such a transformation yields a better distribution can be explained by the following reasoning. Consider again Figure 11. Clause C

(after the final transformation) begins with a base literal. When it is resolved with G_j, the resolvent R_j also begins with a base literal (due to preorder search). This implies that the clause, C', that will be used to resolve R_j will always be an assertion. Since, in general, there are more assertions than rules in a given program, there will be a greater distribution of the history list than in case 3.

Algorithm C gives the details of the final preorder resolution with goal termination implemented as suggested in case 4 above. In comparing this algorithm with Algorithm B of Section 3.2, we note the following main differences. In Algorithm B, the recording and checking of repeated goals is done in step 4, using the global history list H; this occurs each time a recursive goal G is encountered. In Algorithm C, these tasks have been moved from step 4 to a new step 2a. Note that they are performed whenever an assertion is used as the side clause C, when solving a recursive goal G. A temporary variable G_{LR} is used to hold the current recursive goal. A related change is found in step 1; in Algorithm B, a global history list H was initialized; in Algorithm C, on the other hand, a separate history (sub)list H_C is created and initialized for each assertion.

5.2. Example for Mutual Recursion with Cyclic Assertions

In this section, we illustrate Algorithm C by presenting a complete example. Consider the mutually recursive loop consisting of the clauses (5) through (8) in Figure 12.

Given the query $q(a, W)$, Algorithm C produces the refutation tree shown in Figure 13. The first column records the resolution steps numbered 1a through 6a. The second column indicates when goals are recorded onto assertions; each entry shows the assertion and, as a subscript, the goal being recorded during that step. The next two columns show the sequences of the center and the side clauses, respectively. The last column shows the current recursive goal G_{LR}.

```
(1)   Negate query to become the current goal $G$;
      for each assertion $C$, initialize $H_C \leftarrow nil$;
(2)   if $G$ is recursive then $G_{LR} \leftarrow [G]$ else $G_{LR} \leftarrow nil$;
      Scan input clauses;
      if an unused input clause $C$ is unifiable with goal $G$
      then mark $C$ as used; (* goto step (2a) *)
      else  if $G$ is the root of the derivation tree
            then terminate the process
            else  fail $G$ and backtrack to the parent clause $P$ of $G$
                  (undo the goals recorded along the path)
                  set current resolvent to $P$; goto step (4);
(2a)  if $G_{LR}$ is not $nil$ and $C$ is an assertion (with $H_C$)
      then if $G_{LR}$ is not in $H_C$
            then append $G_{LR}$ to $H_C$
            else  fail the $C$ and $G$; (*According to the goal termination strategy*)
                  backtrack to the parent clause $P$ of $G$;
                  (undo the goals recorded along the path;)
                  set current resolvent equal to $P$; goto step (5);
(3)   Resolve $G$ with $C$;
      if an empty clause is derived
      then succeed and return bindings
            fail $C$; (* in order to collect all answers*)
            goto step (2);
      else  the residual literals of $C$ are placed at the leftmost position;
(4)   Set goal $G$ to the leftmost literal of the current resolvent;
      Re-initialize all input clauses that are marked in step (2);
      goto step (2);
```

A Preorder Resolution with Distributed Goal Recording Scheme
Algorithm C

At resolution step 1a, a recursive goal $\neg\ q(a, W)$ is encountered and hence G_{LR} is set to $q(a, W)$. By scanning the input clauses, we find the unused clause (5) (Figure 12), that is unifiable with the current goal. The unification step is carried out with the appropriate substitution list $[X/a\ Y/W]$.

At step 2a, the side clause is the assertion $p(a, b)$. According to step 2a of Algorithm C, we record the current recursive goal $q(a, W)$ onto $p(a, b)$. The same

(1) $p(a, b)$.
(2) $p(b, a)$.
(3) $d(b, a)$.
(4) $d(a, e)$.
(5) $\neg\, p(X, A) \vee \neg\, l(A, B) \vee \neg\, q(B, Y) \vee q(X, Y)$.
(6) $\neg\, d(X, Y) \vee q(X, Y)$.
(7) $\neg\, d(X, Y) \vee l(X, Y)$.
(8) $\neg\, p(X, A) \vee \neg\, q(A, B) \vee \neg\, l(B, Y) \vee l(X, Y)$.
where predicates q and l are mutually recursive

Figure 12

An Example of Mutual Recursion

process is repeated for the resolution steps 2a through 6a. In step 6a, the history list of the side clause $p(a, b)$ contains the goal $[q(a, W)]$. Since this matches the current goal, the goal is failed and the resolution backtracks to step 4a.

At this point, there is no other unused clause unifiable with $\neg d(b, B_1)$ and so the resolution backtracks to step 3a, removing the last goal recorded with each assertion history along the backtracking path. This removes the goal $q(a, W)$ from the assertion $d(b, c)$, that was recorded during resolution step 4a.

The resolution process continues and generates the sequence of steps shown in Figure 14. At step 6b, the goal $\neg p(a, A_4)$ is failed again, since it is already recorded in the history list of the assertion $p(a, b)$. (Note that A_4 and W are both free variables and hence they match.) Execution then backtracks to step 4b. Since there is no other unifiable clause, the process backtracks to step 3a and further to 2a and 1a (Figure 13). The goals that were recorded during previous forward execution steps are removed while backtracking.

The process now continues as shown in Figure 15. It succeeds and returns $[W = e]$ as one possible answer. In order to search for other answers, we fail the current goal and backtrack to step 1f. At this time, there is no other unused unifiable clause, and thus the process terminates with the binding $W = [e]$ as the only answer.

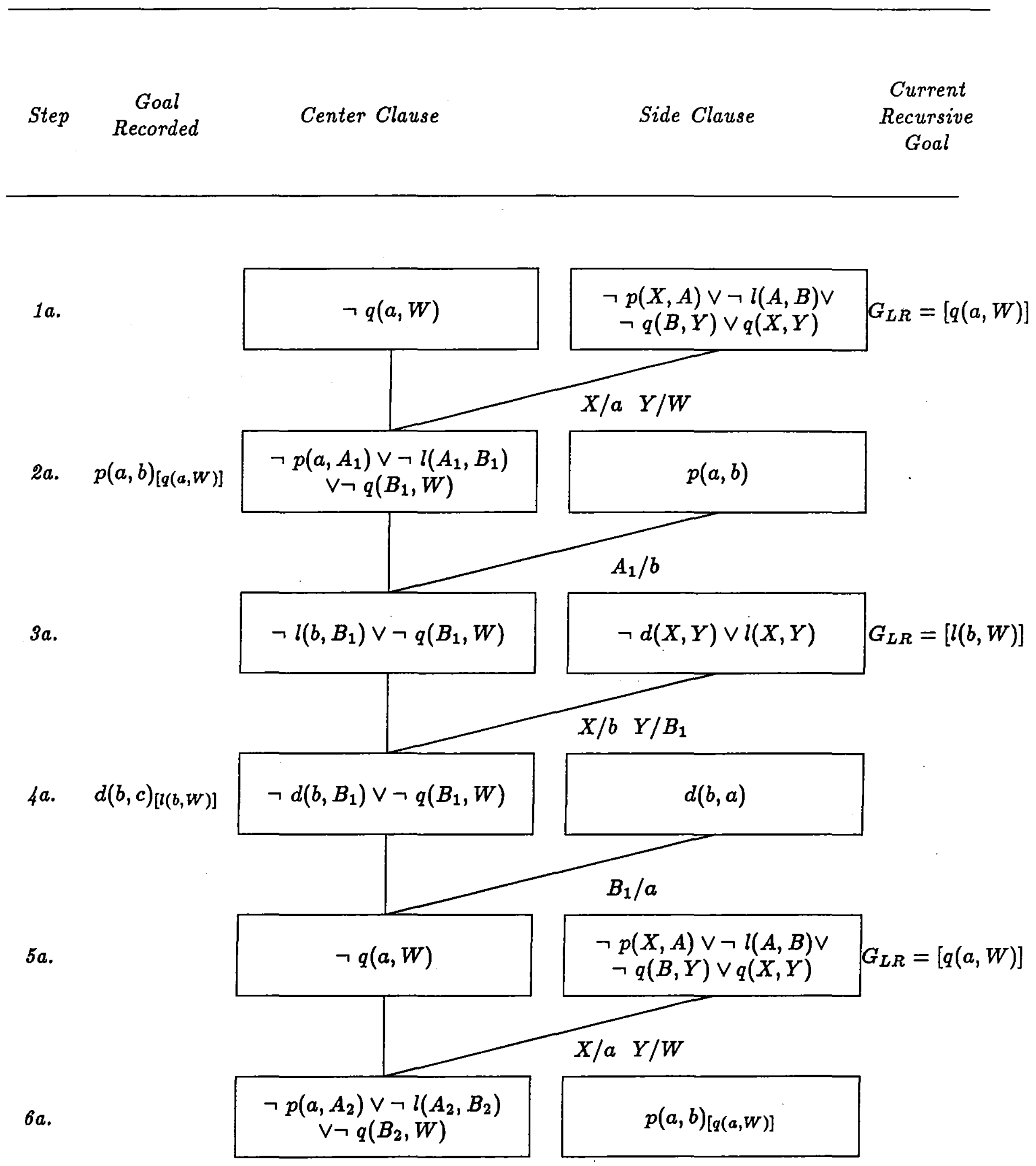

Step
Goal Recorded
Center Clause
Side Clause
Current Recursive Goal
1a.
$\neg\, q(a,W)$
$\neg\, p(X,A) \vee \neg\, l(A,B) \vee \neg\, q(B,Y) \vee q(X,Y)$
$G_{LR} = [q(a,W)]$
$X/a \;\; Y/W$
2a.
$p(a,b)_{[q(a,W)]}$
$\neg\, p(a,A_1) \vee \neg\, l(A_1,B_1) \vee \neg\, q(B_1,W)$
$p(a,b)$
A_1/b
3a.
$\neg\, l(b,B_1) \vee \neg\, q(B_1,W)$
$\neg\, d(X,Y) \vee l(X,Y)$
$G_{LR} = [l(b,W)]$
$X/b \;\; Y/B_1$
4a.
$d(b,c)_{[l(b,W)]}$
$\neg\, d(b,B_1) \vee \neg\, q(B_1,W)$
$d(b,a)$
B_1/a
5a.
$\neg\, q(a,W)$
$\neg\, p(X,A) \vee \neg\, l(A,B) \vee \neg\, q(B,Y) \vee q(X,Y)$
$G_{LR} = [q(a,W)]$
$X/a \;\; Y/W$
6a.
$\neg\, p(a,A_2) \vee \neg\, l(A_2,B_2) \vee \neg\, q(B_2,W)$
$p(a,b)_{[q(a,W)]}$

Figure 13

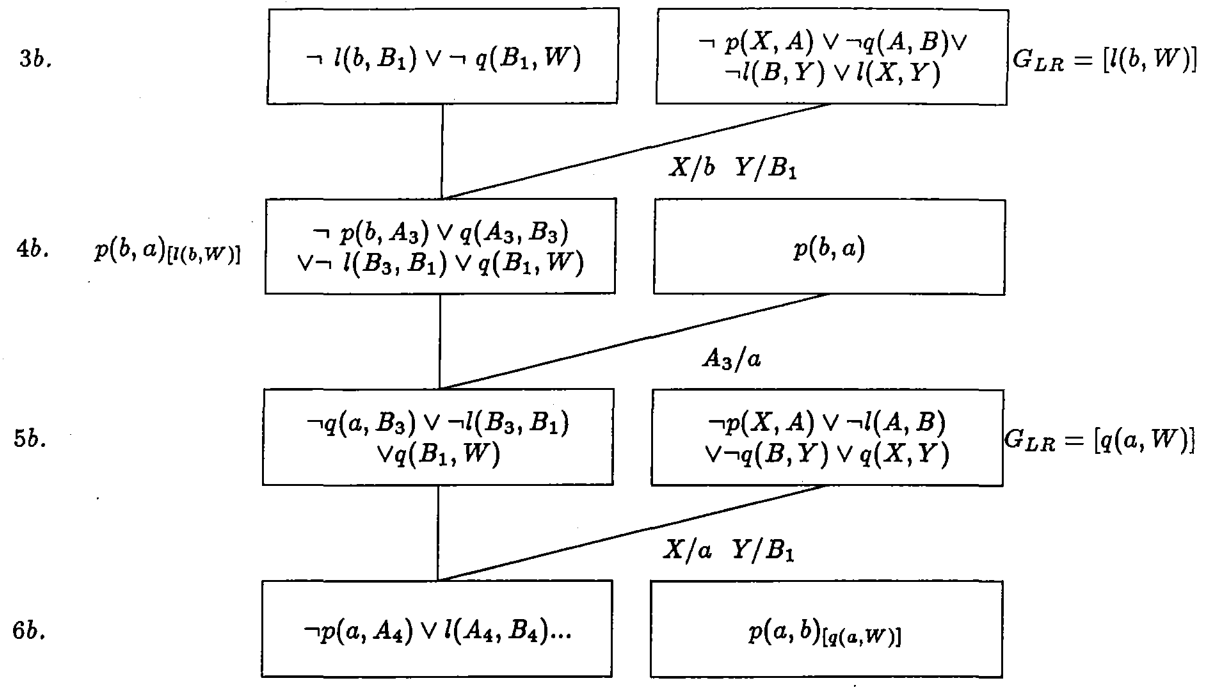

3b.
$\neg\, l(b, B_1) \vee \neg\, q(B_1, W)$
$\neg\, p(X, A) \vee \neg q(A, B) \vee \neg l(B, Y) \vee l(X, Y)$
$G_{LR} = [l(b, W)]$
$X/b \;\; Y/B_1$
4b.
$p(b, a)_{[l(b,W)]}$
$\neg\, p(b, A_3) \vee q(A_3, B_3) \vee \neg\, l(B_3, B_1) \vee q(B_1, W)$
$p(b, a)$
A_3/a
5b.
$\neg q(a, B_3) \vee \neg l(B_3, B_1) \vee q(B_1, W)$
$\neg p(X, A) \vee \neg l(A, B) \vee \neg q(B, Y) \vee q(X, Y)$
$G_{LR} = [q(a, W)]$
$X/a \;\; Y/B_1$
6b.
$\neg p(a, A_4) \vee l(A_4, B_4)...$
$p(a, b)_{[q(a,W)]}$

Figure 14

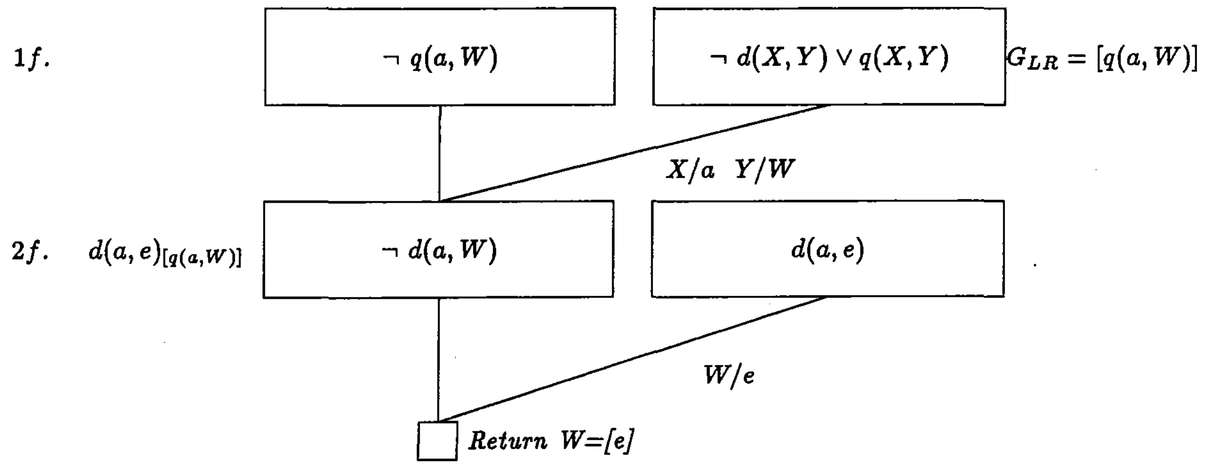


Figure 15

6. Concluding Remarks

Goal termination was proposed to prevent infinite recursion in function-free Horn logic. Unfortunately, it fails to terminate the computation in the case of a left recursive loop; hence, it is not complete. In this paper, we have presented an approach to eliminate left recursion through transformation of clauses. When goal termination is applied to the transformed set, the strategy becomes complete, in that it always terminates and generates all possible answers.

Another problem with the goal termination is to find an efficient implementation. The original proposal assumed a centralized history list that must be searched whenever a recursive goal is encountered. We have presented three other possible ways to improve this approach by distributing the history list throughout the clauses constituting the program. The largest distribution is achieved when the history list is distributed over only assertions, which, typically, form the bulk of any given program (database). We have shown that this can always be achieved by transforming the program (after removing any left recursion) such that all resulting clauses begin with a base literal.

Appendix

A permutation can be represented by "cycles". For example, let σ_1 be the following permutation:

$\sigma_1 = \binom{1,2,3}{3,1,2} = (1,2,3)$

The permutation σ_1 can be represented by a cycle $(1,2,3)$ which is said to have a length of 3. We may interpret σ_1 as carrying 1 into 3, 3 into 2 and 2 into 1. When we permute the elements of a set in a cyclic order, we will return to the original sequence, as for example in:

$(1,2,3) \rightarrow (3,1,2) \rightarrow (2,3,1) \rightarrow (1,2,3)$.

A permutation may contain one or more "disjoint" cycles, as for example in:

$\sigma_2 = \binom{1,2,3,4,5,6}{4,5,6,3,2,1} = (1,4,3,6)(2,5)$

Theorem: The order of a permutation is the least common multiple of the lengths of the cycles [FRA82].

Therefore, the order of $\sigma_2 = \frac{(4)(2)}{2} = 4$

This implies that applying the same permutation 4 times will return to the original sequence. For example,

$$(1,2,3,4,5,6) \rightarrow (4,5,6,3,2,1) \rightarrow (3,2,1,6,5,4)$$
$$\rightarrow (6,5,4,1,2,3) \rightarrow (1,2,3,4,5,6)$$

References

[BW84] Brough, D.R. and A. Walker Some Practical Properties of Logic Programming Interpreters. In *Proceedings of the Int'l Conf on Fifth Generation Computer Systems*, Institute of New Generation Computing, Tokyo, Japan, 1984, pp. 149–156.

[ChLe73] Chang, Chin-Liang and Lee, Richard Char-Tung *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.

[FRA82] Fraleigh, John B. *A First Course in Abstract Algebra*, Addison-Wesley, Reading, Massachusetts, 1982, pp. 38-56.

[Har78] Harrison, Michael A. *Introduction to Formal Language Theory*, Addison-Wesley, Reading, Massachusetts, 1978, pp. 111–115.

[KOW79b] Kowalski, R. *Logic for Problem Solving*, North-Holland, New York, 1979.

[KoKu71] Kowalski, R.A. and Kuehner, D., Linear Resolution with Selection Function. In *Artificial Intelligence, 2(1971)*, pp. 227-260.

[LLO84] Lloyd, J.W. *Foundations of Logic Programming*, Springer-Verlag, New York, 1984.

[SIC76] Sickel, S. A search technique for clause interconnectivity graphs. *IEEE Trans. Comput. C-25*, 8 (Aug 1976), 823-834.

[WOLB84] Wos, Larry, Overbeek, Ross, Lusk, Ewing and Boyle, Jim *Automated Reasoning, Introduction and Applications*, Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1984.