A METHODOLOGY FOR PROVING TERMINATION OF LOGIC PROGRAMS *

Bal Wang†
Graduate Institute of Information Engineering,
Tam Kang University (City Campus),
Li Sui Street, Taipei, Taiwan

R.K. Shyamasundar‡
Tata Institute of Fundamental Research
Bombay 400 005, India
e-mail: shyam@tifrvax.bitnet

Abstract

In this paper, we describe a methodology for proving termination of logic programs. First, we introduce U-graphs as an abstraction of logic programs and establish that SLDNF-derivations can be realized by instances of paths in the U-graphs. Such a relation enables us to use U-graphs for establishing the *universal termination* of logic programs. In our method, we associate pre- and post-assertions to the nodes of the graph and *ordered assertions* to selected edges of the graph. With this as the basis, we develop a simple method for establishing the termination of logic programs. The simplicity/practicality of the method is illustrated through examples.

1 Introduction

One of the most important features of logic programming is its declarative semantics. That is, one can consider the programs to be self-specifying as they are non-procedural, and hence do not need elaborate correctness proofs. There can be a debate about whether it is meaningful to talk about verifying logic programs. It will however be difficult to justify arguments against methods for establishing termination of logic programs. In fact, any

*An earlier version of the paper was presented at STACS 91.

†Most of the work was done while the author was with Department of Computer Science, Pennsylvania State University, University Park, PA 16802.

‡Correspondences may be sent to this address.

one who has written logic programs has to confront sooner or later, the possibility of a program unleashing an infinite computation either because of a bug in one's program or due to the idiosyncrasy of an interpreter being used for a logic programming language. The need for additional information (assertions) for proving termination of logic programs from the declrative semantics follows from the fact that properties of logic programs such as form of procedure calls, success patterns of terms *etc.* that are usually used by programmers for reasoning about the correctness (partial or total) of programs, are inexpressible in terms of the declarative semantics [13].

The primary aim of this paper is to develop a methodology for establishing termination of logic programs. From Floyd's [17] fundamental work on termination, it is clear that any method[1] of showing termination involves discovery of a well-founded set with respect to a given set of initial assertions. Our approach is motivated by the works on termination of term-rewriting techniques [12]. In our approach the discovery of well-founded ordering is localized by representing the program as an *U-graph.* The basis of the method can be informally understood by analyzing SLDNF derivations. For showing the universal termination of the program with respect to the given query, we have to show that all the subgoals give rise to only finite derivations (whether it succeeds or fails). Let us consider an SLDNF derivation where the ith goal (G_i) is given by: $\leftarrow A_1, A_2, \ldots, A_m, \ldots A_k$. Consider the derivation of G_{i+1} from G_i with the input clause $C_i \equiv A \leftarrow B_1, \ldots, B_q$ using the unifier θ_{i+1} between A_m and A. Then,

$$G_{i+1} \equiv \leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots A_k)\theta_{i+1}$$

Now, if we can show that the *order* of G_{i+1} decreases with respect to G_i, then it would enable us to show the termination of the programs. However, it is important to note that just establishing the decreasing nature for the right-hand side (or the body) of the chosen clause with respect to the unifier θ_{i+1} is not sufficient; the decreasing order should be established for the set of all goals in G_{i+1} with respect to the unifier. It can be easily seen that if the predicates are non-recursive, then the task is trivial. However, if there are recursive predicates then showing the finiteness of the SLDNF tree is non-trivial. It is here that the abstraction of the program in terms of the U-graph helps. Recursive predicates can be identified through the cycles of the graph. For proving the correctness of programs, we introduce assertions on vertices and edges in a selective manner and a well-founded order. For purposes of *covering* each goal, a set of edges that belong to cycles of the U-graph are selected and assertions (referred to as *ordered assertions*) are attached. Using this set of assertions, the termination of logic programs is established relative to the given set of assertions (for nodes of the U-graph and selected edges of the graph). The reasons for using assertions only for selected edges rather than for every edge will become clear in the sequel. Such an approach has two advantages: first, the effort of finding a well-founded order is localized and secondly, the localization property enables us to use some of the techniques of term-rewriting (cf. [12, 20]) for arriving at appropriate assertions.

[1] Irrespective of whether the program is deterministic, nondeterministic, backtracking *etc.*

The main contribution of the paper is that it provides a method for showing the termination of the programs by analyzing the term-structure of the program. The method can be used for establishing the universal termination of logic programs. Further, it can be used for proving termination of Prolog programs (as in [2]) taking into account the selection rule.

The rest of the paper is organized as follows: Section 2 provides a brief survey of termination of logic programs. Section 3 introduces the U-graph and the specification language of assertions is described in section 4. Various safety properties of assertions are introduced in Section 5. The method of establishing termination is described with an example in Section 6. In Section 7, we establish the soundness and completeness issues of the method. Section 8 discusses the heuristics for arriving at the pre- and the post-assertions followed by a discussion of the method in section 9. Throughout the paper, we follow the definitions and notation given in [3, 27] unless otherwise stated.

2 Termination of Logic Programs

In this section, we provide a brief account of the spectrum of methods that have been used for proving termination of logic programs. Our primary aim is to highlight different techniques that have been used for proving termination of logic programs rather than provide an exhaustive survey. For a survey, the reader is referred to [15]. In the brief account given here, we first introduce some of the widely used notions of termination of logic programs and then, discuss the main features behind the various techniques of proving termination of logic programs.

2.1 Notions of Termination of Logic Programs

Broad notions of termination of logic programs have been envisaged in [35]. One interpretation is that a program *terminates* iff it either fails finitely or produces a successful derivation. This is referred to as *existential termination*. Another interpretation is that a program *terminates* if and only if all the derivations are finite. This is referred to as *universal termination*. That is, the above notions correspond to finding one and every solution of a program respectively. It may be noted that the property of universal termination is sensitive only to computation rule (i.e., subgoal selection rule) whereas existential termination depends on the selection rule as well as the clause selection rule (i.e., search rule).

A logic program P is said to *strongly terminate* with respect to G if *every* SLD-tree for $P \cup \{G\}$ is finite. That is, G terminates for all computation rules. Note that the notion of strong termination is stronger than universal termination. A program P is said to *weakly terminate* with respect to G if there *exists* a finite SLD-tree for $P \cup \{G\}$. That is, G terminates for some computation rule.

Another notion of termination referred to as *left termination* has been introduced in [2] taking into account Prolog's left-to-right selection rule. A program P is said to be *left* *terminating* with respect to goal G if the SLD-tree for $P \cup \{G\}$ is finite with respect to the *leftmost* (as in Prolog) selection rule.

2.2 Techniques of Proving Termination

There has been a considerable amount of work in the past few years devoted to the issue of termination in the area of logic programming. For the sake of convenience, we broadly categorize (not necessarily disjoint) the works on termination of logic programs into:

1. *Proof Systems*: These works resemble the works on total correctness of traditional programs. The work of [16] falls into this category. Baudinet's work [5] can also be seen as a proof system for Prolog programs.
2. *Characterization of Termination*: The broad rationale of the works in this category is to provide semantic frameworks and techniques to formulate and solve various questions of termination of logic programs. Some of the works belonging to this category are reported in [2, 4, 5, 7, 9, 33, 35].
3. *Automatable Techniques for Proving Termination*: The primary aim of the works in this category is to find automatable and practical techniques for proving termination of classes of logic programs (and deductive databases). The works of [6, 8, 30, 21, 34, 14, 36] fall into this category.

Salient features of some of the above mentioned works are briefed below.

Proof Systems

Francez et al. [16] formulate proof rules for proving the *existential* termination of Prolog programs, (essentially, total correctness[2] in the traditional sense) based on the notions of guarded directions and parameterized invariants for guarded commands. Basically, they seek for a well-founded variant function that is decreasing on the sequence of computation states. This approach appears to be unduly complicated, as one has to use the proof rules for describing the execution behavior of the interpreter also for reasoning about the termination of programs. For this reason, the proofs of even very simple programs become very complex with such an approach. However, the most interesting aspect of the technique is that it caters to Prolog's search and computation rules.

Baudinet [5] describes semantics of Prolog programs taking into account various control aspects including *cuts*. The method consists of associating a system of functional equations whose least fix point defines the meaning of the program. Many termination and nontermination issues can be formulated as a problem in *first order logic* and proved using structural induction. Such a formulation makes it possible to use some of the existing theorem provers. However, it is not clear as to how effectively the termination properties of logic programs can be automatically established.

[2]The work of [13] is concerned with the development of a proof system for establishing partial correctness of logic programs using the inductive assertion method.

Characterization of Termination

Vasak and Potter [35] have introduced various notions of termination discussed above and have developed characterizations of the class of universal terminating goals for a given program with respect to selected computation rules using fix point operators. However, these characterizations cannot be used effectively for proving the termination of logic programs.

Bezem [7] characterizes a class of logic programs referred to as *recurrent programs* based on the notion of *level mappings* (i.e., a function assigning natural numbers to ground atoms). A program is called *recurrent* with respect to level mapping $|\ |$, if for every clause $A \leftarrow B_1, \cdots, B_n$ in $ground(P)$, $|A| > |B_i|$ for all i. Bezem has shown that a logic program terminates if and only if it is recurrent and every totally recursive program is computed by a recurrent program. Note that the characterization ignores selection rules. Apt and Bezem [1] combine the notions of stratification of programs and level mappings and introduces a class of programs referred to as *acyclic* programs. It is also argued that acyclic programs terminate for a large and a natural class of general goals and further, the class of programs can be used for temporal reasoning.

Apt and Pedreschi [2] provide a framework for studying left-terminating programs. The framework is provided by combining the notions of level mapping and recurrent programs envisaged in [7, 9] with model construction methods. The authors introduce a new class of programs referred to as *acceptable* programs. Informally, a program is said to be acceptable if for all ground instances of the clauses of the program and some level mapping and a model, the level of the head is smaller than the level of atoms in a certain prefix (determined by the model of the program) of the body. The authors establish that the two notions of left-termination and acceptability coincide.

One of the distinguishing features between logic programs and term rewriting systems is the presence of *local variables* in the former. Shyamasundar et al. [33] provide a characterization of termination of logic programs through the termination characteristics of term rewriting systems. The study shows that termination characterization of term rewriting systems can be effectively used for establishing termination of logic programs. Practical techniques for establishing termination are reported in [21].

Automatable Techniques for Proving Termination

One of the most versatile tools that has been in use for checking termination/nontermination has been the tools for *loop checking*. The main purpose of loop checks are to reduce search space without pruning solution space – thus, gaining efficiency without loosing soundness. A theoretical basis for loop checking has been provided in [8]. A loop check is said to be *sound* if it does not prune an SLD-tree to such an extent that solutions are lost. It is said to be *weakly sound* if its application results in the loss of some solutions but not the loss of all solutions. If the loop check results in a *finite space* it is said to be *complete*. A spectrum of concrete loop checks lying between sound loop checks and complete loop checks has been discussed in [8].

Ullman and van Gelder [34] has been one of the first works in the development of automatable methods for proving termination of logic programs and deductive databases. The method is based on generating a set of linear inequalities of the form $p_i + c \geq p_j$ to describe inter-argument inequalities of the predicates for the given program and the query under the assumption that there are no function symbols other than the *cons* operator on lists. The satisfaction of these inequalities provides a sufficient condition for establishing termination of the program. Plümer [30] extends the method of [34] for studying the termination of well-moded programs by generalizing the form of inequalities to $\Sigma p_i + c \geq \Sigma p_j$ and allowing other function symbols. Both the methods have various restrictions such as *uniqueness* and *existence of admissible solution graphs*. In fact, these properties do not hold in general for programs where a variable occurs in input positions of more than one atom in a program clause. Such restrictions make some interesting class of programs (such as *multiplication program*) beyond the scope of such techniques. Further, the method requires pre-processing.

De Schreye and Verschaetse [14] extend the work on recurrent- and acceptable-programs by Bezem [7] and provide methods to arrive at natural level mappings using abstract interpretations with the intention of making the method amenable for automation.

A transformational methodology of transforming a given well moded logic program to a rewrite system preserving termination characteristics has been envisaged in [21]. The primary motivation of such an approach has been to exploit the powerful techniques and the tools available for showing the termination of term rewriting systems. In fact, the method does not have restrictions such as those prescribed for the methods of [30, 34] and the method has been used to show the termination of several benchmark programs illustrated in [2] and to prove the termination of prototype ProCoS compiler [22]. It may be noted that proving termination of the ProCoS compiler falls outside the scope of [30, 34]. One of the most interesting aspect of this method has been the mechanizability of the technique and the effective use of theorem provers such as RRL [20], REVE [25], ORME [26] etc. The method has also been adapted for proving termination of parallel logic programs [24].

In the following sections, we discuss a formal approach for proving termination of logic programs using a graphical abstraction of logic programs. The method has the distinct advantage of exploiting programmer's intuition and term structure of the program for proving formal properties (including termination) of the program effectively.

3 U-graphs

In this section, we introduce the main tool, U-graph[3], for analyzing logic programs. Relationship between U-graphs and SLDNF derivations is established through the notion of *g-trees* that relates a path in the U-graph and a sub-derivation of a given SLDNF derivation.

Graphical abstraction of logic programs has been in use for the analysis of normal programs. One of the most widely used abstractions is the signed dependency graph [23, 10].

[3] Here, U stands for unification.

In a signed dependency graph, the set of predicate symbols form the vertex set and the set of edges are pairs $< p, q >$ where p is the predicate symbol of the head of a clause C and q is the predicate symbol of a literal occurring in the body of C. In a sense, the relation is obtained by ignoring the arguments of the literals in each clause of normal programs. Though the construct is a succinct abstraction for several purposes, it discards too much information. In this section, we introduce U-graphs as an abstraction of a normal program from which we can abstract not only the signed-dependency relation but also the necessary unification information for establishing termination.

Without loss of generality, we assume that there are no common variables between any two clauses in a given normal program. In any clause C, we consider literals in the head and the body of C as distinct literals; we resort to subscripting the literals whenever one or more literals with the same predicate symbol appear more than once in the clause. For instance, the literals with the same predicate symbol in the clause

$$p(s(X)) \leftarrow p(s(X)), \neg p(X)$$

are subscripted as illustrated below:

$$p_h(s(X)) \leftarrow p_{b,1}(s(X)), \neg p_{b,2}(X)$$

Definition 3.1 *(U-graph)* Given a normal program P, the *U-graph* $U(V, E)$ of P is a directed graph, where the set of vertices V is the bag of all atoms occurring in P, and edge $< A, B > \in E$ if either

1. There is a clause of the form $A \leftarrow \cdots, B, \cdots,$ (or $A \leftarrow \cdots, \neg B, \cdots$) ; this edge is referred to as *signed edge*.
2. There are two clauses C_1, C_2 such that A (or $\neg A$) occurs in the body of C_1, B is the head of C_2, and A unifies with B; in this case the edge is called an *u-edge* (i.e., unification edge). $\diamond$

Notation 3.1 Given a normal program P, let V_H denote the set of all heads of clauses in P, and V_B denote the set of all atoms occurring in the body of clauses in P. Clearly, $V = V_H \cup V_B$. $\diamond$

Remark 3.1

1. All in-edges of a vertex in V_H (V_B) must be from some vertices in V_B (V_H), and all out-edges of a vertex in V_H (V_B) must be leading to vertices in V_B (V_H). Hence, every path in a U-graph consists of vertices in V_H and V_B alternately.
2. Each cycle has even length.

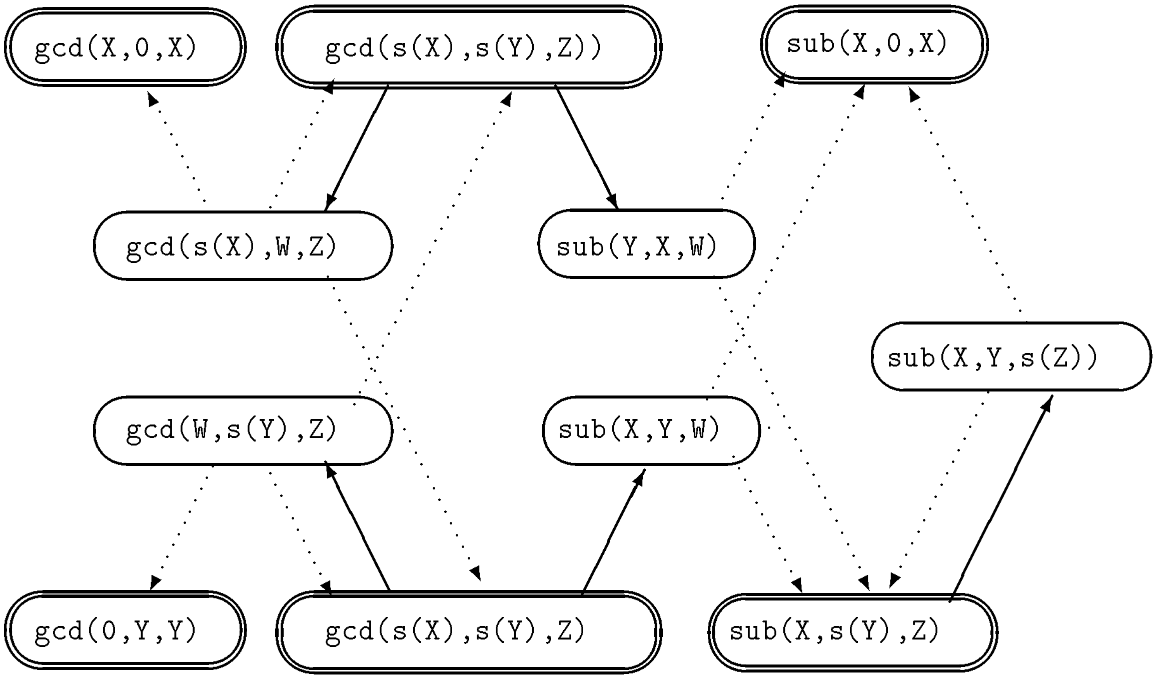

Figure 3.1 : U-graph of Program GCD
u-edges are shown in dotted arrows and signed edges in solid arrows

3. The notion of U-graphs has some resemblance to the notion of *connection graphs* [29]. It may be noted that the purpose of connection graphs is to provide a template for generating the AND/OR solution tree for the problem at hand; however, the purpose of the U-graphs is to capture the static information of the logic program as far as possible.

4. All hierarchical programs (cf. [11, 32, 27]) have acyclic U-graphs. Hence, SLDNF-derivations are finite. $\diamond$

Example 3.1 Consider the following GCD program (cf.[18]) which computes the greatest common divisor of two integers:

$$C_1 : gcd(X, 0, X) \leftarrow$$
$$C_2 : gcd(0, Y, Y) \leftarrow$$

$$C_3 : gcd(s(X), s(Y), Z) \leftarrow sub(Y, X, W), gcd(s(X), W, Z)$$
$$C_4 : gcd(s(X), s(Y), Z) \leftarrow sub(X, Y, W), gcd(W, s(Y), Z)$$

$$C_5 : sub(X, 0, X) \leftarrow$$
$$C_6 : sub(X, s(Y), Z) \leftarrow sub(X, Y, s(Z))$$

The U-graph of P is shown in Figure 3.1. $\diamond$

The relationship between SLDNF derivations and U-graphs enables us to confine ourselves to U-graphs for analyzing the termination of programs. The formal equivalence between instances of paths in U-graphs and SLDNF -derivations is given in the appendix.

4 Specification Language for Assertions

In this section, we define the metalanguage for assertions. The assertions are basically used for specifying properties that are inexpressible in terms of the declarative semantics. Typical properties include the actual form of procedure calls and successes, mode declarations etc.

The metalanguage as specified below can also refer to nonground terms (this aspect turns out to be useful in explaining the concept of a logical variable in a program). We first extend the Herbrand base to the set of terms which consists of functions, constant symbols and an enumerable set of variables. We define *extended Herbrand universe* of program P, denoted U_P^E, to be the Herbrand base obtained from the set of terms consisting of function symbols, constant symbols and an enumerable set of variables. Our method requires assertions for the vertices and edges for the U-graph derived from the program. The specification language for the assertions is an extension of the specification language defined in [13]. Let the *object language* be the language of clauses. For the meta- language, we use the extended Herbrand base of the object language as our domain of interpretation since we are interested in describing relations on (object language) terms. The specification language is formally described below. The functors and teh predicate symbols of the meta language define below pertain to some basic operations and relations.

Definition 4.1 *(The Specification Language for Assertions)*

1. *Variables*:
   (a) We use variables denoted by $i.A$ and $A.i$ for denoting the values of the i-th argument of the A *at* invocation, and *after* invocation respectively.
   (b) $T, U, V, \ldots$ refer to variables.
2. *n-ary function symbols* ($n \geq 0$):
   (a) n-ary function symbols of the object languages.
   (b) Variables of the object language: $X, Y, Z, \ldots$ ($n = 0$).

   Note that the interpretation of a function symbol corresponding to (a) or (b) is the function symbol itself.
3. *Terms*: Standard definition.
4. *Predicate Symbols*: $=, var, ground, \triangleleft, \approx, \ldots$. The interpretation of these predicates is given below:

- $=$ term equality.
- $var(T)$ iff T is a (object language) variable.
- $ground(T)$ iff T is a (object language) ground term.
- $T \lhd U$ iff T is a proper sub-term of U.
- $T \approx U$ iff the terms T and U are variants of each other (they differ at most in the names of their variables).
- $disconnected(V_1, \ldots, V_n)$ iff no variable occurs in more than one of the terms in $\{V_1, \ldots, V_n\}$.
- $subterm(T, U, I)$ iff $T \lhd U$ and I is the corresponding selector (assuming any fixed way of assigning selectors to subterm occurrences).

5. Logical *constants*: *true*, *false*, *quantifiers* and *connectives*: $\wedge$, $\vee$, $\Rightarrow$, ...

6. *Formulas*: standard definition.

7. *Assertions*: For any vertex in the U-graph (say corresponding to a literal, A), an assertion $< F_A^b, F_A^a >$ can be attached with the interpretation that F_A^b is the pre-assertion and F_A^a is the post-assertion. Note that F_A^b is a formula which contains only variables of the form $i.A$ where i ranges over all the argument positions of A. However, F_A^a is a formula containing variables of the form $i.A$ or $A.i$ where i ranges over all the argument positions of A.

8. *Order assertions*: For any vertex A in a non-trivial SCC[4] in the U-graph, an *order assertion* denoted by $O(1.A, \ldots, n.A)$ is attached, where O is a partial mapping[5] from $U_P^{E^n}$ to a well-founded ordered set W with least element 0, and n is the arity of A. For the sake of convenience and easy reference, we sometimes use the notation $O_A(A\theta)$ to denote the ordered assertion of literal A. We refer the tuple of assertions (O_A, O_B) where O_A and O_B are the order assertions associated with nodes A and B respectively of the edge $< A, B >$ as the order assertion associated with edge $< A, B >$. $\diamond$

Remark 4.1 1. It must be noted that certain characteristics of the program cannot be represented in the U-graph. For instance, the property of sharing variables among the literals in the body of a clause is not available in the U-graph. However, the assertion language is rich enough to carry this information.

2. It is only when a signed edge $< A, B >$ is used in a derivation, i.e., $B\theta$ becomes the selected subgoal, for some substitution θ, the assertion attached to A becomes applicable. Using the producer-consumer concept, one could arrive at a proper ordering with reference to a given computation rule. This would become clear in the sequel.

[4] A non-trivial SCC is a strongly connected component consisting of more than one vertex.

[5] Note that for different vertices, we associate different mappings. In other words, the mapping O is defined in the context of a particular vertex. However, we will not explicitly mention the vertex whenever it is clear from the context.

3. Another aspect we need to consider is the computation rule. In this section, we consider a general computation rule rather than just the Prolog computation rule. The main idea of the computation rule is to provide the order of evaluation of the goals in each clause. This can be captured as a partial ordered set. For example, the partial order relation $A < B$ corresponds to the interpretation that A has to succeed before B can be selected. For example, consider a Prolog clause having n literals in its body. If $B_1, \ldots, B_n$ is the body, then the computation rule can be captured by the partial ordered set $B_1 < B_2 < \ldots < B_n$ to depict the left-to-right order of the computation rule.

$\diamond$

5 Consistency of Assertions and Ordered Assertions

First, we define conditions under which the assertions attached to the vertices of the U-graph remain consistent.

Definition 5.1 Let A be a vertex in V_H, and $\{< A, B_1 >, \ldots, < A, B_n >\}$ be the set of all out-edges of A, and $< F_A^b, F_A^a >$, $< F_{B_1}^b, F_{B_1}^a >$, $\ldots$, $< F_{B_n}^b, F_{B_n}^a >$ be the assertions attached to $A, B_1, \cdots, B_n$ respectively. Then, we refer the set $\{< F_A^b, F_A^a >, < F_{B_1}^b, F_{B_1}^a >, \ldots, < F_{B_n}^b, F_{B_n}^a >\}$ as the *A-set* of A. $\diamond$

Definition 5.2 Let H be a vertex in V_H, and $\{< H, B_1 >, \ldots, < H, B_n >\}$ be the set of all out-edges of H, P_{OS}, be the partial order set over $\{1, \cdots, n\}$ corresponding to the given computation rule. The set $\{< F_H^b, F_H^a >, < F_{B_1}^b, F_{B_1}^a >, \ldots, < F_{B_n}^b, F_{B_n}^a >\}$ of pre- and post-assertions attached to the set of vertices $\{H, B_1, \cdots, B_n\}$ is said to be *consistent* with respect to P_{OS}, if

$$\forall\, k, \quad F_H^b \ \wedge \ (\bigwedge_{(i,k) \in P_{OS}} F_{B_i}^a) \ \Rightarrow \ F_{B_k}^b \tag{1}$$

$$F_H^b \ \wedge \ (\bigwedge_{i=1}^{n} F_{B_i}^a) \ \Rightarrow \ F_H^a \tag{2}$$

$\diamond$

Note:

(i) $P_{OS} = \phi$ corresponds to the fact that any literal in the body can be chosen in any order. In this case, condition 1 simply becomes

$$\forall k, F_H^b \ \Rightarrow \ F_{B_k}^b \tag{3}$$

(ii) If n = 0, i.e., $H = p(t_1, \ldots, t_m)$ is the head of a unit clause, then condition (2) reduces to

$$F_H^b \;\Rightarrow\; F_H^a \tag{4}$$

and condition (1) becomes vacuous.

(iii) Essentially, relation (1) shows that if the subgoal B_k in the clause depends on the variables of some subset of the literals in the clause, say $\{H, B_{i_1}, \ldots, B_{i_m}\}$, i.e., $(i_j, k) \in P_{OS}, j = 1, \ldots, m$, then the pre-assertion of H after unification with the goal together with the post-assertions of B_{i_j}, for $1 \le j \le m$, should imply the pre-assertion of B_k. Relation (2) shows that the pre-assertion of H and the post-assertions of all the subgoals B_i in the body of the clause whose head is H should imply the post-assertion of the head of the clause.

(iv) If it is a Prolog clause then (1) becomes

$$F_H^b \wedge (\bigwedge_{j=1}^{k-1} F_{B_j}^a) \;\Rightarrow\; F_{B_k}^b$$

That is, the partial order set is nothing but left-to-right order.

(v) Thus, P_{OS} reflects the dependence of an atom in the body on the othere literals in the body. Thus, it is natural to keep it minimal for proving stronger properties.

Example 5.1 Consider the following program PERMUTE given below:

$R_1 : permute([\,], [\,]) \leftarrow$
$R_2 : permute(T, [H|P]) \leftarrow remove(T, H, R), permute(R, P)$
$R_3 : remove([H|L], H, L) \leftarrow$
$R_4 : remove([B|C], D, [B|E]) \leftarrow remove(C, D, E)$

Let H, B_1, B_2 be $permute(T, [H|P])$, $remove(T, H, R)$, $permute(R, P)$ in R_2, respectively. Let

$< F_H^b,\ F_H^a >=< ground(1.H),\ true >$
$< F_{B_1}^b,\ F_{B_1}^a >=< ground(1.B_1),\ ground(B_1.1) \wedge ground(B_1.3) \wedge (B_1.3 \lhd B_1.1) >$
$< F_{B_2}^b,\ F_{B_2}^a >=< ground(1.B_2),\ true >$

be the assertions attached to H, B_1, B_2, respectively. We can see that the A-set of H is consistent with respect to partial order $\{(1, 2)\}$. $\diamond$

Remark 5.1

1. The selection rule defined through the partial ordered set can be understood in an easy way if we consider the special case of moding of the clauses or program[6]. In the case of moding, we have to make sure that there is partial ordering of the subgoals.

[6] Moding can be represented by using pre-assertions specifying the arguments that are grounded.

2. It may be noted that the variables in $F^b_{B_k}$ and $F^a_{B_k}$ are sensitive to the substitutions applied in a SLDNF-derivation. In condition (1), although we do not explicitly show the substitutions, it should be clear that the condition only depends on the answer substitutions of the subgoals $B_{i_1}, \ldots, B_{i_m}$, where $(i_j, k) \in P_{OS}$. This also shows that the computation (selection) rule also plays a very important role in assuring that condition (1) can be satisfied; we shall discuss this in a later section.

3. Instead of using the natural order (Prolog order, i.e., from left to right) of subgoals in the body of a clause, we use a partial order set which contains the necessary dependent relation among the subgoals to highlight the ability of our method for general computation rules including parallel computation rules.

$\diamond$

Definition 5.3 Let $A \leftarrow B_1, \cdots, B_m$ be a clause. If B_k can be selected/invoked only after the success of literals $B_{i_1}, \cdots, B_{i_r}$ in the body of the clause with respect to the selection rule given by P_{OS} such that $\{B_{i_1}, \cdots, B_{i_r}\} \subset \{B_1, \cdots, B_m\}$ then the *preassertion* to be satisfied before B_k can be invoked is given by

$$B_k^{pre} = F^b_A \wedge F^a_{B_{i_1}} \wedge \cdots \wedge F^a_{B_{i_r}} \wedge F^b_{B_k}$$

$\diamond$

It may be noted that the *preassertions* enable to use the side-ways informations. Having considered the consistency of vertices in V_H, we consider consistency of vertices in V_B.

Definition 5.4 Let B be a vertex in V_B, and let $\{< B, H_1 >, \ldots, < B, H_n >\}$ be the set of all out-edges of B. We say that the set of pre- and post- assertions attached to vertices $B, H_1, \ldots, H_n$ given by $\{< F^b_B, F^a_B >, < F^b_{H_1}, F^a_{H_1} >, \ldots, < F^b_{H_n}, F^a_{H_n} >\}$ of B is *safe* if

$$F^b_B \;\Rightarrow\; F^b_{H_j} \;\text{ for all j} \tag{5}$$

$$F^a_{H_j} \wedge F^b_B \;\Rightarrow\; F^a_B \;\text{ for all j} \tag{6}$$

$\diamond$

The relation (5) corresponds to saying that after unifying the subgoal $B\theta$ with the heads of clauses, the pre-assertion of B should imply the pre-assertions of the heads of clauses with respect to their mgu's. In other words, this condition corresponds to the *enablement* of the input clauses. The relation (6), corresponds to saying that the values consistently passed by the heads of the clauses to the subgoals. satisfy the post-condition of the subgoals.

Conditions (1), (2), and (5) are similar to the sufficient condition (SC) of the main theorem in [13]; however, there is no condition in SC corresponding to (6). The reason is that the analysis in [13] is based on global analysis, i.e., using predicate symbols as the underlying elements; whereas, we use a local analysis, i.e., use occurrence of the literals as the underlying elements. In [13], for each u-edge[7] $< A, B >$, the post-assertions of A

[7] Note that there is no notion of u-edge in [13]; this is only our interpretation for purposes of comparison.

and B are essentially the same, since A and B must have the same predicate symbol. In this case, formula (6) simply becomes a tautology. But in our approach, we may attach different assertions to them. We believe that the local analysis approach provides a good intuitive understanding without requiring any extra work as compared to the global analysis approach.

Example 5.2 In program PERMUTE in example 5.1, let B correspond to $r(T, H, R)$ in clause R_2, and H_1 and H_2 correspond to the heads $remove([H|L], H, L)$, $remove([B|C], D, [B|E])$ occurring in R_3 and R_4, respectively. Let

$$< F_B^b,\ F_B^a >=< ground(1.B),\ ground(B.1) \wedge ground(B.3) \wedge (B.3 \lhd B.1) >$$
$$< F_{H_1}^b,\ F_{H_1}^a >=< ground(1.H_1),\ ground(H_1.1) \wedge ground(H_1.3) \wedge (H_1.3 \lhd H_1.1) >$$
$$< F_{H_2}^b,\ F_{H_2}^a >=< ground(1.H_2),\ ground(H_2.1) \wedge ground(H_2.3) \wedge (H_2.3 \lhd H_2.1) >$$

be the assertions attached to B, H_1, H_2, respectively. The A-set of B is safe can be easily verified. $\diamond$

Before discussing the consistency of ordered assertions, we define the *cycle cut* of a graph which becomes handy for our definition of consistency of order assertions.

Definition 5.5 *(Cycle Cut)*
Let S be an SCC of a directed graph $G(V, E)$. A subset S_{CUT} of edges in S is said to be a *cycle cut* of S if every cycle in S contains at least one edge in S_{CUT}. $\diamond$

Remark 5.2 If S_{CUT} is a cycle cut of S then removing all edges in S_{CUT} from S reduces S to a forest. From Theorem 10.1 given in the appendix, it must be clear that there is no need to attach order assertions to edges which do not belong to any cycle. The property of *cycle cut* provides a natural choice for selecting edges in the U-graph for which order assertions should be attached. As mentioned already, order assertions consist of pair of mappings into well-founded order set, and are attached to selected signed-edges of nontrivial SCC's.

$\diamond$

Definition 5.6 *(Safety of Order Assertions)*
Let S be a non-trivial SCC in the U-graph. We say that a set of order assertions $\{ O_A | A \in S\}$, are *safe ordered assertions on S relative to the given assertions of the nodes and the* P_{OS}, if there is a cycle cut subset S_{CUT} consisting of signed-edges of S such that

1. Consider edge $< A, B >$ in S with arities n and m for A and B respectively,

   (a) If $< A, B >$ is a signed-edge then

   $$F_A^b \wedge F_B^{pre} \wedge (O(1.A, \ldots, n.A) > 0) \;\Rightarrow\; O(1.A, \ldots, n.A) \geq O(1.B, \ldots, m.B)$$

   where F_B^{pre} is the assertion to be satisfied before invoking literal B as defined in definition 5.3.

(b) if $< A, B >$ is an u-edge, and A^I and B^I are instances of A and B, respectively, such that $A^I = B^I$, then $O_A(A^I) = O_B(B^I)$

2. If $< A, B >$ is an edge in S_{CUT} then

$$F_A^b \wedge F_B^{pre} \wedge (O(1.A, \ldots, n.A) > 0) \;\Rightarrow\; O(1.A, \ldots, n.A) > O(1.B, \ldots, m.B)$$

where F_B^{pre} is the preassertion to be satisfied before invoking the literal B. Edge $< A, B >$ is called a strictly decreasing edge.

3. If $O(1.A, \ldots, n.A) > 0$, then for any substitution θ,

$$F_A^a \;\Rightarrow\; O(1.A, \ldots, n.A) \geq O(A.1\theta, \ldots, A.n\theta)$$

4. Let n and m be the arities of A and B, and 0 be the generic name of the zero element of the well-founded ordered set to which O maps. If $< A, B >$ is a strictly decreasing edge, and $O_A(A\theta) = 0$, for some substitution θ, then for all B' such that $< B', A >$ is an in-edge of A, B' and $A\theta$ are not unifiable.

$\diamond$

Intuitively, (1) and (2) say that for each simple cycle, if the least element of the order assertions has not been reached yet, then no edge in the cycle increases order (on the well-founded set W), and there must be an edge in each cycle that strictly decreases the order. Condition (1.b) is necessary due to the fact that the order mappings attached to the vertices B and H may not be the same, where $< B, H >$ is an u-edge. In this case, if they do not agree with each other when the instances of them are essentially the same, then we may fail to show the termination. Consider the following simple program:

$E_1 : p(X) \leftarrow p(X)$

Let us rewrite above clause as $H \leftarrow B$. If we mistakenly choose order assertions of H and B as $size(1.H) + 1$ and $size(1.B)$, respectively, we may find assertions satisfying all conditions (except (1.b)). However, it is easy to see that once clause E_1 is used as an input clause, there is no finite SLDNF-derivation. Condition (3) says that after unification, the order should not be increased. The necessity of this condition follows from the following example.

Example 5.3 Consider the following program:

$E_2 : p(f(X)) \leftarrow p(X)$

It appears that we can establish order assertions to satisfy all the above conditions except (3), by letting $O_H = size(1.H)$ and $O_B = size(1.B)$. Consider the goal $\leftarrow p(X)$. We can see that in each derivation step, the unification essentially raises all orders of subgoals obtained in earlier derivation steps. Hence, even if we locally find the reduction of orders of the head and the subgoal in E_2, we will not be able to establish termination. $\diamond$

Finally, condition (4) says that an instance of the head of clause whose body containing a strictly decreasing edge is mapped to the least element of W, the well-founded set to which all ordered assertions map, then no derivations can apply the instance of an input clause. In other words, unification will not be enabled at this point.

In the next section, we shall see an example of safe order assertions.

6 Steps of the Method

In this section, we introduce the steps of the method to prove termination of logic programs. Before proceeding with the steps, we define the notion of *extended programs* that accounts for the goal also[8] as in [6].

Definition 6.1 Given a normal program P and a normal goal G $\equiv \leftarrow L_1, \ldots, L_m$, we define the *extended program* $P^*(G) = P \cup \{goal \leftarrow L_1, \ldots, L_m\}$, where *goal* is a new predicate symbol with arity 0. $\diamond$

As mentioned above, $(W, <)$ denotes a well-founded set, and 0 is used to denote a generic name for representing the zero elements in W.

Let P be the given normal program and G $\equiv \leftarrow L_1, \ldots, L_m$ be the given normal goal. Let $P^*(G)$ be the extended program of P and and $U(V, E)$ be the U-graph of $P^*(G)$.

Steps

Given

1. the pre- and post-assertions at the vertices of V, and
2. the ordered assertions for selected signed cyclic-edges of $U(V, E)$,

the method establishes[9]

$$\models \{true\} \ \leftarrow goal \ \{true\}$$

provided the following conditions are satisfied:

1. The A-set of each vertex in V_H is consistent.
2. The A-set of each vertex in V_B is safe.

[8] Note that this is necessary since the termination of SLDNF-derivations depends on the goal also.

[9] Here, the assertion *true* corresponds to either success or failure using the computation rule satisfying the pre-assertions of the clause.

3. The set of ordered assertions is safe.

The explanation for the safety and consistency of assertions has already been given through definitions 5.2 - 5.4. The crucial step is the third step which establishes that the computation descends through a well-founded chain. That is, if there is a cycle then there must be a signed edge $< A, B >$ in the cycle with ordered assertion

$$< O(1.A, \ldots, n.A), O(1.B, \ldots, n.B) >$$

such that

$$F_A^b \wedge F_B^b \wedge (O(1.A, \ldots, n.A) > 0) \;\Rightarrow\; (O(1.A, \ldots, n.A) > O(1.B, \ldots, m.B))$$

where $< F_A^b, F_A^a >$ and $< F_B^b, F_B^a >$ are the assertions attached to A and B, respectively. The above relation shows that choosing this direction corresponds to climbing down the well-founded chain. Assuming that the vertex assertions are properly chosen, it can be observed that in the case of a vertex with non-zero number of outgoing edges and $O(1.A, \ldots, n.A) \neq 0$, the subgoal unifies to at least one head of some clause without increasing the order. From the above informal argument, one can infer the existence of a global well-founded ordering. As we are not considering the search rule, it should be clear that we establish universal termination.

The method is illustrated by the permutation program given below.

Example 6.1 Consider the following program PERMUTE discussed in example 5.1 and goal $G \equiv\leftarrow permute(s, t)$. The U-graph of PERMUTE*(G) is shown in Figure 6.1.

Let P_{OS} be the set corresponding to the left-to-right selection (computation rule). The assertions associated with the various vertices (using A_i as the general name of the vertex) are given below:

1. $A_1 = goal: \quad < F_{A_1}^b, F_{A_1}^a > = < true, true >$
2. $A_2 = permute(s, t): \quad < F_{A_2}^b, F_{A_2}^a > = < ground(1.A_2), true >$
3. $A_3 = permute(T, [H|P]): \quad < F_{A_3}^b, F_{A_3}^a > = < ground(1.A_3), true >$
4. $A_4 = r(T, H, R): \quad < F_{A_4}^b, F_{A_4}^a > \; where$
   $F_{A_4}^b = ground(1.A_4), F_{A_4}^a = [ground(A_4.1) \wedge ground(A_4.3) \wedge (A_4.3 \lhd A_4.1)]$
5. $A_5 = p(R, P): \quad < F_{A_5}^b, F_{A_5}^a > = < ground(1.A_5), true >$
6. $A_6 = p([\,], [\,]): \quad < F_{A_6}^b, F_{A_6}^a > = < true, true >$
7. $A_7 = r([B|C], D, [B|E]): \quad < F_{A_7}^b, F_{A_7}^a > \; where$
   $F_{A_7}^b = ground(1.A_7), F_{A_7}^a = [ground(A_7.1) \wedge ground(A_7.3) \wedge (A_7.3 \lhd A_7.1)]$

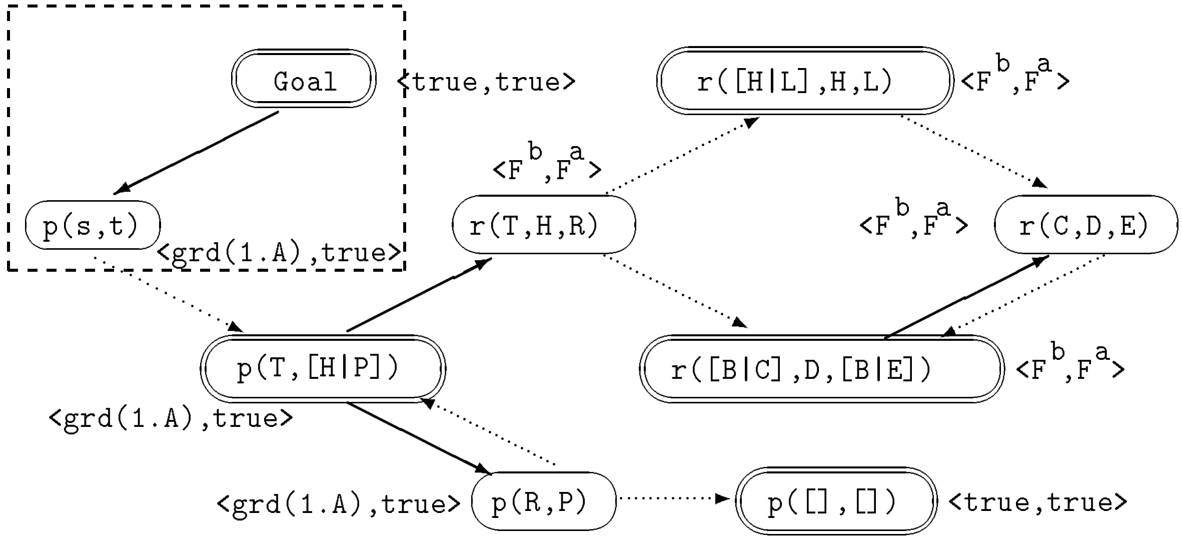

Figure 6.1 : Program PERMUTE

8. $A_8 = r([H|L], H, L):\quad < F^b_{A_8}, F^a_{A_8} > \ where$
   $F^b_{A_8} = ground(1.A_8), F^a_{A_8} = [ground(A_8.1) \wedge ground(A_8.3) \wedge (A_8.3 \lhd A_8.1)]$

9. $A_9 = r(C, D, E):\quad < F^b_{A_9}, F^a_{A_9} > \ where$
   $F^b_{A_9} = ground(1.A_9), F^a_{A_9} = [ground(A_9.1) \wedge ground(A_9.3) \wedge (A_9.3 \lhd A_9.1)]$

$\diamond$

Recall that $\lhd$ stands for proper subterm relation (note that it is a well-founded ordering). Let the well-founded set W be the set of natural numbers. There are two non-trivial SCC's in Figure 6.1, and each of them contains only one signed-edge. Therefore the only choice to form cycle cuts for these two non-trivial SCC's are the sets $\{< p(T, [H|P]) >, p(R, P) >\}$, and $\{< r([B|C], D, [B|E]), r(C, D, E) >\}$ (i.e., $\{< A_3, A_5 >\}$ and $\{< A_7, A_9 >\}$). For the two edges involved in cycles, the order assertions are given below:

1. $< A_3, A_5 >=< p(T, [H|P]), p(R, P) >$ where
   $O(1.A_3, 2.A_3) = size(1.A_3)$
   $O(1.A_5, 2.A_5) = size(1.A_5)$

2. $< A_7, A_9 > = < r([B|C], D, [B|E]), r(C, D, E) >$ where
   $O(1.A_7, 2.A_7, 3.A_7) = size(1.A_7)$
   $O(1.A_9, 2.A_9, 3.A_9) = size(1.A_9)$ where $size(T)$ is the number of (function and constant) symbols occurring in a ground term T.

It is not difficult to see that

1. the A-set of $permute(T,[H|P])$ and $remove([B|C],D,[B|E])$ are consistent with respect to partial ordered sets $\{(1,2)\}$ and ϕ, respectively.
2. A-set of vertices in V_B are safe, and
3. the set of ordered assertions is safe; let us consider the order assertion attached to edge $< A_3, A_5 >$. The preassertion for A_5 is given by $F_{A_5}^{pre} = F_{A_3}^b \wedge F_{A_4}^a \wedge F_{A_5}^b$. From this, we get $1.A_5 \triangleleft 1.A_3$; that is, $1.A_5$ is a proper subterm of $1.A_3$. Thus, $O(1.A_3, 2.A_3) > O(1.A_5, 2.A_5)$ and hence, the proof that the assertion is safe follows.

Therefore, if the first parameter of the goal $\leftarrow permute(s,t)$ is ground, i.e., if the precondition $F^b_{permute(s,t)} = ground(1.A_1)$ of the vertex $permute(s,t)$ is true, then the program terminates by a computation rule (i.e., the partial ordered set) associated with the A-sets of the heads of all non-unit clauses.

It may be noted from the discussion in [2] that the permute program is not recurrent but acceptable. This follows from the fact that recurrent programs ignore computation rules whereas acceptable programs do consider computation rules. Our method considers the computation rules for showing termination. Thus, if the query $\leftarrow$ $permute(x,t)$ where x is not ground and t is ground, we can arrive at a computation rule (in this case, right-to-left) for which the termination can be proved.

7 Soundness of the Method

In the appendix (cf. Theorem 10.1), we have established that an inconclusive SLDNF-derivation corresponds to an infinite path in the U-graph through the notions of extended SLDNF derivation and g-trees. In this section, we show that the method described in the earlier section is sound using the results proved in the appendix.

Definition 7.1 Given an extended normal program $P^*(G)$ of a normal program P and a normal goal G with U-graph $U(V,E)$, let $\mathcal{F}$ be the set of assertions attached to vertices in V. Assume that for each H in V_H, the A-set of H is consistent with respect to a partial order set $P_{OS}(H)$. As explained already in the discussion of definition 5.2, the partial order set captures the minimal information relative to the selection rule. We say that a computation rule is *consistently safe* with respect to $\mathcal{F}$ if, for each clause $H \leftarrow B_1, \ldots B_n$, subgoal B_i is selected only when all B_j succeed, for all j, $(j,i) \in P_{OS}(H)$. $\diamond$

Theorem 7.1 Given a normal program P and a normal goal G $\equiv \leftarrow L_1, \ldots, L_m$, let $P^*(G)$ be the extended program with extended U-graph $U(V,E)$. If

1. there is a set $\mathcal{F}$ of assertions attached to all vertices in V, such that
   (a) for each A-set of a vertex $A \in V_H$ is consistent, and

(b) for each A-set of a vertex $A \in V_B$ is safe.

2. there is a safe set of order assertions attached to all signed edges involved in cycles,

then P $\cup$ $\{\leftarrow goal\}$ has finite SLDNF tree via a consistently safe computation rule with respect to $\mathcal{F}$.
proof:

By simple induction on the length of SLD (SLDNF)-derivations, it is easy to see that under any consistently safe rule with respect to $\mathcal{F}$, the pre-condition of selected subgoal is always satisfied at any SLDNF derivation. Further, by the safety of order assertions it follows that there is a well-founded ordering over all instances of paths in the strongly connected component. Now, let us suppose that there is no finite SLDNF tree. Then, from Lemma 10.2, there must be an infinite extended SLDNF-derivation (cf. Definition 10.7), having the g-tree G_0 = G, $G_1, \ldots$. Now, from Theorem 10.1, it follows that the infinite branch corresponds to an instance of an infinite path in the U-graph. This contradicts the premises that the order assertions form a well-founded order set. Thus, there cannot be an infinite SLDNF-derivation. Hence, the theorem follows. $\diamond$

Now, let us see whether for every terminating logic program, we can derive an asserted program as described above. For this purpose, we confine our attention to positive[10] Horn clause programs (hence, SLD derivations). We do this by applying our method for recurrent programs discussed[11] in [7]. It has been shown in [7] that every total recursive recursive function can be computed by the class of recurrent programs and further, recurrent programs terminate for all bounded goals, i.e., goals whose instances are below some fixed level. In the following, we sketch a method of getting order assertions from the U-graph of a recurrent program. Let P be a recurrent program, i.e., it is recurrent with respect to a level mapping $|\ |$. Then, we have,

For every clause of the form $A \leftarrow B_1, \cdots, B_n$ in $ground(P)$, $|A| > |B_i$ for all i.

Now, we can use ">" as the order for the edge $< A, B_i >$ in the U-graph; in this way, we can assign an order assertion for every edge. With this as the basic set of order assertions, the proof of termination follows in a trivial way relative to the bounded (which again can be specified as pre-assertion in our meta-language) queries.

Thus, it follows, that we can indeed derive an asserted program for a terminating definite logic program. In a similar way, we can derive an asserted program for left-terminating programs as well.

[10] Aspects of SLDNF-completeness in the context of termination has been discussed in [37].

[11] As already discussed while discussing the permutation program, our method considers computation rules – which are not considered by recurrent programs.

8 Heuristics for Assertions

In this section, we briefly discuss some heuristics to arrive at assertions and order assertions for a given program and a given goal. Although our method can be used for establishing logical correctness as well as termination of the programs, we shall confine our attention to termination aspects in the sequel.

Besides the information of the (syntax) structure of the given program, the most important information lies with the given goal. In most cases, the information from the given goal leads to input-output patterns of arguments of the heads and subgoals of the clauses of the programs. Thus, the first step for guessing assertions is to obtain as much information as possible from the goal. This can be done by examining the corresponding U-graph from the root *goal* of the extended program to all reachable nodes in a top-down fashion. It may be noted that the top-down data-flow analysis focuses on the input information (e.g., modings) as illustrated in the following example.

Example 8.1 Consider the following program SUBTRACT which computes the difference (the third argument) of two integers (the first two arguments).

$S_1 : sub(X, 0, X) \leftarrow$
$S_2 : sub(X, s(Y), Z) \leftarrow sub(X, Y, s(Z))$

Let us abbreviate the program as follows (associations of the literals with symbols U,H and B is assumed in the usual manner):

$S_1 : U \leftarrow$
$S_2 : H \leftarrow B$

Consider the goal $\leftarrow sub(t_1, t_2, t_3)$, where t_i is a ground term, $i = 1, 2$, or 3. A simple top-down analysis shows that $ground(i.A)$ could be a part of pre-assertion of A, for all $A \in \{U, H, B\}$. Therefore, as a first step, we may attach the following assertions to nodes U, H, B:

$$< F^b_U, F^a_U > = < ground(i.U), \;\; true >$$

$$< F^b_H, F^a_H > = < ground(i.H), \;\; true >$$

$$< F^b_B, F^a_B > = < ground(i.B), \;\; true >$$

It is not hard to see that the assertions are indeed consistent and safe. $\diamond$

For simple cases such as SUBTRACT, the next step to derive termination would be to derive order assertions for recursive calls (subgoals), i.e., nodes in non-trivial SCC's in the U-graph. In this example, the only signed-edge $< H, B >$ involved in the non-trivial SCC has the following property:

> The literal B is unifiable with heads of rules S_1 and S_2 whenever the second argument is unifiable irrespective of the terms to which the first and third arguments

get bound from H (they always succeed).

Thus, for establishing termination, we have to look for the possibility of non-unifiability of the second argument; otherwise, the SLDNF-derivation will be nonterminating. From the above analysis, it can be seen that each time the recursive subgoal B appears in the derived goal, the second argument of B remains ground if the second argument in the top level goal $\leftarrow sub(t_1, t_2, t_3)$ is ground. In addition, we may find from the syntax an ordering relation based on the size of symbols in the second argument of B. That is, we can define the order assertions for $< H, B >$ defined as

$O_{<H,B>} = size(2.B)$.

Let us place the restriction that the second argument t_2 in any goal $\leftarrow sub(t_1, t_2, t_3)$ is ground. This corresponds to saying that $2.H$ and $2.B$ are ground. From clause S_2, we can observe the following relation between the second argument of head H and the second argument of subgoal B (recall that $s \triangleright t$ means that s is a subterm of term t):

$$2.H = s(Y)\theta \triangleright Y\theta = 2.B.$$

This shows that for each recursive call, $size(2.B)$ does decrease. Furthermore, when $size(2.B) = size(Y\theta)$ reaches 1, then $B\theta$ fails to unify to H. Now, the safety of the order assertion attached to $< H, B >$ follows since the U-graph contains only one simple cycle consisting of $< H, B >$. Hence, we can conclude the termination of SUBTRACT $\cup$ $\{\leftarrow sub(t_1, t_2, t_3)\}$ provided t_2 is a ground term.

Next, let us look at a more complicated example.

Example 8.2 Consider the GCD program given earlier.

$D_1 : gcd(X, 0, X) \leftarrow$
$D_2 : gcd(0, X, X) \leftarrow$

$D_3 : gcd(s(X), s(Y), Z) \leftarrow sub(Y, X, W), gcd(s(X), W, Z)$
$D_4 : gcd(s(X), s(Y), Z) \leftarrow sub(X, Y, W), gcd(W, s(Y), Z)$

Again, let us rewrite clauses $C_3 - C_6$ as follows:

$D_1 : H_1 \leftarrow$
$D_2 : H_2 \leftarrow$

$D_3 : H_3 \leftarrow S_3, B_3$
$D_4 : H_4 \leftarrow S_4, B_4$

In this example, it should be clear that a simple data-flow (top-down) analysis will not be enough for showing the termination of the non-trivial SCC, S_{GCD} consisting of $\{H_3, B_3, H_4, H_4\}$. However, it can be seen that the producer-consumer concept and a bottom-up style analysis with reference to subgoal S_3 (resp. S_4) in the body of clause D_3 (resp. D_4) can provide information necessary for establishing termination. Bottom-up analysis shows that whenever the subgoals S_3 and S_4 succeed, the success pattern of their arguments (outputs) always carry some information which was discarded in the analysis of the previous example. Producer-consumer concept enables us to show that the information generated by S_3 (resp. S_4) in the body of D_3 (resp. D_4) can be passed to B_3 (resp. B_4) for establishing the well-founded relation over recursive subgoals B_3, B_4 and heads H_3, H_4. An easy way to carry output information of subgoals S_3 and S_4, is to strengthen especially the post-assertions of predicate sub used earlier, as follows:

$< F^b_U, F^a_U > = < ground(1.U),\ ground(U.3) \wedge (U.1 = U.3) >$

$< F^b_H, F^a_H > = < ground(1.H) \wedge ground(2.H),\ ground(H.3) \wedge (H.1 \trianglerighteq H.3) >$

$< F^b_B, F^a_B > = < ground(1.B) \wedge ground(2.B),\ ground(B.3) \wedge (B.1 \trianglerighteq B.3) >$

Let θ denote an answer substitution. Since $U = sub(X, 0, X)$, therefore $U.1 = X\theta = U.3$ is trivial. Note that $H = sub(X, s(Y), Z)$ and $B = sub(X, Y, s(Z))$. If $B.1 = X\theta \trianglerighteq s(Z)\theta = B.3$ then we have $H.1 = X\theta \triangleright Z\theta = H.3$. From these simple facts, we can find that the consistency and safety still hold.

It is interesting to point out that the safety property of assertions is very useful to verify the effectiveness of output information when the recursive subgoals are involved in general cases.

From the above newly derived assertions, we can get the following relations,

$$2.H_3 = s(Y)\theta \triangleright Y\theta = S_3.1 \trianglerighteq S_3.3 = W\theta = 2.B_3, \quad (i.e., 2.H_3 \triangleright 2.B_3) \tag{7}$$

$$1.H_3 = s(X)\theta \triangleright X\theta = S_3.1 \trianglerighteq S_3.3 = W\theta = 1.B_3, \quad (i.e., 1.H_3 \triangleright 1.B_3) \tag{8}$$

$$2.H_4 = s(Y)\theta \triangleright Y\theta = S_4.1 \trianglerighteq S_4.3 = W\theta = 2.B_4, \quad (i.e., 2.H_4 \triangleright 2.B_4) \tag{9}$$

$$1.H_4 = s(X)\theta \triangleright X\theta = S_4.1 \trianglerighteq S_4.3 = W\theta = 1.B_4, \quad (i.e., 1.H_4 \triangleright 1.B_4) \tag{10}$$

where θ is an answer substitution of S_3. Later, we shall see that this information indeed can help us to derive a safe order assertions for edge $< H_3, B_3 >$ and $< H_4, B_4 >$. Based on (7) – (10), we can derive assertions for all nodes in the U-graph of GCD as given below:

$< F^b_{H_1}, F^a_{H_1} > = < ground(1.H_1),\quad ground(H_1.1) \wedge ground(H_1.3) >$

$< F^b_{H_2}, F^a_{H_2} > = < ground(2.H_2),\quad ground(H_2.2) \wedge ground(H_2.3) >$

$< F^b_{H_3}, F^a_{H_3} > = < ground(1.H_3) \wedge ground(2.H_3),\quad ground(H_3.3) >$

$< F^b_{S_3}, F^a_{S_3} > = < ground(1.S_3) \wedge ground(2.S_3),\quad ground(S_3.3) \wedge (S_3.1 \triangleright S_3.3)$

$< F^b_{B_3}, F^a_{B_3} > = < ground(1.B_3) \wedge ground(2.B_3),\quad ground(B_3.3) >$

$< F^b_{H_4}, F^a_{H_4} > = < ground(1.H_4) \wedge ground(2.H_4),\quad ground(H_4.3) >$

$< F^b_{S_4}, F^a_{S_4} > = < ground(1.S_4) \wedge ground(2.S_4),\ ground(S_4.3) \wedge (S_4.1 \triangleright S_4.3) >$

$< F^b_{B_4}, F^a_{B_4} > = < ground(1.B_4) \wedge ground(2.B_4),\quad ground(B_4.3) >$

Together with the newly derived assertions for U, H, B of the previous example, we can easily check that the assertions are safe and consistent.

The remaining work is to look for order assertions for the non-trivial SCC S_{GCD}. The only possible cycle cut in S_{GCD} is $\{<H_3, B_3>, <H_4, B_4>\}$. Using analysis given above, it is easy to arrive at the following assertions:

$O_{<H_3,B_3>} = (size(1.B_3), size(2.B_3))$
$O_{<H_4,B_4>} = (size(1.B_4), size(2.B_4))$

In this example, the well-founded order adopted is ordering of pairs of integers. $(m, n) \geq (m', n')$ iff $m \geq m'$ and $n \geq n'$.

Since $1.H_3$ and $2.H_3$ (resp. $1.H_4$ and $2.H_4$) are ground, and subgoal B_3 (resp. B_4) is selected only when S_3 (resp. S_4) succeeded, by (7) – (10) we get

1. in each recursive call of B_3 (resp. B_4), the one of $size(1.B_3)$ and $size(2.B_3)$ (resp. $size(1.B_4)$ and $size(2.B_4)$) order assertion of $<H_3, B_3>$ (resp. $<H_4, B_4>$) always decreases,

2. when either component of the assertions reaches 1 then it fails to unify to heads H_3 and H_4.

The safety of order assertions now follows easily.

To sum up, the above analysis establishes the termination of GCD $\cup \{\leftarrow gcd(s_1, s_2, s_3)\}$ provided that s_1 and s_2 are ground terms. $\diamond$

In general, we suggest the following broad steps for showing the correctness of programs through our method:

step 1. Derive pre-assertions from the information obtained from the given goal, e.g., modings for all reachable nodes from the root in the U-graph by top-down data-flow analysis.

step 2. Establish consistency and safety of assertions.

step 3. Derive all possible order assertions for signed-edges in a bottom-up fashion, i.e., derive order assertions starting from the lowest SCC's to the root SCC's, in the condensed graph. Then, check if (i) order assertions are safe, and (ii) For each non-trivial SCC, there is selected edge that forms the cycle-cut for the SCC. If so, we are done; otherwise, go to the next step.

step 4. Derive output information which can be used in the producer-consumer relation in a bottom-up fashion, and integrate it to the assertions attached to each node. Then, go to step 3.

9 Discussion

In the preceding sections, we have described a method for showing the termination of logic programs using U-graphs. The method is simple and allows us to concentrate only on those parts of the program that are not easily amenable for understanding the termination aspects intuitively. In other words, the U-graph abstraction allows us to localize the task of finding the well-founded set and enables us to use some of the termination techniques of term rewriting systems such as those described in [12] for arriving at order assertions. From the U-graph, several useful properties such as disjointness (for instance, the U-graph for the program consisting of two clauses $p(X) \leftarrow p(f(X))$ and $p(X) \leftarrow$ will have two disjoint graphs), reachability *etc.* can be observed easily. In [36], we have identified a class of terminating general programs based on ths static analysis of U-graphs and embedding orders.

Further, one can use semantic information for arriving at orders. For instance, consider the following example:

same_generation(x,y) :- parent(x,xp), parent(y,yp), same_generation(xp, yp)

same_generation(x,x) :-

In the above program, *parent* consists of all facts (parent-children relations). The termination of such a program is based on the condition that there is no circular relation of data items in the relation *parent*. For example, having $parent(a, b)$ and $parent(b, a)$ leads to a circular relation. In other words, if we assume that there is no cyclicity, then we can define or find an ordering among data items to realize a well-founded order – which leads to a proof of termination. In fact, one can use type information effectively in proving the termination of logic programs; the metalanguage can be used for specifying certain typing informations. It may be noted that the type assertions are in fact used as annotations for speeding up the execution, checking errors of a program *etc.* [28].

Another interesting point may be observed from the analysis of the two cycles in Example 6.1. For instance, we can have the following two sets of order assertions:

- The first set corresponds to a goal $\leftarrow perm(s, t)$, where s is a ground term, which has a finite SLD-derivation under the Prolog computation rule.
- The second set can be derived by considering the goal $\leftarrow perm(s, t)$, where t is a ground term. In this case, we can see that we have a different order to keep the consistency of clause

  $$permute(T, [H|T]) \leftarrow remove(T, H, R), permute(R, P)$$

  and hence, we need different computation rules to achieve termination. This is also reflected in our earlier discussion for the termination of queries of the form $\leftarrow permute(x, y)$ where either the first is ground or the second is ground.

Another advantage of the method is that pre-assertions for the predicates can be distributed; this is very helpful in the understanding of the program. Further, we can check the pre-assertions to see whether the logic program can be transformed to a functional program; for instance conditions given in [31] for the transformation of logic programs to functional programs can be derived in a natural way.

In short, the method described is a simple pragmatic method for proving termination of logic programs. It permits the effective use of programmer's intuition and the term structure for deriving various properties of logic programs. The power and scope of the method lies in effectively using annotations for specifying:

1. directionality, producer-consumer relations, and moding annotations (which describe sets of possibly non-ground atoms).

2. type annotations; type annotations use sets of ground atoms only and thus, one can use it to define a particular classes of computations only.

3. control information such as call patterns, parallel implementations, efficient implementations, *etc.*.

4. order on term-structure based on the reachability of the predicates; this feature helps in achieving modularity. This feature is helpful is using proofs of original fragments in modified programs whenever possible.

5. selection rules: As discussed already, the computation rule is captured through a partial order set (cf. Definition 5.2). The consistency rule follows the order of selection as indicated after Definition 5.2 for the Prolog's computation rule.

Various termination notions as highlighted earlier which do not involve search rules as well as various computation (selection) rules (cf. [27]) can be handled [12] using the assertion language described earlier. For including search rule, one needs to include assertions to reflect the unifiability/otherwise of the subgoals relative to the clause chosen under the given search rule. Our main interest has been to pursue the verification of logic programs using a local analysis. For this reason, in our investigations we have considered selection rules and ignored search rules. Our efforts in the verification of logic programs using our method has been quite encouraging. We are working towards adapting our technique for loop checking [8]. In particular, we are working towards checking the acceptability condition on infinitely many ground instances of a clause. As we have unification information in the U-graphs, we are exploring methods of verification of Prolog programs that uses extensions of execution models as envisaged in [19].

[12] Most of the termination methods [4] ignore the search rule.

Acknowledgments

It is a pleasure to thank Deepak Kapur, SUNY, Albany and M.R.K. Krishna Rao for many invaluable discussions. The authors are grateful to the referees whose comments and suggestions have greatly improved the presentation of the paper.

References

[1] Apt, K.R., and M. Bezem, *Acyclic programs*, New Generation Computing, 9, pp. 335-363, 1991.

[2] Apt, K.R. and D. Pedreschi, *Studies in pure Prolog: termination*, Proc. Symposium on Computational Logic (edited by J.W. Lloyd), Springer-Verlag, 1990, pp. 150-176.

[3] Apt, K.R., *Logic Programming*, Handbook of Theoretical Computer Science, Edited by J. van Leeuwen, Elsevier Science Publishers, 1990, pp. 494-574.

[4] Apt, K.R. and D. Pedreschi, *Reasoning about termination of Prolog programs*, TR, CWI 1992.

[5] Baudinet, M., *Proving termination properties of Prolog programs: a semantic approach*, Proc. LICS, 1988, pp. 336-347.

[6] Barbuti, R., M. Martelli, *A Tool to Check the Non-floundering Logic Programs and Goals*, First International Workshop on Programming Languages Implementation and Logic Programming, LNCS 348. pp. 58-67. 1988.

[7] Bezem, M., *Characterizing termination of logic programs with level mappings*, In E.L. Lusk and R.A. Overbeek, Editors, Proc. of the North American Conference on Logic Programming, pp. 69-80, MIT Press, 1989.

[8] Bol, R., **Loop Checking in Logic Programming**, Ph.D. Thesis, CWI, Amsterdam, 1991.

[9] Cavedon, L., *Continuity, consistency, and completeness properties for logic programs*, Proc. ICLP 89, pp.571-584, June 1989.

[10] Cavedon, L., J. W. Lloyd, *A Completeness Theorem for SLDNF-resolution*, Tech. Report CS-87-06, University of Bristol, 1987.

[11] Clark, K. L., *Negation as Failure*, in Symposium on Logic and Data Bases, (Gallaire, H. and J. Minker Eds.), Plenum Press, 1978.

[12] Dershowitz, N., *Termination of Rewriting*, J. of Symbolic Computation, (1987) 3, pp. 69-116.

[13] Drabent, W., J. Maluszynski, *Inductive Assertion Method for Logic Programs*, J. Theoretical Computer Science, 59 (1988) 133-155.

[14] De Schreye, D., and K. Verschaetse, *Termination Analysis of definite logic programs with respect to call patterns*, TR, Leuven, 1992.

[15] De Schreye, D., and S. Decorte, *Termination of LP: The never-ending story*, submitted for publication, 1993.

[16] Francez, N., O. Grumberg, S. Katz, A. Pnueli, *Proving Termination of Prolog Programs*, Proc. of Logics of Programs Conference, Brooklyn NY, LNCS 193, 89-105, 1985.

[17] Floyd, R. M., *Assigning Meanings to Programs*, Proc. AMS symposium on Applied Mathematics, 19, Providence R.I., 1967.

[18] Heck, N., J. Avenhaus, *On Logic Programs with Data-Driven Computations*, LNCS No.225, 1986, 433-443.

[19] Kanamori, T., H. Seki, *Verification of PROLOG Programs Using an Extension of Execution*, Proc. of the Third Intern. Conf. on Logic Programming 1987.

[20] Kapur, D. and H. Zhang, *An overview of Rewrite Rule Laboratory (RRL)*, Proc. of Rewrite Techniques and Applications, Springer-Verlag, LNCS, 355, pp. 559-563.

[21] Krishna Rao, M.R.K., D. Kapur and R.K. Shyamasundar, *A transformational methodology for proving termination of logic programs*, Proc. of the Computer Science Logic 91, Lecture Notes in Computer Science, 626, pp. 213-226, Springer Verlag, 1992.

[22] Krishna Rao, M.R.K., P. Pandya and R.K. Shyamasundar, *Verification Tools in the Development of Provably Correct Compilers*, Proceedings of the Conference on Formal Methods Europe, LNCS 670, pp. 442-661, April 1993.

[23] Kunen, K., *Signed Data Dependencies in Logic Programs*, Computer Science Technical Report 719 University of Wisconsin - Madison, 1987.

[24] Krishna Rao, M.R.K., D. Kapur and R.K. Shyamasundar, *Proving Termination of GHC Programs*, Proc. ICLP 93, Budapest, MIT Press, Cambridge, pp. 720-736.

[25] P. Lescanne, P., *Computer experiments with the REVE term rewriting systems generator*, in proc. 10^{th} ACM POPL (1983), pp. 99-108.

[26] Lescanne, P., *Termination of rewriting systems by elementary interpretations*, in proc. Algebraic and Logic Programming, ALP'92, Pisa.

[27] Lloyd, J. W., **Foundation of Logic Programming,**, Revised Extended Version, Springer-Verlag, 1987.

[28] Mycroft, A. and O'Keefe, R.A., *A polymorphic type system for Prolog*, Artificial Intelligence, 23, pp. 295-307, 1984.

[29] Nilsson, N. J., **Principles of Artificial Intelligence,** Spring-Verlag, Berlin, 1982.

[30] Plümer, L., **Termination Proofs for Logic Programs**, Ph.D. Thesis, Lecture Notes in Artificial Intelligence, 446, Springer-Verlag, Berlin, 1990.

[31] Reddy, U. S., *On the Relationship between Logic and Functional Languages*, in Logic Programming: Functions, Relations, and Equations (Degroot, D. and G. Lindstrom, Eds), Prentice-Hall, New Jersey, 1986.

[32] Shepherdson, J. C., *Negation as Failure: A Comparison of Clark's Completed Data Base and Reiter's Closed World Assumption*, J. Logic Programming, 1984:1:51-79.

[33] Shyamasundar, R.K., M.R.K. Krishna Rao and D. Kapur, *Rewriting Concepts in the Study of Termination of Logic Programs*, Proc. ALPUK'92 Conference (edited by K. Broda), Workshops in Computing Series, Springer-Verlag, 1992, pp. 3-20.

[34] Ullman, J.D. and A. Van Gelder, *Efficient tests for top-down termination of logical rules*, JACM, 35 (2), 1988, pp. 345-373.

[35] Vasak, T., J. Potter, *Characterization of Terminating Logic Programs*, IEEE Symposium on Logic Programming, 1986.

[36] Wang, B., R. K. Shyamasundar. *Towards a characterization of termination of logic programs*, Proc. of PLIPS, LNCS 456, pp. 204-221, Springer-Verlag.

[37] Wang, B., **Verification, Termination, and Completeness of Logic Programs**, Ph. D. Thesis, Computer Science Department, The Pennsylvania State University, University Park, PA, August 1990.

10 APPENDIX

Relationship between U-graphs and SLDNF-derivations

In this section, we show that U-graph abstraction is a succinct abstraction of a program from the point of view of termination. Using the notion of *g-trees* we relates a path in the U-graph and a sub-derivation of a given SLDNF-derivation. Using *g-trees* we establish a correspondence between paths of U-graphs and SLDNF-derivations. through

Note, that we consider SLDNF derivations instead of SLD derivations as the latter can be obtained as a special case of the former. First, we recall some definitions related to SLDNF-derivations from [27].

The basic idea for the SLNDNF-derivation lies in using SLD-resolution, augmented by the negation as failure rule. When a positive literal is selected, we use essentially SLD-derivation to derive a new goal. However, when a ground negative literal is selected, the goal answering process is entered recursively. The negative subgoals must be answered individually and is referred to as *lemmas* which must be established to compute the result. Note that the lemmas do not create any bindings; they only succeed or fail. That is, negation as failure is purely a test.

In the following, first we give the definition of SLDNF-derivation assuming the definition of SLDNF-refutation and finitely failed SLDNF-tree as in [27].

Definition 10.1 *(SLDNF-Derivation)* Let P be a normal program and G a normal goal. An SLDNF-derivation of $P \cup \{G\}$ consists of a (finite or infinite) sequence $G_0 = G, G_1, \cdots$ of normal goals, a sequence $C_1, C_2, \cdots$ of variants of program clauses (called *input clauses*) of P or negative ground literals[13] and a sequence $\theta_1, \theta_2, \cdots$ of substitutions satisfying the following conditions:

1. For each i, either
   (a) G_{i+1} is derived from G_i and an input clause C_{i+1} using θ_{i+1}, or
   (b) G_i is $\leftarrow L_1, \cdots, L_m, \cdots, L_p$, the selected literal L_m in G_i is ground negative literal $\neg A_m$ and there is a finitely failed SLDNF-tree for $P \cup \{\leftarrow A_m\}$. In this case, G_{i+1} is $\leftarrow L_1, \cdots, L_{m-1}, L_{m+1}, \cdots, L_p$, θ_{i+1} is the identity substitution and C_{i+1} is $\neg A_m$.
2. If the sequence $G_0, G_1, \cdots$ of goals is finite, then either
   (a) The last goal is empty, or

[13] In Apt's definition (cf. [3]) one finds the term C_i *arbitrary* in place of our disjunctive phrases here. In the extended-SLDNF-derivation to be given subsequently, we explicitly define the clauses so that there is no anomaly.

(b) The last goal is $L_1, \cdots, L_m, \cdots, L_p$ and there is no program clause (variant) in P whose head unifies with the selected atom L_m, or

(c) The last goal is $L_1, \cdots, L_m, \cdots, L_p$, such that the selected literal L_m is a ground negative literal $\neg A_m$ and there is an SLDNF-refutation of $P \cup \{\leftarrow A_m\}$.

$\diamond$

Definition 10.2 A *Safe Computation Rule* is a function from a set of normal goals, none of which consists entirely of non-ground negative literals, to a set of literals such that the value of the function for such a goal is either a positive literal or a ground negative literal, called the *selected* literal, in that goal. $\diamond$

Definition 10.3 *Safe* SLDNF-derivations are SLDNF derivations using safe computation rules. $\diamond$

Definition 10.4 *(SLDNF-tree)* Let P be a normal program and G be a normal goal. An SLDNF-tree of $P \cup \{G\}$ is a tree satisfying:

1. Each node of the tree is a (possibly empty) normal goal.

2. The root node is G.

3. Let $\leftarrow L_1, \cdots, L_m, \cdots, L_p$ be a non-leaf node in the tree and suppose that L_m is selected. Then, either

   (a) L_m is an atom and for each program clause (variant) $A \leftarrow M_1, \cdots, M_q$ such that L_m and A are unifiable with mgu θ, the node has a child $\leftarrow (L_1, \cdots, L_{m-1}, M_1, \cdots, M_q, L_{m+1}, \cdots, L_p)\theta$, or

   (b) L_m is aground negative literal $\neg A_m$ and there is a finitely failed SLDNF-tree for $P \cup \{\leftarrow A_m\}$ in which case the only child is $L_1, \cdots, L_{m-1}, L_{m+1}, \cdots, L_p$.

4. Let $\leftarrow L_1, \cdots, L_m, \cdots, L_p$ $(p \geq 1)$ be a leaf node in the tree and suppose that L_m is selected. Then, either

   (a) L_m is an atom and there is no program clause (variant) in P whose head unifies with L_m, or

   (b) L_m is ground negative literal, $\neg A_m$ and there is an SLDNF-refutation of $P \cup \leftarrow A_m$.

5. Nodes with empty goal have no children.

$\diamond$

Definition 10.5 *(Main tree)* Given a normal program P and a normal goal G, we refer the SLDNF-tree for $P \cup \{G\}$ as the *main tree* and the SLDNF-tree for $P \cup \{\leftarrow A\}$ as the *lemma tree* where $\neg A$ is the selected negative subgoal in the main tree/lemma tree. $\diamond$

Definition 10.6 We say that an SLDNF-derivation is *successful* if it is finite and the last goal is the empty goal. An SLDNF-derivation is *failed* if it is finite and the last goal is not the empty goal. An SLDNF-derivation is *inconclusive* if it neither succeeds nor fails. $\diamond$

From the above definition, it must be evident that an inconclusive derivation may be finite (due to lemma trees becoming infinite) or infinite (due to some branch of SLDNF tree becoming infinite), since derivations corresponding to lemma trees of ground negative literals is not considered. This is the main reason for defining an extension of SLDNF-derivation in the sequel.

The following lemma aids in establishing the relationship between U-graphs and SLDNF-derivations.

Lemma 10.1 Given two atoms A and B, if A and B are not unifiable then $A\theta$ and B are not unifiable, for any substitution θ such that $lvar(\theta) \cap var(B) = \phi$[14].
Proof:
Suppose there is a substitution δ such that

$$lvar(\delta) \cap var(B) = \phi$$

and $A\delta$ and B are unifiable with mgu σ. Since all the variables occurring in δ are not in B, we get

$$B = B\delta$$
$$A\delta\sigma = B\sigma = B\delta\sigma$$

The last equation implies that A and B are unifiable, contradicting the hypothesis. Hence, the lemma. $\diamond$

It may be noted that in U-graphs, there are no u-edges that correspond to a literal in the body of a clause and the head of a clause having the same predicate symbol that are not unifiable; in other words, the paths in the U-graph relates to the possible reachable SLDNF-derivations.

It can be easily seen that a program is nonterminating if:

1. Some SLDNF-derivation is infinite, or

[14] $var(B)$ denotes all variables occurring in literal B and $lvar(\theta)$ denotes all variables replaced by substitution θ.

2. Some lemma tree is infinite, or

3. There are infinite number of lemma trees.

The situation corresponding to (1) is already integrated in the definition of the SLDNF-derivation itself. In this case, the derivation tree itself will be infinite and hence, there will be an infinite branch (from König's lemma). However, (2)-(3) are not kept track of in the definition of SLDNF-derivation since negative literals do not provide an answer substitution. In other words, the underlying SLDNF-tree need not be infinite even though the program is nonterminating. We would like to capture a structure which will enable us to use the König's lemma in a straightforward way. Such a relation would establish a formal relationship between paths in U-graphs and SLDNF-derivations. We do this by extending the definition of SLDNF-derivation to account for the lemma tree also. The formal extension is captured in the following definition:

Definition 10.7 *(Extended SLDNF Derivation)*
Let S_D be an SLDNF derivation of P $\cup$ {G} which consists of a sequence $G_0 = \mathrm{G}, G_1, G_2, \ldots$, a sequence of normal goals, a sequence $C_1, C_2, \ldots$ of input clauses, and a sequence $\theta_1, \theta_2, \ldots$ of mgu's. An *extended SLDNF derivation* S_D^d of S_D consists of a sequence $G'_0 = \mathrm{G}, G'_1, G'_2, \ldots$ of (sequences of) normal goals, a sequence $C'_1, C'_2, \ldots$ of (sequences of) input clauses, and a sequence $\theta'_1, \theta'_2, \ldots$ of (sequences of) mgu's, such that

1. *No lemma trees*: In this case S_D^d and S_D would be the same.

2. *Lemma trees* : Let $L_T^{(1)}, L_T^{(2)}, \ldots$ be the sequence (possibly infinite) of lemma trees generated due to some subgoal during the derivation, and $G_D^{(i)} \equiv G_0^{(i)}, G_1^{(i)}, \ldots, G_{n_i}^{(i)}$ be the branch in $L_T^{(i)}$ with corresponding sequences $C_D^{(i)} \equiv C_1^{(1)}, \ldots, C_{n_i}^{(i)}$ of input clauses and $\theta_D^{(i)} \equiv \theta_1^{(i)}, \ldots, \theta_{n_i}^{(i)}$ of mgu's such that the selected subgoal (negative) in $G_{n_i}^{(i)}$ is the root of $L_T^{(i+1)}$. Let $G_D^{(i)'} \equiv G_1^{(i)}, \ldots, G_{n_i}^{(i)}$. Then S_D^d consists of the concatenations of (i) normal goal sequence of S_D and $G_D^{(1)'}, G_D^{(2)'}, \ldots$, (ii) input clause sequence of S_D and $C_D^{(1)}, C_D^{(2)}, \ldots$, and (iii) mgu sequence of S_D and $\theta_D^{(1)}, \theta_D^{(2)}, \ldots$. $\diamond$

If S_D is an infinite or inconclusive SLDNF derivation, then an extended SLDNF derivation is a collection of derivations in the main and lemma trees.

Definition 10.8 *(Instances of Paths)*
Given a program P with U-graph $U(V, E)$, the instance of path $H_1, B_1, H_2, B_2, \ldots, H_n, B_n$ with respect to substitutions θ_1,δ_1,δ_2,$\ldots \delta_n$ is defined as the sequence

$$H_1\theta_1,\ B_1\theta_1\delta_1,\ H_2\theta_2,\ B_2\theta_2\delta_2,\ \ldots,\ H_i\theta_i,\ B_i\theta_i\delta_i,\ \ldots,\ H_n\theta_n, B_n\theta_n\delta_n$$

θ_i is the mgu of $B_{i-1}\theta_{i-1}\delta_{i-1}$ and H_i, $\forall i, 2 \leq i$. We use the notation H_1^I,B_1^I,H_2^I,B_2^I,$\ldots$ B_n^I to denote an instance. $\diamond$

	goals	input clauses
G_0	$\equiv \ \leftarrow gcd(s(s(s(0))), s(s(0)), X), sub(X, s(0), Y)$	
G_1	$\equiv \ \leftarrow sub(s(s(0)), s(0), W_1), gcd(W_1, s(s(0)), X), sub(X, s(0), Y)$	C_4
G_2	$\equiv \ \leftarrow sub(s(s(0)), 0, s(W_1)), gcd(W_1, s(s(0)), X), sub(X, s(0), Y)$	C_6
G_3	$\equiv \ \leftarrow gcd(s(0), s(s(0)), X), sub(X, s(0), Y)$	C_5
G_4	$\equiv \ \leftarrow sub(s(0), 0, W_2), gcd(s(0), W_2, X), sub(X, s(0), Y)$	C_3
G_5	$\equiv \ \leftarrow gcd(s(0), s(0), X), sub(X, s(0), Y)$	C_5
G_6	$\equiv \ \leftarrow sub(0, 0, W_3), gcd(s(0), W_3,), sub(X, s(0), Y)$	C_3
G_7	$\equiv \ \leftarrow gcd(s(0), 0, X), sub(X, s(0), Y)$	C_5
G_8	$\equiv \ \leftarrow sub(s(0), s(0), Y)$	C_1
G_9	$\equiv \ \leftarrow sub(s(0), 0, s(Y_1))$	C_6
G_{10}	$\equiv \ \leftarrow []$	C_5

Figure 10 : $SLDNF_{GCD}$ **of GCD**$\cup$**{**$\leftarrow gcd(s(s(s(0))), s(s(0)), X), sub(X, s(0), Y)$**}**

Later, we shall see the role played by substitutions $\theta_0, \delta_1, \delta_2, \ldots \delta_n$.

Definition 10.9 *(G-trees)*
Let S_D^d be an extended SLDNF derivation of P $\cup$ {G} which consists of a sequence of normal goals G_0 = G, G_1, $G_2, \ldots$, a sequence of input clauses C_1, $C_2, \ldots$, and a sequence of substitutions θ_1, $\theta_2, \ldots$. We define the *g-tree* $G_T(S_D^d)$ of S_D^d as follows. The set of nodes in $G_T(S_D^d)$ are the non-empty goals $G_0, G_1, \ldots$ and an extra node G_{-1}. The root of the $G_T(S_D^d)$ is G_{-1}. $< G_i, G_j >$ is an edge in $G_T(S_D^d)$, $i \geq 0$, if the selected subgoal of G_j is from the input clause C_{i+1}. $< G_{-1}, G_j >$ is an edge in $G_T(S_D^d)$ if the selected subgoal of G_j is from the top level goal G (i.e., G_0). $\diamond$

The purpose of introducing g-trees is to establish a relationship between an extended SLDNF derivation and a set of instances of paths in a U-graph. In general, a subsequence of normal goals of a derivation does not correspond to an instance of a path in U-graph. However, each path $G_{-1}, G_{i_0}, G_{i_1}, G_{i_2}, \ldots$ in a g-tree is obtained along (corresponds to) the instance H_1^I, $B_1^I, \ldots$ of a path $H_1, B_1, \ldots$, where H_k is the head of input clause $C_{i_{k-1}+1}$, and B_k^I is the selected subgoal of G_{i_k}, for all $k \geq 1$.

Example 10.1 Consider the GCD program shown in example 3.1 with the goal

$\leftarrow gcd(s(s(s(0))), s(s(0)), X), sub(X, s(0), Y)$.
Figure 10 shows the SLDNF-derivation $SLDNF_{GCD}$ under Prolog rule.

The g-tree, G_T, of $SLDNF_{GCD}$ is shown in Figure 10. It is easy to see that the subgoals selected in the path G_0, G_3, G_5, G_7 in G_T, correspond to the heads of input clauses D_4, D_3, D_3, D_1. In other words,

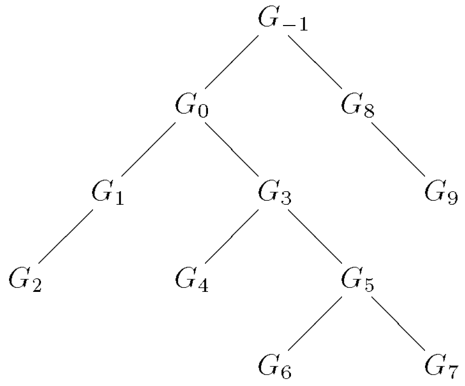

Figure 10 of $SLDNF_{GCD}$

$$gcd(s(s(s(0))), s(s(0)), X),\ gcd(s(0), s(s(0)), X),\ gcd(s(0), s(s(0)), X),\ gcd(s(0), s(0), X),$$
$$gcd(s(0), s(0), X),\ gcd(s(0), 0, X),\ gcd(s(0), 0, s(0))$$

indeed forms an instance of path

$$gcd(s(X), s(Y), Z),\ gcd(W, s(Y), Z),\ gcd(s(X), s(Y), Z),\ gcd(s(X), W, Z),$$
$$gcd(s(X), s(Y), Z),\ gcd(s(X), W, Z),\ gcd(X, 0, X).$$

in Figure 3.1. $\diamond$

Lemma 10.2 Let $G_{-1}, G_{i_0}, G_{i_1}, G_{i_2}, \ldots$ be a path in the g-tree of a given extended SLDNF derivation. If the head of input clause $C_{i_{k-1}+1}$ is H_k ($H_k^I = H_k\theta_{i_{k-1}+1}$), and the selected subgoal of G_{i_k} is B_k, for $k \geq 1$, then $H_1^I, B_1^I, H_2^I, B_2^I, \ldots$ is an instance of a path in the U-graph.
proof:

By definition, $H_1, B_1, H_2, B_2, \ldots$ is a path in the given U-graph. Since B_k is obtained from input clause $C_{i_{k-1}+1}$, by definition of g-tree, $B_k^I = B_k\theta_{i_{k-1}+1}\cdots\theta_{i_k}$.

Let $\delta_k = \theta_{i_{k-1}+1}\cdots\theta_{i_k}$, $k \geq 1$. We can easily see that $H_1^I, B_1^I, H_2^I, B_2^I, \ldots$ is indeed an instance of H_1, B_1, H_2, B_2, ... $\diamond$

In the above lemma, we say $G_{i_1}, G_{i_2}, \ldots$ is a subsequence of goals *corresponding to the selected subgoal* B_0 *and path* H_1, B_1, H_2, ...

Lemma 10.3 There is an inconclusive SLDNF derivation if and only if there is an infinite extended SLDNF derivation.

proof:

Follows trivially from the definition. $\diamond$

Lemma 10.4 There is an inconclusive SLDNF derivation if and only if there is an infinite branch in the g-tree.

proof:

(if): Trivial.

(only if): By the above lemma, we can conclude that if there is an inconclusive SLDNF derivation, then there is an infinite extended SLDNF derivation. Let S_D^d consisting of a sequence of goals G_0, $G_1, \ldots$ be an infinite extended SLDNF derivation. Since G_i $\forall i, i \geq 1$ is a node in $G_T(S_D^d)$, it follows that $G_T(S_D^d)$ contains infinite nodes. Secondly, since each goal in the sequence $G_0, G_1, \ldots$ is obtained by using one input clause only, by definition of g-tree, each node in $G_T(S_D^d)$ has bounded degree. Now, by König's lemma it follows that an infinite tree with bounded degree must have an infinite branch. $\diamond$

By lemma 10.3, we know that if an extended SLDNF derivation is infinite then there is an instance of an infinite path in the U-graph. In other words, if there is no instance of an infinite path then there is no possible infinite extended SLDNF derivations. The connections between success- or finite failure- SLDNF derivations and the g-trees follow from the definitions in a natural way. The following theorem captures the inconclusive SLDNF derivations in terms of U-graphs.

Theorem 10.1 Let P be a normal program and G be a goal. P $\cup$ {G} has an inconclusive SLDNF derivation if and only if there is an instance H_1^I, B_1^I, ... of an infinite path H_1, B_1, ... in the U-graph of P, such that there is a subgoal A in G such that A and H_1 are unifiable, and H_1^I is an instance of A.

proof:

(only if): From lemma 10.4 we can conclude that if there is an inconclusive SLDNF derivation then there is an infinite branch in the g-tree. Together with the fact that each path in the g-tree corresponds to an instance of a path in the U-graph, the result follows.

(if): The proof follows easily by constructing the computation rule from the infinite instance of a path in the u-graph. $\diamond$

Note that the above theorem does not imply that if there is an instance of an infinite path in the U-graph, then P $\cup$ {G} does not have any conclusive SLDNF-derivation. What all it means is that there exists a computation rule that will find the corresponding inconclusive derivation tree.