NORTH-HOLLAND

TRANSFORMATIONAL METHODOLOGY FOR PROVING TERMINATION OF LOGIC PROGRAMS*

M. R. K. KRISHNA RAO, DEEPAK KAPUR, AND R. K. SHYAMASUNDAR

▷ A methodology for proving the termination of well-moded logic programs is developed by reducing the termination problem of logic programs to that of term rewriting systems. A transformation procedure is presented to derive a term rewriting system from a given well-moded logic program such that the termination of the derived rewrite system implies the termination of the logic program for all well-moded queries under a class of selection rules. This facilitates applicability of a vast source of termination orderings proposed in the literature on term rewriting, for proving termination of logic programs. The termination of various benchmark programs has been established with this approach. Unlike other mechanizable approaches, the proposed approach does not require any preprocessing and works well, even in the presence of mutual recursion. The transformation has also been implemented as a front end to Rewrite Rule Laboratory (RRL) and has been used in establishing termination of nontrivial Prolog programs such as a prototype compiler for ProCoS, PL_0 language. © Elsevier Science Inc., 1998 ◁

1. INTRODUCTION

Termination is an important property of imperative as well as declarative programs, and proving termination is one of the main steps in arriving at a sound methodology and for proving the correctness of programs. Recently, termination of logic programs has attracted a lot of attention, and many approaches are reported in the literature (see De Schreye and Decorte [14] for a comprehensive survey). In this paper, we present a transformational approach for proving termination of logic

*This is a revised and extended version of [24].

Address correspondence to Dr. R. K. Shyamasundar, Tata Institute of Fundamental Research, Computer Science Group, Homi Bhabha Road, Bombay 400 005, India, Email: shyam@tcs.tifr.res.in.

Received March 1995; revised February 1996; accepted March 1997.

programs by reducing the termination problem of logic programs to that of term rewriting systems. The termination problem of term-rewriting systems has been well studied, and many useful techniques and tools have been developed for proving termination of term-rewriting systems. The prime motivation of our approach is to facilitate the use of this vast source of termination techniques and tools in proving termination of logic programs.

Before describing our method, let us discuss the differences between the paradigms of logic programming and term rewriting and see why termination techniques of rewriting cannot be adopted for logic programs in a straightforward way.

1. Unification is the basic step in the computations of logic programs, whereas matching plays a similar role in term rewriting. The backward propagation of substitutions due to unification complicates termination analysis of logic programs, and the termination techniques of rewriting are not directly applicable to logic programs (e.g., a logic program containing a clause `p(f(X))` $\leftarrow$ `p(X)` does not terminate for query $\leftarrow$ `p(Y)`, whereas the corresponding term-rewriting system containing rule `p(f(X))` $\rightarrow$ `p(X)` terminates on all terms).
2. Logic programs have local variables (variables that occur in the body but not in the head of a clause) playing the crucial role of sideways information passing, whereas in term-rewriting literature, it is generally assumed that all of the variables in the right-hand term also occur in the left-hand term of the rewrite rules. The reason for avoiding extra variables on the right-hand sides of rewrite rules is that they trivially lead to nontermination. Almost all of the termination techniques of rewriting work only under this restriction.
3. In general, logic programs are not directed, in the sense that there is no notion of *input* and *output*. A variable (or an argument position) can be used as either input or output; for example, with the factorial program it is possible to ask, "What is factorial of 6?" ($\leftarrow$ `factorial(6,O)`) as well as "What is the value of `I` if the factorial of `I` is 720?" ($\leftarrow$ `factorial(I,720)`). In fact, this invertibility is often seen as the principal difference between logic and functional programming paradigms. Term rewriting is directional in the sense that left-hand terms are replaced by the corresponding right-hand terms.

We present a transformation procedure to derive a term-rewriting system from a given logic program such that termination of the derived term-rewriting system implies termination of the logic program and thereby reduces the termination problem of logic programs to that of term-rewriting systems. To get the directionality, we assume that every predicate has an associated "mode" specifying which arguments are "input" and which are "output" and consider the class of well-moded programs, so that input terms of selected atoms are always ground and there is no backward propagation through input positions. The transformation removes the *local variables* present in the logic programs through a kind of Skolemization procedure, using mode information while deriving term-rewriting systems. The absence of backward propagation through input positions and the local variables in the derived term-rewriting systems facilitate the applicability of termination techniques of rewriting in proving termination of logic programs. The transformation derives the term-rewriting system in an incremental fashion by transforming each clause in the program into a set of rewrite rules. In Section 5, we establish that the given logic program terminates for all well-moded queries under a class of selection rules if the derived term-rewriting system is terminating. To summarize, our method consists of two steps: (i) transforming the given well-moded logic program into a term-rewriting system and then (ii) proving termination of the resulting rewrite system using various techniques available in the literature on term-rewriting systems.

The transformation procedure is purely syntactical and has been implemented as a front end to RRL—a theorem prover based on rewrite techniques—that supports techniques such as recursive path ordering for proving termination of term-rewriting systems in an interactive fashion. The tool developed has been used in establishing termination of a prototype compiler for **ProCoS** language PL_0. This compiler has been developed using Hoare's refinement algebra approach. Refinement algebra provides elegant proofs for partial correctness (ensuring that the compiler only generates correct code) of compilers developed in this approach. A proof of termination ensures that the compiler indeed generates an output (object code). In this respect, our tool plays an important role in the development of provably correct compilers. The fact that termination of this compiler cannot be established by the other mechanizable approaches available in the literature demonstrates the practicality of our approach.

We follow the notations of Lloyd [29] and Apt [2] for logic programming concepts and Dershowitz and Jouannaud [19] for rewriting concepts. The rest of the paper is organized as follows. In Section 2, we give definitions of well-modedness and related concepts. In Section 3, the transformation of logic programs into rewriting systems is explained through examples. Section 4 provides a formal description of the transformation procedure. In Section 5, we prove that the termination of the derived term-rewriting system implies the termination of the logic program for well-moded queries; a brief review of important termination techniques of rewrite systems is also provided. Section 6 briefly discusses the automation of termination proofs of logic programs using our approach. The paper concludes by a comparative evaluation of the methods in Section 7.

2. PRELIMINARIES

In this section, first, we define the notion of well-moded logic programs (queries) and prove some properties of well-moded programs (queries). The moding information essentially specifies which arguments are input arguments and which are output arguments in a predicate. Second, we highlight the basic concepts underlying term-rewriting systems.

2.1. Well-Modedness and Related Concepts

Definition 1. A mode m of an n-ary predicate p is a function from $\{1,\ldots,n\}$ to the set $\{in, out\}$. The set $\{i \mid m(i) = in\}$ is the set of input positions of p and $\{o \mid m(o) = out\}$ is the set of output positions of p.

NOTATION. The terms $invar(L)$ and $outvar(L)$ denote the sets of variables occurring in the input and output positions of a literal L, respectively, and $Var(L) = invar(L) \cup outvar(L)$.

REMARK 1. It may be noted that some predicates may be used in different modes in a single program. We use different subscripts to a predicate to differentiate between different modings (usages).

In the rest of the paper, we assume that the moding information of all of the predicates is available. However, this does not mean that the programmer has to supply this information, as there are many techniques available in the literature (e.g., [15]) for deriving moding information from a given logic program.

The notion of well-moded programs has been invented to constrain the "flow of data" and thereby obtain SLD-derivations with certain desirable properties. One of these desirable properties is the data-drivenness of computations, i.e., input terms of every selected atom are ground. Since groundness of input terms of the selected atom depends on the selection rule employed, two alternatives are possible: (i) to fix a selection rule and consider the class of programs for which the input terms of the selected atom are ground under this section rule and (ii) to consider the class of programs for which there exists at least one selection rule such that the input terms of the selected atom are ground. Obviously, the class of programs considered in the second alternative is larger than the class considered in the first alternative. Here we adopt the second alternative and define the notion of well-modedness independent of the selection rule using the concepts of producers and consumers. Then we give a characterization of the class of selection rules suitable for the execution of any given well-moded program (i.e., computations under those selection rules are data-driven). De Schreye and Decorte [14] call our well-moded programs *well-moded** programs to differentiate between the two notions of well-moded programs.

Definition 2. Let $A \leftarrow B_1, \ldots, B_k$ be a clause and X be a variable occurring in B_i. The atom B_i is a consumer of X if $X \in invar(B_i)$, otherwise B_i is a *producer* of X (i.e., if $X \in Var(B_i) - invar(B_i)$). The head A is a *producer*[1] of variable X if $X \in invar(A)$, and A is a consumer of X if $X \in Var(A) - invar(A)$.

REMARK 2. Note that we say that B_i is a producer of X if $X \in Var(B_i) - invar(B_i))$ rather than $X \in outvar(B_i)$, since we have to consider the possibility of a variable, X, occurring in input as well as output positions of an atom, say A. We resolve the conflict in the situation by saying that A is a consumer of X as we want all the input terms to be ground at the time of selection; in other words, X should have another producer that binds X before A is selected.

Definition 3. The producer-consumer relation of a clause c: $A \leftarrow B_1, \ldots, B_k$ is defined as $\{\langle B_i, B_j \rangle \mid B_i$ and B_j are producer and consumer of a variable X in c respectively$\}$.

Definition 4 (Well-moded programs and queries). A clause c is *well-moded* if (a) its producer-consumer relation is acyclic and (b) every variable in c has at least one producer. A program P is *well-moded* if every clause in it is well-moded. A *well-moded query* is nothing but a well-moded clause without head.

[1] Our notion of *producer* is similar to the notion of *generator* used in Conery and Kibler [10] for studying AND/OR parallelism in logic programming.

Since the producer-consumer relation of a well-moded clause $c{:}A \leftarrow B_1, \ldots, B_n$ is *acyclic*, it defines a partial order $<$ on the atoms in the body of c as follows: $B_i < B_j$ if $\langle B_i, B_j \rangle$ is in the producer-consumer relation of c.

Definition 5. An element $a \in A$ is *minimal* in the poset $(A, \leq)$ if $\forall b \in A$, $b \not< a$.
The following example illustrates these definitions.

Example 1. Consider the following quick-sort program; here, $\leq$ and $<$ are the built-in's with the usual moding information.

```
moding: q(in, out); s(in, in, out, out) and a(in, in, out)
```

```
1. q(nil, nil) ←
2. q(c(H, L), S) ← s(L, H, A, B),q(A, A1), q(B, B1), a(A1,
   c(H, B1), S)
3. s(nil, Y, nil, nil) ←
4. s(c(X, Xs), Y, c(X, Ls), Bs) ← X ≤ Y, s(Xs, Y, Ls, Bs)
5. s(c(X, Xs), Y, Ls, c(X, Bs)) ← X > Y, s(Xs, Y, Ls, Bs)
6. a(nil, X, X) ←
7. a(c(H, X), Y, c(H, Z)) ← a(X, Y, Z)
```

Except for the second clause, the producer-consumer relation of all other clauses is empty. For the second clause, it is {⟨`s(L, H, A, B), q(A, A1)`⟩, ⟨`s(L, H, A, B), q(B, B1)`⟩, ⟨`q(A, A1), a(A1, c(H, B1), S)`⟩, ⟨`q(B, B1), a(A1, c(H, B1), S)`⟩}. It is easy to see that for every clause, (i) the producer-consumer relation is acyclic and (ii) all the variables in it have at least one producer. So the program is well-moded.

In the following lemmas, we capture some of the properties of well-modedness properties.

Lemma 1. If $H \leftarrow B_1, \ldots, B_n$ is a well-moded clause and X is a variable in outvar(H), then (a) $X \in$ invar(H) or (b) there exist a B_i in the body such that $X \in$ outvar(B_i).

PROOF. By Definition 4, every variable in a well-moded clause has at least one producer, i.e., either (a) H is a producer of X or (b) some B_i is a producer of X. By Definition 2, H is a producer of X only if $X \in invar(H)$ and B_i is a producer of X only if $X \in (Var(B_i) - invar(B_i)) = (outvar(B_i) - invar(B_i))$. □

Lemma 2. If $H \leftarrow$ is a well-moded unit clause, then $outvar(H) \subseteq invar(H)$.

PROOF. Follows from Definitions 2 and 4 and the above lemma. □

Lemma 3. Let $A \leftarrow B_1, \ldots, B_n$ be a well-moded clause and B_i be a minimal element under the partial order defined by the producer-consumer relation of the clause. Then, $invar(B_i) \subseteq invar(A)$.

PROOF. Since B_i is a minimal element under the partial order defined by the producer–consumer relation of the clause, there is no pair $\langle B_j, B_i \rangle$ in the producer–consumer relation. Therefore, input variables of B_i (if any) are not produced by other atoms in the body. Hence they should be produced by A; that is, $invar(B_i) \subseteq invar(A)$. □

Lemma 4. Let $\leftarrow B_1, \ldots, B_n$ *be a well-moded query and* B_i *be a minimal element under the partial order defined by the producer–consumer relation. Then the input terms of* B_i *are ground.*

PROOF. By Definition 4, well-moded query is a well-moded clause without head and by the above lemma $invar(B_i) \subseteq invar(head) = \phi$. Therefore the set of input variables of B_i is empty and hence the input terms of B_i are ground. □

In the following, we characterize selection rules that are suitable for well-moded programs. First, we formally define the selection rule.

Definition 6. A *computation rule* (or *selection rule*) [29] is a function from the set of goals to the set of atoms such that the value of the function for a goal is an atom, called *selected* atom, in that goal.

If $G_0, G_1, \ldots, G_n$ is an SLD-derivation such that $G_i = G_j$, $i \neq j$, then the selected atoms of G_i and G_j are the same [29].[2] This notion can be extended to the clauses in the following way: given a clause, the selection rule gives an evaluation order among the atoms in the body of the clause. It can be captured by a partial order (if $l_i < l_j$ in the partial order, it means that l_i should be selected before l_j is selected).

The definition of well-moded programs (queries) given earlier is very concise and is a generalization of the existing notions. The earlier notions are closely linked to the Prolog's selection rule, and the producers of a variable should precede the consumers of that variable in the textual order [16]. Our notion is not linked with any selection rule. One of our aims has been to define the notion of well-modedness independent of selection rule, and we have been able to achieve this by saying that the producer–consumer relation is acyclic. Our definition is quite general as compared to other definitions. However, it still does not completely capture some notions. For instance, in a well-moded Prolog program (using another notion), the producer–consumer relation can be cyclic; it can be seen that we exclude such a possibility. Once the selection rule is fixed, certain cycles can indeed be handled; for example,

```
head(X, Y, Z,W):-a1(X, Y), a2(Y, Z), a3(Z, W, Y)
```

with modes

```
head(in, out, out, out); a1(in, out); a2(in, out)
                                    and a3(in, out, out)
```

is well-moded w.r.t. Prolog's selection rule, although its producer–consumer relation is cyclic (`Z` is produced by `a2` and consumed by `a3` and `Y` is produced by `a3` and consumed by `a2`). In fact, we have used some of these notions in the context of application of our method to GHC programs.

Now we characterize the class of selection rules suitable for the execution of a given well-moded logic program.

[2] Apt [2] considered a more general notion of selection rule that also takes the history of the derivation into account in selecting an atom from the goal. With such a selection rule it is possible to select two different atoms in G_i and G_j, even though $G_i = G_j$. A selection rule that selects leftmost and rightmost atoms alternately is an example.

We would be interested in selection rules satisfying the following property:

Consider the following SLD-resolution step. Let $G_i = \leftarrow q_1(\cdots), \ldots, q_m(\cdots)$ be a goal, $q_j(\cdots)$ be the selected atom, $A \leftarrow B_1, \ldots, B_l$ be the input clause and σ be the mgu of $q_j(\cdots)$ and A. Then the goal G_{i+1} will be $\leftarrow q_1(\cdots)\sigma, \ldots, q_{j-1}(\cdots)\sigma,\ B_1\sigma, \ldots, B_l\sigma, q_{j+1}(\cdots)\sigma, \ldots, q_m(\cdots)\sigma$. We assume that the evaluation order induced by the selection rule satisfies the following: If the evaluation order for goals G_i, G_{i+1} and $\leftarrow B_1, \ldots, B_l$ are $\prec G_i$, $\prec G_{i+1}$ and $\prec Body$, respectively, then

1. $B_{l_1}\sigma \prec_{G_{i+1}} B_{l_2}\sigma$ if and only if $B_{l_1} \prec_{Body} B_{l_2}$.
2. $B_{l_3}\sigma \prec_{G_{i+1}} q_{j_2}(\cdots)\sigma$ if and only if $1 \le l_3 \le l$ and $q_j(\cdots) \prec_{G_i} q_{j_2}(\cdots)$
3. $q_{j_1}(\cdots)\sigma \prec_{G_{i+1}} q_{j_2}(\cdots)\sigma$ if and only if $j_1 \ne j \ne j_2$ and $q_{j_1}(\cdots) \prec_{G_i} q_{j_2}(\cdots)$.

The above conditions essentially say that to obtain the evaluation order of G_{i+1}, extend the evaluation order of G_i with the evaluation order of the input clause and make the order of the new atoms, B's, smaller than those atoms in G_i that were greater than the selected atom. For example, in Prolog's computation strategy, the leftmost atom is selected every time and the body of the input clause is added at the left end of the goal. Thus the newly added atoms (leftmost) will be selected first.

The following definition characterizes the class of selection rules suitable for a given well-moded program.

Definition 7. A selection rule is *implied* by the moding information of a well-moded program if and only if the partial order induced by the selection rule for each clause in the program is an *extension* of the producer–consumer relation of that clause.

When P is a well-moded program and Q is a well-moded query, we say that a selection rule S is implied by $P \cup \{Q\}$ if S is implied by the moding information of P and the evaluation order given by S for Q is an extension of the producer–consumer relation of Q.

One can easily check whether a given selection rule is implied by the moding information of a given well-moded program, by checking whether the evaluation order given for each clause by the selection rule is an extension of the producer–consumer relation of that clause.

Theorem 1. It is decidable whether a given selection rule is implied by the moding information of a given well-moded program.

Proof. Since every logic program has finitely many clauses, it is always possible to check whether the evaluation order given for each clause by the selection rule is an extension of the producer–consumer relation of that clause. In fact, it is easy to visualize an algorithm with time complexity $O(n)$, where n is the number of pairs in the relation for verifying the property. □

Example 2. It is very easy to see that the textual order of the atoms in the body of each clause in the quick-sort program (given in Example 1) is a linear extension of the producer–consumer relation. Therefore, Prolog's left-to-right selection rule is *implied* by the moding information.

Example 3. The following permutation program is not well-moded according to earlier definitions (e.g., the definition given in [16, 32]—consumer $ap_1(X1s, c(X, X2s), Xs)$ of variables `X1s` and `X2s` precede their producer `ap₂(X1s, X2s, Zs)`) but is well-moded according to our definition, and the right-to-left selection rule is implied by the moding information.

```
moding: perm (in, out); ap₁ (out, out, in)
                                        and ap₂ (in, in, out).
```

```
1. ap₁(nil, X, X) ←
2. ap₁(c(H, X), Y, c(H, Z)) ← ap₁(X, Y, Z)
3. ap₂(nil, X, X) ←
4. ap₂(c(H, X), Y, c(H, Z)) ← ap₂(X, Y, Z)
5. perm(nil, nil) ←
6. perm(Xs, c(X, Ys)) ← ap₁(X1s, c(X, X2s), Xs), ap₂(X1s,
   X2s, Zs), perm(Zs, Ys).
```

The clauses defining the predicates `ap₁` and `ap₂` are essentially the clauses in the standard append program.

In the following, we establish that the computations of well-moded programs under the implied selection rules are data-driven.

Definition 8. Let P be a well-moded program and Q be a well-moded query. An evaluation (SLD-derivation) of $P \cup \{Q\}$ is said to be *data driven* if at every resolution step, the selected atom is ground on all of its input positions.

Theorem 2. If P is a well-moded program, Q is a well-moded query and S is a selection rule implied by $P \cup \{Q\}$, then every SLD-derivation of $P \cup \{Q\}$ is a data-driven evaluation.

PROOF. See Appendix. □

2.2. *Term-Rewriting Systems*

In this subsection, we briefly explain the basic concepts of term-rewriting systems.

Definition 9. A term-rewriting system (TRS, for short) $\mathcal{R}$ is a pair $(\mathcal{F}, R)$ consisting of a set $\mathcal{F}$ of function symbols and a set R of rewrite rules of the form $l \to r$ satisfying

(i) $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, the set of terms built from functions in $\mathcal{F}$ and variables in $\mathcal{X}$,
(ii) left-hand-side l is not a variable, and
(iii) $Var(r) \subseteq Var(l)$.

A rule $l \to r$ applies to term t in $\mathcal{T}(\mathcal{F}, \mathcal{X})$ if a subterm s of t matches with l through some substitution σ, i.e., $s \equiv l\sigma$, and the rule is applied by replacing the subterm s in t by $r\sigma$, resulting in a new term u. This is formalized in the following definitions.

Definition 10. A context $C[,\dots,]$ is a term in $\mathcal{T}(\mathcal{F}\cup\{\square\},\mathcal{X})$. If $C[,\dots,]$ is a context containing n occurrences of $\square$ and $t_1,\dots,t_n$ are terms, then $C[t_1,\dots,t_n]$ is the result of replacing the occurrences of $\square$ from left to right by $t_1,\dots,t_n$. A context containing precisely one occurrence of $\square$ is denoted $C[\ ]$.

Definition 11. The rewrite relation $\Rightarrow_{\mathcal{R}}$ induced by a TRS $\mathcal{R}$ is defined as follows: $s \Rightarrow_{\mathcal{R}} t$ if there is a rewrite rule $l \to r$ in $\mathcal{R}$, a substitution σ, and a context $C[\ ]$ such that $s \equiv C[l\sigma]$ and $t \equiv C[r\sigma]$.

We say that s *reduces* to t in *one rewrite* (*or reduction*) *step* if $s \Rightarrow_{\mathcal{R}} t$ and say s *reduces* to t if $s \Rightarrow^*_{\mathcal{R}} t$ (the relation $\Rightarrow^*_{\mathcal{R}}$ is the reflexive-transitive closure of $\Rightarrow_{\mathcal{R}}$).

Definition 12. A term-rewriting systems $\mathcal{R}$ is terminating if there is no infinite rewriting derivation $t_1 \Rightarrow_{\mathcal{R}} t_2 \Rightarrow_{\mathcal{R}} t_3 \Rightarrow_{\mathcal{R}} \cdots$.

3. TRANSFORMING A LOGIC PROGRAM INTO A REWRITE SYSTEM

Our main objective is to reduce the termination problem of logic programs to the termination problem of term-rewriting systems so that we can use the many techniques available in the rewriting literature. It may be noted that term-rewriting systems do not have local variables, and all of the termination results of term-rewriting systems crucially depend on this property. With this as the motivation, we first eliminate local variables by introducing Skolem functions.

For each n-ary predicate p having a moding with k output positions, we introduce k new function symbols $p^1,\dots,p^k$ of arity $n-k$. These k-function symbols correspond to the k output positions of the predicate p. (If $k=0$, we introduce an n-ary function symbol p^0.) Then we construct a set of rewrite rules to compute these new functions. For example, in Example 8 discussed in the sequel, we associate three binary function symbols $\mathtt{r}^1$, $\mathtt{r}^2$ and $\mathtt{r}^3$ with the predicate `r` with moding(`in, in, out, out, out`). These functions take first two arguments of `r` as inputs and give one output each, corresponding to the third, fourth and fifth arguments of `r`. Rewrite rules for these functions are constructed in Example 8. In the following, we illustrate the transformational approach through a series of examples.

Example 4. Consider the following multiplication program:

```
moding: add (in, in, out) and mult (in, in, out)

add(0, Y, Y) ←
add(s(X), Y, s(Z)) ← add(X, Y, Z)
mult(0, Y, 0) ←
mult(s(X), Y, Z) ← mult(X, Y, Z1), add(Z1, Y, Z)
```

From these clauses and the moding information, we obtain the following rewriting rules:

1. Since the output of predicate add for inputs 0 and `Y` is `Y`, we get $\mathtt{add}^1\mathtt{(0,Y)} \to \mathtt{Y}$.
2. Since the output of `mult` for inputs 0 and `Y` is 0, we get $\mathtt{mult}^1\mathtt{(0,Y)} \to \mathtt{0}$.

3. In the second clause, the output of add for inputs s(X) and Y is s(Z), where Z is the output of add for the inputs X and Y. We get add^1(S(X),Y) $\rightarrow$ s(add^1(X,Y)).
4. In the last clause, the output of mult for inputs s(X) and Y is Z, where Z is the output of add for the inputs Z1 and Y, where Z1 is the output of mult for inputs X and Y. So we get mult^1(s(X),Y) $\rightarrow$ add^1(Z1,Y), where Z1 = mult^1(X,Y).

The resulting rule is mult^1(s(X),Y) $\rightarrow$ add^1(mult^1(X, Y),Y).

Example 5. Consider the permutation program given in Example 3. It was shown that the right-to-left selection rule is implied by the moding. First, we get the following rules from clauses 1 to 5:

ap_1^1(nil,X) $\rightarrow$ X

ap_1^1(c(H, X), Y) $\rightarrow$ c(H, ap_1^1(X, Y))

ap_2^1(X) $\rightarrow$ nil

ap_2^1(c(H, Z)) $\rightarrow$ c(H, ap_2^1(Z))

ap_2^2(X) $\rightarrow$ X

ap_2^2(c(H, Z)) $\rightarrow$ ap_2^2(Z)

perm^1(nil) $\rightarrow$ nil

Let us now consider clause 6. The output of predicate perm for input c(X, Ys) is Xs, where Xs is the output of predicate ap_1 for inputs X1s, c(X, X2s), i.e., Xs is ap_1^1(X1s, c(X, X2s)). Here, X1s and X2s are outputs of predicate ap_2 for input Zs and Zs is the output of perm for input Ys. Therefore, we get the rewrite rule

perm^1(c(X,Ys)) $\rightarrow$ ap_1^1(ap_2^1(perm^1(Ys)), c(X, ap_2^2(perm^1(Ys)))).

When a nonvariable term appears in an output position of a body literal, we need to introduce *inverse functions* as illustrated in the following example.

Example 6. Consider the permutation program given in Example 3 with moding perm(in, out); ap_1(out, out, in) and ap_2 (in, in, out). It is easy to see that we get the following rules from the clauses 1 to 5:

ap_1^1(X) $\rightarrow$ nil

ap_1^1(c(H, Z)) $\rightarrow$ c(H,ap_1^1(Z))

ap_1^2(X) $\rightarrow$ X

ap_1^2(c(H,Z)) $\rightarrow$ ap_1^2(Z)

ap_2^1(nil, X) $\rightarrow$ X

ap_2^1(c(H, X), Y) $\rightarrow$ c(H, ap_2^1(X, Y))

perm^1(nil) $\rightarrow$ nil

Let us now consider clause 6. The output of predicate perm for input Xs is c(X, Ys), where Ys is the output of predicate perm for input Zs and X is a part of the second output of predicate $\mathtt{ap}_1$ for input `Xs`. How do we extract `X` and `X2s` from `c(X, X2s)`?

They can be extracted through inverse functions of `c`. The operators `car` and `cdr` can be used as inverse functions of `c`, and we get a rewrite rule $\mathtt{perm}^1\mathtt{(Xs)} \rightarrow \mathtt{c(car(ap}_1^2\mathtt{(Xs)),perm}^1\mathtt{(ap}_2^1\mathtt{(ap}_1^1\mathtt{(Xs),\ cdr(ap}_1^2\mathtt{(Xs)))))}$. To evaluate the functions `car` and `cdr`, we add the following rules: `car(c(H, T))` $\rightarrow$ `H` and `cdr(c(H, T))` $\rightarrow$ `T`. Thus, for clause (6), we get the following rules:

$$\mathtt{perm}^1\mathtt{(Xs)} \rightarrow \mathtt{c(car(ap}_1^2\mathtt{(Xs)),\ perm}^1\mathtt{(ap}_2^1\mathtt{(ap}_1^1\mathtt{(Xs),}$$
$$\mathtt{cdr(ap}_1^2\mathtt{(Xs)))))}$$
$$\mathtt{car(c(H,\ T))} \rightarrow \mathtt{H}$$
$$\mathtt{cdr(c(H,\ T))} \rightarrow \mathtt{T}.$$

As shown in the above example, inverse functions are needed when a nonvariable term occurs in an output position of an atom in the body. Appropriate inverse functions are generated as follows:

1. Build a representation of the nonvariable output term.
2. Identify for each variable in this term (and consumed by other atoms in the clause) a path from root to an occurrence of that variable in the above tree.
3. Traverse upward path (from leaf to root), collecting suitable inverse function symbols, which will be used in constructing the right-hand sides of the rewrite rules as illustrated in the following example.

Example 7. Consider the following clause:

```
moding: a(in, out); b(in, in, out); and c (in, out)
a(X, Y) ← b(X, 0, f(X, g(h(0, Z), X), X, 1)), c(Z, Y)
```

A nonvariable term `f(X, g(h(0, Z), X), X, 1)` is occurring in the output position of `b` and the local variable `Z` occurring in this term is consumed by atom `c(Z, Y)`. Function symbols `f`, `g`, and `h` occur in the path from the root to variable Z, and the appropriate inverse functions are collected as follows.

Since Z is the second argument of `h`, the inverse function $\mathtt{h2}^{-1}$ is collected (and a rewrite rule $\mathtt{h2}^{-1}\mathtt{(h(X1,\ X2))} \rightarrow \mathtt{X2}$ is added to the rewrite system). Since the subterm `h(0, Z)` is the first argument of `g`, the inverse function symbol $\mathtt{g1}^{-1}$ is collected (and a rewrite rule $\mathtt{g1}^{-1}\mathtt{(g(X1,\ X2))} \rightarrow \mathtt{X1}$ is added). Since $\mathtt{g}(\cdots)$ is the second argument of `f`, the inverse function symbol $\mathtt{f2}^{-1}$ is collected (and a rewrite rule $\mathtt{f2}^{-1}\mathtt{(f(X1,\ X2,\ X3,\ X4))} \rightarrow \mathtt{X2}$ is added). The transformation procedure derives the following rewrite rules for the above clause:

$$\mathtt{a}^1\mathtt{(X)} \rightarrow \mathtt{c}^1\mathtt{(h2}^{-1}\mathtt{(g1}^{-1}\mathtt{(f2}^{-1}\mathtt{(b}^1\mathtt{(X,\ 0)))))}$$
$$\mathtt{h2}^{-1}\mathtt{(h(X1,\ X2))} \rightarrow \mathtt{X2}$$
$$\mathtt{g1}^{-1}\mathtt{(g(X1,\ X2))} \rightarrow \mathtt{X1}$$
$$\mathtt{f2}^{-1}\mathtt{(f(X1,\ X2,\ X3,\ X4))} \rightarrow \mathtt{X2}$$

In the above examples, all of the variables occurring in the output positions of an atom in the body are also occurring either in output positions of the head or in input positions of some other atom in the body. And the rewrite systems derived in both of the examples capture the termination of the corresponding logic programs correctly. The above transformation is basically capturing the data flow in the program execution. When there are some variables occurring only in output positions of atoms in the body, we need to *add additional rewrite rules*, as illustrated below.

Example 8. Consider the following (nonterminating) logic program:

```
moding: a(in, out); b(in, out); c(in, out) and r(in, in,
  out, out).
a(X, f(X)) ←
b(X, X) ←
c(X, Y) ← a(X, Z), r(X, Z, Y, Z1)
r(X, Y, O1, O2) ← b(X, O1), c(Y, O2)
```

We get the following rewrite system according to the above transformation:

$$\begin{aligned}&a^1(X) \to f(X)\\&b^1(X) \to X\\&c^1(X) \to r^1(X, a^1(X))\\&r^1(X, Y) \to b^1(X)\\&r^2(X, Y) \to c^1(Y)\end{aligned}$$

It is easy to see that this rewrite system captures the data flow in the program execution correctly. This rewrite system is terminating (it can be proved using recursive path ordering with precedence $r^2 > c^1 > a^1 > f$, $c^1 > r^1 > b^1$), whereas the above program is a nonterminating one.

For the query ← c(t, Y), where t is a ground term, the program has an infinite SLD-derivation:

```
← c(t, Y)
← a(t, Z), r(t, Z, Y, Z1)
← r(t, f(t), Y, Z1)
← b(t, Y), c(f(t), Z1)
← c(f(t), Z1)
    ⋮
← c(f(f(t)), Z')
    ⋮
```

In the rewrite system, we compute $c^1(t)$ for some ground term t through the rewrite derivation $c^1(t) \Rightarrow r^1(t, a^1(t)) \Rightarrow r^1(t, f(t)) \Rightarrow b^1(t) \Rightarrow t$, whereas for the execution of c(t, Y) in the logic programming paradigm, we need to execute r(t, f(t), Y, Z1), which in turn needs the execution of b(t, Y) and $c(f(t), Z1)$—thus leading to an infinite SLD-derivation. Computing c^1 in rewriting involves only a partial computation of r (i.e., computation of r^1 only), whereas this kind of partial computation is not possible in logic programming and execution of r does not stop after computing r^1 but continues to evaluate r^2 (here it needs to execute c, which leads to a loop).

In the above example, the computation of the second output of r(⋯) causes the looping of the logic program, and the computation does not contribute anything useful to the evaluation of the initial query. The corresponding rewrite system does not enter any loop, as it does not involve the computation of the second output of r(⋯). To cover all of the computation paths in the SLD-trees, we include additional rewrite rules reflecting the (possible) nontermination due to the computation of unnecessary values (the computation of these values does not provide any information directly or indirectly to the head). The derivation of these rewrite rules will be clear in the next section.

Example 8 (continued). We include the following rewrite rules to capture the unnecessary computations:

$$\mathtt{c}^{1}(\mathtt{X}) \to \#(\mathtt{r}^{2}(\mathtt{X},\ \mathtt{a}^{1}(\mathtt{X})))$$
$$\mathtt{c}^{1}(\mathtt{X}) \to \#(\mathtt{r}^{3}(\mathtt{X},\ \mathtt{a}^{2}(\mathtt{X})))$$

It is easy to see that the resulting rewrite system is a nonterminating one.

4. FORMAL DESCRIPTION OF THE TRANSFORMATION PROCEDURE

Although input and output positions of a predicate can combine in all possible ways, for notational convenience we write all of the input positions first, followed by all of the output positions. We write $p(t_{i_1},\ldots,t_{i_j}, t_{o_1},\ldots,t_{o_k})$ to denote an atom $p(\cdots)$ containing the terms $t_{i_1},\ldots,t_{i_j}$ in input positions and $t_{o_1},\ldots,t_{o_k}$ in output positions.

The main step in our transformation is the elimination of local variables. Basically, the right-hand sides of rewrite rules are derived from an output term of the head by repeatedly replacing a local variable (to be precise, variables in $Var(c) - invar(head)$) by a term corresponding to one of its producers. When a variable has more than one producer, one has to consider all possible choices. The following function, ELIMINATE-LOC-VARS, has been designed to perform the elimination of local variables repeatedly until there is no local variable to replace. This function needs the computation of the set of producers for each local variable.

It is easy to see that computation of an output term in the body does not provide any information directly or indirectly to the head or any other atom if and only if none of the variables in that output term have any consumer. Therefore, the unnecessary computations can be captured by computing the set of variables without consumers. To derive the rewrite rules corresponding to unnecessary computations, we compute the following sets *Consvar* and *Unsry* for each clause in the program. In all, the transformation procedure computes the following sets for each clause $c: head \leftarrow body$ in the program:

1. $Prod(X) = \{\langle p^l(t_{i_1},\ldots,t_{i_j}), t_{o_l}\rangle \mid p(t_{i_1},\ldots,t_{i_j}, t_{o_1},\ldots,t_{o_k})$ is an atom in the body of the clause c, $X \in Var(t_{o_l})$ and $X \notin Var(\{t_{i_1},\ldots,t_{i_j}\})\}$, the set of producers, for each variable X not occurring in input positions of the head.
2. $Consvar = \{X \in Var(c) - invar(head) \mid X \in outvar(head)$ or X occurs in an input position of an atom in the body$\}$, the set of variables in the clause *consumed at least once*.
3. $Unsry = \{p^l(t_{i_1},\ldots,t_{i_j}) \mid Var(t_{o_l}) \cap Consvar = \phi\} \cup \{q^0(s_{i_1},\ldots,s_{i_k}) \mid$ predicate q does not have output positions$\}$, where $p(\cdots)$ and $q(\cdots)$ are atoms in the body.

This set corresponds to the set of computations (we call them *unnecessary computations*) that do not contribute to the outputs of head directly or indirectly.

The algorithm TRANSFORM and the function ELIMINATE-LOC-VARS are formally described below.

```
algorithm TRANSFORM (P :in; $R_P$ : out);
begin
  $R_P := \phi$;            {* $R_P$ contains rewrite rules. *}
  for  each clause $c$: $a(t_{i_1},\ldots,t_{i_k}, t_{o_1},\ldots,t_{o_{k'}}) \leftarrow B_1,\ldots,B_n \in P$ do
    begin  $INhead := Var(\{t_{i_1},\ldots,t_{i_k}\})$
      Compute $Consvar$ and $Unsry$;
      Compute $Prod(X)$ for every variable in $Var(c) - INhead$;
      for $j := 1$ to $k'$ do
        begin
          $S :=$ ELIMINATE-LOC-VARS($\{t_{o_j}\}$);
              {* $S$ contains the right-hand sides of the rules in $R_P$ *}
          $R_P := R_P \cup \{a^j(t_{i_1},\ldots,t_{i_k}) \rightarrow t \mid t \in S\}$
        end;
    {* Following code derives rewrite rules corresponding to unnecessary computations. *}
      $S :=$ ELIMINATE-LOC-VARS($Unsry$);
      $R_P := R_P \cup \{a^{k'}(t_{i_1},\ldots,t_{i_k}) \rightarrow \#(t) \mid t \in S\}$
    end
end TRANSFORM.
function ELIMINATE-LOC-VARS($T$)
```

{* *This function goes on replacing the local variables in the set of terms T by the terms corresponding to their producers as long as there are local variables. Since the producer–consumer relation of every well-moded clause is acyclic, this function is guaranteed to terminate. Here, INhead is a global variable.* *}

```
begin $V := Var(T) - INhead$;
  while $V \neq \phi$ do
    begin
      for each $X \in V$ do
        begin $T' := \phi$;
          for each $\langle p^l(\cdots), t\rangle \in Prod(X)$ do
            if $t = X$ then $T' := T' \cup T\{X/p^l(\cdots)\}$
              {* Replace local var X by its producer-term. *}
            else if $t = f(X)$ then
                begin
                  $T' := T' \cup T\{X/f^{-1}(p^l(\cdots))\}$;   {* Introduce inverse functions. *}
                  $R_P := R_P \cup \{f^{-1}(X)) \rightarrow X\}$
                end;
          $T := T'$
        end;
      $V := Var(T) - INhead$
    end;
  Return($T$)
end ELIMINATE-LOC-VARS;
```

The following series of examples illustrates the transformation procedure.

Example 9. For the quick-sort program given in Example 1, the algorithm TRANSFORM derives the following term rewriting system:

1. q^1(nil) $\rightarrow$ nil
2. q^1(c(H, L)) $\rightarrow$ a^1(q^1(s^1(L, H)), c(H, q^1(s^2(L, H))))
3. s^1(nil, Y) $\rightarrow$ nil

3′. s^2(nil, Y) $\rightarrow$ nil

4. s^1(c(X, Xs), Y) $\rightarrow$ c(X, s^1(Xs, Y))

4′. s^2(c(X, Xs), Y) $\rightarrow$ s^2(Xs, Y)

5. s^1(c(X, Xs), Y) $\rightarrow$ s^1(Xs, Y)

5′. s^2(c(X, Xs), Y) $\rightarrow$ c(X, s^2(Xs, Y))

6. a^1(nil, X) $\rightarrow$ X
7. a^1(c(H, X), Y) $\rightarrow$ c(H, a^1(X, Y))

Here, we explain how rule 2 is derived from the second clause. The other rules can be derived in a similar fashion. The head q(c(H, L), S) contains c(H, L) in the input position and variable S in the output position; hence the left-hand side of the rewrite rule is q^1(c(H, L). Producers of variables not occurring in the input positions of the head are as follows: *Prod*(A) = {⟨s^1(L, H), A⟩}, *Prod*(B) = {⟨s^2(L, H), B⟩}, *Prod*(A1) = {⟨q^1(A), A1⟩}, *Prod*(B1) = {⟨q^1(B), B1⟩}, *Prod*(S) = {⟨a^1(A1, c(H, B1)), S⟩}. To construct the right-hand term, algorithm TRANSFORM calls function ELIMINATE-LOC-VARS with argument $T = \{S\}$. Values of T at the end of various iterations of the **while** loop in ELIMINATE-LOC-VARS are given below.

Iteration 1 T = {a^1(A1, c(H, B1))} *variable S is replaced by its producer.*

Iteration 2 T = {a^1(q^1(A), c(H, q^1(B)))} *local variables* A1 *and* B1 *are replaced by their producers.*

Iteration 3 T = {a^1(q^1(s^1(L, H)), c(H, (q^1(s^2(L, H))))} *local variables* A *and* B *are replaced.*

Since there are no local variables in T after the third iteration, ELIMINATE-LOC-VARS returns this T to TRANSFORM, which produces the rewrite rule 2 given above.

Example 10. Let us consider the third clause of the program given in Example 8:

c(X, Y) ← a(X, Z), r(X, Z, Y, Z1, Z2)

The following producers of the variables are computed:

prod(Z) = {⟨a^1(X), Z⟩},

prod(Y) = {⟨r^1(X, Z), Y⟩},

prod(Z1) = {⟨r^2(X, Z), Z1⟩},

prod(Z2) = {⟨r^3(X, Z), Z2⟩},

and *Unsry* = {r^2(X, Z), r^3(X, Z)}.

The function ELIMINATE-LOC-VARS is first invoked with input $T = \{Y\}$. The values of T at the end of various iterations of the **while** loop are {Y}, {r^1(X, Z)}, {r^1(X, a^1(X))} and we get the rewrite rule c^1(X) $\rightarrow$ r^1(X, a^1(X)).

The function ELIMINATE-LOC-VARS is then invoked with input T = *Unsry* = {r^2(X, Z), r^3(X, Z)}. The value of T after the first iteration of the **while** loop is {r^2(X, a^1(X)), r^3(X, a^1(X))}, which does not have any local variables.

Corresponding to this set, algorithm TRANSFORM adds the following two rewrite rules:

$$\mathtt{c}^1\mathtt{(X)} \to \#(\mathtt{r}^2\mathtt{(X,\ a}^1\mathtt{(X))}) \text{ and } \mathtt{c}^1\mathtt{(X)} \to \#(\mathtt{r}^3\mathtt{(X,\ a}^1\mathtt{(X))})$$

Example 11. Let us consider the following program:

$$\mathtt{p} \leftarrow \mathtt{q},\ \mathtt{p}$$

Here, we have propositions (no in/out arguments). So the associated function symbols $\{\mathtt{p}^0,\ \mathtt{q}^0\}$ are of arity zero (i.e., *constants*). The value of *Unsry* is $\{\mathtt{p}^0,\ \mathtt{q}^0\}$. The algorithm TRANSFORM derives R_P with the following two rewrite rules:

$$\mathtt{p}^0 \to \#(\mathtt{q}^0) \quad \text{and} \quad \mathtt{p}^0 \to \#(\mathtt{p}^0).$$

5. FORMAL CORRECTNESS

In this section, we first prove that the algorithm TRANSFORM terminates for a given input and study some properties of the rewrite system derived by the algorithm TRANSFORM from a given well-moded logic program. Then we establish that termination of the derived rewriting system implies termination of the logic program for all well-moded queries under all selection rules implied by the moding information. Figure 1 gives the interdependence of the technical lemmas and theorems leading to the proof of our main result (Theorem 5).

Lemma 5. The algorithm TRANSFORM *terminates.*

PROOF. Since there are only a finite number of clauses in the program and every predicate has a finite number of output positions, the number of iterations in the **for**-loop is finite. Since each body has a finite number of atoms, *Unsry* is finite for every clause. Therefore, there are only a finite number of calls to the function ELIMINATE-LOC-VARS, and it is enough to prove termination of ELIMINATE-LOC-VARS for proving termination of TRANSFORM.

It is easy to see that for proving termination of ELIMINATE-LOC-VARS, it is enough to prove termination of the **while** loop. The main step in the **while** loop can be abstracted as follows: *application of substitution* $\{X/p^j(\cdots)\}$ *to the terms in* T, *where* $p^j(\cdots)$ *is a producer of* X. That is, an occurrence of a variable in a term corresponding to its consumer is replaced by its producer. Since the producer–consumer relation of a well-moded clause is acyclic, the **while** loop is bound to terminate. □

5.1. Properties of the Derived TRS

In this subsection we study certain properties of the derived term-rewriting systems using the transformation procedure from the given well-moded programs. Our

$$\begin{tikzcd}
\text{Lem. 7} \arrow[r] \arrow[rr, bend right] & \text{Lem. 8} \arrow[r] \arrow[rrr, bend left] & \text{Lem. 9} \arrow[r] & \text{Lem. 11} \arrow[r] & \text{Th. 4} \arrow[r] & \text{Th. 5} \arrow[r] & \text{Th. 6} \\
 & & & \text{Lem. 10} \arrow[u] & & &
\end{tikzcd}$$

FIGURE 1. Road-map of the technical results.

basic aim is to establish that the termination of the derived term-rewriting system implies the termination of the logic program. Toward such a goal, we show that corresponding to each resolution step in the SLD-derivations, there is at least one reduction step in the rewrite derivations.

The following lemma establishes that there are no extra variables on the right-hand side of the rewrite rules in R_P, and thus all of the termination techniques of rewriting systems can be used in proving the termination of logic programs.

Lemma 6. $Var(r) \subseteq Var(l)$ for each rewrite rule $l \to r$ in R_P.

PROOF. Each rule associated with inverse function is of the form $f_i^{-1}(f(X_1, \ldots, X_n)) \to X_i$, and hence the lemma holds for such rewrite rules. Other rewrite rules are of the form $p^j(\overline{t_{in}}) \to t$ or $p^j(\overline{t_{in}}) \to \#(t)$, where $\overline{t_{in}}$ is the sequence of input terms of a head and t is a term in the set T returned by ELIMINATE-LOC-VARS. The function ELIMINATE-LOC-VARS terminates with the condition $(Var(T) - INhead) = \phi$. Therefore, $Var(T) \subseteq INhead$ and hence $Var(t) \subseteq INhead = Var(p^j(\overline{t_{in}})) = Var(l)$. Thus, $Var(r) \subseteq Var(l)$ for each rule $l \to r$ in R_P. □

The basic ingredient in the transformation is the *application of substitution $\{X/p^j(\cdots)\}$ to the terms in T, where $p^j(\cdots)$ is a producer of X.* A study of the derived rewrite rules involves a study of these substitutions and their effect. We need the following notation.

Definition 13. Let $c: head \leftarrow body$ be a well-moded clause and V be the set of variables $Var(c) - invar(head)$. For any variable X, we denote by $ELV(X)$, the set of terms returned by the function ELIMINATE-LOC-VARS for input $\{X\}$. We denote by Θ the set of substitutions $\{\sigma \mid X\sigma \in ELV(X)$ for each variable $X \in V\}$. For any term t, $\Theta(t)$ denotes the set $\{\sigma \mid X\sigma \in ELV(X)$ for each variable $X \in Var(t)\}$. We call the substitutions in Θ *Skolem substitutions*.

REMARK 3. If X is a variable in a term $s \equiv C[X]$ for some context $C[\ ]$, we can construct an appropriate context $C'[\ ]$ of inverse functions such that $C'[C[X]]$ can be reduced to X by the rewrite rules defining the inverse functions. In the sequel, we use the phrase "context of inverse functions" to mean the context built from the inverse functions as illustrated.

Lemma 7. Let $c: head \leftarrow body$ be a well-moded clause, Θ be the set of its Skolem substitutions and X be a variable in $Var(c) - invar(head)$ such that $\langle p^l(t_{i_1}, \ldots, t_{i_n}), t_{o_1} \rangle$ is an element in $Prod(X)$ and $q(s_{i_1}, \ldots, s_{i_k}, s_{o_1}, \ldots, s_{o_{k'}})$ is a consumer of X. Then,

1. *There exists a Skolem substitution σ in Θ such that $X\sigma \equiv C[p^l(t_{i_1}, \ldots, t_{i_n})\sigma]$, where C is a context of inverse functions such that $C[t_{o_1}] \Rightarrow {}^*X$.*
2. *Corresponding to every Skolem substitution $\sigma \in \Theta(p^l(t_{i_1}, \ldots, t_{i_n}))$ there is a substitution $\sigma' \in \Theta(q^m(t_{i_1}, \ldots, t_{i_n}))$ such that $X\sigma' \equiv C[p^l(t_{i_1}, \ldots, t_{i_n})\sigma'] \equiv C[p^l(t_{i_1}, \ldots, t_{i_n})\sigma]$ for each $m \in [1, k']$.*
3. *The output of the function ELIMINATE-LOC-VARS for input $\{t\}$ is the set $\{t\sigma \mid \sigma \in \Theta\} = \{t\gamma \mid \gamma \in \Theta(t)\}$.*

PROOF (sketch). In the function ELIMINATE-LOC-VARS, a local variable is replaced by the terms corresponding to its producers. If a local variable (say, Y)

occurs in a nonvariable output term (say, t), the term corresponding to the producer is adorned with a context of inverse functions $C1$ such that $C1[t]$ is reducible to Y by the rewrite rules defining the inverse functions. It can easily be seen that statement (1) of the lemma follows easily. Since the local variable X (with a producer $\langle p^l(t_{i_1},\dots,t_{i_n}),t_{o_l}\rangle$) occurs in the input terms of the atom $q(\cdots)$, the variable X in any term $q^m(t_{i_1},\dots,t_{i_n})$ is replaced with $C[p^l(t_{i_1},\dots,t_{i_n})]$ by the function ELIMINATE-LOC-VARS. Statement 2 of the lemma now follows. Statement 3 of the lemma follows from the above definition of Θ. □

The following example illustrates this lemma:

Example 12. Consider the following clause:

```
moding: d(in, in, out); a(in, out, out); and q, b, c(in, out)
q(X, W) ← a(X, O, U), b(X, U), c(O, V), d(U, V, W)
```

For this clause, $ELV(\mathrm{O}) = \{\mathrm{a}^1(\mathrm{X})\}$, $ELV(\mathrm{U}) = \{\mathrm{a}^2(\mathrm{X}), \mathrm{b}^1(\mathrm{X})\}$, $ELV(\mathrm{V}) = \{\mathrm{c}^1(\mathrm{a}^1(\mathrm{X}))\}$, and $ELV(\mathrm{W}) = \{\mathrm{d}^1(\mathrm{a}^2(\mathrm{X}), \mathrm{c}^1(\mathrm{a}^1(\mathrm{X}))), \mathrm{d}^1(\mathrm{b}^1(\mathrm{X}), \mathrm{c}^1(\mathrm{a}^1(\mathrm{X})))\}$.

The value of Θ is $\{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$, where

$$\sigma_1 = \{\mathrm{O}/\mathrm{a}^1(\mathrm{X}),\ \mathrm{U}/\mathrm{a}^2(\mathrm{X}),\ \mathrm{V}/\mathrm{c}^1(\mathrm{a}^1(\mathrm{X})),\ \mathrm{W}/\mathrm{d}^1(\mathrm{a}^2(\mathrm{X}),\ \mathrm{c}^1(\mathrm{a}^1(\mathrm{X})))\},$$

$$\sigma_2 = \{\mathrm{O}/\mathrm{a}^1(\mathrm{X}),\ \mathrm{U}/\mathrm{b}^1(\mathrm{X}),\ \mathrm{V}/\mathrm{c}^1(\mathrm{a}^1(\mathrm{X})),\ \mathrm{W}/\mathrm{d}^1(\mathrm{a}^2(\mathrm{X}),\ \mathrm{c}^1(\mathrm{a}^1(\mathrm{X})))\},$$

$$\sigma_3 = \{\mathrm{O}/\mathrm{a}^1(\mathrm{X}),\ \mathrm{U}/\mathrm{a}^2(\mathrm{X}),\ \mathrm{V}/\mathrm{c}^1(\mathrm{a}^1(\mathrm{X})),\ \mathrm{W}/\mathrm{d}^1(\mathrm{b}^1(\mathrm{X}),\ \mathrm{c}^1(\mathrm{a}^1(\mathrm{X})))\},$$

$$\sigma_4 = \{\mathrm{O}/\mathrm{a}^1(\mathrm{X}),\ \mathrm{U}/\mathrm{b}^1(\mathrm{X}),\ \mathrm{V}/\mathrm{c}^1(\mathrm{a}^1(\mathrm{X})),\ \mathrm{W}/\mathrm{d}^1(\mathrm{b}^1(\mathrm{X}),\ \mathrm{c}^1(\mathrm{a}^1(\mathrm{X})))\}.$$

The output of ELIMINATE-LOC-VARS for input $\{\mathrm{d}^1(\mathrm{U},\ \mathrm{V})\}$ is the set

$$\{d^1(a^2(X),c^1(a^1(X))),d^1(b^1(X),c^1(a^1(X)))\}$$

which is equal to $\{\mathrm{d}^1(\mathrm{U},\ \mathrm{V})\sigma \mid \sigma \in \Theta\}$. The term $\mathrm{d}^1(\mathrm{U},\ \mathrm{V})$ is an element in *Prod*(W) and substitutions σ_1 and σ_4 satisfy the equation $\mathrm{W}\sigma \equiv \mathrm{d}^1(\mathrm{U},\ \mathrm{V})\sigma$.

The following lemma plays a crucial role in proving our main result. It shows that corresponding to each atom in the body of a clause there are some (sub)terms in the derived rewrite rules.

Lemma 8. Let c : head ← body be a well-moded clause and Θ be the set of Skolem substitutions. Then, for every atom $p(t_{i_1},\dots,t_{i_k}, t_{o_1},\dots,t_{o_{k'}}) \in$ body, for each $\sigma \in \Theta$ and for each j such that $1 \le j \le k' \ne 0$ or $j = k' = 0$, the term $p^j(t_{i_1},\dots,t_{i_k})\sigma$ occurs as a subterm of the right-hand side of a rewrite rule derived from clause c.

PROOF. There are two cases:

($k' = 0$) In this case, the term $p^0(t_{i_1},\dots,t_{i_k})$ is included in the set *Unsry* and the function ELIMINATE-LOC-VARS is called with input *Unsry*. Rewrite rules are constructed with terms from the output of ELIMINATE-LOC-VARS on the right-hand sides. It follows from Lemma 7 that the lemma holds in this case.

($k' > 0$) The lemma is proved in this case by using Noetherian induction with the following Noetherian relation $\prec$. The relation $\prec$ is defined over the terms of the form $p^j(t_{i_1},\dots,t_{i_k})$, where $p(t_{i_1},\dots,t_{i_k},t_{o_1},\dots,t_{o_{k'}})$ is an atom in the body of c and $1 \leq j \leq k'$, as follows:

$$q^l(s_{i_1},\ldots,s_{i_n}) \prec p^j(t_{i_1},\ldots,t_{i_k})$$

if and only if $\langle p^j(t_{i_1},\ldots,t_{i_k}),t_{o_j}\rangle$ is a producer of a variable in $Var(q^l(s_{i_1},\ldots,s_{i_n})) - invar(head)$. Since the producer-consumer relation of a well-moded clause is acyclic, the relation $\prec$ is Noetherian.

Minimal elements: A minimal element (say, $r^k(u_{i_1},\ldots,u_{i_m})$) in this relation is either a producer of a variable in $(outvar(head) - invar(head))$ or is a member of *Unsry*.

- If it is a member of *Unsry*, the lemma holds for this element because the function `ELIMINATE-LOC-VARS` is called with *unsry* as the input and the output of this function is $\{t\sigma \mid t \in Unsry, \sigma \in \Theta\}$. Each term in this set is a subterm of the right-hand term of a rewrite rule.
- If it is not a member of *Unsry*, it is a producer of a variable (say, X) in an output term (say, t) of the head. By Lemma 7, there is a substitution $\sigma \in \Theta$ such that $X\sigma \equiv C[r^k(u_{i_1},\ldots,u_{i_m})\sigma]$, where C is a (possibly empty) context of inverse function symbols such that $C[u_{o_k}] \Rightarrow {}^*X$. The output of the function `ELIMINATE-LOC-VARS` for input $\{t\}$ is $\{t\gamma \mid \gamma \in \Theta\}$. Each term in this set is the right-hand term of a rewrite rule. Since $X \in Var(t)$, $X\sigma \equiv C[r^k(u_{i_1},\ldots,u_{i_m})\sigma]$ is a subterm of $t\sigma$, the lemma holds.

Nonminimal elements: Now we prove that the lemma holds for the element $p^j(t_{i_1},\ldots,t_{i_k})$ if it holds for an element $q^l(s_{i_1},\ldots,s_{i_n})$, such that $q^l(s_{i_1},\ldots,s_{i_n}) \prec p^j(t_{i_1},\ldots,t_{i_k})$. From the definition of the relation $\prec$ it follows that $\langle p^j(t_{i_1},\ldots,t_{i_k}),t_{o_j}\rangle$ is a producer of a variable (say, Y) in $Var(q^l(s_{i_1},\ldots,s_{i_n})) - invar(head)$. By Lemma 7, corresponding to every substitution $\sigma \in \Theta(p^j(t_{i_1},\ldots,t_{i_k}))$ there is a substitution $\sigma' \in \Theta(q^l(s_{i_1},\ldots,s_{i_n}))$, such that

$$Y\sigma' \equiv C\left[p^j(t_{i_1},\ldots,t_{i_k})\sigma'\right] \equiv C\left[p^j(t_{i_1},\ldots,t_{i_k})\sigma\right],$$

for some (possibly empty) context C of inverse function symbols. Hence $p^j(t_{i_1},\ldots,t_{i_k})\sigma$ is a subterm of $q^l(s_{i_1},\ldots,s_{i_n})\sigma'$. By hypothesis, $q^l(s_{i_1},\ldots,s_{i_n})\sigma'$ occurs as a subterm of the right-hand side of a rewrite rule and hence the lemma holds for the element $p^j(t_{i_1},\ldots,t_{i_k})$. □

The following lemma describes the structure of rewrite rules in R_P. It is shown that the function symbols p^i and q^j occur in the right-hand side of rewrite rules in such a way that p^i occurs inside q^j if $q(\cdots)$ is a consumer of a variable (say, X) that occurs in the ith output term of $p(\cdots)$ (i.e., $p(\cdots)$ is a producer of X). In particular, the terms corresponding to the minimal atoms in the body of a clause occur at the innermost level of the right-hand terms of the rewrite rules.

Lemma 9. Let $p(t_{i_1},\ldots,t_{i_k},t_{o_1},\ldots,t_{o_{k'}})$, $q(s_{i_1},\ldots,s_{i_n},s_{o_1},\ldots,s_{o_{n'}})$ be two atoms in the body of a well-moded clause c such that $q(\cdots)$ is a consumer of a variable X and $\langle p^l(t_{i_1},\ldots,t_{i_k}),t_{o_l}\rangle$ is an element in Prod(X). Then, for each $1 \leq j \leq n'$, the term $q^j(s_{i_1},\ldots,s_{i_n})\sigma$ occurs as a subterm of the right-hand side of a rewrite rule derived from c, where σ is a substitution in Θ such that $X\sigma \equiv C[p^l(t_{i_1},\ldots,t_{i_k})\sigma]$ and C is a (possibly empty) context of inverse functions such that $C[t_{o_l}] \Rightarrow {}^*X$.

PROOF. Follows from Lemma 7 (part 2) and Lemma 8. □

The following lemma establishes a correspondence between the computations of logic programs and the derivations of the derived term-rewriting systems.

Lemma 10. Let P be a well-moded program, $Q = \leftarrow q(s_{i_1}, \ldots, s_{i_m}, s_{o_1}, \ldots, s_{o_{m'}})$, *be a well-moded query and* R_P *be the term rewriting system derived from P. If* θ *is a computed answer substitution of* $P \cup \{Q\}$, *then* $q^j(s_{i_1}, \ldots, s_{i_m}) \Rightarrow_{R_P} s_{o_j}\theta$, *for each* $j \in [1, m']$.

PROOF. Induction on the length l of the SLD-refutation of $P \cup \{\leftarrow q(\cdots)\}$. □

5.2. *Termination of a Given Logic Program and the Derived TRS*

Now we prove that the termination of R_P implies termination of the given program P for all well-moded queries under all selection rules implied by the moding information. For this purpose, we introduce the notion of a *rewrite tree* of a well-moded query and establish the relationship between the SLD tree starting with a query and the rewrite tree of that query.

Definition 14 (Rewrite Tree). Let P be a well-moded program, R_P be the term rewrite system derived from P by the transformation, $Q = \leftarrow q_1(\cdots), \ldots, q_n(\cdots)$ be a well-moded query, and R_Q be the set of rewrite rules derived from the clause $q_0 \leftarrow q_1(\cdots), \ldots, q_n(\cdots)$, where q_0 is a fresh predicate of arity "0" not occurring in $P \cup \{Q\}$. The *rewrite tree* RT_{PQ} of P and Q is defined as follows:

1. $\text{Root}(RT_{PQ}) = q_0^0$
2. Children of a node $t \in RT_{PQ}$ are $\{s \mid t \Rightarrow_{R_Q \cup R_P} s\}$.

The rewrite tree RT_{PQ} essentially contains all of the (rewriting) derivations of the rewrite system $R_Q \cup R_P$, starting from the initial term q_0^0. The following theorems establish the relationship between RT_{PQ} and the SLD-derivations of $P \cup \{Q\}$. Before giving the formal theorems, we illustrate the relationship between RT_{PQ} and the SLD-derivations through an example.

Example 13. Consider the program given in Example 8 and query $Q = \leftarrow$ `c(t, Y)`, where t is a ground term. Here we give two possible SLD-derivations and show their correspondence with RT_{PQ}:

```
←c(t,Y)                              ←c(t,Y)
←a(t,Z), r(t,Z,Y,Z1,Z2)              ←a(t,Z), r(t,Z,Y,Z1,Z2)
←r(t,f(t),Y,Z1,Z2)                   ←r(t,f(t),Y,Z1,Z2)
←b(t,Y), c(f(t),Z1),                 ←b(t,Y), c(f(t),Z1),
  d(t,f(t),Z2)                         d(t,f(t),Z2)
←c(f(t),Z1), d(t,f(t),Z2)            ←b(t,Y), c(f(t),Z1)
          ⋮                                    ⋮
```

Figure 2 shows a (top) portion of the rewrite tree RT_{PQ}. The dots "$\cdots$" denote a subtree. Observe that corresponding to every selected atom in the SLD-derivations there is a term in RT_{PQ}.

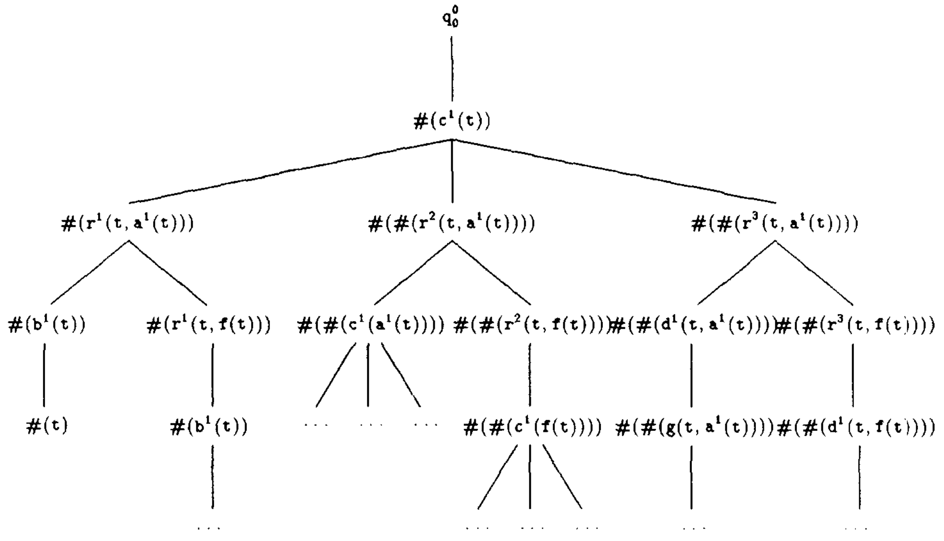

FIGURE 2.

In the following, we show that corresponding to every resolution step in SLD-derivations of $P \cup \{Q\}$, there is at least one rewrite step in RT_{PQ}. For establishing this property, we need the following technical lemma.

Lemma 11. Let $G_0, G_1, \ldots, G_n$ be an SLD-derivation of a well-moded program P and a well-moded query Q under a selection rule implied by the moding information of P and Q. Furthermore assume that $H \leftarrow B_1, \ldots, B_m$ is the input clause and θ_{n_1} is the mgu used in deriving G_{n_1+1} from G_{n_1}, and $B_i\theta$ is the selected atom in $G_{n_1+n_2}$, where θ is the composition of mgus used in the derivation $G_{n_1}, \ldots, G_{n_1+n_2}$. Then, $p^j(s_{i_1}, \ldots, s_{i_k})\sigma\theta_{n_1} \Rightarrow^*_{R_P} p^j(s_{i_1}, \ldots, s_{i_k})\theta$, if $B_i \equiv p(s_{i_1}, \ldots, s_{i_k}, s_{o_1}, \ldots, s_{o_{k'}})$ for each $\sigma \in \Theta(p^j(s_{i_1}, \ldots, s_{i_k}))$.

PROOF. Induction on n_2.
Basis: $n_2 = 1$.
In this case, $\theta = \theta_{n_1}$ and B_i is a minimal element in the producer–consumer relation of the clause $H \leftarrow B_1, \ldots, B_m$. By definition, $domain(\sigma) \cap invar(H) = \phi$. Since B_i is minimal, $Var(p^j(s_{i_1}, \ldots, s_{i_k})) = invar(B_i) \subseteq invar(H)$, and hence $p^j(s_{i_1}, \ldots, s_{i_k})\sigma \equiv p^j(s_{i_1}, \ldots, s_{i_k})$. Since $B_i\theta_{n_1}$ is the selected atom in G_{n_1+1}, its input terms $s_{i_1}\theta_{n_1}, \ldots, s_{i_k}\theta_{n_1}$ are ground. Hence $p^j(s_{i_1}, \ldots, s_{i_k})\sigma\theta_{n_1} \equiv p^j(s_{i_1}, \ldots, s_{i_k})\theta$. The lemma holds.
Induction Hypothesis: Assume that the lemma holds for all $n_2 < l$.
Induction Step: $n_2 = l$.
Let $r^{j1}(t_{i_1}, \ldots, t_{i_{k_r}})\sigma\theta_{n_1}$ be a maximal proper subterm (having a Skolem function at the root) of $p^j(s_{i_1}, \ldots, s_{i_k})\sigma\theta_{n_1}$. Let $G_{n_1+n_r}$ be the goal in which $r(t_{i_1}, \ldots, t_{i_{k_r}}, t_{o_1}, \ldots, t_{o_{k'_r}})\theta'$ is selected and θ' be the composition of mgus used in the derivation $G_{n_1}, \ldots, G_{n_1+n_r}$. It is clear from Lemma 9 that $r(\cdots)$ is a producer of a variable (say, X) in $invar(p(\cdots))$ and $r(\cdots)$ should be selected before $p(\cdots)$ is selected, i.e., $n_r < n_2$. By induction hypothesis,

$$r^{j1}\left(t_{i_1}, \ldots, t_{i_{k_r}}\right)\sigma\theta_{n_1} \Rightarrow^*_{R_P} r^{j1}\left(t_{i_1}, \ldots, t_{i_{k_r}}\right)\theta'.$$

By Lemma 10, $r^{j1}(t_{i_1},\dots,t_{i_{k_r}})\theta' \Rightarrow^*_{R_P} t_{o_{j1}}\theta'' \equiv t_{o_{j1}}\theta$, where θ'' is the ground computed answer substitution of $r(\cdots)\theta'$. Furthermore, it is easy to see that $C[t_{o_{j1}}\theta''] \Rightarrow^*_{R_P} X\theta''$ if C is the context of inverse function symbols around the subterm $r^{j1}(t_{i_1},\dots,t_{i_{k_r}})\sigma\theta_{n_1}$ in $p^j(s_{i_1},\dots,s_{i_k})\sigma\theta_{n_1}$.

It is easy to see that $p^j(s_{i_1},\dots,s_{i_k})\sigma\theta_{n_1}$ can be reduced to $p^j(s_{i_1},\dots,s_{i_k})\theta$ by reducing each maximal proper subterm $u^{j2}(s'_{i_1},\dots,s'_{i_{k_u}})$ (having a Skolem function at the root) of $p^j(s_{i_1},\dots,s_{i_k})\sigma\theta_{n_1}$ to $s'_{o_{j2}}\theta$ and then reducing the contexts of inverse functions. □

The above lemma establishes a correspondence between SLD-derivations and the derivations in the rewrite tree. Figure 3 depicts such a correspondence.

In the following, we prove that corresponding to every selected atom in an SLD-derivation, there is at least one term in RT_{PQ}. This helps us to show that corresponding to every resolution step in the SLD-derivations of $P \cup \{Q\}$, there is at least one rewrite step in RT_{PQ}.

Theorem 3. If P is a well-moded program, Q is a well-moded query and S is a selection rule implied by $P \cup \{Q\}$, then for each goal G_i in any SLD-derivation $G_1,\dots,G_n$ of $P \cup \{Q\}$ (under selection rule S), the following holds: for each minimal (under the evaluation order of G_i) atom $p(t_{i_1},\dots,t_{i_k},t_{o_1},\dots,t_{o_{k'}})$ in G_i and for each j such that $1 \le j \le k' \ne 0$ or $j = k' = 0$, the term $p^j(t_{i_1},\dots,t_{i_k})$ occurs as a subterm of a node in the rewrite tree RT_{PQ} of Q.

PROOF. Let *query* be a fresh predicate symbol of arity 0 and P' be the well-moded program $P \cup \{query \leftarrow Q\}$. It is easy to see that $G_0, G_1,\dots,G_n$ is a SLD-derivation of $P' \cup \{\leftarrow query\}$ if $G_1,\dots,G_n$ is a SLD-derivation of $P \cup \{Q\}$ and $G_0 = \leftarrow query$. Now we prove the theorem using induction on i.

Basis: $i = 1$. The goal G_1 is Q itself. From the validity of Lemma 8 over the rewrite

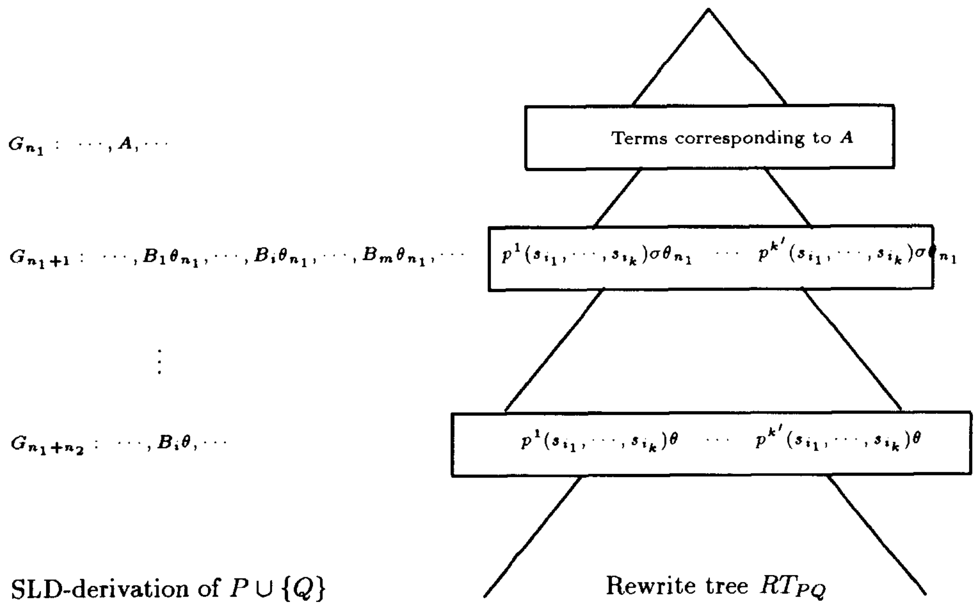
G_{n_1} : $\cdots, A, \cdots$
G_{n_1+1} : $\cdots, B_1\theta_{n_1}, \cdots, B_i\theta_{n_1}, \cdots, B_m\theta_{n_1}, \cdots$
$G_{n_1+n_2}$: $\cdots, B_i\theta, \cdots$
Terms corresponding to A
$p^1(s_{i_1},\cdots,s_{i_k})\sigma\theta_{n_1}$ $\cdots$ $p^{k'}(s_{i_1},\cdots,s_{i_k})\sigma\theta_{n_1}$
$p^1(s_{i_1},\cdots,s_{i_k})\theta$ $\cdots$ $p^{k'}(s_{i_1},\cdots,s_{i_k})\theta$
SLD-derivation of $P \cup \{Q\}$
Rewrite tree RT_{PQ}

FIGURE 3.

rules in R_Q, it follows that a term $p^j(t_{i_1},\ldots,t_{i_k})\sigma$ occurs as a subterm of the right-side of rewrite rule in R_Q and hence occurs as a subterm of a node in RT_{PQ}. Since $p(\cdots)$ is a minimal element in the well-moded query, it does not have any variable in input terms, and hence $p^j(t_{i_1},\ldots,t_{i_k}) \equiv p^j(t_{i_1},\ldots,t_{i_k})\sigma$ for every substitution σ.
Induction hypothesis: Let us assume that the theorem holds for all $i < d$.
Induction step: Now we prove that the theorem holds for $i = d$.
Let (i) $\leftarrow q_1(\cdots),\ldots,q_{n'}(\cdots)$ be the goal G_{i-1}, (ii) $q_l(\cdots)$ be the selected atom, (iii) $H \leftarrow B_1,\ldots,B_m$ be the input clause, and (iv) θ_i be the mgu used in deriving G_i from G_{i-1}. Then

$$G_i \text{ is } \leftarrow q_1(\cdots)\theta_i,\ldots,q_{l-1}(\cdots)\theta_i, B_1\theta_i,\ldots,B_m\theta_i, q_{l+1}(\cdots)\theta_i,\ldots,q_{n'}(\cdots)\theta_i.$$

Now we have two cases: (a) $m \neq 0$, that is, the input clause is not a unit clause, and (b) $m = 0$, that is, the input clause is a unit clause.

Case (a). By the assumption on selection rules, minimal elements, $min(G_i) = \{q_j(\cdots)\theta_i \mid q_j(\cdots) \in min(G_{i-1}) \text{ and } j \neq l\} \cup min(B_1\theta_i,\ldots,B_m\theta_i\}$. For those minimal elements that are already in G_{i-1}, the theorem holds by the induction hypothesis. For minimal elements in $B_1\theta_i,\ldots,B_m\theta_i$, the theorem holds because of Lemma 8 (argument similar to that in the base case).

Case (b). The set of minimal elements, $min(G_i)$, contains $\{q_j(\cdots)\theta_i \mid q_j(\cdots) \in min(G_{i-1}) \text{ and } j \neq l\}$ and atoms $q_{l'}(\cdots)\theta_i, l' \neq l$, such that $q_{l'}(\cdots)$ is only greater than $q_l(\cdots)$ in the evaluation order of G_{i-1}. For minimal elements that are already in $min(G_{i-1})$, the theorem holds by the induction hypothesis.

Now assume that $p(t_{i_1},\ldots,t_{i_k},t_{o_1},\ldots,t_{o_{k'}})$ is an element in $min(G_i)$ but not in $min(G_{i-1})$. Since $p(\cdots)$ is an atom in the SLD-derivation, there must be an atom $A \equiv p(s_{i_1},\ldots,s_{i_k},s_{o_1},\ldots,s_{o_{k'}})$ in the body of c' such that $p(\cdots) \equiv A\theta$, where c' is the input clause used in deriving the goal $G_{i'+1}$ from $G_{i'}$ for some $i' < i$ and θ is the composition of mgus used in $G_{i'},\ldots,G_i$. Let $r(u_{i_1},\ldots,u_{i_m},u_{o_1},\ldots,u_{i_{m'}})$ be the selected atom in $G_{i'}$. By the induction hypothesis, $r^j(u_{i_1},\ldots,u_{i_m})$, $1 \leq j \leq m'$, occurs as a subterm of a node in RT_{PQ}. Now, by Lemma 8 and the construction of RT_{PQ}, $p^j(s_{i_1},\ldots,s_{i_k})\sigma\theta_{i'}$ occurs as a subterm of a node in RT_{PQ} for each $\sigma \in \Theta(p^j(s_{i_1},\ldots,s_{i_k}))$ and $1 \leq j \leq k'$, where $\theta_{i'}$ is the mgu used in deriving $G_{i'+1}$ from $G_{i'}$. It follows from Lemma 11 that $p^j(s_{i_1},\ldots,s_{i_k})\sigma\theta_{i'}$ reduces to $p^j(t_{i_1},\ldots,t_{i_k})$ by R_P, that is, $p^j(t_{i_1},\ldots,t_{i_k})$ is a subterm of a node in RT_{PQ}. □

The following theorem establishes that corresponding to every resolution step in SLD-derivations of $P \cup \{Q\}$, there are some rewriting steps in RT_{PQ}.

Theorem 4. If P is a well-moded program, Q is a well-moded query and S is a selection rule implied by $P \cup \{Q\}$, then corresponding to every resolution step in every SLD-derivation of $P \cup \{Q\}$ (under selection rule S), there are reduction steps in the rewrite-tree RT_{PQ} of Q.

PROOF. Let us consider a resolution step in which $p(t_{i_1},\ldots,t_{i_k},t_{o_1},\ldots,t_{o_l})$, $l \neq 0$ is resolved using an input clause c (case $l = 0$ can be handled similarly). By Theorem 3, corresponding to this atom there are terms $p^j(t_{i_1},\ldots,t_{i_k})$, $1 \leq j \leq l$, occurring as subterms of nodes in the rewrite tree of Q. By Theorem 2, input terms $t_{i_1},\ldots,t_{i_k}$ of the selected atom $p(\cdots)$ are ground.

Let θ be the mgu and $p(s_{i_1},\ldots,s_{i_k},s_{o_1},\ldots,s_{o_l})$ be the head of the input clause used in the resolution step. Corresponding to this clause, we have rewrite rules $p^j(s_{i_1},\ldots,s_{i_k}) \to r_j$. By definition of the SLD-resolution, $p(s_{i_1},\ldots,s_{i_k},s_{o_1},\ldots,s_{o_l})\theta \equiv p(t_{i_1},\ldots,t_{i_k},t_{o_1},\ldots,t_{o_l})\theta$. Therefore,

$$p^j(s_{i_1},\ldots,s_{i_k})\theta \equiv p^j(t_{i_1},\ldots,t_{i_k})\theta \equiv p^j(t_{i_1},\ldots,t_{i_k}), \qquad 1 \le j \le l$$

(since $t_{i_1},\ldots,t_{i_k}$ are ground). Hence terms $p^j(t_{i_1},\ldots,t_{i_k})$, $1 \le j \le l$, in RT_{PQ} match with the left-hand sides of rewrite rules derived from clause c and can be reduced. □

The following theorem establishes the relationship between the termination of a given well-moded logic program and that of the derived term-rewriting system. Informally, the theorem says that a well-moded logic program terminates for all well-moded queries under any selection rule implied by the moding information, if the derived term-rewriting system terminates.

Theorem 5. If P is a well-moded program, Q is a well-moded query, S is a selection rule implied by $P \cup \{Q\}$ and R_P is the term rewriting system derived from P, then all the SLD-derivations (under S) of $P \cup \{Q\}$ are of finite length if R_P is a terminating system.

PROOF. Follows from Theorem 4 and König's lemma. □

Before illustrating the application of this theorem in proving termination of logic programs, we briefly review important termination techniques of rewrite systems.

5.3. *Termination of Term-Rewriting Systems*

Although termination of term-rewriting systems in general is undecidable, it has been proved that one can simulate a Turing machine by using a single rewrite rule [11]; several techniques and tools have been proposed in the literature for proving termination of term-rewriting systems. One standard technique is to look for a well-founded ordering $\succ$ over $\mathcal{T}$ satisfying a condition: if $s \Rightarrow {}^*t$ then $s \succ t$. If the well-founded ordering $\succ$ has the properties of *monotonicity* ($t \succ u$ implies $C[t] \succ C[u]$) and stability ($t \succ u$ implies $t\sigma \succ u\sigma$), it is enough to check that $l \succ r$ for each rewrite rule $l \to r$ in the system. Dershowitz [18] has discussed several such well-founded orderings and techniques for proving termination of term-rewriting systems. Here we explain two termination techniques, namely, recursive path ordering with status and interpretation-based techniques.

Definition 15. A quasi-ordered set $(\mathcal{S}, \succsim)$ consists of a set $\mathcal{S}$ and a transitive-reflexive binary relation $\succsim$ defined over $\mathcal{S}$. We define the associated equivalence relation $\approx$ as $s \approx t$ if and only if $s \succsim t$ and $t \succsim s$, and the associated strict partial ordering $\succ$ as $s \succ t$ if and only if $s \succsim t$ but not $t \succsim s$.

We allow an operator to have one of the following three statuses: multiset, lexicographic left-to-right (LR), and lexicographic right-to-left (RL). Multiset status is taken as default status of an operator if its status is not specified.

Definition 16 (Recursive Path Ordering with Status [17, 22]). Let $\succsim$ be a quasi-ordering (called *precedence*) over a finite set $\mathcal{F}$ of function symbols. The *recursive path ordering* $\succ_{\text{rpos}}$ on the set $\mathcal{T}(\mathcal{F},\mathcal{X})$ of terms induced by the precedence $\succsim$ is defined recursively as follows:

$s = f(s_1,\dots,s_m) \succsim_{\text{rpos}} g(t_1,\dots,t_n) = t$ if and only if one of the following is true:

(i) $s_i \succsim_{\text{rpos}} t$ for some $i = 1,\dots,m$ or
(ii) $f \succ g$ and $s \succ_{\text{rpos}} t_j$ for all $j = 1,\dots,n$ or
(iii) $f \approx g$, f, g have *multiset* status and $\{s_1,\dots,s_m\} \succsim_{\text{rpos}} \{t_1,\dots,t_n\}$, or
(iv) $f \approx g$, $s \succ_{\text{rpos}} t_j$ for all $j = 1,\dots,n$ and
  (a) f and g have LR status and $(s_1,\dots,s_m) \succsim^*_{\text{rpos}} (t_1,\dots,t_n)$ or
  (b) f and g have RL status and $(s_m, s_{m-1},\dots,s_1) \succsim^*_{\text{rpos}} (t_n, t_{n-1},\dots,t_1)$,

where $\succsim_{\text{rpos}}$ is the multiset ordering induced by $\succsim_{\text{rpos}}$, and $\succsim^*_{\text{rpos}}$ is the lexicographic ordering induced by $\succsim_{\text{rpos}}$.

Theorem 6. For any well-founded quasi-ordering $\succsim$ and status on a given finite set of function symbols, the recursive path ordering with status $\succ_{\text{rpos}}$ is a monotonic and stable well-founded ordering [17, 22].

The above theorem makes the recursive path ordering with status a powerful tool in proving termination of rewrite systems. This ordering has been implemented in theorem provers like RRL, REVE, etc.

Example 14. This example illustrates the use of recursive path ordering with status in proving termination of term-rewriting systems. To prove termination of the following term-rewriting system (derived from the multiplication program), take the precedence as `mult`$^1 \succ$ `add`$^1 \succ$ `s`:

```
add¹(0, Y) → Y
add¹(s(X), Y) → s(add¹(X, Y))
mult¹(0, Y) → 0
mult¹(s(X), Y) → add¹(mult¹(X, Y), Y)
```

Proving that $l \succ_{\text{rpos}} r$ for rules 1 and 3 is easy. Consider rule 2. Since `add`$^1 \succ$ `s`, to prove that `add`^{1}`(s(X), Y)` $\succ_{\text{rpos}}$ `s(add`^{1}`(X, Y))`, it is enough to prove that `add`^{1}`(s(X), Y)` $\succ_{\text{rpos}}$ `add`^{1}`(X, Y)`. And to prove `add`^{1}`(s(X), Y)` $\succ_{\text{rpos}}$ `add`^{1}`(X, Y)`, we need to prove that {`s(X), Y`} $\succ_{\text{rpos}}$ {`X, Y`}, which is indeed the case, because `s(X)` $\succ_{\text{rpos}}$ `X`.

Now consider rule 4. Since `mult`$^1 \succ$ `add`^{1}, to prove that `mult`^{1}`(s(X), Y)` $\succ_{\text{rpos}}$ `add`^{1}`(mult`^{1}`(X, Y), Y)`, it is enough to prove that `mult`^{1}`(s(X), Y)` $\succ_{\text{rpos}}$ `mult`^{1}`(X, Y)` and `mult`^{1}`(s(X), Y)` $\succ_{\text{rpos}}$ `Y`. To prove `mult`^{1}`(s(X), Y)` $\succ_{\text{rpos}}$ `mult`^{1}`(X, Y)`, we need to prove that {`s(X), Y`} $\succ_{\text{rpos}}$ {`X, Y`}, which is indeed the case. Furthermore, `mult`^{1}`(s(X), Y)` $\succ_{\text{rpos}}$ `Y` by the subterm property of recursive path ordering. This completes the termination proof.

Example 15. This example illustrates the use of status information. To prove termination of the term-rewriting system with just one rule $(X+Y)+Z \rightarrow X+(Y+Z)$, we use recursive path ordering with status $\succ_{\text{rpos}}$ as follows.

Since the two terms have the same function symbol at the top level, to prove that $(X+Y)+Z \succ_{\text{rpos}} X+(Y+Z)$, it is enough to prove (i) $(X+Y, Z) \succ_{\text{rpos}}$

$*(X, Y+Z)$, (ii) $(X+Y)+Z \succ_{\text{rpos}} X$, and (iii) $(X+Y)+Z \succ_{\text{rpos}} (Y+Z)$. By the subterm property, $X+Y \succ_{\text{rpos}} X$, and hence (i) follows. Again by the subterm property (ii) follows.

To prove (iii), it suffices to prove (a) $(X+Y, Z) \succ_{\text{rpos}} *(Y, Z)$, (b) $(X+Y)+Z \succ_{\text{rpos}} Y$, and (c) $(X+Y)+Z \succ_{\text{rpos}} Z$. The conditions (b) and (c) follow directly from the subterm property, and (a) follows from the fact that $X+Y \succ_{\text{rpos}} Y$.

Interpretation-based termination proofs try to map the set of terms onto a well-founded, partially ordered set and prove that left-hand sides are mapped onto bigger elements than the corresponding right-hand sides. This is formally described below.

Definition 17. A terminating function τ from a set of terms $\mathcal{T}(\mathcal{F},\mathcal{X})$ to a well-founded, partially ordered set $(\mathcal{W}, \succ)$ is composed of a set of functions $\{f_\tau : \mathcal{W}^n \to \mathcal{W} | f$ is a function symbol of arity n in $\mathcal{F}\}$ such that $\tau(f(t_1,\ldots,t_n)) = f_\tau(\tau(t_1),\ldots,\tau(t_n))$ for every term $f(t_1,\ldots,t_n) \in \mathcal{T}(\mathcal{F},\mathcal{X})$ and $w \succ w'$ implies $f_\tau(\cdots \tau(w) \cdots) \succ f_\tau(\cdots \tau(w') \cdots)$ for all $w, w' \in \mathcal{W}$ and $f \in \mathcal{F}$.

Basically, terminating function consists of a set of monotonic mappings. Polynomial and exponential interpretations are special instances of terminating functions. A terminating function τ from terms to natural numbers, where each f_τ is a polynomial, is called a polynomial interpretation. Polynomial interpretations are implemented in REVE [27]. A terminating function τ where each f_τ is a polynomial or an exponential is called an exponential interpretation. Exponential interpretations are implemented in the ORME [28].

Example 16. To prove termination of the following system over a set of terms constructed from constants 0 and 1 and the function symbols $+$ and $\times$,

$$X \times (Y+Z) \to (X \times Y) + (X \times Z)$$
$$(Y+Z) \times X \to (Y \times X) + (Z \times X)$$
$$(X+Y) + Z \to X + (Y+Z),$$

we can use the following polynomial interpretation:

$$\tau(0) = 2 \qquad \tau(s \times t) = \tau(s).\tau(t)$$
$$\tau(1) = 2 \qquad \tau(s+t) = 2\tau(s) + \tau(t) + 1.$$

5.4. *Termination Proofs for Logic Programs*

Now we illustrate the application of our main theorem in proving termination of logic programs through examples.

Example 17. In Example 14, using recursive path ordering, we proved termination of the rewrite system derived from the multiplication program. From Theorem 5, it follows that the `multiplication` program terminates for all well-moded queries.

Example 18. Termination of the quick-sort program for well-moded queries can be established by using the above theorem by proving termination of the derived term-rewriting system given in Example 9. Termination of this term-rewriting system can be proved by using the following elementary interpretation in the ORME system [28]:

$$
\begin{array}{ll}
[[\mathtt{a}^1]]((x,y),(u,v)) = (x+u, 2y+v) & \\
[[\mathtt{s}^1]]((x,y),(u,v)) = (x+u, 2y+v) & [[\mathtt{q}^1]]((x,y)) = (2^x, y) \\
[[\mathtt{s}^2]]((x,y),(u,v)) = (x+u, 2y+v) & [[\mathtt{nil}]] = (0,0) \\
[[\mathtt{c}]]((x,y),(u,v)) = (x+u+2, y+v). &
\end{array}
$$

Here, for $f \in \{\mathtt{a}^1,\ \mathtt{s}^1,\ \mathtt{s}^2,\ \mathtt{q}^1,\ \mathtt{c},\ \mathtt{nil}\}$, the function $[[f]]$ denotes the function f_τ given by the elementary interpretation τ.

5.5. *Converse of the Main Theorem*

That the converse of Theorem 5 does not hold is demonstrated by the following example.

Example 19. Let us consider the following well-moded logic program:

```
moding: p(in, out) and tc (in, out)
p(a, b) ←
p(b, c) ←
tc(X, Y) ←p(X, Z), tc(Z, Y)
```

It is easy to see that this program terminates for all well-moded queries. But the term-rewriting system:

$$
\begin{array}{l}
\mathtt{p}^1(\mathtt{a}) \rightarrow \mathtt{b} \\
\mathtt{p}^1(\mathtt{b}) \rightarrow \mathtt{c} \\
\mathtt{tc}^1(\mathtt{X}) \rightarrow \mathtt{tc}^1(\mathtt{p}^1(\mathtt{X}))
\end{array}
$$

derived from the above program has an infinite derivation:

$$
\begin{aligned}
&\mathtt{tc}^1(\mathtt{a}) \Rightarrow \mathtt{tc}^1(\mathtt{p}^1(\mathtt{a})) \Rightarrow \mathtt{tc}^1(\mathtt{p}^1(\mathtt{p}^1(\mathtt{a}))) \\
&\qquad \Rightarrow \cdots \Rightarrow \mathtt{tc}^1\big(\mathtt{p}^1\big(\cdots \mathtt{p}^1(\mathtt{a}) \cdots\big) \Rightarrow \cdots
\end{aligned}
$$

In other words, termination of the derived term-rewriting system is not a necessary condition for termination of the logic program. Since our aim is to prove termination of logic programs using termination techniques of term rewriting systems, the sufficient conditions for termination of logic programs are more important than the necessary conditions, and the above theorem provides a sufficient condition for termination of logic programs.

It is an interesting to find classes of programs for which the converse of Theorem 5 holds as well. In the following, we identify two such classes of programs.

Definition 18. A logic program P is a *non-variable-including program* (*nvi* program) if all of the clauses satisfy the property: *all of the variables in the body occur in the head as well.*

The following theorem establishes the converse of Theorem 5 for the class of *nvi* programs.

Theorem 7. Let P, S, Q, and R_P be as in Theorem 5 such that (i) P is an nvi-program, (ii) each variable in P has precisely one producer and (iii) the output terms in Q and the bodies of clauses in P are distinct variables. Then, all the SLD-derivations (under S) of $P \cup \{Q\}$ are of finite length if and only if R_P is terminating.

PROOF. The if-part follows from Theorem 5.
Only-if part: A nice property of SLD-derivations of $P \cup \{Q\}$ is that the output terms of each goal are distinct variables. Therefore, a selected atom unifies with the head of any clause whose input terms can match with the input terms of the selected atom. A nice property of R_P is that the Skolem functions are not nested on the right-hand sides of the rewrite rules. Therefore, nesting of Skolem functions is forbidden in all of the terms of any rewrite derivation of $R_P \cup \mathscr{R}_Q$ starting from Q_0, and it is easy to exhibit an infinite SLD-derivation of $P \cup \{Q\}$ corresponding to an infinite rewrite derivation of $R_P \cup \mathscr{R}_Q$ starting from Q_0. □

Another class of programs for which the converse of Theorem 5 holds is the following.

Definition 19. A logic program P is a *binary program* if no clause in it has more than one atom in the body.

The following theorem establishes the converse of Theorem 5 for the class of binary programs.

Theorem 8. Let P, S, Q and R_P be as in Theorem 5 such that (i) P is a binary program having just variables in the output positions in the body of each clause, (ii) each variable in P has precisely one producer, and (iii) Q has only one atom and has distinct variables in the output positions. Then, all of the SLD-derivations (under S) of $P \cup \{Q\}$ are of finite length if and only if R_P is terminating.

PROOF. Same as that of the above theorem. □
In fact, there is no sideways information passing in the execution of the above class of programs, and every selection rule is implied by the moding information. Therefore, termination of R_P implies termination of P for Q under all selection rules (i.e., strong termination).

6. A TOOL FOR PROVING TERMINATION

In the previous sections, we have reduced the termination problem of logic programs to that of term-rewriting systems. The transformation procedure has been implemented as a front end to Rewrite Rule Laboratory (RRL); RRL is a theorem prover based on techniques developed by Kapur, Zhang, and Sivakumar [23] to obtain a semi-automatic interactive system for proving termination of term-rewriting systems. RRL supports techniques such as recursive path ordering for proving termination of term-rewriting systems in an interactive fashion. The block diagram in Figure 4 gives the module structure of the system.

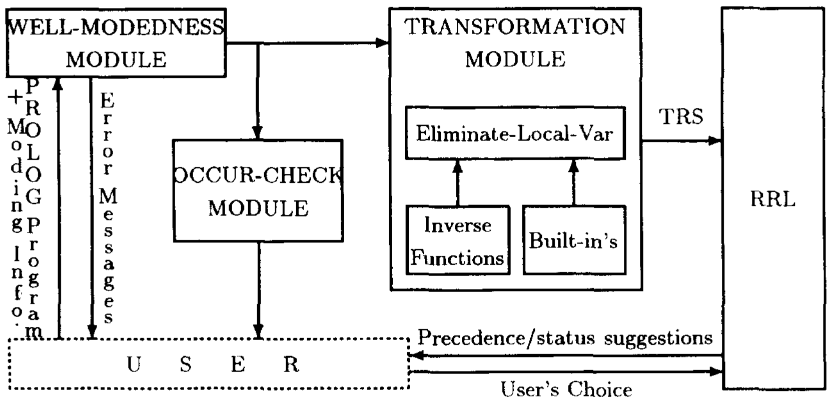

FIGURE 4. Block diagram of the modules in the system.

6.1. Well-Modedness Module

This module checks well-modedness of the given program, applying the following steps on each clause:

1.1 A producer–consumer graph with atoms in the body as nodes is constructed.
1.2 The producer–consumer graph is checked for acyclicity (using depth-first-search).
1.3 If it is not acyclic, a warning is given saying that a particular clause is not well-moded.
2.1 The set of producers of each variable in the clause is computed.
2.2 It is checked whether every variable has at least one producer.
2.3 If any variable has no producer, a warning is given saying that a particular variable has no producer.

6.2. Occur-Check Module

This module takes well-moded Prolog programs and checks whether they can be executed soundly on Prolog interpreters without an occur-check test. This module essentially checks whether any variable occurs more than once in output terms of head of any clause. If no variable occurs more than once in output terms of heads, the program is declared to be not subjected to occur-check (NSTO). If there is a variable occurring more than once in output terms of some head, this module notifies the user of the offending variable and the clause.

6.3. Transformation Module

This module implements the transformation procedure described in the previous section. For each clause, it computes the sets *Unsry* and *Consvar* (as defined in the formal transformation given earlier) and the set of producers of each local variable.[3] The major submodule "eliminate-local-variable" implements the proce-

[3] Actually, a "well-modedness" module checks every clause for well-modedness and passes the set of producers of each variable to the "transformation" module. So, there is no repetition of work.

dure with the same name described earlier. In addition, there are two more modules, one for generating inverse functions, and the other for *built-ins*.

6.4. Built-In Predicates

Prolog has built-in predicates $=$, $<$, $<=$, $>$, $>=$, $\backslash=$ with moding `(in, in)`, and `is` with moding `(out, in)`. According to the definition of *Unsry* in the transformation algorithm, for each built-in atom $b(t1, t2)$, $b \in \{=, <, <=, >, >=, \backslash=\}$ in a clause, $b^0(t1, t2)$ is included in *Unsry*, and rewrite rules of the form $lhs \to \#(b^0(t1, t2))$ will be present in R_p. Since we know that the above built-in atoms always terminate, our implementation does not make rules of the above form (this is achieved by removing the above terms from *Unsry* before making the # rules). An `is`-atom is a producer of the variables occurring in its first argument, and the Skolem function is^1 occurs on the right-hand side of the derived rewrite rules. Because of the semantics of `is`, the right-hand sides of the rules are reduced by the rule $is^1(X) \to X$ before outputting the rules in R_p.

Rao, et al. [25] explored the application of our system as a verification tool in the development of provably correct compilers. In particular, they have established the termination of a prototype compiler for **Pro-CoS** level 0 language PL_0 using our tool. The compiler has been developed by using Hoare's refinement algebra approach. The fact that termination of this compiler cannot be shown by using other approaches to termination of logic programs (e.g., approaches of Ullman and van Gelder [35], Plümer [32], De Schreye and Verschaetse [12]) demonstrates the practicality of the transformational approach.

Consider, for instance, the following clause (from the compiler) with moding: `c(in, in, out, out, in, in)`; `ce (in, in, out, out, in, in)`; `mtrans (in, in, out, out)`; `psi (in, in, out)` and `flatten (in, out)`. It is not very difficult to check that this clause does not have an admissible solution graph. Therefore, the approaches of Ullman and van Gelder [35] and Plümer [32] cannot be used in proving termination of the **ProCoS** compiler.

```
c(Output!E, S, F, M, Psi, Omega) ←
     ce(E, S, L1, M1, Psi, Omega),
     psi(Psi, outputbuf, Psioutputbuf),
     psi(Psi, Output, PsiOutput),
     mtrans(stl(Psioutputbuf), L1, L2, M2),
     mtrans(ldlp(Psioutputbuf), L2, L3, M3),
     mtrans(ldc(PsiOutput), L3, L4, M4),
     mtrans(ldc(4), L4, L5, M5),
     mtrans(out, L5, F, M6),
     flatten([M1, M2, M3, M4, M5, M6], M), !.
```

The following clause (and two more similar clauses) is problematic[4] for the tool

[4] In this case, the problem appears to be due to the fact that in the method it is not possible to compare the two function symbols $\langle\ \rangle$ and $eq1$. However, in this case, it is possible to handle the clause using "unfolding" techniques.

built by De Schreye and Verschaetse [12]:

```
ce(E1〈 〉E2, S, F, M, Psi, cons(Loc, Omega))←
     ce(E1 eql E2, S, L1, M1, Psi, cons(Loc, Omega)),
     mtrans(eqc(0), L1, F, M2),
     append(M1, M2, M),!.
```

7. RELATED WORKS

In this section, we briefly survey the related works on termination analysis of logic programs. For more details on these works, the reader is referred to a recent survey on termination of logic programs by De Schreye and Decorte [14].

The results on termination of logic programs can be broadly classified into two categories:

1. **Characterizations of terminating programs.** Papers in this category usually present a set of *necessary and sufficient* conditions for the termination of logic programs. Since logic programs have the expressive power of Turing machine, the termination is undecidable and hence these necessary and sufficient conditions are undecidable, too. Therefore, the approaches presented in these works may not be easily mechanizable.
2. **Automatic / semi-automatic procedures** for proving termination of logic programs. Papers in this category usually present a set of *sufficient* conditions (which can be easily tested) for the termination of logic programs. Again, because of the undecidability of termination of logic programs, none of these sufficient conditions can certify each and every terminating program—for every such sufficient condition there exists a terminating program that does not satisfy it.

In the following, we briefly discuss the works in both of these categories.

7.1. Characterizations of Terminating Programs

Vasak and Potter [36] were the first ones (to our knowledge) to attempt to give a mathematical treatment to the termination problem of logic programs. They introduced various notions of termination, such as strong termination, weak termination, universal termination, and existential termination. However, there is no attempt toward automatic verification of termination.

Baudinet [6] took a semantic approach to termination of logic programs. She associated with each program a system of equations whose least fixpoint is the meaning of the program. By analyzing this fixpoint, various termination properties of the program can be proved. Structural induction is used in her termination proofs. The approach is very general and can be used for studying termination of normal logic programs (i.e., with negation), existential termination (which cannot be studied by most of the other approaches), and the effect of cuts (!) in the programs. Baudinet [6] suggested the use of theorem provers for proving all of these properties.

Francez et al. [20] took an assertion-based approach to termination of logic programs. Proofs are very similar to the termination proofs of imperative logic programs. They give characterizations of both universal and existential termination. There is, however, not much effort toward automatic verification of termination.

Using level mappings, Bezem [7] introduced the class of *recurrent programs*, which strongly terminate for a class of queries. He proved that every total recursive function can be computed by some recurrent program. `Append` and `Merge` are examples of recurrent programs, whereas `Quick-sort`, `Merge-sort`, etc. do not belong to the class of recurrent programs. Using models in addition to level mappings, Apt and Pedreschi [4] generalized the notion of recurrent programs to *acceptable* programs that terminate for a class of queries under Prolog's left-to-right selection rule. They called this notion of termination *left-termination.* Apt and Pedreschi [4] proved that a program terminates for all ground queries under Prolog's selection rule if and only if it is an acceptable program. The main steps in proving a program to be recurrent/acceptable involve finding a suitable level mapping and a model in the case of acceptability.

Wang and Shyamasundar [38, 39] and Bossi et al. [8] use graph abstraction to localize the task of finding suitable mappings. These two approaches are very similar. With a given logic program P and a goal G, [38, 39] associated a U-graph and [8] associated a specific graph. The nodes in these graphs are atoms in the program P and goal G, and there are two kinds of edges, namely signed edges and unification edges. A signed edge goes from the head of a clause to an atom in the body of the same clause, whereas unification edges go from an atom in the body of a clause/query to a head of a (not necessarily the same) clause, with which it unifies. The problem of finding a suitable mapping is locally solved by analyzing each strongly connected component (SCC) of the graph. The main steps in proving termination of logic programs using these approaches are construction of the graph, finding a suitable mapping, associating suitable pre/post assertions, and proving their correctness and a few simple conditions. The approaches are quite interesting. However, their mechanizability is not very clear.

Shyamasundar et al. [34] characterized strong termination of logic programs by using unification closures. The concept of unification closure is closely related to the concepts of forward closure and overlap closure used in term-rewriting literature to characterize the termination of linear term-rewriting systems. With a given logic program P, they associate a term-rewriting system RR_P and define unification closure of P for a given query Q (denoted UC_{P_Q}) in terms of RR_P and Q. Unification closure is a set of pairs of atoms of the form $\langle A,B \rangle$ such that corresponding to every atom in SLD-derivations of P starting with Q, there is a pair in the unification closure UC_{P_Q} and vice versa. This relationship between SLD-derivations and the unification closures led to the following characterization of strong termination: a logic program P is strongly terminating for a query Q if and only if UC_{P_Q} is finite and does not contain any pair of the form $\langle a_1, a_2 \rangle$ such that $a_1 = a_2 \sigma$ for some substitution σ.

7.2. *Automatic / Semi-Automatic Procedures*

We classify approaches proposed in this direction into two categories: (a) approaches based on linear inequalities and (b) transformational approaches.

Ullman and van Gelder's work [35] is the first to propose an automatic procedure for proving termination of logic programs. Their approach is to generate a set of linear predicate inequalities (of the form $p_i + c \geq p_j$) from a given well-moded program and a goal so that the satisfiability of these inequalities implies the termination of that program for that goal. The intended meaning of the inequality $p_i + c \geq p_j$ is that the size of the jth argument of predicate p is greater than the size of the ith argument of p by at most c units. The size of a term is defined as the number of function symbols in the term, with an assumption that the only function symbol available is '.', the cons operator. Ullman and van Gelder [35] presented an algorithm to generate a set of such inequalities using data flow analysis through variable/argument graphs. For this algorithm to work, all of the clauses in the program must satisfy the "uniqueness" property. A rule violates this property if a variable occurs more than once in the input positions of the literals in the body (see [35] for details).

Plümer [32] improved this method by generalizing the form of inequalities to $\Sigma p_i + c \geq \Sigma p_j$ and allowing function symbols other than the cons '.' operator. This resulted in a more powerful method. The method also uses data-flow analysis for deriving the inequalities, some (not all) of the programs violating the "uniqueness" property can be handled, and it uses AND/OR data flow graphs to generate the linear predicate inequalities. The algorithm for generating linear predicate inequalities from a given logic program with the above programs requires that the associated AND/OR data-flow graph satisfies the condition that each OR node has exactly one incoming edge. A graph satisfying this condition is called an *admissible solution graph*. This condition is in general violated by (not *all*) programs having a variable occurring in input positions of more than one atom in the body of a clause. (See [32] for an example program with a variable occurring in input positions of more than one atom in the body of a clause and having an admissible solution graph.) Apparently, this requirement is placed to get an efficient algorithm. This approach cannot handle programs with mutual recursion. Termination of `Append`, `Merge`, `Quick-sort`, and `Permutation` under Prolog's selection rule can be proved by using this approach, whereas termination proofs for multiplication and **ProCoS** compiler are beyond its scope, as some clauses in these programs do not have admissible solution graphs.

In contrast to our approach, Plümer's approach is automatic. Our approach often needs precedence information or polynomial/elementary interpretations from the user, whereas Plümer's algorithm derives the linear predicates from the program and moding information. The time complexity of Plümer's algorithm as well as our transformation procedure are *linear* in the number of atoms in the program, and both process the given program clause by clause. The transformation procedure has to analyze the literal dependency graph of each clause, which is simpler than the AND/OR data flow graph (of the clause) used by Plümer. Therefore, the running time of the transformation might be shorter than that of the algorithm for deriving linear predicate inequalities.

De Schreye and Verschaetse [12] proposed an amalgamated approach for termination analysis of logic programs using level mappings, linear inequalities, and abstract interpretation. This work is motivated by the difficulties involved in deriving automatic proofs of recurrences [7] and acceptability [4] of logic programs. It is pointed out by the authors that the difficulty is mainly due to the requirement that the level of the head is greater than the levels of all of the atoms in the body (in the context of acceptability, this is needed when all of the predecessors are satisfiable in the model), despite the fact that nontermination is possible only through recursion in logic programming. By relativizing the notions of recurrent and acceptable programs with respect to the set of queries, they refine these notions in such a way that the level of the head has to be greater than the levels of (only) those atoms in the body that are in (mutual) recursion with the head. Given a logic program, they try to derive a system of linear inequalities using abstract interpretation, so that the *unsatisfiability* of these inequalities implies acceptability of that program. Termination of `Append`, `Merge`, `Quick-sort`, `Multiplication`, and `Permutation` under Prolog's selection rule can be proved with this approach. However, termination of **ProCoS** compiler cannot be proved with this approach—the algorithm needs some of the clauses in the compiler to be unfolded to derive a suitable set of linear inequalities.

7.3. *Related Work in the Transformational Approach*

Our main result in the previous sections deals with termination of logic programs under a class of selection rules rather than a particular selection rule. In fact, the notion of *implied selection rules* suggests a class of suitable selection rules for the execution of a given well-moded program; furthermore, the termination of the derived rewrite system implies the termination of the logic program under all of these selection rules. In this sense, the result is stronger than the other results on termination of a logic program, which deal with termination under a particular selection rule (typically, Prolog's *leftmost* selection rule). However, it is also desirable to study termination under a particular selection rule, as the program need not be terminating under all of the implied selection rules, but can be terminating under a particular selection rule (typically, one of the implied selection rules). Our transformation returns a nonterminating rewriting system in that case, and we cannot establish termination under the particular selection rule. In the following, we discuss the results obtained using Prolog's *leftmost* selection rule.

As illustrated below, Krishna Rao et al. [24] proposed a transformation of a given logic program, so that only the Prolog's selection rule is implied by the moding information. Then one can use the results presented in the previous sections for proving termination of the transformed program, which in turn implies termination of the original program under Prolog's selection rule.

Example 20. Consider the following reachability program. Given a graph $G(V, Ed)$ with set of vertices V and set of edges Ed represented as list of pairs $f(X, Y)$, the problem is to find all of the vertices reachable from a given vertex.

```
moding: r (in, out, in, in), e (in, in, out) and nm (in, in)

r(X, Y, Ed, V) ← e(X, Ed, Y)
r(X, Y, Ed, V) ← e(X, Ed, Z), nm(Z, V), r(Z, Y, Ed, [Z|V])
e(X, [f(X, Y)|T], Y) ←
e(X, [f(X1, Y)|T], Z) ← X ≠ X1, e(X, T, Z)
nm(X, []) ←
nm(X, [H|T]) ← X ≠ H, nm(X, T)
```

This program terminates for the query $\leftarrow r(a, Y, [f(a,b), f(b,c), f(c,a), f(c,d)], [])$ under Prolog's selection rule, even though it has an infinite evaluation under the *rightmost atom* selection rule (which is also an implied selection rule). In [24], the following program is derived from the above program. The new program essentially encodes Prolog's selection rule into the original program.

```
moding: r_new(in, in, out, in, in), e_new (in, in, in, out)
  and nm_new (in, in, in, out)

r_new(true, X, Y, Ed, V) ← e_new(true, X, Ed, Y)
r_new(true, X, Y, Ed, V) ← e_new(true, X, Ed, Z), nm_new(true, Z,
  V, T), r_new(T, Z, Y, Ed, [Z|V])
e_new(true, X, [f(X, Y)|T], Y) ←
e_new(true, X, [f(X1, Y)|T], Z) ← noteq(X, X1, S), e_new(S, X,
  T, Z)
nm_new(true, X, [], true) ←
nm_new(true, X, [H|T], S1) ← noteq(X, H, S), nm_new(S, X, T, S1)
```

The transformation derives the following rewrite system:

```
r¹_new(true, X, Ed, V) → e¹_new(true, X, Ed)
r¹_new(true, X, Ed, V) → r¹_new(nm¹_new(true, e¹_new(true, X, Ed), V),
            e¹_new(true, X, Ed), Ed, [e¹_new(true, X, Ed)|V])
e¹_new(true, X, [f(X, Y)|T]) → Y
e¹_new(true, X, [f(X1, Y)|T]) → e¹_new(noteq(X, X1), X, T)
nm¹_new(true, X, []) → true
nm¹_new(true, X, [H|T]) → nm¹_new(noteq(X, H), X, T)
```

It is easy to observe that this rewrite system terminates because the first two rules can be applied only when the first argument of $\mathtt{r}^1_{\mathrm{new}}$ is *true*, which is passed by $\mathtt{nm}^1_{\mathrm{new}}$, and the value of $\mathtt{nm}^1_{\mathrm{new}}$ is true only when `X` is not a member of the visited vertices. So once all of the vertices in a cycle are visited, $\mathtt{nm}^1_{\mathrm{new}}$ will not pass `true` to the $\mathtt{r}^1_{\mathrm{new}}$, preventing it from entering into a cycle.

Ganzinger and Waldmann [21] proposed to transform a given logic program into a conditional term-rewriting system such that termination of the conditional term-rewriting system implies termination of the logic program under Prolog's selection rule. The most appealing thing about this transformation is that there is no need to introduce inverse functions. Their transformation can be explained as follows. They introduce two function symbols p^{in} and p^{out} of arities n and m, respectively, for each predicate symbol p with n input positions and m output positions. If $p(t_{i_1}, \ldots, t_{i_n}, t_{o_1}, \ldots, t_{o_m})$ is an atom, $p^{in}[t]$ denotes $p^{in}(t_{i_1}, \ldots, t_{i_n})$ and $p^{out}[t]$ denotes $p^{out}(t_{o_1}, \ldots, t_{o_m})$, where t is the tuple of terms $t_{i_1}, \ldots, t_{i_n}, t_{o_1}, \ldots, t_{o_m}$. From each clause $A_0(t_0) \leftarrow A_1(t_1), \ldots, A_k(t_k)$, they derive a conditional rewrite rule,

$$A_0^{in}[t_0] \rightarrow A_0^{out}[t_0] \Leftarrow A_1^{in}[t_1] \rightarrow A_1^{out}[t_1], \ldots, A_k^{in}[t_k] \rightarrow A_k^{out}[t_k].$$

Example 21. For the transitive-closure program given in Example 19, the following *terminating* conditional term rewriting system is derived:

```
pin(a) → pout(b)
pin(b) → pout(c)
tcin(X) → tcout(Y) ⇐ pin(X) → pout(Y)
tcin(X) → tcout(Y) ⇐ pin(X) → pout(Z), tcin(Z) → tcout(Y).
```

As demonstrated by the above example, the class of programs for which the transformation of [21] derives a terminating CTRS properly contains the class of programs for which our transformation derives a terminating TRS.

The fact that the termination of general term-rewriting systems is well studied and easier to understand than the termination of conditional term-rewriting systems motivated Aguzzi and Modigliani [1] and Chtourou and Rusinowitch [9] to derive a term-rewriting system from a given logic program (rather than a conditional term-rewriting system), such that its termination implies the termination of the logic program under Prolog's selection rule. Aguzzi and Modigliani [1] and Chtourou and Rusinowitch [9] independently came up with two similar transformations. Basically, [1, 9] derive a term-rewriting system from the conditional term-rewriting system derived by Ganzinger and Waldmann. From a non-unit clause $A_0(t_0) \leftarrow A_1(t_1), \ldots, A_k(t_k)$, they derive the following set of rewrite rules:

$$\{A_0^{in}[t_0] \to f_1(\mathscr{V}_0, A_1^{in}[t_1])\}$$
$$\bigcup_{j=1}^{k-1} \{f_j(\mathscr{V}_0, \mathscr{V}_1, \ldots, \mathscr{V}_{j-1}, A_j^{out}[t_j]) \to f_{j+1}(\mathscr{V}_0, \mathscr{V}_1, \ldots, \mathscr{V}_j, A_{j+1}^{in}[t_{j+1}])\}$$
$$\bigcup \{f_k(\mathscr{V}_0, \mathscr{V}_1, \ldots, \mathscr{V}_{k-1}, A_k^{out}[t_k]) \to A_0^{out}[t_0]\},$$

where $\mathscr{V}_i$ is the sequence (without repetitions) of variables produced by the atom $A_i(t_i)$ and $f_1, \ldots, f_k$ are distinct function symbols not occurring in the program and the rewrite rules derived from other clauses. From a unit clause $A_0(t_0) \leftarrow$, they derive a rewrite rule $A_0^{in}[t_0] \to A_0^{out}[t_0]$.

Example 22. The following term-rewriting system is derived by [1, 9] from the transitive-closure program given in Example 19:

```
pin(a) → pout(b)
pin(b) → pout(c)
tcin(X) → f1(X, pin(X))
f1(X, pout(Y)) → tcout(Y)
tcin(X) → g1(X, pin(X))
g1(X, pout(Z)) → g2(X, Z, tcin(Z))
g2(X, Z, tcout(Y)) → tcout(Y).
```

As the basic idea of [1, 9] is to derive a TRS from the CTRS derived by [21], the class of programs for which the above transformation derives terminating TRSs is same as that of [21].

Aguzzi and Modigliani [1] identified a class of programs, called *input-driven* programs, for which the derived TRS terminates if and only if the logic program terminates for well-moded queries. Arts and Zantema [5] worked on this issue further and investigated the classes of programs for which the derived TRS (their transformation is essentially same as that of [1, 9]) can be proved to be terminating by using recursive path ordering and semantic labeling. Using the results of Zantema [40], they show that their transformation derives from any structural recursive program (cf. [32]) a terminating TRS whose termination can be proved by using semantic labeling.

Using the results of [31] on unification-freeness, Marchiori [30] presented transformations for two subclasses (called simply well-moded and flatly well-moded programs) of the class of well-moded programs such that they derive a terminating TRS from any terminating program in these classes. The added advantage of unification-freeness of these classes of programs helps in proving the termination of logic programs, which cannot be proved by the other results in the field. The following program with moding `p(in, out)` is an example of this:

```
p(X, g(X)) ←
p(X, f(Y)) ← p(X, g(Y)).
```

8. CONCLUSION

A transformational approach to the termination analysis of logic programs has been presented by reducing the termination problem of logic programs to that of a term-rewriting system. Unlike the methods of Ullman and van Gelder [35], Plümer [32], and De Schreye and Verschaetse [12], our method does not need any preprocessing and can prove termination of programs that cannot be handled by them. Mutual recursion does not present a problem for the proposed method. The proposed transformation is purely syntactical and has been implemented as a front end to RRL. Using this tool, we have successfully proved termination of a prototype compiler for **ProCoS** language PL_0, which cannot be proved by the other mechanizable approaches.

We study a stronger notion of termination, i.e., termination under a class of selection rules, whereas almost all of the other approaches available in the literature deal with termination with respect to a particular selection rule, typically Prolog's leftmost atom selection rule. In fact, our notion of implied selection rules characterizes the class of selection rules suitable for a given well-moded logic program.

Although we have considered only well-moded queries in the previous sections, our results hold for a larger class of queries. The well-modedness ensures that the input terms of every selected atom are ground and the unification of input terms is one-sided (i.e., matching). This is the main property that enables applicability of termination techniques of rewriting in proving termination of logic programs. However, unification of input terms is one-sided for many non-well-moded queries. For example, this is true for queries with (not necessarily ground) lists as inputs in the queries to programs such as `append`, `permutation`, `reverse`, etc. In fact, our results extend to all unification-free programs and queries, where unification of input terms is always a matching (see [3, 31, 26] for many classes of unification-free programs). This extension of our main result is comparable to the results of Plümer [33].

APPENDIX

The following lemma plays a crucial role in proving that the computations of well-moded programs under implied selection rules are data-driven.

Lemma 12. If P is a well-moded program, G_0 is a well-moded query and S is a selection rule implied by $P \cup \{G_0\}$ and $G_0, G_1, \ldots, G_n$ be a SLD-derivation of $P \cup \{G_0\}$, then all the goals (queries) G_i satisfy the following property: If an atom A in G_i contains a variable X in its input terms then there exists a producer (say B) of X in G_i such that $B \prec_{G_i} A$, where $\prec_{G_i}$ is the evaluation order of the goal G_i.

PROOF. Induction on i.
Basis: $i = 0$. By the hypothesis of the lemma, G_0 is well-moded. By definition 4, variable X has a producer (call it B). It remains to show that $B \prec_{G_i} A$. By the hypothesis of the lemma, the evaluation order of G_0 is an extension of the producer-consumer relation of G_0. Hence $B \prec_{G_i} A$.
Induction hypothesis: Assume that lemma holds for $i = n$.
Induction step: $i = n + 1$. Let G_n be $\leftarrow q_1(\cdots), \ldots, q_m(\cdots)$. Let $q_j(\cdots)$ be the selected atom, $H \leftarrow B_1, \ldots, B_l$ be the input clause and σ be the mgu in the nth resolution step. Goal G_{n+1} will be $\leftarrow q_1(\cdots)\sigma, \ldots, q_{j-1}(\cdots)\sigma,\ B_1\sigma, \ldots, B_l\sigma,\ q_{j+1}(\cdots)\sigma, \ldots, q_m(\cdots)\sigma$.

We make a few observations before continuing with the proof. The mgu σ instantiates only those variables in G_n that occur in $q_j(\cdots)$. The selection rule always selects a minimal element, so $q_j(\cdots)$ is minimal and hence *its input terms are ground* by the induction hypothesis. Now we prove the lemma case by case.

Consider an atom $q_k(\cdots)\sigma$, $k \neq j$ containing a variable X in input position. There are two cases: (i) X is occurring in G_n and (ii) X does not occur in G_n. In case (i), by the induction hypothesis there is a producer (say, C) of X such that $C \prec_{G_n} q_k(\cdots)$. There are two subcases: (a) C is $q_j(\cdots)$ or (b) C is $q_{k'}(\cdots)$, $k \neq k' \neq j$. In subcase (a), since $q_j(\cdots)\sigma \equiv H\sigma$, X occurs in $H\sigma$ and therefore X occurs in the well-moded clause $H\sigma \leftarrow B_1\sigma, \ldots, B_l\sigma$. By Definition 4, X has a producer in the clause. Since input terms of $H\sigma (\equiv q_j(\cdots)\sigma)$ are ground, $H\sigma$ is not a producer of X. So some atom in the body (say $B_{l'}\sigma$) is a producer of X, i.e., X occurs in output positions of $B_{l'}\sigma$ and by the above construction, $B_{l'}\sigma \prec_{G_{n+1}} q_k(\cdots)\sigma$. In subcase (b), $q_{k'}(\cdots)\sigma$ is a producer of X and the lemma holds. This completes case (i).

Case (ii): X does not occur in G_n. That is, σ replaces a variable (say, Y) in $q_k(\cdots)$ by a term containing X. By the above observation, variable Y occurs in output terms of $q_j(\cdots)$ (note that input terms are ground). Proof of the lemma in this case is similar to that of subcase (a) of case (i).

Consider an atom $B_m\sigma$, $1 \leq m \leq l$, containing a variable X in input terms. It is obvious that there is a variable Y in the input clause $H \leftarrow B_1, \ldots, B_l$ such that $X \in Var(Y\sigma)$. By Definition 4, Y has a producer in well-moded clause $H \leftarrow B_1, \ldots, B_l$. Since input terms of $H\sigma$ are ground, Y does not occur in input terms of H, and hence H is not a producer of Y. Therefore, some atom (say $B_{m'}$) in the body is a producer of Y and hence $B_{m'}\sigma$ is a producer of every variable in $Y\sigma$; in particular it is a producer of X. By the definition of the implied selection rule, $B_{m'}\sigma \prec_{G_{n+1}} B_m\sigma$. This completes the proof of the lemma. □

Now we can prove Theorem 2 (cf. Section 2.2).

Theorem 2. If P is a well-moded program, Q is a well-moded query and S is a selection rule implied by $P \cup \{Q\}$, then every SLD-derivation of $P \cup \{Q\}$ is a data-driven evaluation.

Proof (By Contradiction). Assume that the selected atom at a resolution step has variables in its input positions. By the above lemma, there are producers that are smaller (under the evaluation order) than the selected atom contradicting the minimality of the selected atom. Therefore, input terms of the selected atom are ground at every resolution step. Hence every SLD-derivation starting with a well-moded query is a *data-driven* evaluation. □

We thank Prof. K. R. Apt and Prof. J. W. Klop of CWI for their helpful comments. Thanks are also due to the anonymous referees, whose comments have greatly improved the presentation. Dr. Kapur was partially supported by National Science Foundation Grant CCR-9303394 and National Science Foundation Indo-US Grant INT-9416687. Dr. Shyamasundar was partially supported by National Science Foundation Indo-US Grant INT-9416687.

REFERENCES

1. Aguzzi, G. and Modigliani, U., Proving Termination of Logic Program by Transforming Them into Equivalent Term Rewriting Systems, in: *Proc. 13th Conf. Foundations Software Technol. Theoret. Comput. Sci.* (FST&TCS'93), *Lecture Notes in Computer Science* 761, Springer Verlag, New York, 1993, pp. 114–124.
2. Apt, K. R., Logic Programming, in: J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, North-Holland, Amsterdam, 1990, Vol. B, pp. 493–574.
3. Apt, K. R. and Etalle, S., On the Unification Free Prolog Programs, in: *Proc. MFCS'93, Lecture Notes in Computer Science* 711, Springer Verlag, New York, 1993, pp. 1–19.
4. Apt, K. R. and Pedreschi, D., Reasoning about Termination of Pure Prolog Programs, *Information Computation* 106:109–157 (1993).
5. Arts, T. and Zantema, H., Termination of Logic Programs via Labelled Term Rewrite Systems, TR UU-CS-1994-20, Utrecht University, 1994.
6. Baudinet, M., Proving Termination Properties of Prolog Programs: A Semantic Approach, *Proc. LICS'88* (revised version in *J. Logic Programming* 15:1–29 (1988)).
7. Bezem, M., Strong Termination of Logic Programs, *J. Logic Programming* 15:79–98 (1989).
8. Bossi, A., Cocco, N., and Fabris, M., Proving Termination of Logic Programs by Exploiting Term Properties, *Proc. TAPSOFT'91, Lecture Notes in Computer Science* 494, Springer Verlag, New York, 1991, pp. 153–181.
9. Chtourou, M. and Rusinowitch, M., A Transformation from Logic Programs to Term Rewrite Systems and Its Application to Termination, draft, CRIN-INRIA, Lorraine, France, 1993.
10. Conery, J. S. and Kibler, D. F., AND Parallelism and Nondeterminism in Logic Programming, *New Generation Comput.* 3:43–70 (1985).
11. Dauchet, M., Simulation of Turing Machines by a Left-Linear Rewrite Rule, *Proc. RTA'89, Lecture Notes in Computer Science* 355, Springer Verlag, New York, 1989, pp. 109–120.
12. De Schreye, D. and Verschaetse, K., *Termination Analysis of Definite Logic Programs with Respect to Call Patterns*, TR CW 138, Department of Computer Science, K. U. Leuven, Belgium, 1992.
13. De Schreye, D., Verschaetse, K., and Bruynooghe, M., A Framework for Analyzing the Termination of Definite Logic Programs, *Proc. FGCS92, ICOT*, 1992, pp. 481–488.

14. De Schreye, D. and Decorte, S., Termination of Logic Programs: The Never-Ending Story, *J. Logic Programming* 19/20:199–260 (1993).
15. Debray, S. K. and Warren, D. S., Automatic Mode Inference for Logic Programs, *J. Logic Programming* 5:207–229 (1988).
16. Dembinski, P. and Maluszinski, J., And-Parallelism with Intelligent Backtracking for Annotated Logic Programs, *Int. Symp. Logic Programming*, 1985.
17. Dershowitz, N., Orderings for Term-Rewriting Systems, *TCS* 17:279–301 (1982).
18. Dershowitz, N., Terminiation of Rewriting, *J. Symb. Comp.* 3:69–116 (1987).
19. Dershowitz, N. and Jouannaud, J.-P., Rewrite Systems, in: J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, North-Holland, Amsterdam, 1990, Vol. B, pp. 243–320.
20. Francez, N., Grumberg, O., Katz, S., and Pnueli, A., Proving Termination of Prolog Programs, in: *Proc. Logics Programs, Lecture Notes in Computer Science* 193, Springer Verlag, New York, 1985, pp. 89–105.
21. Ganzinger, H. and Waldmann, U., Termination Proofs of Well-Moded Logic Programs via Conditional Rewrite Systems, in: *Proc. CTRS'92, Lecture Notes in Computer Science* 656, Springer Verlag, New York, 1992, pp. 430–437.
22. Kamin, S. and Levi, J.-J., Two Generalizations of Recursive Path Ordering, Unpublished note, Department of Computer Science, University of Illinois, Urbana, IL, 1980.
23. Kapur, D. and Zhang, H., An Overview of Rewrite Rule Laboratory (RRL), in: *Proc. RTA'89, Lecture Notes in Computer Science* 355, Springer Verlag, New York, 1989, pp. 559–563.
24. Krishna Rao, M. R. K., Kapur, D., and Shyamasundar, R. K., A Transformational Methodology for Proving Termination of Logic Programs, in: *Proc. Comput. Sci. Logic, CSL'91, Lecture Notes in Computer Science* 626, Springer Verlag, New York, 1991, pp. 213–226.
25. Krishna Rao, M. R. K., Pandya, P. K., and Shyamasundar, R. K., Verification Tools in the Development of Provably Correct Compilers, in: *Proc. 5th Symp. Formal Methods Europe, FME'93, Lecture Notes in Computer Science* 670, Springer Verlag, New York, 1993, pp. 442–461.
26. Krishna Rao, M. R. K. and Shyamasundar, R. K., Unification-Free Execution of Well-Moded and Well-Typed Prolog Programs, in: *Proc. Static Analysis Symp., SAS'95, Lecture Notes in Computer Science* 983, Springer Verlag, New York, 1994, pp. 243–260.
27. Lescanne, P., Computer Experiments with the REVE Term Rewriting Systems Generator, in: *Proc. 10th ACM POPL'83*, 1983, pp. 99–108.
28. Lescanne, P., Termination of Rewriting Systems by Elementary Interpretations, in: *Proc. Algebraic Logic Prog., ALP'92, Lecture Notes in Computer Science* 632, Springer Verlag, New York, 1992, pp. 21–36.
29. Lloyd, J. W., *Foundations of Logic Programming*, Springer Verlag, New York, 1987.
30. Marchiori, M., Logic Programs as Term Rewriting Systems, in: *Proc. Algebraic Logic Prog. ALP '94, Lecture Notes in Computer Science* 850, Springer Verlag, New York, 1994, pp. 223–241.
31. Marchiori, M., Localizations of Unification Freedom Trough Matching Directions, *Proc. Int. Symp. Logic Programming, ILPS '94*, MIT Press, Cambridge, MA, 1994.
32. Plümer, L., Termination Proofs for Logic Programs, Ph.D. Thesis, University of Dortmund (also appears as *Lecture Notes in Computer Science* 446, Springer Verlag, New York, 1990).
33. Plümer, L., Automatic Termination Proofs for Prolog Programs Operating on Nonground Terms, in: *Proc. ILPS'91*, MIT Press, Cambridge, MA, 1991, pp. 503–517.
34. Shyamasundar, R. K., Krishna Rao, M. R. K., and Kapur, D., Rewriting Concepts in the Study of Termination of Logic Programs, in: K. Broda (ed.), *Proc. ALPUK'92 Conf. Workshops in Computing Series*, Springer Verlag, New York, 1990, pp. 3–20.
35. Ullman, J. D. and van Gelder, A., Efficient Tests for Top-Down Termination of Logical Rules, *JACM* 35:345–373 (1988).

36. Vasak, T. and Potter, J., Characterization of Terminating Logic Programs, *Third IEEE Symp. Logic Programming* 140–147 (1986).
37. Verschaetse, K., Static Termination Analysis for Definite Horn Clause Logic Programs, Ph.D. Thesis, K. U. Leuven, Leuven, Belgium, 1992.
38. Wang, B. and Shyamasundar, R. K., Towards a Characterization of Termination of Logic Programs, in: *Proc. PLILP'90, Lecture Notes in Computer Science* 456, Springer Verlag, New York, 1990, pp. 203–221.
39. Wang, B. and Shyamasundar, R. K., Methodology for Proving the Termination of Logic Programs, in: *Proc. STACS'91, Lecture Notes in Computer Science* 480, Springer Verlag, New York, 1991, pp. 214–227.
40. Zantema, H., Termination of Term Rewriting by Semantic Labelling, *Fundamenta Informaticae* 24:89–105 (1995).